НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.75:004.056.5

«До захисту допущено»
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис) (Ім’я ПРІЗВИЩЕ)
«__» _____2025 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Комп’ютерні системи та мережі»
зі спеціальності 123 «Комп’ютерна інженерія»
на тему «Відмовостійка система реального часу з адаптивною автентифікацією»

Виконав:
студент ІІ курсу, групи ІО-41мп
Ліненко Костянтин Миколайович _____
(підпис)

Керівник:
доцент, к.т.н.
Волокита Артем Миколайович _____
(підпис)

Консультант з нормоконтролю:
проф., д.т.н.
Кулаков Юрій Олексійович _____
(підпис)

Рецензент

_____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет *інформатики та обчислювальної техніки*
Кафедра *обчислювальної техніки*

Рівень вищої освіти – другий (магістерський)

Спеціальність *123 Комп'ютерна інженерія*

Освітньо-професійна програма *Комп'ютерні системи на мережі*

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис) (Ім'я ПРІЗВИЩЕ)
«__» _____2025 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Ліненку Костянтину Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації *Відмовостійка система реального часу з адаптивною автентифікацією*

керівник дисертації *доцент, к.т.н. Волокита Артем Миколайович*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «*06*» *листопада* 2025 р. № *4841-с*

2. Термін подання студентом дисертації *05.12.2025*

3. Об'єкт дослідження – *процеси проєктування, побудови та експлуатації відмовостійких розподілених веб-систем реального часу з механізмами автентифікації користувачів*

4. Предмет дослідження – *архітектурні патерни високої доступності, методи та алгоритми адаптивної автентифікації, моделі потокової обробки подій*

5. Перелік завдань, які потрібно розробити:
1) Провести огляд та аналіз сучасних архітектурних рішень
2) Дослідити та систематизувати сучасні підходи безпечної автентифікації
3) Реалізувати прототип програмного комплексу з можливістю обробки подій в реальному часу, оцінки ризику та автоматичного масштабування
4) Виконати тестування системи та сформулювати висновки щодо результатів роботи прототипу

6. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	проф., д.т.н. Кулаков Ю.О.		
2	проф., д.т.н. Кулаков Ю.О.		
3	проф., д.т.н. Кулаков Ю.О.		
4	проф., д.т.н. Кулаков Ю.О.		

7. Дата видачі завдання *1 вересня 2025 р.*

Календарний план

Назва етапів виконання дипломного проєкту	Строк виконання етапів проєкту	Примітка
Затвердження теми роботи	*04.09.2025*	
Аналіз предметної області, огляд літератури	*15.09.2025*	
Дослідження існуючих архітектур систем реального часу та методів забезпечення відмовостійкості	*25.09.2025*	
Аналіз методів автентифікації	*30.09.2025*	
Проєктування архітектури системи реального часу з адаптивною автентифікацією	*10.10.2025*	
Реалізація програмного прототипу системи	*05.11.2025*	
Тестування системи, аналіз результатів	*12.11.2025*	
Написання загальних висновків	*14.11.2025*	
Передзахист		
Захист	*24.12.2025*	

Студент _____ Костянтин ЛІНЕНКО
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник _____ Артем ВОЛОКИТА
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Структура та обсяг роботи. Робота складається із вступу та чотирьох розділів. Загальний обсяг роботи: 118 аркушів основного тексту, 59 ілюстрацій, 36 таблиць, 8 формул. При підготовці використовувалася література з 25 джерел.

Актуальність теми. Сьогодні більшість онлайн-сервісів працює в режимі реального часу, тож усі вони мають уміти швидко масштабуватися під час пікових навантажень і залишатися доступними навіть попри збої інфраструктури чи кібератак. Водночас зростають вимоги до безпеки автентифікації – дедалі частіше трапляються випадки компрометації паролів, бот-атак, входів із незвичних пристроїв або локацій. Усе це підштовхує до створення відмовостійких архітектур із адаптивною, ризик-орієнтованою автентифікацією, яка посилює перевірки лише тоді, коли це справді потрібно, не погіршуючи UX-досвід більшості легітимних користувачів.

Мета роботи – спроєктувати й реалізувати прототип відмовостійкої системи реального часу з адаптивною автентифікацією; розробити модель оцінки ризику доступу; побудувати архітектурні механізми високої доступності (HA, балансування навантаження, автоматичне відновлення) та експериментально підтвердити працездатність і ефективність запропонованого підходу.

Об’єкт дослідження – процеси проєктування, побудови та експлуатації відмовостійких розподілених веб-систем реального часу з механізмами автентифікації користувачів.

Предмет дослідження – архітектурні патерни високої доступності, методи та алгоритми адаптивної автентифікації, моделі потокової обробки подій.

Методи дослідження: системний і порівняльний аналіз, моделювання потоків подій і відмов, розробка евристичної моделі ризику, експериментальні навантажувальні, latency- та failover-тести; моніторинг метрик продуктивності та надійності.

Інноваційність одержаних результатів роботи полягає в наступному:

- Запропоновано цілісний підхід, який поєднує відмовостійку потокову архітектуру з адаптивною автентифікацією. У цьому рішенні risk-engine у реальному часі визначає, коли потрібно активувати step-up автентифікацію, спираючись на багатофакторну оцінку контексту.
- Розроблено вагову модель ризику з нормалізацією факторів і динамічними порогами рівнів LOW / MEDIUM / HIGH, що враховують індивідуальний профіль користувача.
- Запропоновано архітектурну схему “stateless-JWT + stateful risk-context”: сесійна токенізація залишається безпечною та масштабованою, тоді як контекст ризику обробляється потоково й кешується для швидкого виконання step-up-перевірки.
- Представлено комбінацію патернів високої доступності для навантажень у реальному часі: балансування за кількістю з’єднань, ідемпотентні споживачі подій, автоматичний failover і self-healing.

Особистий внесок магістранта. Автор самостійно спроєктував архітектуру системи та реалізував ключові мікросервіси, серед яких – автентифікація з підтримкою JWT, TOTP і OAuth2, сервіс оцінки ризику та потокова взаємодія через брокер подій. Було розроблено модель і алгоритм обчислення ризику з урахуванням вагових коефіцієнтів і політик step-up автентифікації, інтегровано кешування та сховище секретів для підвищення безпеки. Налаштовано балансування трафіку та механізми автоматичного відновлення, створено дашборди моніторингу, проведено навантажувальні та відмовостійкі експерименти, що підтвердили ефективність підходу. Також були підготовлені UX-кейси для фронтенду, які демонструють сценарії роботи користувача в умовах адаптивної автентифікації та змін рівня ризику в реальному часі.

Практична цінність. Запропоноване рішення виступає готовим референс-шаблоном для впровадження у фінтех-, e-commerce-, e-gov- та телемедичних системах, де критично важливими є низька затримка, висока доступність і гнучка безпека. Архітектура дає змогу зменшити кількість зайвих MFA-перевірок для

легітимних користувачів, одночасно підсилюючи контроль для підозрілих сесій. Завдяки механізмам високої доступності система скорочує простої, а прозорий моніторинг спрощує операційне адміністрування. Розроблений прототип може бути безпосередньо використаний для подальших промислових упроваджень і масштабування, забезпечуючи надійний фундамент для реальних застосувань у критично важливих галузях.

КЛЮЧОВІ СЛОВА: ВІДМОВОСТІЙКА СИСТЕМА, РЕАЛЬНИЙ ЧАС, АДАПТИВНА АВТЕНТИФІКАЦІЯ, ОЦІНКА РИЗИКУ, МІКРОСЕРВІСНА АРХІТЕКТУРА, БРОКЕР ПОВІДОМЛЕНЬ, ОРКЕСТРУВАННЯ, БАЛАНСУВАННЯ НАВАНТАЖЕННЯ.

ABSTRACT

Structure and scope of the work. The thesis consists of an introduction and four chapters. Total volume: 118 pages of main text, 59 illustrations, 36 tables, 8 formulas. During preparation, literature from 25 sources was used.

Relevance of the topic. Today, most online services operate in real-time mode, which requires them to scale quickly during peak loads and remain available despite infrastructure failures or cyberattacks. At the same time, authentication security requirements are increasing – cases of password compromise, bot attacks, and logins from unusual devices or locations are becoming more frequent.
All of this drives the development of fault-tolerant architectures with adaptive, risk-based authentication, which strengthens security checks only, when necessary, without worsening the user experience for legitimate users.

Purpose of the work – to design and implement a prototype of a fault-tolerant real-time system with adaptive authentication; to develop a model for access risk assessment; to build architectural mechanisms of high availability (HA) – including load balancing and automatic recovery – and to experimentally validate the performance and efficiency of the proposed approach.

Object of research – processes of designing, building, and operating fault-tolerant distributed real-time web systems with user authentication mechanisms.

Subject of research – architectural high-availability patterns**,** methods and algorithms of adaptive (risk-based) authentication**,** and models of event stream processing**.**

Research methods: system and comparative analysis, modeling of event and failure flows, development of a heuristic risk model, experimental load, latency, and failover testing; monitoring of performance and reliability metrics.

Scientific novelty:

- A comprehensive approach is proposed that integrates a fault-tolerant streaming architecture with adaptive authentication. In this model, the real-time risk

engine determines when to activate step-up authentication based on a multifactor contextual evaluation.

- A weighted risk model is developed, including normalization of factors and dynamic LOW / MEDIUM / HIGH thresholds that adapt to the individual user profile.
- An architectural scheme “stateless-JWT + stateful risk-context” is introduced: session tokenization remains secure and scalable, while the risk context is processed in a streaming manner and cached for fast step-up verification.
- A combination of high-availability (HA) patterns is presented for real-time loads: connection-based load balancing, idempotent event consumers, automatic failover, and self-healing mechanisms.

Author’s contribution. The author independently designed the system architecture and implemented key microservices, including authentication with JWT, TOTP, and OAuth2, a risk assessment service, and stream interaction through an event broker. A risk calculation model and algorithm were developed with weighted coefficients and step-up policies, caching and secret storage were integrated to enhance security. Traffic balancing and auto-recovery mechanisms were configured; monitoring dashboards were built; load and fault-tolerance experiments were conducted to confirm the approach’s efficiency. Additionally, system documentation and frontend UX cases were prepared, demonstrating user scenarios under adaptive authentication and real-time risk level changes.

Practical significance. The proposed solution provides a reference architecture suitable for fintech, e-commerce, e-government, and telemedicine, where low latency, high availability, and adaptive security are essential. It reduces unnecessary MFA for legitimate users while strengthening verification for suspicious sessions. High-availability mechanisms minimize downtime, and integrated monitoring simplifies maintenance. The prototype and testing approach can be directly applied in industrial deployments, offering a reliable foundation for mission-critical systems.

KEYWORDS: FAILURE-TOLERANT SYSTEM, REAL-TIME, ADAPTIVE AUTHENTICATION, RISK ASSESSMENT, MICROSERVICE ARCHITECTURE, MESSAGE BROKER, ORCHESTRATATION, LOAD BALANCING.

ЗМІСТ

ВСТУП

Актуальність проблеми швидкості обробки та достовірності інформації стає все більш критичною для прийняття рішень у бізнесі, промисловості, транспорті, охороні здоров'я та державному секторі – багатьох сфер, які швидко розвиваються у сучасному цифровому світі. Системи реального часу, що забезпечують миттєву реакцію на зовнішні події, дедалі частіше виконують роль ядра інфраструктури, від надійності якої залежить безперервність бізнес-процесів, безпека користувацьких даних та ефективність менеджменту. Збої в роботі таких систем, навіть короткочасні, можуть призвести до значних фінансових втрат, репутаційних ризиків, або навіть загрози життю людей.

Поєднання вимог до відмовостійкості та адаптивної автентифікації створює низку інженерних та наукових викликів. Необхідно розробити таку архітектуру системи, що здатна забезпечувати стабільну роботу при часткових збоях та пікових навантаженнях, також масштабуватися з практично нульовим ризиком зниження продуктивності, а також додатково інтегрувати адаптивні механізми автентифікації без шкоди для часу відгуку.

Сучасні рішення, доступні на ринку, часто або зосереджені виключно на питаннях масштабування й продуктивності, або фокусуються на безпеці, не враховуючи обмеження систем реального часу. Це створює незадоволену потребу в комплексних підходах, які одночасно поєднують обидва аспекти.

Метою магістерської роботи є створення відмовостійкої архітектури системи реального часу з вбудованими адаптивними механізмами автентифікації, яка відповідатиме сучасним вимогам безпеки та здатна працювати в умовах високої динаміки навантажень.

У межах дослідження буде здійснено:

- аналіз сучасних архітектур і методів автентифікації;
- проєктування нової архітектурної моделі;
- розроблення прототипу системи з реальними сценаріями використання;

- проведення навантажувальних та безпекових тестів;
- оцінка можливостей комерціалізації результатів у вигляді стартап-проєкту.

Практична цінність роботи полягає у розробленні архітектурних та інженерних рішень, які можуть бути безпосередньо впроваджені у високонавантажених критичних системах, де важлива безперервність функціонування та гнучкий контроль доступу користувачів.

Запропонований підхід може бути застосований:

- у сервісах оперативного моніторингу та реагування (системи «розумного мосту», транспорт, енергетика тощо);
- у фінансово-банківській сфері для захисту онлайн-транзакцій у реальному часі;
- у платформах з мільйонами одночасних користувачів, де критично важлива низька затримка та стійкість до збоїв;
- у державних/військових інформаційних системах з підвищеними вимогами до безпеки.

РОЗДІЛ 1
АНАЛІЗ СТАНУ ПРОБЛЕМИ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Актуальність теми дослідження

Поняття систем реального часу сформувалося у 1950–1960-х роках у зв'язку з розвитком перших обчислювальних комплексів для військових, аерокосмічних і промислових потреб. Одними з перших прикладів таких систем стали проєкти раннього попередження про ракетні атаки (SAGE, США, 1958 р.) та системи управління польотами космічних апаратів NASA у програмі Apollo (1960–1970-ті роки) [1]. Ці системи вимагали гарантованої реакції у реальному часі, високої точності часової синхронізації та безперебійності роботи, оскільки будь-яка затримка або відмова могла призвести до катастрофічних наслідків.

У подальші десятиліття системи реального часу вийшли за межі спеціалізованих галузей і стали невід'ємною частиною автоматизованих виробництв, транспортної інфраструктури, медичних приладів, фінансових сервісів та телекомунікацій. Якщо раніше такі системи функціонували у закритих середовищах з обмеженою кількістю користувачів та чітко контрольованими умовами експлуатації, то сьогодні вони працюють у глобальних розподілених мережах, де обсяг запитів може сягати мільйонів на секунду, а вплив людського фактору та зовнішніх загроз зростає.

1.1.1. Зростання вимог до відмовостійкості

З кожним роком очікування користувачів і бізнесу щодо доступності сервісів стають жорсткішими. Якщо у 1990-х допустимі простої вимірювалися годинами, то сьогодні провідні компанії орієнтуються на SLA 99.99 % і вище, що еквівалентно максимум 52 хвилинам простою на рік [2]. У високочутливих сферах (наприклад, біржова торгівля, медичні системи моніторингу, системи управління дорожнім рухом) допустимі затримки вимірюються мілісекундами, а будь-який збій може призвести до реальних фінансових або людських втрат.

Крім того, розвиток мікросервісних архітектур, контейнеризації та мультихмарних середовищ, з одного боку, відкрив можливості масштабування та гнучкості, а з іншого – створив нові виклики. Кожен мікросервіс є потенційною точкою відмови, а загальна складність системи зростає експоненційно. Збої можуть виникати через мережеві затримки, втрату повідомлень, розсинхронізацію станів або некоректну взаємодію між сервісами.

Додатковим фактором є кіберзагрози, які часто спрямовані саме на порушення безперервності роботи сервісів (наприклад, DDoS-атаки). У поєднанні з технічними збоями це створює складне середовище, у якому забезпечення відмовостійкості стає не просто бажаною властивістю, а необхідною умовою існування системи.

1.1.2. Важливість безпечної автентифікації в умовах розподілених архітектур

Автентифікація – процес перевірки автентичності користувача або системи – пройшла шлях від простих паролів до складних багатофакторних та поведінкових методів. На початкових етапах становлення мережевих технологій (1980–1990-ті роки) автентифікація виконувалася в централізованих середовищах, де всі користувачі взаємодіяли з єдиним сервером. Основним способом контролю доступу були паролі, що зберігалися на одному вузлі, а мережеві взаємодії відбувалися у межах контрольованої інфраструктури [3].

Сучасні системи суттєво відрізняються від таких моделей. Вони базуються на розподілених архітектурах, що включають безліч сервісів, розміщених у різних дата-центрах, регіонах або навіть хмарних провайдерах. Ці сервіси взаємодіють між собою через публічні або напівпублічні мережі, використовуючи API, брокери повідомлень, шини даних та інші комунікаційні механізми. В таких умовах безпечна автентифікація набуває критичного значення, оскільки кожен сервіс та кожна точка взаємодії стають потенційною мішенню для атак.

Основні виклики автентифікації в розподілених системах пов’язані насамперед із відсутністю єдиного контрольного центру. У централізованих архітектурах контроль доступу відбувається на одному сервері, тоді як у

розподілених системах автентифікація має бути узгодженою між багатьма вузлами. Це ускладнює управління обліковими даними, токенами доступу та політиками безпеки.

Ще однією проблемою є підвищена вразливість каналів передачі даних. Оскільки взаємодія між сервісами відбувається через мережу, зловмисники можуть використовувати атаки «man-in-the-middle», перехоплювати токени або підробляти запити. Недостатній захист автентифікаційних даних може відкривати доступ до внутрішніх компонентів системи.

Також виникають проблеми масштабування та продуктивності. Класичні методи автентифікації, які добре працюють у монолітних додатках, часто не витримують навантаження в розподіленому середовищі з мільйонами одночасних з'єднань. Затримки при верифікації користувачів можуть негативно впливати на загальний час відгуку системи.

Не менш важливою є необхідність безперервної синхронізації автентифікаційних даних. У середовищі з кількома сервісами токени, ключі та сесії повинні бути синхронізовані в реальному часі між різними вузлами для забезпечення узгодженості доступу. Будь-яка розсинхронізація може призвести до помилок доступу або створити вразливості безпеки.

Окремо слід відзначити зростання ролі машинного доступу. У сучасних мікросервісних системах значну частину взаємодій здійснюють не користувачі, а сервіси між собою. Це потребує використання безпечних протоколів автентифікації сервісів, таких як OAuth 2.0, OpenID Connect, mutual TLS або спеціалізовані системи управління секретами.

1.1.3. Проблематика класичних методів резервування та статичних механізмів автентифікації

Традиційні підходи до забезпечення відмовостійкості та безпеки у системах реального часу формувалися в епоху, коли більшість інформаційних систем мали централізовану архітектуру, обмежену кількість користувачів та передбачувані сценарії навантаження. З розвитком розподілених, мікросервісних та

мультихмарних архітектур, ці методи дедалі частіше виявляються недостатніми, що створює реальні технічні та безпекові ризики.

1.1.3.1. Класичні методи резервування: обмеження у масштабованих середовищах

Історично для підвищення надійності використовувались два основні підходи [4]:

- Активне/пасивне резервування (hot-standby). Один вузол виконує основну роботу, інший перебуває у режимі очікування та бере на себе навантаження у разі збою.
- Активне/активне резервування (load balancing). Декілька вузлів працюють одночасно, розподіляючи запити, що підвищує продуктивність і дозволяє компенсувати відмову одного з них.

Хоча класичні підходи до резервування ефективні в обмежених середовищах, у сучасних динамічних системах вони стикаються з низкою проблем. По-перше, обмежена гнучкість і статичні конфігурації. Резервування зазвичай вимагає попередньої ручної настройки вузлів, IP-адрес та маршрутів, і в розподілених архітектурах з десятками або сотнями мікросервісів така модель стає складною в адмініструванні та масштабуванні.

По-друге, нерівномірний розподіл навантаження. У схемах «гарячого резерву» пасивні вузли часто простоюють, витрачаючи ресурси неефективно. Під час аварійного перемикання можуть виникати затримки та втрата контексту сесій, що знижує загальну продуктивність системи.

По-третє, відсутність автоматичної адаптації. Класичні механізми резервування не враховують динаміку мережевих умов, зміни навантаження або появу нових вузлів і не пристосовані до автоматичного масштабування, яке є типовим для контейнеризованих та хмарних середовищ.

Нарешті, складність синхронізації стану. Для систем реального часу важливо не лише відновити працездатність, а й зберегти контекст виконання. У класичних схемах резервування часто відсутні ефективні механізми синхронізації

оперативних даних, що може призводити до втрати повідомлень або непослідовності даних при перемиканні.

У підсумку, ці методи працюють у відносно статичних сценаріях, але погано масштабуються в умовах мультихмарних, географічно розподілених та високодинамічних систем, де відмовостійкість має бути гнучкою та автоматизованою.

1.1.3.2. Статичні механізми автентифікації: вразливість у динамічних середовищах

Класичні механізми автентифікації – паролі, базові токени, статичні API-ключі – десятиліттями залишалися стандартом для контролю доступу.

Проте в умовах розподілених архітектур класичні підходи до автентифікації мають низку критичних недоліків. Використання фіксованих облікових даних, таких як статичні токени або ключі без регулярної ротації, створює серйозну вразливість: у разі витоку зловмисник отримує необмежений доступ до сервісу або внутрішньої мережі. Крім того, відсутність контекстної перевірки у статичних методах не враховує поведінкові та ситуаційні фактори, такі як місцезнаходження, пристрій або аномальна активність, що спрощує виконання атак типу credential stuffing, session hijacking або man-in-the-middle. Складність централізованого керування в великих системах із десятками сервісів призводить до дублювання ключів, неузгоджених політик безпеки та проблем контролю доступу. Також такі механізми погано інтегруються з масштабуванням у хмарних середовищах, де сервіси динамічно створюються та видаляються, і статичні методи не встигають адаптуватися до змін, створюючи «дірки» у системі безпеки. Нарешті, класичні підходи не здатні ефективно протидіяти сучасним атакам: більшість зломів великих компаній пов’язані не зі складними експлойтами, а з крадіжкою або підбором облікових даних, оскільки статичні токени часто зберігаються у коді, CI/CD системах або відкритих репозиторіях, стаючи легкою мішенню.

1.1.3.3. Синергетична проблема для систем реального часу

У системах реального часу ці недоліки посилюються додатковими вимогами:

- затримки автентифікації безпосередньо впливають на продуктивність та SLA;
- ручне резервування не здатне забезпечити миттєве перемикання без втрати даних;
- компрометація одного токена у розподіленому середовищі може поставити під загрозу всю систему;
- статичність конфігурацій суперечить динамічній природі контейнеризованих і мікросервісних архітектур.

У результаті класичні методи, які ще донедавна вважалися надійними, сьогодні часто стають причиною серйозних інцидентів. Наприклад, у 2022 році злам одного з API-ключів у CI/CD середовищі GitHub дозволив зловмисникам проникнути у приватні репозиторії кількох компаній, що спричинило витоки даних і фінансові збитки. Аналогічно, у фінансовому секторі статичні токени часто використовуються для інтеграцій між банківськими сервісами, що створює серйозні ризики при їх витоку [5].

1.2. Огляд існуючих архітектур систем реального часу

1.2.1. Централізовані архітектури

У централізованих архітектурах усі компоненти системи – обробка запитів, бізнес-логіка, доступ до бази даних, керування чергами, безпека – виконуються в одному процесі або в межах єдиної програми. Зазвичай застосунок спирається на одну спільну базу даних, з якою взаємодіють усі модулі.

Для систем реального часу така архітектура означає, що усі події та запити надходять до центрального вузла, який відповідає за обробку в реальному часі, а також що затримки мінімізуються за рахунок відсутності міжсервісної комунікації – усі виклики відбуваються всередині процесу

У таблицях 1.1-1.2 наведені переваги та недоліки централізованої архітектури.

Таблиця 1.1

Переваги централізованої архітектури

Перевага	Опис
Низька затримка внутрішніх викликів	Усі операції виконуються локально, без потреби в мережевих запитах між сервісами, що зменшує latency
Проста модель даних	Єдина база даних спрощує транзакційність, узгодженість станів та відсутність потреби у складних механізмах синхронізації
Спрощене управління безпекою та автентифікацією	Контроль доступу здійснюється у межах однієї програми, що мінімізує кількість потенційних точок атак
Легша розробка на початкових етапах	Команді розробників не потрібно проєктувати міжсервісні протоколи, шини подій чи окремі сервіси – достатньо працювати з одним застосунком
Простота розгортання та адміністрування у малих масштабах	Розгортання полягає у встановленні та запуску однієї програми або контейнера

Таблиця 1.2

Недоліки централізованої архітектури

Недолік	Опис
Єдина точка відмови	Збій центрального вузла призводить до повної недоступності всієї системи. Навіть резервування серверів не гарантує безперервності роботи у реальному часі через затримки перемикання та втрату сесій
Складність масштабування	Горизонтальне масштабування монолітів є нетривіальним: копіювання всього застосунку на кілька вузлів часто призводить до проблем синхронізації станів і колізій у базі даних
Обмежена гнучкість при змінах	Будь-яка зміна в одній частині застосунку потребує повторного розгортання всього моноліту. Це ускладнює безперервну інтеграцію та оновлення у реальному часі
Погана стійкість до локальних збоїв	Помилка в одному модулі або витік пам’яті можуть зупинити роботу всієї системи, що особливо критично для реальних виробничих процесів
Складнощі при інтеграції з зовнішніми системами	Монолітна структура не передбачає гнучкої міжсистемної взаємодії через API або шини повідомлень, що обмежує розширюваність

Низькі затримки та простота роблять їх привабливими для невеликих проєктів або систем з обмеженим навантаженням. Проте в умовах сучасних вимог до відмовостійкості, масштабованості та географічної розподіленості, монолітні підходи дедалі частіше стають вузьким місцем і потребують еволюції до розподілених архітектур.

1.2.2. Розподілені та мікросервісні архітектури

Розподілена архітектура передбачає, що система складається з кількох вузлів (серверів або процесів), які спільно виконують обчислення та обмінюються даними через мережу. Кожен вузол може мати власні завдання: обробку запитів, зберігання даних, кешування, черги повідомлень тощо. Мікросервісна архітектура є еволюцією розподіленої моделі. Її суть полягає у розбитті системи на невеликі, незалежні сервіси, кожен з яких виконує окрему функцію та може розроблятися, тестуватися та масштабуватися незалежно [6]. У таблицях 1.3-1.4 наведені переваги та недоліки мікросервісної архітектури.

Таблиця 1.3

Переваги мікросервісної архітектури

Перевага	Опис
Масштабованість на рівні компонентів	Сервіси, які відчувають найбільше навантаження можна масштабувати незалежно від решти системи
Підвищена відмовостійкість	Вихід з ладу одного мікросервісу не обов’язково зупиняє всю систему
Гнучкість розробки	Різні команди можуть працювати над різними сервісами паралельно, використовуючи різні технології. Це прискорює інновації та оновлення систем

Таблиця 1.3 (продовження)

Переваги мікросервісної архітектури

Готовність до географічного розподілу та edge-обчислень	Мікросервіси можуть бути розгорнуті у різних регіонах, ближче до кінцевих користувачів, що особливо важливо для зниження latency у системах реального часу

Таблиця 1.4

Недоліки мікросервісної архітектури

Недолік	Опис
Зростання мережевих взаємодій	Внутрішні виклики між сервісами відбуваються через мережу, що додає затримки й потребує оптимізації маршрутизації
Складність забезпечення транзакційності та узгодженості даних	На відміну від моноліту з єдиною БД, мікросервіси потребують складних патернів (Saga, Event Sourcing, eventual consistency), щоб гарантувати коректність операцій
Безпека	Кожен мікросервіс стає потенційною точкою атаки. Необхідно впроваджувати надійні механізми автентифікації
Складність спостереження та діагностики	Для відстеження стану системи потрібні централізовані системи моніторингу, логування, трасування

Розподілені та мікросервісні архітектури стали логічним розвитком монолітних систем, відповідаючи на виклики масштабування, гнучкості та географічної розподіленості. Для систем реального часу вони відкривають нові можливості – високу відмовостійкість, гнучке масштабування та низькі затримки, якщо обрати правильну тактику оптимального розподілу компонентів. Водночас вони вимагають застосування спеціалізованих архітектурних рішень, щоб подолати виклики узгодженості, мережевих затримок та безпеки.

1.2.3. Використання черг повідомлень та брокерів

Перші реалізації черг повідомлень з'явилися у великих корпоративних системах на основі middleware-рішень, як-от IBM MQ Series та Microsoft MSMQ. Вони дозволяли асинхронно обмінюватися повідомленнями між додатками, не вимагаючи їх одночасної доступності.

Згодом ці концепції еволюціонували у брокери повідомлень та подій – спеціалізовані системи, що забезпечують:

- гарантовану доставку повідомлень між сервісами;
- буферизацію та черги;
- маршрутизацію за темами або ключами;
- масштабування обробки потоків у реальному часі.

Такі системи стали невід'ємною частиною сучасних архітектур реального часу, особливо мікросервісних, де компоненти повинні обмінюватися даними швидко, надійно й без жорсткої синхронізації.

У таблицях 1.5-1.6 наведені переваги та недоліки брокерів повідомлень.

Таблиця 1.5

Переваги брокерів повідомлень

Перевага	Опис
Асинхронна взаємодія між компонентами	Сервіси не залежать від одночасної доступності один одного
Буферизація пікових навантажень	Черги згладжують навантаження на споживачів, що особливо важливо у реальному часі
Підвищення відмовостійкості	У разі збою споживача повідомлення зберігаються в черзі до його відновлення
Зменшення зв’язності між сервісами	Компоненти можуть змінюватися, оновлюватися або масштабуватися незалежно
Гнучкі моделі комунікації	Підтримка як point-to-point, так і publish–subscribe схем, що дозволяє будувати гібридні сценарії обробки подій

Таблиця 1.6

Недоліки брокерів повідомлень

Недолік	Опис
Додаткова інфраструктура	Потребує налаштування та адміністрування брокерів, що ускладнює систему
Збільшення мережевих затримок	На відміну від внутрішніх викликів у моноліті, з’являються додаткові hops через брокер

Таблиця 1.6 (продовження)

Недоліки брокерів повідомлень

Складність відлагодження та трасування	Асинхронна природа ускладнює пошук помилок і причинно-наслідкових зв’язків
Необхідність гарантувати узгодженість	Можуть виникати дублікати або зміна порядку повідомлень, що потребує спеціальної обробки
Підвищені вимоги до безпеки	Брокери стають критичними точками атаки, що вимагає ретельного контролю доступу, шифрування та моніторингу
Можливість накопичення “чергового боргу”	Якщо споживачі не встигають обробляти повідомлення, черга може рости неконтрольовано

Брокери повідомлень забезпечують надійний обмін даними між сервісами, ізолюючи відправника від отримувача та дозволяючи масштабувати систему без жорстких зв’язків між компонентами. Їхні переваги – асинхронність, стійкість до пікових навантажень, повторна доставка, гарантії порядку та зручна інтеграція різних технологій. Водночас брокери ускладнюють архітектуру, потребують додаткових ресурсів та адміністрування, можуть стати “єдиною точкою” відмови без кластеризації, а затримки у чергах роблять поведінку системи менш передбачуваною порівняно з прямими синхронними викликами.

1.2.3.1. Apache Kafka

Apache Kafka – це розподілена платформа потокової обробки даних, створена у LinkedIn у 2010 році та пізніше передана до Apache Foundation. Сьогодні Kafka є де-факто стандартом для високонавантажених систем реального часу. Вона базується на публікаційно-підписній моделі (publish-subscribe) з використанням topics, гарантує доставку повідомлень із можливістю повторного зчитування, підтримує горизонтальне масштабування через партиції та кластери брокерів,

зберігає повідомлення на диску протягом певного часу, що дозволяє обробляти події з відставанням або повторно, а також забезпечує низьку затримку та здатність обробляти мільйони повідомлень на секунду [7].

Kafka ефективно застосовується для обробки потоків даних у реальному часі, побудови шини подій між мікросервісами, логування, моніторингу та аналітики в реальному часі, а також для створення відмовостійких систем із реплікацією даних.

1.2.3.2. RabbitMQ

RabbitMQ – це брокер повідомлень, який реалізує протокол AMQP (Advanced Message Queuing Protocol) і більше підходить для традиційних черг завдань та складної маршрутизації. Він підтримує черги з підтвердженнями доставки, маршрутизацію через exchange та binding (direct, topic, fanout, headers), забезпечує гнучку інтеграцію з різними мовами та фреймворками, має меншу затримку у сценаріях point-to-point комунікації та просте розгортання у порівнянні з Kafka, хоча його горизонтальна масштабованість обмежена [8].

RabbitMQ часто застосовується для обробки фонового або відкладеного виконання завдань, зв’язку між мікросервісами у транзакційних сценаріях, маршрутизації повідомлень за складними правилами та організації черг у системах середнього навантаження.

1.2.4. Підсумок

У межах огляду існуючих архітектур систем реального часу було розглянуто різні підходи – від централізованих монолітів до сучасних мікросервісних рішень із використанням брокерів повідомлень та технологій високої доступності. Аналіз показав, що класичні моноліти мають перевагу у простоті розгортання та низьких затримках внутрішніх викликів, однак їхні обмеження щодо масштабованості, відмовостійкості та гнучкості роблять їх менш придатними для побудови складних, навантажених і географічно розподілених систем, які мають працювати безперервно у реальному часі.

Зважаючи на вимоги до надійності, продуктивності та масштабованості, найбільш доцільним для реалізації проєкту є перехід до мікросервісної архітектур.

Такий підхід дає змогу розділити систему на незалежні компоненти з чіткими зонами відповідальності, масштабувати окремі сервіси залежно від навантаження та підвищити стійкість до відмов завдяки ізоляції збоїв.

Ключову роль у такій архітектурі відіграють брокери повідомлень, зокрема Kafka або RabbitMQ, які забезпечують асинхронну взаємодію між сервісами, буферизацію пікових навантажень та підвищену відмовостійкість. Їх використання особливо важливе у контексті систем реального часу, де критичною характеристикою є не максимальна швидкість обробки подій, а гарантована та передбачувана реакція системи у визначений момент часу. Завдяки механізмам гарантованої доставки, впорядкування повідомлень та стійкості до збоїв брокери дозволяють забезпечити детерміновану обробку подій, яка є фундаментальною вимогою для таких систем.

Не менш важливим елементом архітектури є впровадження рішень з високої доступності, кластеризації та балансування навантаження. Вони дозволяють уникнути єдиних точок відмови, забезпечують автоматичне відновлення після збоїв та гарантують стабільну продуктивність у динамічних умовах. Кластеризація критичних компонентів у поєднанні з балансуванням запитів між сервісами створює технічне підґрунтя для надійної роботи системи у реальному часі, а також робить можливим поступове масштабування без зупинки сервісів.

Таким чином, для практичної реалізації проєкту оптимальним є поєднання мікросервісної архітектури, подієвої комунікації через брокери повідомлень та рішень з високої доступності. Такий підхід забезпечує гнучкість, стійкість і масштабованість системи, відповідає сучасним архітектурним практикам і водночас дає можливість природно перейти від традиційної монолітної моделі розробки до повноцінної розподіленої екосистеми, що є логічним і стратегічно правильним кроком для подальшого професійного розвитку.

1.3. Огляд існуючих підходів забезпечення відмовостійкості

1.3.1. Кластеризація

Кластеризація – це об’єднання декількох вузлів у єдину систему з метою підвищення доступності та продуктивності. Якщо один вузол виходить з ладу, його функції автоматично переходять до іншого без помітного для користувача простою. Основні типи кластерів [9]:

- Активно–пасивні кластери (Active–Passive). Один вузол виконує всі запити, а інший перебуває у резерві. У разі збою відбувається перемикання на пасивний вузол (failover). Цей підхід простий, але передбачає тимчасову затримку під час перемикання та неефективне використання резервних ресурсів.
- Активно–активні кластери (Active–Active). Усі вузли одночасно обробляють запити, а навантаження розподіляється між ними. Якщо один вузол виходить з ладу, решта продовжують роботу без зупинок. Цей підхід забезпечує вищу продуктивність і гнучкість, але потребує складнішої синхронізації станів і даних.

Переваги кластеризації включають автоматичне відновлення після збоїв (failover/failback), можливість масштабування без зупинки сервісів, зменшення ризику єдиної точки відмови та потенційне географічне розподілення вузлів для підвищення відмовостійкості.

Разом із тим, кластеризація має свої проблеми та виклики: це необхідність синхронізації стану між вузлами у реальному часі, складність налаштування та адміністрування кластерів, а також можливі затримки під час переключення на резервний вузол у пасивних конфігураціях.

1.3.2. Балансування навантаження

Балансування навантаження – це метод рівномірного розподілу вхідних запитів між кількома серверами, який дозволяє уникнути перевантаження окремих вузлів, забезпечити стабільну продуктивність та підвищити загальну відмовостійкість системи [10]. Існують різні типи балансування: на DNS-рівні, коли трафік розподіляється через декілька IP-адрес за допомогою round-robin; на

мережевому рівні (L4), коли балансування здійснюється на основі IP та портів і забезпечує низькі накладні витрати та високу продуктивність; та на прикладному рівні (L7), що дозволяє маршрутизувати HTTP/HTTPS-запити на основі контенту та забезпечує гнучку маршрутизацію [11].

Серед алгоритмів балансування популярні Round Robin, який рівномірно розподіляє запити по черзі; Least Connections, що спрямовує запит на вузол із найменшою кількістю активних з'єднань; IP Hash або Sticky Sessions, які закріплюють користувача за певним вузлом для збереження сесії; а також Weighted, що враховує потужність вузлів при розподілі навантаження. Найпоширенішими технологіями для реалізації балансування є HAProxy – високопродуктивний L4/L7 балансувальник з можливістю health-check і failover, та NGINX – веб-сервер і L7 балансувальник з великою гнучкістю.

1.4. Огляд підходів до автентифікації в системах реального часу

У системах реального часу автентифікація має забезпечувати не лише коректність особи чи сервісу, а й робити це з мінімальною затримкою та з високою стійкістю до збоїв. Важливо відокремлювати автентифікацію користувача (user-to-service) та взаємну автентифікацію сервісів (service-to-service). Ще одна практична вісь – stateful vs stateless підходи, адже керування сесіями впливає на горизонтальне масштабування та час відгуку.

1.4.1. Класичні методи

Класичні методи автентифікації охоплюють два основні підходи: парольна перевірка облікових даних та використання токенів доступу. Вони залишаються базовим шаром автентифікації в більшості систем, у тому числі з реальним часом, але відрізняються архітектурними вимогами та поведінкою при масштабуванні.

1.4.1.1. Парольна автентифікація

У типовому сценарії користувач надсилає логін і пароль, які сервер перевіряє шляхом порівняння з хешованим значенням у базі даних. Хешування виконується

з використанням алгоритмів, стійких до перебору. Після успішної перевірки система або видає токен, або створює сесію. Основна проблема цього підходу для систем реального часу – синхронна залежність від центральної бази під час входу. Якщо база недоступна або перевантажена, автентифікація затримується або блокується. Також зберігання паролів потребує надійного захисту та політик складності, що ускладнює керування на масштабі.

1.4.1.2. Сесійні токени

У stateful-схемі після автентифікації сервер створює сесію та зберігає її в оперативній пам’яті або в зовнішньому сховищі. Клієнт отримує session ID та надсилає його з кожним запитом. Сервер за session ID відновлює контекст користувача.

У реальному часі цей підхід має суттєві обмеження:

- кожен запит вимагає доступу до централізованого сесійного сховища, що створює додаткову латентність і точку відмови;
- горизонтальне масштабування ускладнюється – потрібні sticky sessions або розподілене кешування;
- при падінні вузла або Redis сесії можуть бути втрачені, що безпосередньо впливає на SLA.

Тому stateful-сесії малопридатні для високоінтенсивних потокових сценаріїв, де важлива мінімальна затримка при перевірці доступу.

1.4.1.3. JWT

На відміну від stateful-сесій, токени є самодостатніми: вони містять усю необхідну інформацію про користувача і криптографічно підписані, що дозволяє будь-якому серверу валідувати їх без доступу до спільного сховища.

У типовій схемі користувач один раз автентифікується, отримує токен доступу, а далі при кожному запиті надсилає його у заголовку. Сервер перевіряє підпис і термін дії токена локально, що значно зменшує затримки й дозволяє легко масштабувати систему горизонтально [12]. Це особливо важливо для систем

реального часу, де потрібна швидка обробка тисяч одночасних запитів або підтримка довготривалих WebSocket-з’єднань.

1.4.2. Двохфакторна автентифікація

Двохфакторна автентифікація (далі 2FA) додає до базової схеми «логін + пароль» другий фактор, який належить до іншої категорії:

- щось, що користувач знає (пароль, PIN);
- щось, що має (фізичний токен, смартфон, OTP-додаток);
- щось, чим є (біометрія).

У сучасних веб-системах найчастіше застосовують комбінацію першого та другого варіантів – пароль + одноразовий код (TOTP, SMS, push). У контексті архітектур реального часу 2FA має бути впроваджена так, щоб мінімізувати вплив на затримки, забезпечити масштабованість і не створювати єдиних точок відмови.

1.4.2.1. TOTP (Time-based One-Time Password)

Користувач і сервер мають спільний секрет, з якого обидві сторони обчислюють одноразовий код на основі поточного часу. Користувач вводить цей код під час другого етапу автентифікації.

Технічні особливості для реалізації 2FA на базі TOTP у реальному часі [13]:

- обчислення кодів не потребує зовнішніх API – все виконується локально на сервері;
- код дійсний короткий час (зазвичай 30 с), тому перевірка повинна бути синхронною й точно узгодженою з часовими інтервалами;
- для запобігання відмовам потрібно використовувати стійке до збоїв сховище секретів;
- у розподіленій системі кожен вузол повинен мати доступ до однакових секретів, тому секрети синхронізують або кешують з централізованого сховища.

TOTP добре масштабується, оскільки не вимагає зовнішніх звернень (на відміну від SMS), але критично залежить від надійності сховища секретів та точності часу.

1.4.2.2. SMS та Email коди

Це простий у впровадженні метод, але має високу латентність і залежить від сторонніх сервісів (операторів або поштових систем). У реальному часі це може стати bottleneck – перевірка 2FA затримує повну автентифікацію користувача на десятки секунд. До того ж SMS-канали вразливі до перехоплень, а email – до компрометації поштових скриньок.

У розподілених архітектурах SMS/Email-відправлення часто винесені у окремий сервіс-чергу, щоб не блокувати основний потік автентифікації. Замість очікування відповіді система надсилає запит у чергу, а користувач вводить код у другому кроці.

1.4.2.3. Push-повідомлення та App-based 2FA

Push-нотифікації через мобільні додатки дозволяють підтвердити вхід у один клік. Технічно цей метод вимагає зовнішнього push-сервера та механізму асинхронного очікування відповіді. У системах реального часу push-підтвердження інтегрують через:

- асинхронні черги подій (користувачу надсилається push, а сервер чекає підтвердження паралельно з іншими операціями);
- короткочасні WebSocket-сесії між мобільним додатком і бекендом для миттєвого зворотного зв’язку.

Push-підтвердження дає найкраще співвідношення безпеки й UX, але потребує стабільного push-каналу, керування токенами пристроїв та захисту мобільних клієнтів.

1.4.3. Багатофакторна автентифікація

Це розширення концепції 2FA, яке передбачає використання двох і більше факторів одночасно або динамічне комбінування факторів для досягнення вищого рівня гарантії автентичності користувача. На відміну від 2FA, багатофакторна автентифікація (далі MFA) не обмежується фіксованою парою факторів: система може послідовно або паралельно застосовувати декілька незалежних механізмів, залежно від контексту, ризику або політик безпеки [14].

1.4.3.1. Типові комбінації факторів

- Пароль + TOTP + Push. Базова комбінація, де пароль перевіряється першим, TOTP – другим, а push-запит слугує додатковим захистом при підозрілих діях.
- Пароль + біометрія + токен пристрою. Використовується у мобільних додатках, де біометрія локально розблоковує криптографічний ключ, що виступає як третій фактор.
- Токен доступу + TOTP / App confirm. У stateless архітектурах токен видається після первинної автентифікації, а додаткові фактори вимагаються при критичних операціях.

1.4.3.2. Архітектурні підходи до реалізації MFA [15]

- Послідовна перевірка факторів (step-chain)

Користувач проходить фактори поетапно. Система зберігає тимчасовий стан автентифікації між кроками. Перевага – гнучкість: можна відмовити після будь-якого етапу. Недолік – необхідність надійного керування проміжними сесіями та їх реплікації між вузлами.

- Паралельна перевірка факторів (step-parallel)

Сервер запускає перевірку кількох факторів одночасно (наприклад, TOTP + push). Рішення про успішну автентифікацію приймається після надходження позитивних результатів. Це знижує загальний час перевірки, що критично для систем реального часу, але вимагає асинхронної обробки та подієвої шини.

- Динамічне комбінування (policy-based)

Найгнучкіший варіант: застосовуються правила, які визначають набір факторів залежно від контексту (тип операції, IP, пристрій, поведінкові метрики тощо). Наприклад, для входу з довіреного пристрою може вимагатися лише пароль + токен, а для фінансової операції – додатковий біометричний фактор або push-підтвердження.

1.4.3.3. Комунікаційна інфраструктура

Для масштабованої реалізації MFA фактори автентифікації не обробляються монолітно. Зазвичай застосовують сервісну архітектуру:

- Auth Core. Відповідає за основний облік користувачів, токени та політики.
- MFA Service. Незалежний компонент, який керує факторами, верифікацією TOTP, push-запитами, біометричними ключами тощо.
- Message Broker. Використовується для асинхронної обробки push-запитів і біометричних підтверджень без блокування основного потоку.
- Temporary State Store. Зберігає короткочасні дані про прогрес проходження факторів.

1.4.4. Адаптивна автентифікація: принципи та сучасні рішення

1.4.4.1. Поняття та базові принципи

Адаптивна автентифікація – це підхід, який динамічно змінює рівень перевірки користувача залежно від контексту доступу та оціненого ризику. На відміну від статичних схем, де набір факторів фіксований, адаптивні системи використовують контекстні сигнали, поведінкові патерни та алгоритми ризик-аналізу, щоб вирішити, чи потрібно:

- пропустити користувача після базової перевірки;
- вимагати додаткові фактори (step-up MFA);
- повністю заблокувати спробу автентифікації.

У системах реального часу це особливо актуально, оскільки дозволяє підтримувати високий рівень безпеки без збільшення затримок для “нормальних” сценаріїв.

Основні принципи адаптивної автентифікації [16]:

- Контекстна оцінка кожної сесії. Автентифікація залежить не лише від введених даних, а й від метаданих запиту (IP, пристрій, геолокація, час доби, швидкість запитів тощо).

- Динамічна зміна рівня довіри. Замість фіксованих правил система обирає потрібний набір перевірок залежно від ситуації.
- Step-up MFA. У разі виявлення аномалій система вимагає додаткові фактори, не перериваючи основний потік.
- Автоматизація рішень. На основі правил або моделей (scoring/ML) приймається рішення про ризик без ручного втручання.
- Прозорість для користувача. При низькому ризику вхід виконується без додаткових кроків, без зниження безпеки.

Під час реалізації адаптивної автентифікацію будуть враховані наступні фактори, що перераховані у таблиці 1.7.

Таблиця 1.7

Контекстні та поведінкові фактори

Категорія	Приклади факторів
Мережеві	IP-адреса, ASN, країна, TOR/VPN
Пристрої	User-Agent, OS fingerprint, Device ID
Часові	Час доби, частота входів, сезонність
Поведінкові	Швидкість введення пароля, послідовність рухів миші, натискання клавіш, розкладка, аномальні патерни
Географічні	Відстань між поточним та попереднім входом, нетипова країна чи регіон

Основні контекстні фактори включають геолокацію, IP-адресу та репутацію мережі, час доби, тип і характеристики пристрою, налаштування браузера, модель поведінкового профілю користувача, а також контекст сесії. Поведінкові фактори охоплюють швидкість і динаміку введення тексту, характерні шаблони руху миші чи жестів, частоту та порядок виконання дій, типові маршрути навігації, звичну послідовність операцій і аномальні дії, що відрізняються від історичного профілю користувача. Комбінація цих сигналів дозволяє системі адаптивно визначати рівень ризику та автоматично обирати потрібний рівень автентифікації.

1.4.4.2. Ризик-аналіз та scoring

Після збору сигналів система обчислює ризиковий бал, який визначає сценарій подальших дій. Найпростіша схема – це набір правил:

- знайомий пристрій + відома IP → низький ризик;
- новий пристрій + інша країна → високий ризик.

Більш просунуті системи застосовують алгоритмічний scoring або моделі машинного навчання, що враховують історичну поведінку користувача та нетипові патерни. До таких підходів відносяться логістична регресія та дерева рішень для класифікації «звичайна/аномальна», unsupervised методи, як-от Isolation Forest та One-Class SVM, для виявлення нових аномалій, а також моделі, які оновлюються у реальному часі на стрімінгових даних за допомогою платформ типу Kafka у поєднанні з Flink або Spark Streaming [18].

1.4.4.3. Архітектурна реалізація

У типових мікросервісних системах адаптивна автентифікація реалізується як окремий сервіс ризик-оцінки, що працює у зв'язці з основним Auth-сервісом:

- Auth Service виконує базову перевірку облікових даних.
- Context Collector збирає сигнали з HTTP-заголовків, cookies, fingerprinting, telemetry.
- Risk Scoring Service синхронно або асинхронно обчислює бал ризику.
- Залежно від балу Auth Service або видає токен, або надсилає MFA-челендж, або відхиляє запит.

Для мінімізації затримок у реальному часі низькорівневі сигнали оцінюються синхронно (<10–30 мс), а поведінковий аналіз може виконуватись асинхронно, оновлюючи профіль користувача для майбутніх сесій. Часто використовують кешування для швидкого доступу до попередніх оцінок пристроїв та IP.

1.4.5. Підсумок

Системи автентифікації, що застосовуються у середовищах реального часу, мають суттєві відмінності від класичних веб-рішень через жорсткі вимоги до латентності, масштабованості та відмовостійкості. Використання stateful-сесій та

централізованих сховищ контексту значно обмежує горизонтальне масштабування та створює критичні точки відмови, тому у сучасних архітектурах перевага надається JWT, які дозволяють виконувати перевірку автентичності локально на кожному вузлі з мінімальною затримкою.

2FA та MFA залишаються базовими механізмами підвищення рівня безпеки, однак у потокових і мікросервісних сценаріях вони мають бути реалізовані через асинхронну обробку, розділення сервісів і, за можливості, паралельну валідацію факторів для уникнення блокування критичних потоків даних. Найбільш ефективними виявляються TOTP і push-сценарії, які легко інтегруються в розподілену архітектуру, тоді як SMS та email-коди мають вищу затримку і гірше масштабуються.

Адаптивна автентифікація формує наступний рівень безпеки, поєднуючи контекстні та поведінкові фактори з ризик-аналізом для динамічного вибору рівня перевірки. Завдяки цьому можливо зменшити кількість MFA-викликів для низькоризикових сценаріїв, не жертвуючи загальним рівнем захисту. Інтеграція адаптивних механізмів через окремі сервіси ризик-скорингу та кеші дозволяє забезпечити низьку латентність навіть у високонавантажених потокових системах.

Загалом, ефективна автентифікація в архітектурах реального часу передбачає комбінацію JWT, MFA та адаптивної логіки, реалізованої у розподілений спосіб з мінімальною залежністю від централізованих компонентів. Такий підхід забезпечує баланс між безпекою, продуктивністю та стійкістю системи до збоїв.

1.5. Порівняльний аналіз існуючих рішень

У таблицях 1.8-1.9 зведено порівняльну характеристику архітектур систем реального часу та методів автентифікації, які допоможуть резюмувати проведений аналіз по цим напрямкам та сформувати кінцеве бачення системи.

Таблиця 1.8

Порівняльна таблиця архітектур систем реального часу

Архітектура	Переваги	Недоліки
Моноліт	Простота розгортання та обслуговування, низький поріг входу для розробки	Одна точка відмови, вища латентність при високому навантаженні
Мікросервіс без брокерів	Гнучкіша масштабованість порівняно з монолітом, можливість поступового відмовостійкого розділення компонентів, краща ізоляція модулів	Відсутність централізованої подієвої шини ускладнює синхронізацію, можливі затримки при прямій комунікації між сервісами
Мікросервіс з брокерами	Висока відмовостійкість завдяки реплікації брокерів, природна горизонтальна масштабованість	Складніше налаштування та підтримка інфраструктури брокерів, вища початкова складність архітектури
Мікросервіс + stateless токени	Висока масштабованість без залежності від стану, низька latency через локальну перевірку токенів, легка інтеграція MFA та risk scoring, добра сумісність з потоковими WS/SSE сценаріями	Потрібне безпечне керування ключами підпису, необхідна політика ротації токенів та refresh-механізми, вища початкова складність конфігурації безпеки

Таблиця 1.8 (продовження)

Порівняльна таблиця архітектур систем реального часу

Мікросервіс + HA	Дуже висока відмовостійкість, автоматичне балансування навантаження та autoscaling, низька latency завдяки локальній перевірці токенів, повна сумісність із MFA та адаптивною автентифікацією	Складність розгортання, підвищені вимоги до DevOps-експертизи, потрібно продумане керування конфігураціями та ключами

Таблиця 1.9

Порівняльна таблиця методів автентифікації

Метод автентифікації	Переваги	Недоліки
Парольна автентифікація	Простота реалізації, не потребує додаткових пристроїв або каналів, сумісна з більшістю систем	Вразлива до підбору, фішингу та повторного використання, вимагає захисту бази паролів; залежить від центральної БД, що збільшує затримку при навантаженні

Таблиця 1.9 (продовження)

Порівняльна таблиця методів автентифікації

Stateful сесії	Зрозуміла модель роботи, простий контроль сесій з боку сервера	Погана горизонтальна масштабованість через sticky sessions, залежність від централізованого сховища, додаткова latency через lookup у Redis/БД, єдина точка відмови
Push-підтвердження	Зручність для користувача, можливість асинхронного step-up MFA	Потребує push-інфраструктури, керування токенами пристроїв
Stateless токени	Локальна перевірка, легка горизонтальна масштабованість, зручна інтеграція з мікросервісами та API Gateway, добра сумісність з MFA та адаптивними схемами	Потрібне безпечне керування ключами, необхідна політика ротації та refresh-механізми, ризик компрометації при витоку токена
TOTP	Локальна перевірка без зовнішніх сервісів, низька затримка перевірки, висока стійкість до перехоплення	Потрібно безпечно зберігати секрети, необхідна точна синхронізація часу, управління секретами у масштабі може бути складним

Таблиця 1.9 (кінець)

Порівняльна таблиця методів автентифікації

SMS/Email коди	Простота для користувача, не потребують попереднього налаштування	Висока затримка через зовнішні сервіси, вразливість до перехоплення, залежність від сторонніх каналів
MFA	Високий рівень безпеки, можливість комбінації факторів залежно від контексту, підтримка політик доступу з різною строгістю	Підвищена складність реалізації, потрібна узгоджена логіка факторів між сервісами, вимоги до тимчасових сховищ та брокерів для паралельної перевірки
Адаптивна автентифікація	Динамічне підвищення рівня захисту, мінімізація затримок у “звичайних” сценаріях, контекстна оцінка ризиків у реальному часі	Підвищена архітектурна складність, потребує збору контекстних даних та ML/rules рушія

Для розроблюваної системи найбільш доцільним є використання мікросервісної архітектури з брокером повідомлень у поєднанні з stateless токенами та механізмами високої доступності. Такий підхід забезпечує гарантовану доставку подій, горизонтальне масштабування, ізоляцію збоїв, а також можливість обробляти пікові навантаження без втрати повідомлень. Використання брокера дозволяє реалізувати принцип «подія прийде будь-коли, але ніколи не загубиться», що критично важливо для реальних систем, де затримка допустима, а втрата – ні.

Щодо автентифікації, найкраще поєднання – це stateless токени як основний механізм доступу та TOTP як другий фактор. Stateless токени мінімізують навантаження на центральні сховища сесій, добре працюють у розподіленому середовищі та поєднуються зі схемою HA. Для високоризикових дій, відповідних висновкам risk-engine, доцільно використовувати push-підтвердження або TOTP, які забезпечують сильну аутентифікацію без значного погіршення UX.

Висновки до розділу 1

Узагальнюючи результати проведеного аналізу, можна зробити висновок, що традиційні централізовані та stateful-підходи до побудови систем реального часу не відповідають сучасним вимогам до масштабованості, відмовостійкості та швидкодії. Наявність єдиних точок відмови, складність горизонтального масштабування та підвищена латентність роблять їх малопридатними для критично важливих застосувань. Натомість сучасні практики проєктування ефективних рішень ґрунтуються на використанні мікросервісної архітектури з елементами високої доступності та подієвих брокерів, що забезпечує гнучкість, стійкість до збоїв і здатність працювати в умовах високих навантажень у режимі реального часу.

У частині автентифікації оптимальним є застосування JWT як базового механізму авторизації, TOTP як надійного другого фактора та адаптивної автентифікації з ризик-аналізом, що дозволяє динамічно підвищувати рівень безпеки без негативного впливу на продуктивність. Така комбінація забезпечує високий рівень захисту користувачів і сервісів за умов мінімальної затримки та без втрати масштабованості.

Обраний підхід, що поєднує мікросервісну архітектуру з елементами високої доступності, подієвими брокерами та адаптивними механізмами автентифікації, виходить за рамки традиційних схем проєктування. Інноваційність полягає у поєднанні відмовостійкої потокової інфраструктури з інтелектуальними механізмами автентифікації, які динамічно змінюють рівень захисту залежно від ризикового контексту без збільшення базової затримки.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ВІДМОВОСТІЙКОЇ СИСТЕМИ РЕАЛЬНОГО ЧАСУ З АДАПТИВНОЮ АВТЕНТИФІКАЦІЄЮ

2.1. Постановка вимог до системи

2.1.1. Функціональні вимоги

Насамперед, система повинна надавати користувачам можливість реєстрації нових облікових записів із валідацією введених даних. У процесі реєстрації передбачається активація двохфакторної автентифікації (варіант з TOTP), яка здійснюється шляхом генерації та відображення QR-коду для додавання у мобільний автентифікатор. Це забезпечує базовий рівень захисту ще на етапі створення акаунту.

Вхід до системи здійснюється за допомогою комбінації логіну, пароля та TOTP-коду. Після успішної автентифікації користувач отримує короткочасний JWT-токен, який використовується для подальшого доступу до ресурсів без збереження стану на сервері. Для підвищення безпеки при кожному вході здійснюється аналіз контекстних факторів – мережевих, фізичних, поведінкових та часових (див. табл. 1.7).

Перевірка JWT-токена повинна виконуватися локально на кожному вузлі без централізованого сховища, що мінімізує затримки та усуває єдині точки відмови. Система повинна підтримувати автоматичне продовження сесій через refresh токени з визначеними політиками безпеки, що гарантує стабільність роботи при тривалих з’єднаннях.

Для підтримки стабільності система повинна мати вбудовані засоби моніторингу стану компонентів у реальному часі та базову систему оповіщень про відмови або аномальні ситуації. Також передбачається механізм резервування та автоматичного відновлення сервісів після збоїв, що гарантує безперервність роботи навіть у разі часткових відмов інфраструктури. Адміністративний інтерфейс або API має надавати можливість перегляду активних користувачів та сесій.

Користувацький веб-інтерфейс повинен бути інтуїтивним, підтримувати динамічне оновлення даних у реальному часі та забезпечувати захищену комунікацію через HTTPS/WSS. Взаємодія користувачів із системою повинна залишатися стабільною та швидкою незалежно від навантаження або внутрішніх перезапусків сервісів.

2.1.2. Нефункціональні вимоги

Система повинна зберігати працездатність навіть у випадку збоїв окремих компонентів або тимчасової недоступності окремих вузлів. Для цього передбачається використання механізмів кластеризації, балансування навантаження та реплікації даних. У разі відмови одного з сервісів навантаження має автоматично перерозподілятися між іншими екземплярами без переривання потоків даних і втрати сесій користувачів.

Ще однією критичною нефункціональною вимогою є масштабованість. Архітектура системи має забезпечувати можливість горизонтального розширення без змін у бізнес-логіці. Додавання нових вузлів не повинно потребувати простою або ручного втручання. Завдяки використанню брокера повідомлень та stateless-автентифікації система повинна легко обробляти збільшену кількість користувачів і подій у режимі реального часу без деградації продуктивності.

Продуктивність та затримки є особливо важливими для систем реального часу. Середній час обробки повинен залишатися стабільно низьким навіть за умов пікових навантажень. Для цього передбачається використання асинхронної взаємодії між сервісами через брокер повідомлень, кешування даних у Redis, а також локальну перевірку JWT-токенів без додаткових звернень до централізованих сховищ. Система має забезпечувати плавну роботу користувацького інтерфейсу без відчутних затримок у потоках оновлень чи під час аутентифікації.

Вимоги до безпеки передбачають багаторівневий захист усіх компонентів. Усі з’єднання повинні здійснюватися через захищені протоколи з актуальними

TLS-сертифікатами. Чутливі дані мають зберігатися у зашифрованому вигляді з обмеженим доступом.

2.2. Архітектура системи

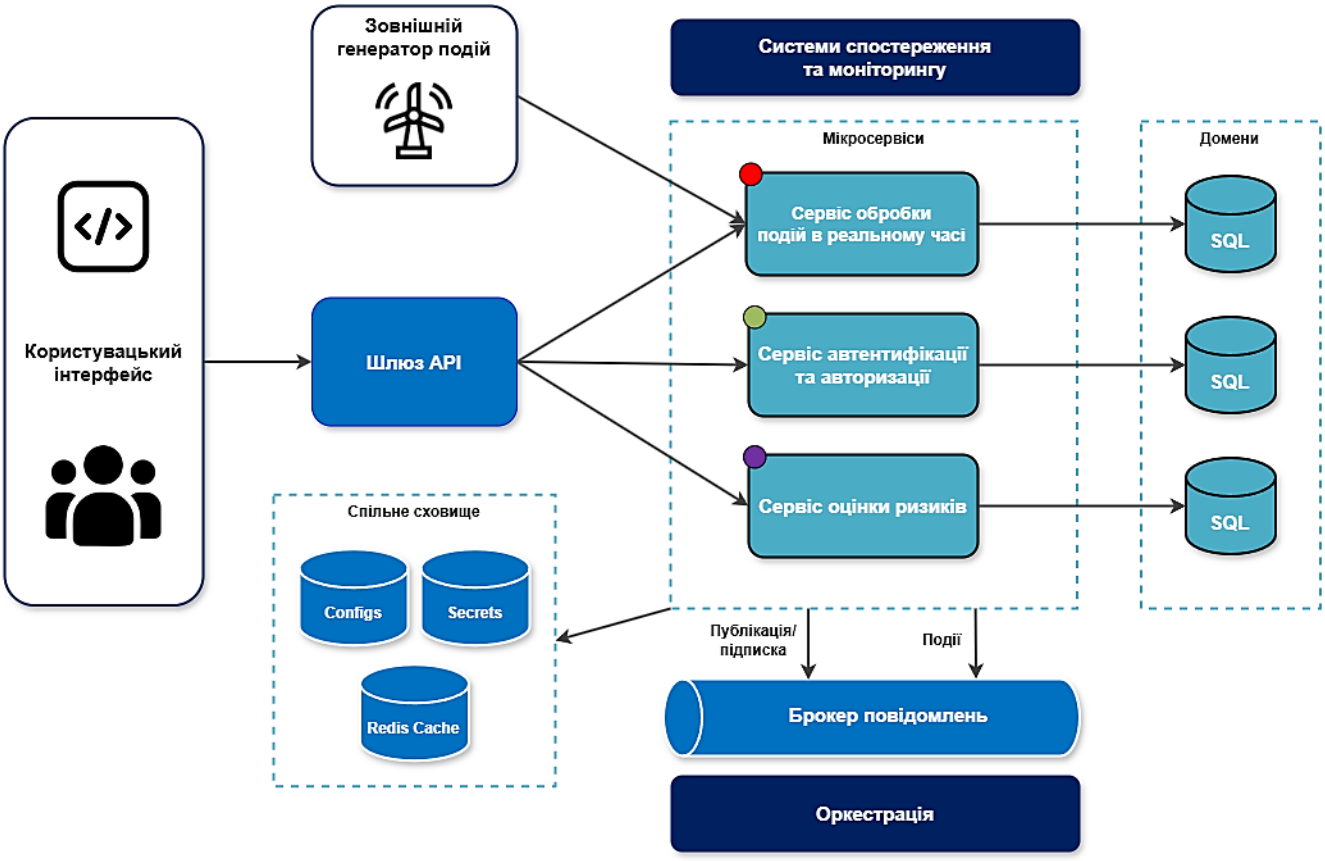

Рис. 2.1. Архітектура системи реального часу

Компонент API Gateway виконує роль єдиної вхідної точки до всіх сервісів системи. Він приймає вхідний трафік від користувачів та маршрутизує його до відповідних backend-компонентів відповідно до типу запиту. Gateway також відповідає за застосування політик безпеки (CORS, rate limiting) та реалізацію механізмів балансування навантаження. При відмові одного з інстансів запити автоматично перенаправляються до інших без втрати сесії користувача. Крім того, через шлюз встановлюються та підтримуються WebSocket/SSE-з'єднання для потокової передачі даних у реальному часі.

Сервіс автентифікації реалізує функції ідентифікації користувачів, керування обліковими записами та контроль доступу до ресурсів. Він обробляє запити на

логін, реєстрацію та видачу короткочасних JWT токенів, що використовуються для подальшої stateless-авторизації без збереження сесій на сервері. Крім базової автентифікації за логіном і паролем, сервіс підтримує двофакторну автентифікацію на основі TOTP, а також механізм step-up MFA – додаткову перевірку при виявленні ризикових умов. Для цього Auth Service взаємодіє з модулем адаптивної безпеки.

Модуль оцінки ризиків виконує адаптивну перевірку під час автентифікації та взаємодії користувачів із системою. Він аналізує контекстні фактори, такі як IP-адреса, геолокація, User-Agent, частота звернень та інші поведінкові фактори. На основі визначених правил або евристик Risk Scoring Service класифікує вхідні події за рівнем ризику та, за необхідності, ініціює додаткові кроки перевірки – наприклад, повторне введення TOTP або підтвердження через інший фактор.

Real-Time Service відповідає за прийом та трансляцію подій між користувачами в режимі реального часу. Він буде реалізований на реактивному стеку, який буде описаний наприкінці цього розділу, що дозволяє ефективно обслуговувати тисячі одночасних з’єднань без блокувань потоків. Сервіс підписується на повідомлення з брокера та доставляє їх клієнтам через WebSocket, забезпечуючи низьку затримку оновлень. Також він може приймати події від користувачів, публікувати їх у брокер та відправляти іншим клієнтам.

Зовнішнім генератором подій у системі виступають самі користувачі, які ініціюють різноманітні дії – спроби входу, створення або зміну даних, виконання чутливих операцій, переходи між пристроями чи мережами. Кожна така взаємодія формує подію, що надходить до системи через публічні API-ендпоінти і обробляється в реальному часі для оцінки ризиків та застосування адаптивних механізмів автентифікації.

Брокер повідомлень є центральною подієвою шиною системи. Він забезпечує асинхронний обмін між сервісами, розв’язує їх залежності та виконує роль буфера під час пікових навантажень. Продюсери публікують події у відповідні топіки, а споживачі отримують ці повідомлення незалежно від часу їх генерації. Завдяки

підтримці реплікації та кластеризації брокер гарантує високу доступність і стійкість до відмов.

Система використовує комбінацію реляційного сховища, кешу та журналу подій. Реляційне сховище відповідає за збереження постійних даних – облікових записів, TOTP-секретів, політик безпеки, журналів входів тощо. Redis використовується як високошвидкісний кеш для профілів ризику та інших тимчасових даних, що потребують частих перевірок із мінімальною затримкою. Окремо ведеться Event Store для зберігання історії подій у реальному часі.

Observability модуль охоплює засоби збору метрик, логів та трасування. Далі будуть обрані технології, які будуть використовуватися для збору технічних показників, трасування запитів між сервісами, централізованого зберігання та аналізу логів, а також візуалізації зібраних метрик.

Всі сервіси системи розгортаються у контейнерах та керуються платформою оркестрування. Це дозволяє автоматично масштабувати сервіси відповідно до навантаження, здійснювати rolling updates без простою, автоматично відновлювати збої та рівномірно розподіляти навантаження між вузлами. Liveness та readiness probes гарантують, що у кластері залишаються лише здорові екземпляри сервісів.

Користувацький інтерфейс є основною точкою взаємодії між кінцевим користувачем та системою. Він надає доступ до основних функцій, таких як реєстрація, автентифікація з використанням TOTP, перегляд та відправлення подій у режимі реального часу. Інтерфейс реалізується як веб-додаток, який взаємодіє з backend-сервісами через HTTPS для звичайних запитів та через WebSocket або Server-Sent Events для отримання динамічних оновлень. Важливою характеристикою є мінімальна затримка оновлення даних та стабільність з’єднання навіть за умов високого навантаження. Таким чином, Frontend не тільки виконує роль представницького шару, але й забезпечує інтерактивність та реалістичну демонстрацію роботи системи у режимі реального часу.

2.3 Архітектурні компоненти системи

2.3.1. Шлюз API та балансувальник навантаження

API Gateway є вхідною точкою для всіх клієнтських запитів до системи. Він забезпечить централізовану маршрутизацію, безпеку та контроль трафіку. Архітектурно шлюз розміщується на периферії системи. Для клієнтів це дійсно єдина точка входу, адже за рахунок внутрішньої маршрутизації URL-шляхів до відповідних сервісів розуміння реальної структури системи є неможливим для користувача.

Шлюз бере участь у механізмах High Availability: постійно перевіряє стан сервісів через health-checks і автоматично виключає «мертві» вузли з пулу за допомогою failover-механізму. При відновленні інстансу він знову включається до маршрутизації.

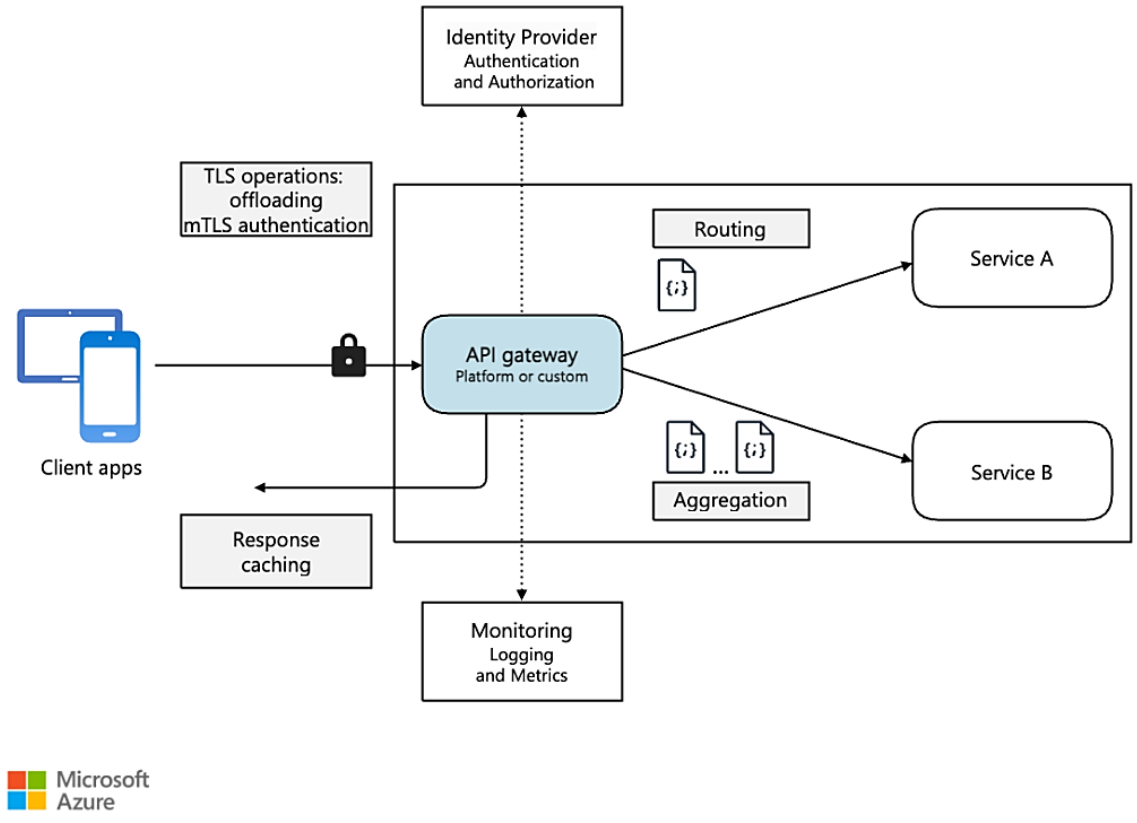

Рис. 2.2. Варіант архітектури API Gateway від Microsoft Azure [16]

Обов’язки компоненту:

- єдина точка входу й маршрутизації запитів;
- перший рубіж безпеки;
- балансування навантаження;
- учасник механізмів відмовостійкості.

2.3.2. Сервіс обробки подій у реальному часі

На відміну від класичного REST, де взаємодія ініціюється користувачем, сервіс реального часу працює в режимі push – система надсилає дані відразу після їх виникнення. Така реактивна модель взаємодії базується на патернах publish-subscribe та observer, де клієнти підписуються на певні потоки подій, а сервер транслює зміни в режимі реального часу.

Реактивна взаємодія суттєво впливає на проєктування серверної частини:

- Зміна ролі сервера. Сервер стає активним генератором подій, а не лише обробником запитів. Це потребує впровадження механізмів маршрутизації подій, керування підписками та широкомовного розповсюдження (fan-out);
- Стан з’єднання. Реактивні канали потребують підтримки відкритих сесій, що вимагає від системи зберігати інформацію про клієнтські підключення, їхній статус та права доступу.
- Вимоги до масштабування. Якщо система обробляє запити тисячі користувачів, блокуюча модель рано чи пізно призведе до вичерпання ресурсів. Тому застосовується неблокуюча, подійно-орієнтована архітектура, що дозволяє обробляти багато підключень у межах обмеженої кількості потоків.
- Затримка. Реактивна модель забезпечує мінімальний час доставки подій, оскільки дані передаються одразу після їх виконання.

Сервіс не генерує події самостійно, а виступає як транслятор подій, попередньо підписавшись на внутрішнє джерело подій (брокер повідомлень), та відправляє інформацію у форматі, зрозумілому клієнтам. Така архітектура відповідає патерну event streaming, де реальний час досягається завдяки часу виконання публікації/підписки, а не опитування.

Обов’язки компоненту:

- забезпечення реактивної push-взаємодії через тривалі з’єднання;
- підтримка великої кількості одночасних підключень;
- споживання подій з внутрішніх джерел і трансляція їх до клієнтів з мінімальною затримкою;
- перетворення повідомлень у зовнішній клієнтський формат.

2.3.3. Сервіс автентифікації та авторизації

У системі використовується обов’язкова двохфакторна автентифікація для всіх користувачів. Процес входу складається з двох етапів:

- Базова перевірка облікових даних. Користувач надсилає логін і пароль, які перевіряються централізовано. Якщо комбінація валідна, система переходить до другого етапу.
- Перевірка другого фактору. Користувач повинен ввести одноразовий код з TOTP-генератора. Без успішного проходження цього етапу токен не видається, а сесія не створюється.

Після успішного проходження обох етапів автентифікації користувачу видається JWT-токен, який містить:

- унікальний ідентифікатор користувача,
- ролі та права доступу,
- час видачі та термін дії,
- додаткові атрибути контексту (за потреби).

JWT підписується приватним ключем сервісу автентифікації й може бути перевірений іншими мікросервісами локально, без звернення до централізованої сесійної бази. Це забезпечує stateless-авторизацію, яка добре масштабується у кластері [19]. Для продовження сесії використовуватиметься механізм refresh-токенів, які також проходять повну перевірку та можуть бути відкликані у разі компрометації.

Інстанс real-time сервісу виконує перевірку токена на етапі handshake, після чого з’єднання вважається автентифікованим до закінчення терміну дії токена або його відкликання.

Обов’язки компоненту:

- обробка первинних облікових даних користувача;
- двохфакторна аутентифікація;
- генерація, підпис та видача JWT access- та refresh-токенів;
- централізоване управління політиками доступу та перевірка токенів;
- можливість відкликання токенів та завершення сесій.

2.3.4. Сервіс оцінки ризиків

Сервіс оцінки ризиків виконує функцію контекстного аналізу поведінкових та мета-даних користувача з метою визначення рівня ризику доступу в конкретний момент часу. У разі виявлення аномалій, виявлених за допомогою описаного нижче алгоритму, блокується доступ до критичних секцій застосунку: стрімінгу чутливих даних або виконання важливих операцій. Сервіс приймає пул інформації, виконує обчислення і повертає оцінку ризику у вигляді одної з категорій: low, medium, high. Контекстні фактори, що аналізуються:

- геолокаційна інформація;
- час доби та нетипові часові інтервали;
- User-Agent та fingerprint пристрою;
- поведінкові патерни роботи за пристроями вводу (клавіатури/миші).

Алгоритм оцінки ризику описаний у п. 2.4.

2.3.5. Брокер повідомлень

Брокер повідомлень виступає посередником між сервісами, які не взаємодіють напряму, а надсилають та отримують повідомлення через публікацію у топіки або черги. Завдяки цьому сервіси роз’єднані в часі – відправник не чекає, поки отримувач обробить повідомлення, можна горизонтально масштабувати споживачів без змін логіки відправника, підвищується відмовостійкість, оскільки повідомлення зберігаються у черзі, якщо споживач тимчасово недоступний, а

також легко додавати нові сервіси, достатньо підписатися на потрібний топік. Поширені патерни використання включають Publish–Subscribe, коли кілька сервісів підписані на один топік і отримують однакові події, що підходить для push-нотифікацій та real-time оновлень; Point-to-Point, де кожне повідомлення обробляється лише одним споживачем для рівномірного розподілу навантаження; Fan-out, коли одне повідомлення публікується в кілька топіків для обробки різними групами сервісів; та Event Sourcing, коли брокер зберігає хронологію подій як лог для повторного звернення. Ключові механізми брокера включають буферизацію та черги для гарантії доставки навіть при тимчасових збоях, підтвердження обробки, щоб повідомлення не втрачалися, ретенцію логів для можливості зчитування історичних подій новими споживачами.

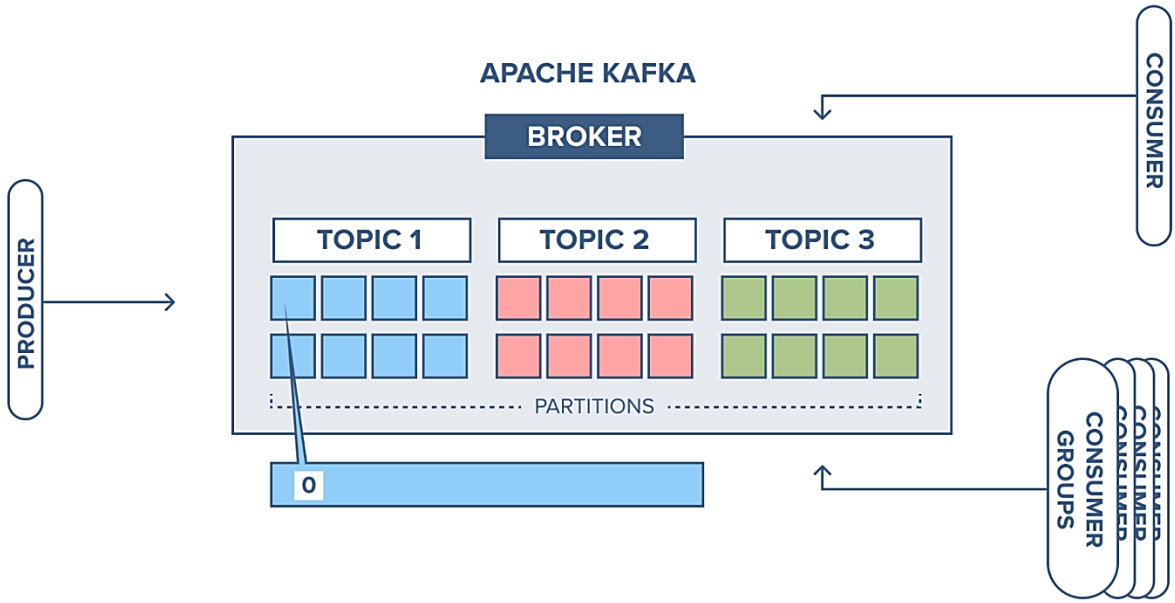

Рис. 2.3. Схема зберігання черги повідомлень Apache Kafka [20]

2.3.6. Сховище даних та кеш

Основне сховище даних виконує роль централізованого джерела істини. У ньому зберігаються усі критичні дані, включаючи профілі користувачів, історію автентифікацій, журнали подій безпеки, поведінкові профілі, а також аналітичну інформацію та дані для аудиту. Зберігання здійснюється у надійному, транзакційному середовищі, яке гарантує цілісність та відновлюваність даних навіть у випадку збоїв.

Система кешування виступає проміжним швидким шаром доступу до інформації, яка часто запитується або використовується в режимі реального часу. Вона дозволяє зменшити затримки при зверненні до даних та знизити навантаження на основну базу. Кеш може зберігати профілі користувачів для швидкої оцінки ризиків, короткострокові сесійні дані, попередньо обчислені показники для дашбордів, проміжні аналітичні результати та інші дані, які важливі для реактивної роботи системи.

Взаємодія між цими двома рівнями може відбуватись за різними патернами: read-through (при відсутності даних у кеші – автоматичне завантаження з БД), write-through або write-back, а також з використанням TTL для автоматичного очищення застарілих записів. У масштабованих середовищах можуть застосовуватись реплікація та шардінг бази даних, що підвищує відмовостійкість та розподіляє навантаження між вузлами.

Сховище даних та кеш взаємодіють з усіма основними сервісами системи. Сервіс автентифікації використовує кеш для швидкої перевірки профілів та токенів. Сервіс оцінки ризиків поєднує історичні дані з БД і кешовані профілі, щоб оперативно реагувати на спроби входу. Real-time сервіс звертається до кешу для миттєвого отримання актуальних значень без прямого доступу до бази. Аналітичні інструменти можуть зчитувати дані з реплік БД, не навантажуючи транзакційне ядро.

Обов’язки компоненту:

- забезпечення надійного довготривалого зберігання критичних даних і журналів подій;
- організація швидкого доступу до даних;
- зниження навантаження на основну базу завдяки використанню проміжного кешу;
- підтримка узгодженості даних між кешем і основним сховищем;
- забезпечення масштабованості та відмовостійкості через реплікацію та TTL-політики.

2.3.7. Сховище секретів та конфігурацій

Компонент відповідає за безпечне централізоване зберігання та контрольований доступ до конфіденційних даних, які необхідні сервісам для роботи. Доступ до секретів здійснюється за допомогою політик авторизації, які чітко визначають, який сервіс або користувач може зчитувати або змінювати конкретні секрети.

Окрім секретів, компонент також керує конфігураційними параметрами середовищ – наприклад, URL сервісів, налаштування брокера повідомлень, параметри кешу чи бази даних, часові зони, політики безпеки тощо. Це дозволяє централізовано оновлювати конфігурації без необхідності перебудови та повторного деплою кожного сервісу.

Сховище секретів та конфігурацій інтегрується з усіма компонентами системи. Сервіс автентифікації зчитує з нього ключі для підпису та валідації JWT-токенів. Real-time сервіс використовує його для збереження даних підключень, внутрішніх ключів доступу до брокера та SSL-сертифікатів. Сховище даних – для зберігання паролів до баз, конфігурацій реплікації та шардінгу.

Обов’язки компоненту:

- централізоване та безпечне зберігання секретів;
- керування доступом до секретів;
- зберігання та поширення конфігураційних параметрів середовищ;
- можливість динамічного оновлення конфігурацій без перезапуску сервісів;
- підтримка узгодженості налаштувань у всій системі.

2.3.8. Системи спостереження та моніторингу

Цей компонент працює на кількох рівнях одночасно:

- Інфраструктурний рівень. Контроль вузлів, контейнерів, мережі, пам’яті, CPU, дисків.
- Рівень застосунків. Відстеження продуктивності кожного сервісу.

- Бізнесовий рівень. Контроль ключових показників (KPI), які відображають правильність роботи функціональних процесів: кількість успішних логінів, відсоток заблокованих входів, аномалії поведінки користувачів тощо.

Для ефективної роботи системи моніторингу необхідно визначити та відстежувати набір критичних метрик, які покривають як інфраструктурну, так і прикладну частину:

Інфраструктурні метрики

- CPU usage / Load Average на кожному вузлі – виявлення перевантаження;
- використання пам’яті – контроль споживання ресурсів сервісами;
- стан дисків: вільне місце, I/O latency, зношування SSD;
- стан мережі: latency, packet loss, відкриті з’єднання;

Прикладні метрики

- кількість активних з’єднань у real-time сервісі – відображає реальне навантаження;
- кількість авторизацій/логінів за хвилину – базова активність користувачів;
- кількість невдалих логінів та MFA-помилок – сигнал потенційних атак або збоїв;
- error rate (5xx відповіді, exceptions) для кожного сервісу;
- cache hit/miss ratio – ефективність кешу;

Бізнесові та поведінкові метрики

- рівень ризикових сесій: LOW/MEDIUM/HIGH за останній період;
- частота тригерів step-up автентифікації;
- аномалії поведінки: різке зростання кількості підозрілих сесій;

Дані агрегуються в централізовану систему, а потім візуалізуються за допомогою дашбордів. На основі встановлених порогів можуть бути налаштовані алерти, які автоматично сповіщають розробників про потенційні проблеми.

2.3.9. Користувацький інтерфейс

Користувацький інтерфейс забезпечує взаємодію кінцевого користувача з усіма функціями платформи. Його основна роль полягає у тому, щоб надавати доступ до сервісів системи у зручній, зрозумілій та швидкій формі. Для систем реального часу інтерфейс повинен працювати не як статична вебсторінка, а як динамічне середовище, яке миттєво реагує на зміни, що надходять від бекенду через реактивні канали.

Компонент тісно взаємодіє з іншими компонентами системи. Всі REST-запити та авторизаційні операції проходять через API-шлюз, який маршрутизує їх до відповідних мікросервісів. Після автентифікації інтерфейс отримує JWT-токен та використовує його для подальших захищених запитів. Під час підключення до реактивного каналу UI підписується на відповідні потоки подій залежно від ролі користувача та автоматично оновлює віджети та аналітичні блоки при надходженні нових даних. Таким чином, інтерфейс стає інтерактивним та самодостатнім компонентом, який синхронно працює з усією системою без потреби в ручних оновленнях сторінок.

Обов’язки компоненту:

- забезпечення зручного, безпечного та швидкого доступу до функцій системи;
- реалізація автентифікації користувачів з підтримкою двофакторної перевірки;
- отримання та відображення даних у реальному часі;
- інтерактивне керування параметрами безпеки та сесіями користувачів;
- адаптація функціональності інтерфейсу згідно з даними сервісу оцінки ризиків;
- безпечна комунікація з бекендом через JWT та захищені протоколи.

2.3.10. Інфраструктура та оркестрація

Компонент відповідає за розгортання, масштабування, управління та підтримку всіх інших елементів архітектури у стабільному та відмовостійкому

середовищі. Він забезпечує роботу системи як цілісного комплексу, включаючи автоматизацію розгортання, керування залежностями, балансування навантаження між вузлами, обробку відмов та оновлення без простоїв.

В основі цієї підсистеми лежить контейнеризація всіх компонентів, що дозволяє розгортати систему у вигляді незалежних модулів з чітко визначеними інтерфейсами. Для керування контейнерами застосовується система оркестрації, яка автоматично розподіляє сервіси між вузлами, підтримує їхню доступність, стежить за станом контейнерів, перезапускає у разі збоїв та масштабовує за потреби. Завдяки цьому система зберігає працездатність навіть при виході з ладу окремих елементів.

Ключовим елементом є також автоматизовані CI/CD-процеси, які забезпечують безперервну інтеграцію змін та їх безпечне розгортання у різних середовищах. Це дозволяє оперативно впроваджувати новий функціонал або оновлення без ручних втручань і без зупинки системи. Також передбачені механізми бекапів, логування та централізованого зберігання артефактів, що спрощує відновлення після інцидентів і забезпечує аудит змін.

Інфраструктура включає налаштування мережевих політик, балансувальників навантаження, DNS та SSL, що гарантує стабільну комунікацію між компонентами та зовнішніми клієнтами. Завдяки горизонтальному масштабуванню та кластеризації, система здатна адаптуватись до зростання навантаження без зміни коду, просто додаючи нові вузли.

Обов’язки компоненту:

- забезпечення стабільного розгортання всіх компонентів у кластерному середовищі;
- автоматизація оновлень та керування залежностями через CI/CD;
- налаштування мережевої взаємодії, балансування та безпечного доступу;
- забезпечення відмовостійкості, резервного копіювання та швидкого відновлення;
- підтримка стабільної роботи всієї архітектури як єдиного цілого.

2.4. Алгоритм оцінки ризиків для застосування механізмів адаптивної автентифікації

Алгоритм оцінки ризику виглядає наступним чином:

1. Збір контекстної інформації. При кожній спробі аутентифікації та участі у критичних операціях вводу/виводу зчитується набір даних для їх перевірки.

2. Оцінка геолокаційного відхилення. Виконується порівняння поточної локації зі списком звичайних локацій користувача за формулою (2.1):

$$S_{geo} = \begin{cases} 1, & \text{нова країна/регіон} \\ \dfrac{d_{curr} - d_{avg}}{d_{max}}, & \text{для проміжних випадків} \\ 0, & \text{звичайна локація} \end{cases}, \quad (2.1)$$

де S_{geo} – частковий бал ризику геолокації, d_{curr} – поточне відхилення від історичної локації, d_{avg} – середнє відхилення, d_{max} – максимальне історичне відхилення.

3. Оцінка часової аномалії. Аналіз того, чи поточний час аутентифікації відрізняється від звичайних годин активності користувача за формулою (2.2):

$$S_{time} = 1 - P_{time}(t_{curr}), \quad (2.2)$$

де S_{time} – частковий бал ризику часового інтервалу, $P_{time}(t_{curr})$ – ймовірність активності користувача в цей часовому проміжку.

4. Оцінка новизни пристрою. Перевірка, чи використовувався це пристрій раніше, за формулою (2.3):

$$S_{device} = \begin{cases} 1, & \text{невідомий пристрій} \\ 0, & \text{зарєестрований пристрій} \end{cases}, \quad (2.3)$$

де S_{device} – частковий бал ризику пристрою.

5. Оцінка кількості невдалих спроб. Якщо під час проходження адаптивної автентифікації до одного з елементів функціоналу було багато невдалих

спроб – це значною мірою підвищує безпеку й, скоріш за все, користувач втратить свою сесію. Використовується наступна формула (2.4):

$$S_{fail} = \begin{cases} 0, & N_{fail} \leq N_{threshold} \\ \dfrac{N_{fail} - N_{threshold}}{N_{max} - N_{threshold}}, & N_{fail} > N_{threshold} \end{cases}, \quad (2.4)$$

де S_{fail} – частковий бал ризику невдалих спроб, N_{fail} – кількість невдалих спроб, $N_{threshold}$ – порогове значення росту ризику, N_{max} – верхня межа шкали.

6. Оцінка поведінкових аномалій. Перевірка інтервалів між натисканням клавіш, швидкість мишки, ритм введення символів за формулою (2.5):

$$S_{behavior} = \frac{dist(b_{curr}, b_{profile})}{dist_{max}}, (2.5)$$

де $S_{behavior}$ – частковий бал ризику поведінки, b_{curr} – поточні поведінкові дані, $b_{profile}$ – середній профіль користувача, $dist_{max}$ – максимальна відстань, що нормалізує результат у діапазон [1, 2].

7. Сума балів та нормалізація. Після обчислення часткових балів за кожним фактором, ці бали зважуються й підсумовуються у спільну величину ризику за формулами (2.6, 2.7):

$$S = \omega_{geo} \cdot S_{geo} + \omega_{time} \cdot S_{time} + \omega_{device} \cdot S_{device} + \omega_{fail} \cdot S_{fail} + \omega_{behavior} \cdot S_{behavior}, (2.6)$$

$$S_{norm} = \frac{S}{\sum \omega_i}, (2.7)$$

де S – сумарний бал ризику, S_{norm} – нормалізований бал ризику, ω_i – вага окремого фактору ризику.

8. Повернення результату та ухвала дії. На основі балу визначається категорія ризику, після чого ухвалюють рішення: LOW – стандартна автентифікація, додаткових методів захисту не потребується, MEDIUM – обмеження деяких функцій з вимогою додаткового фактору, HIGH – аналогічно до MEDIUM, але відбувається відправка повідомлення на пошту або в застосунок на

всіх інших підключених девайсах про аномальну поведінку. Використовується наступна формула (2.8):

$$RiskLevel = \begin{cases} LOW, & S_{norm} < 0.4 \\ MEDIUM, & 0.4 \leq S_{norm} \leq 0.7, \\ HIGH, & S_{norm} > 0.7 \end{cases} \quad (2.8)$$

За особистою експертною оцінкою була сформована таблиця 2.1, що містить відповідність між факторами ризику та їх вагами, що в сумі дає 100.

Таблиця 2.1

Рекомендовані ваги факторів ризику ($\sum \omega_i = 100$)

Фактор	Вага	Пояснення
Геолокація	20	Важлива, але не дуже стабільна через VPN/роумінг або можливі подорожі
Часовий інтервал	10	Корисний, але допоміжний сигнал
Пристрій	30	Найсильніший предиктор викрадення облікового запису
Невдалі спроби	20	Чіткий сигнал зловмисної активності
Поведінка	20	Динаміка набору тексту та руху миші – майже унікальні патерни для конкретної людини

Обрані фактори та їхні вагові коефіцієнти відображають їхню відносну силу впливу на підсумковий ризиковий бал у системі адаптивної автентифікації. Геолокація виступає важливим індикатором, оскільки раптова зміна країни або міста може свідчити про небажану активність, проте цей сигнал нестабільний через VPN, корпоративні проксі чи подорожі, тому його вплив обмежений. Часовий інтервал використовується як допоміжний фактор: аномальний час входу може вказувати на ризик, але сам по собі він рідко є ключовим.

Пристрій має найбільшу вагу, оскільки використання незнайомого або різко зміненого пристрою є одним із найсильніших предикторів компрометації облікового запису. Поведінкові характеристики – такі як динаміка набору тексту, патерни руху миші та особливості взаємодії з інтерфейсом – забезпечують майже

унікальний біометричний профіль користувача, тому є важливим, але ресурсомістким сигналом, що виправдовує високу, але не максимальну вагу. Разом ці ваги збалансовано формують модель, яка чутлива до ключових ознак ризику та достатньо стійка до випадкових відхилень.

2.5. Механізми відмовостійкості

Проведемо порівняльну характеристику наявних алгоритмів балансування навантаження, що визначена у таблиці 2.2.

Таблиця 2.2

Порівняльна характеристика алгоритмів балансування навантаження

Алгоритм	Принцип роботи	Переваги	Недоліки	Рекомендовані випадки використання
Round Robin	Рівномірне розподілення по черзі	Простота в реалізації	Не враховує поточне навантаження	Базові веб-додатки з однорідними серверами
Least Connections	Вибір серверу з найменшою кількістю з’єднань	Гарна продуктивність для тривалих сесій	Потребує постійного моніторингу стану з’єднань	WebSocket, SSE, stateful-з’єднання
IP Hash	Прив’язка клієнта до серверу	«Липкі» сесії без окремого сховища	Нерівномірне навантаження при нерівномірному розподілі IP	Stateful-додатки без JWT

Таблиця 2.2 (продовження)

Порівняльна характеристика алгоритмів балансування навантаження

Weighted Round Robin	Враховується вага (потужність) серверів	Гнучкість налаштувань, все ще досить простий	Не враховує динамічне навантаження в реальному часі	Кластери з різною потужністю
Least Response Time	Вибір серверу з найменшою затримкою	Орієнтація на реальний час	Чутливість до короткочасних піків	Реактивні сервіси

У більшості сучасних розподілених високонавантажених систем за статистикою найефективнішими алгоритмами є Least Connections та Least Response Time. Для даного проєкту, в якому ключову роль відіграє реактивний Real-Time Service з великою кількістю тривалих WebSocket/SSE-з'єднань, найбільш доцільним є саме Least Connections. Алгоритм Least Response Time добре працює для коротких REST-запитів, однак у випадку постійних двосторонніх з'єднань він не відображає реального навантаження на вузол, оскільки базується лише на швидкості відповіді під час початкового встановлення з'єднання. Натомість Least Connections безпосередньо враховує кількість одночасно активних клієнтів і рівномірно розподіляє їх між інстансами, що забезпечує стабільну пропускну здатність, нижчу затримку під час трансляції подій та запобігає перевантаженню окремих вузлів. Крім того, цей алгоритм простіший у реалізації в середовищі Kubernetes.

Failover – це ключовий елемент відмовостійкої архітектури, який гарантує безперервність роботи при збоях окремих компонентів. У цій системі використовується кластерний підхід, коли кожен критичний сервіс має щонайменше два активних інстанси. Ingress-контролер постійно виконує health-check запити до бекендів. У випадку, якщо один з інстансів не відповідає, він автоматично виключається з пулу, а навантаження перерозподіляється на інші доступні екземпляри без втрати з'єднань користувачів [21].

На рівні брокера повідомлень застосовується реплікація та механізм лідерства. Якщо один із брокерів виходить з ладу, лідерство передається іншому вузлу без зупинки обробки подій. Подібні механізми використовуються і в базі даних, де налаштована реплікація та можливий автоматичний промоушн standby-сервера до primary у разі відмови.

Автоматичне відновлення компонентів системи здійснюється за допомогою інструментів оркестрації контейнерів. У разі збоїв або зависань оркестратор автоматично перезапускає контейнер. Якщо проблема полягає у вузлі, на якому розміщено сервіс, pod переміщується на інший доступний вузол.

При оновленнях застосовується стратегія rolling updates, яка дозволяє оновлювати сервіси по черзі, не перериваючи роботу системи. Нові екземпляри запускаються і перевіряються, після чого старі поступово виводяться з пулу.

2.6. Вибір технологічного стеку

Таблиця 2.3 відображає фінальне рішення щодо технологій, які будуть використовуватися у розробці кожного компоненту в наступному розділі.

Таблиця 2.3

Вибір технологічного стеку

Компонент	Технологія/інструмент	Обгрунтування вибору
Backend	Java 17, Spring Boot, Spring WebFlux	Стабільний стек з широкою підтримкою корпоративних рішень. WebFlux призначений для побудови реактивних, неблокуючих сервісів
Frontend	React	Побудова реактивних SPA з ефективним керуванням станом
Messaging	Apache Kafka	Висока пропуска здатність, низька затримка, масштабованість та надійність

Таблиця 2.3 (продовження)

Вибір технологічного стеку

Storage	PostgreSQL, Redis	PostgreSQL – ACID-база з підтримкою реплікації, Redis – швидкий кеш
Security/Config	HashiCorp Vault	Централізоване сховище секретів з підтримкою політик доступу й ротації ключів
Monitoring/ Observability	Prometheus, Grafana	Prometheus – гнучкий збір та зберігання метрик, Grafana – потужна візуалізація та система сповіщень
Infrastructure/ Orchestration	Docker, Kubernetes, Gitlab CI/CD	Розгортання та ізоляція сервісів, балансування та self-healing, автоматизація тестів та деплою

Обрані компоненти системи формують сучасний стек, що забезпечує високу продуктивність, відмовостійкість та безпеку. Backend реалізовано на Java 17 із використанням Spring Boot і Spring WebFlux, що дозволяє будувати реактивні, неблокуючі сервіси з високою масштабованістю та стабільною підтримкою корпоративних рішень. Frontend на React забезпечує ефективне побудування SPA з гнучким керуванням станом, що дозволяє реалізовувати інтерактивний та швидкий інтерфейс користувача.

Messaging організовано через Apache Kafka, яка гарантує низькі затримки, високу пропускну здатність і надійне доставляння повідомлень, що критично для системи реального часу. Storage поєднує PostgreSQL як ACID-базу для надійного зберігання структурованих даних і Redis для швидкого кешування та зменшення затримок. Security/Config реалізовано через HashiCorp Vault, що забезпечує централізоване управління секретами, контроль доступу та автоматичну ротацію ключів.

Для моніторингу та спостережності використано Prometheus і Grafana, де Prometheus збирає та зберігає метрики, а Grafana забезпечує візуалізацію та

налаштування сповіщень. Infrastructure та Orchestration покривають Docker і Kubernetes для ізоляції, балансування навантаження та самовідновлення сервісів, а GitLab CI/CD автоматизує тестування, збірку та деплой. Усі компоненти разом формують стійку, масштабовану та безпечну платформу для реалізації відмовостійкої системи реального часу з адаптивною автентифікацією.

Висновки до розділу 2

У цьому розділі було спроєктовано архітектуру відмовостійкої системи реального часу з адаптивною автентифікацією. На основі попереднього аналізу обрано архітектурний підхід, який поєднує мікросервісну структуру, реактивну обробку подій та використання брокера повідомлень. Такий підхід дозволяє забезпечити високу масштабованість, гнучкість інтеграцій, стабільну роботу у реальному часі та підвищену відмовостійкість завдяки ізоляції компонентів.

Для кожного архітектурного елемента були визначені його функціональна роль, взаємодії з іншими модулями та ключові механізми забезпечення надійності. Ретельно спроєктовано шлюз API, сервіс обробки подій, модуль автентифікації та оцінки ризиків, брокер повідомлень, підсистеми зберігання даних і кешування, сховище секретів, систему моніторингу, користувацький інтерфейс, а також інфраструктурний рівень з механізмами оркестрації.

Окрему увагу приділено вибору технологічного стеку, який забезпечує поєднання стабільності та інноваційності. Використання Java Spring Boot та WebFlux дозволяє ефективно реалізувати реактивні бекенд-сервіси, Kafka – забезпечити асинхронну взаємодію з низькою затримкою та гарантованою доставкою повідомлень у визначені інтервали часу, Redis – прискорити доступ до даних, а Kubernetes – автоматизувати розгортання та масштабування. Обрані інструменти не лише технічно узгоджені між собою, а й забезпечують легку подальшу підтримку та розвиток системи.

РОЗДІЛ 3
РОЗРОБКА ВІДМОВОСТІЙКОЇ СИСТЕМИ РЕАЛЬНОГО ЧАСУ З АДАПТИВНОЮ АВТЕНТИФІКАЦІЄЮ

3.1. Реалізація сервісу автентифікації та авторизації

Модуль aura-auth реалізує функціональність безпечної автентифікації користувачів у прототипі системи реального часу AURA (Adaptive User Risk Authentication). Архітектура модуля дотримується принципів Domain-Driven Design та чистої архітектури, поділяючи логіку на три рівні [23]:

a) Контролери – REST API для обробки запитів;

b) Сервіси – бізнес-логіка

c) Репозиторії – доступ до бази даних через Spring Data JPA

Комунікація з іншими мікросервісами здійснюється через Kafka-topic auth-events, де публікуються події успішних та невдалих входів. Взаємодія з користувачем відбувається через REST API та одноразові 2FA-коди, згенеровані в TOTP-застосунку.

```
src/main/java/com.kpi.fict.aura.auth/
│
├─ config/          → конфігураційні класи (Spring, безпека, Vault, Kafka)
├─ controller/      → контролери та класи для валідації запитів
├─ converter/       → конвертери для шифрування полів
├─ dto/             → об'єкти передачі даних (запити/відповіді API)
├─ exception/       → кастомні типи винятків
├─ filter/          → фільтри запитів (JwtAuthFilter, CORS)
├─ handler/         → обробники подій безпеки
├─ mapper/          → мапери між сутностями та DTO (MapStruct)
├─ model/           → сутності бази даних (User, Credentials, Token, Device)
├─ repository/      → репозиторії JPA для доступу до БД
├─ service/         → бізнес-логіка
├─ validator/       → валідація користувацьких даних
```

Рис. 3.1. Структура модулю автентифікації [23]

Реєстрація користувача – це перший етап взаємодії із системою, під час якого створюється обліковий запис, зберігається у базі даних і генерується секрет для 2FA.

```
@PostMapping("/register")
public ResponseEntity<?> register(@RequestBody SignUpRequest request) {
    Long userId = authService.saveUser(request);
    String totpSecret = totpService.generateSecretForUser(userId);
    String qr = totpService.generateQrCodeUri(request.getEmail(), totpSecret);
    return ResponseEntity.ok(Map.of("userId", userId, "qr", qr));
}
```

Рис. 3.2. Метод реєстрації користувача

Спочатку перевіряється валідність даних, потім шифрується пароль, після чого користувач зберігається в БД та надсилається повідомлення на пошту про підтвердження акаунту.

```
public Long saveUser(SignUpRequest signUpRequest) {
    securityValidator.validateUserRegistration(signUpRequest);
    UserSecurityDto savedUser = userService.saveUser(
            userMapper.toSaveUserDto(signUpRequest, passwordEncoder));
    userService.notifyUserConfirmation(savedUser);
    return savedUser.id();
}
```

Рис. 3.3. Ідемпотентний метод збереження облікового запису

Метод є універсальним як для реєстрації через базову HTML-форму, так і при першому вході через зовнішні сервіси. Тому базовий метод збереження користувача є ідемпотентним: якщо обліковий запис існує – оновлюємо enabled = true, адже вхід через зовнішній сервіс вже підтверджує той факт, що користувач володіє поштою.

```
public UserSecurityDto saveUser(SaveUserDto saveUserDto) {
    Optional<User> optionalUser =
userRepository.findByEmail(saveUserDto.getEmail())
            .map(user -> userMapper.updateRegisteredUser(user, saveUserDto));

    Credentials credentials = optionalUser
            .map(credentialsRepository::findByUser)
            .filter(Optional::isPresent)
            .map(opt -> userMapper.updateCredentials(opt.get(), saveUserDto))
            .orElse(userMapper.toCredentials(saveUserDto));

    if (optionalUser.isEmpty()) {
        credentials.setUser(userRepository.save(userMapper.toUser(saveUserDto)));
        credentials = credentialsRepository.save(credentials);
    }
    return userMapper.toUserSecurityDto(credentials);
}
```

Рис. 3.4. Метод збереження нового користувача

Налаштування безпеки здійснюється у класі WebSecurityConfig, який визначає правила доступу до REST-ендпоінтів, JWT-фільтр, а також інтегрує кастомній хендлери аутентифікації.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
    http.csrf().disable()
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers("/api/auth/register", "/api/auth/login",
"/api/auth/oauth2/**").permitAll()
                    .anyRequest().authenticated())
            .oauth2Login(oauth2 -> oauth2
                    .loginPage("/api/auth/oauth2")
                    .successHandler(authenticationHandler))
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(new JwtAuthFilter(jwtService),
UsernamePasswordAuthenticationFilter.class)
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    return http.build();
}
```

Рис. 3.5. Реалізація ланцюга фільтрів безпеки зі stateless-політикою

Після перевірки пароля користувач повинен підтвердити особу через одноразовий TOTP-код.

```
@PostMapping("/verify-2fa")
public ResponseEntity<?> verifyTwoFactor(@RequestBody TwoFactorRequest request) {
    boolean valid = totpService.verifyCode(request.getUserId(),
request.getCode());
    if (!valid) {
        kafkaProducer.sendAuthEvent(AuthEvent.failed(request.getUserId(), "Invalid
OTP"));
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid OTP");
    }
    String token = jwtService.generateToken(request.getUserId());
    kafkaProducer.sendAuthEvent(AuthEvent.success(request.getUserId(), "2FA
verified"));
    return ResponseEntity.ok(Map.of("accessToken", token));
}
```

Рис. 3.6. Метод перевірки TOTP-коду

Клас JwtService реалізує створення, перевірку та оновлення токенів. Токени підписуються секретним ключем з Vault, термін дії access-токену – 1 година, refresh-токену – 7 днів.

```
public String generateToken(Long userId) {
    return Jwts.builder()
            .subject(userId.toString())
            .issuedAt(Date.from(Instant.now()))

.expiration(Date.from(Instant.now().plusMillis(tokenExpirationMilliseconds)))
            .signWith(secretKey, Jwts.SIG.HS256)
            .compact();
}

public boolean isTokenNotAcceptable(String token) {
    try {
        Claims jwtClaims = extractAllClaims(token);
        return jwtClaims.getExpiration().toInstant().isBefore(Instant.now());
    } catch (JwtException | IllegalArgumentException ex) {
        return true;
    }
}
```

Рис. 3.7. Генерування та валідація JWT-токенів

Сервіс TotpService відповідає за генерацію і верифікацію коду TOTP. Для цього використовується бібліотека com.eatthepath.otp, що створює одноразові паролі на основі алгоритму HMAC-SHA1. Інтервал доступності – 30 секунд.

```
@Service
public class TotpService {
    private static final Duration STEP = Duration.ofSeconds(30);
    private final TimeBasedOneTimePasswordGenerator generator;
    private final VaultService vaultService;
    private final UserRepository userRepository;

    public TotpService(VaultService vaultService, UserRepository userRepository) {
        this.generator = new TimeBasedOneTimePasswordGenerator(STEP);
        this.vaultService = vaultService;
        this.userRepository = userRepository;
    }

    public boolean verifyCode(Long userId, String code) {
        User user = userRepository.findById(userId).orElseThrow();
        String decryptedSecret = vaultService.decrypt(user.getTotpSecret());
        int expected = generator.generateOneTimePassword(
                new SecretKeySpec(Base64.getDecoder().decode(decryptedSecret),
"HmacSHA1"), Instant.now());
        return String.valueOf(expected).equals(code);
    }
}
```

Рис. 3.8. Верифікація TOTP-коду

Події автентифікації та безпеки фіксуються й асинхронно публікуються в Kafka. Ці події мають один з наступних типів, зазначених у таблиці 3.1.

Таблиця 3.1

Типи подій безпеки

Тип події	Призначення
USER_REGISTERED	Створено акаунт
LOGIN_SUCCESS	Валідні логін/пароль
LOGIN_FAILED	Невдала спроба логіну
OTP_VERIFIED	Успішна TOTP-перевірка
OTP_FAILED	Невдала TOTP-перевірка
OAUTH2_LOGIN	Первинна ідентифікація через зовнішній сервіс
LOGOUT	Вихід із системи

Щоб зберегти локальний порядок подій користувача, ключем для топіку auth-events слугує userId. Це гарантує, що всі події одного користувача потраплятимуть в одну партицію та читаються послідовно, водночас дозволяючи горизонтально масштабуватися кількістю партицій.

Витяг з конфігурації Kafka-продюсера у Spring:

```
@Configuration
public class KafkaConfig {

    @Bean
    public ProducerFactory<String, AuthEvent> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    public KafkaTemplate<String, AuthEvent> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

Рис. 3.9. Конфігурація Kafka-продюсера

3.2. Реалізація модулю оцінки ризиків

Сервіс побудовано за event-driven та реактивною моделями. Вхідні дані надходять через події, згенеровані Kafka-consumer, або через потік поведінкових метрик від UI/realtime-service. Як працює сервіс під час взаємодії користувача [24]:

1) Після успішної автентифікації risk-service перехоплює подію LOGIN_SUCCESS та підтягує контекст: останній відомий IP/fingerprint, час доби тощо.

2) Обчислюється risk score і надається відповідна категорія: LOW – нічого не робити, MEDIUM – інтеграція step-up автентифікації на критичний функціонал, HIGH – те саме, що в MEDIUM, але з відправкою попередження на пошту з можливістю розлогувати пристрій.

3) Логування зліпку обчисленного risk score з деякими деталями.

```
src/main/java/com.kpi.fict.aura.risk/
│
├─ config/          → Kafka, Redis, DB (R2DBC/JPA), Vault
├─ dto/             → події, контексти, результати
├─ ingestion/       → слухачі Kafka
├─ enrichment/      → ContextRepository, GeoResolver, ProfileCache, BehaviorTap
├─ evaluation/      → RiskEvaluator (ваги/формули)
├─ pipeline/        → RiskPipeline (керує стадіями: enrich → score → persist → alert)
├─ repository/      → репозиторії
├─ publish/         → StepUpNotifier
├─ observability/   → метрики, логування
```

Рис. 3.10. Структура модулю оцінки ризиків [24]

Згідно з формулами та вагами факторів, наданих у розділі 2, була побудована наступна функція оцінки ризику:

```
@Component
public class RiskEvaluator {

    private static final double W_GEO = 0.20, W_TIME = 0.10, W_DEV = 0.30, W_FAIL
= 0.20, W_BEH = 0.20;

    public RiskResult score(EnrichedAuthEvent e) {
        double sGeo  = geoScore(e);
        double sTime = timeScore(e);
        double sDev  = deviceScore(e);
        double sFail = failuresScore(e);
        double sBeh  = behaviorScore(e);

        double r = 100.0 * (W_GEO*sGeo + W_TIME*sTime + W_DEV*sDev + W_FAIL*sFail
+ W_BEH*sBeh);
        RiskLevel level = (r < 30) ? RiskLevel.LOW : (r < 60 ? RiskLevel.MEDIUM :
RiskLevel.HIGH);

        return new RiskResult(e.userId(), r, level, explain(sGeo, sTime, sDev,
sFail, sBeh));
    }

    private double geoScore(EnrichedAuthEvent e) {
        if (!e.lastGeo().isPresent()) return 0.1;
        boolean countryChanged =
!e.geo().country().equals(e.lastGeo().get().country());
        boolean impossible = e.travelTimeHours() < e.minTravelTimeHours();
        return countryChanged ? (impossible ? 1.0 : 0.6) : 0.0;
    }

    private double timeScore(EnrichedAuthEvent e) {
        return e.isUnusualHour() ? 0.7 : 0.0;
    }

    private double deviceScore(EnrichedAuthEvent e) {
        return e.isKnownDevice() ? 0.0 : 0.8;
    }

    private double failuresScore(EnrichedAuthEvent e) {
        int f = e.failedAttemptsLast15m();
        return Math.min(1.0, f / 5.0);
    }

    private double behaviorScore(EnrichedAuthEvent e) {
        if (!e.behavior().isPresent()) return 0.15;
        BehaviorMetrics b = e.behavior().get();
        return clamp(0.0, 1.0, 0.5*b.keystrokeAnomaly() + 0.5*b.mouseAnomaly());
    }

    private Map<String, Object> explain(double g, double t, double d, double f,
double b) {
        return Map.of("geo", g, "time", t, "device", d, "failures", f, "behavior",
b);
    }

    private double clamp(double lo, double hi, double v) {
        return Math.max(lo, Math.min(hi, v)); }
}
```

Рис. 3.11. MVP функції оцінки ризиків

Пайплайн оцінки ризику у модулі risk-service працює як реактивний конвеєр, що в режимі реального часу перетворює сирі події автентифікації на числову оцінку ризику. Коли користувач виконує будь-яку дію – логін, підтвердження 2FA, зміну пристрою – модуль aura-auth публікує подію в Kafka-топік auth-events. Risk-service, маючи підписку на цей топік, одразу перехоплює повідомлення, обробляє його через послідовність стадій і повертає результат у межах кількох сотень мілісекунд.

Спочатку перевіряється ідемпотентність: якщо подія з тим самим eventId уже була опрацьована, вона ігнорується, що запобігає дублюванню. Далі виконується збагачення контекстом – сервіс підтягує останні відомості про користувача з кешу Redis або таблиці risk_profiles: країну попереднього входу, типовий часовий інтервал активності, довірені пристрої та кількість невдалих спроб. Усі ці атрибути формують розширену структуру EnrichedAuthEvent.

Наступна стадія – фічінг, тобто перетворення сирих даних на нормалізовані показники факторів ризику. Конвеєр працює у реактивному режимі (Reactor Flux): кожен етап виконується неблокуюче, з підтримкою backpressure, щоб забезпечити стабільність при пікових навантаженнях. Обробка має at-least-once семантику з підтвердженням Kafka-офсетів після успішного запису в базу.

Сервіс збирає власні метрики (risk_events_total, risk_high_ratio, processing_latency_ms), які експортуються у Prometheus, дозволяючи контролювати SLO – не більше 500 мс на повний цикл обчислення ризику.

```java
@Component
@RequiredArgsConstructor
public class AuthEventsListener {
    private final RiskPipeline riskPipeline;

    @KafkaListener(topics = "auth-events", groupId = "risk-service")
    public void onMessage(AuthEvent event, Acknowledgment ack) {
        riskPipeline.process(event)
                .doOnSuccess(res -> ack.acknowledge())
                .subscribe();
    }
}

@Service
@RequiredArgsConstructor
public class RiskPipeline {
    private final ContextRepository contextRepo;
    private final RiskEvaluator riskEvaluator;
    private final RiskStorage riskStorage;
    private final AlertsPublisher alertsPublisher;
    private final IdempotencyService idempotency;

    public Mono<RiskResult> process(AuthEvent evt) {
        if (idempotency.seen(evt.getEventId())) {
            return Mono.empty();
        }
        return contextRepo.enrich(evt)
                .map(riskEvaluator::score)
                .flatMap(riskStorage::persist)
                .doOnNext(res -> idempotency.mark(evt.getEventId()));
    })
}
```

Рис. 3.12. Реалізація пайплайну оцінки ризику

3.3. Реалізація модулю обробки подій у реальному часі

Realtime-service відповідає за миттєву доставку безпекових подій та системних нотифікацій до веб-клієнтів і консолі адміністратора. Він споживає події з Kafka, трансформує їх у фронтенд-підтримувані DTO та пушить через SSE або WebSocket. Сервіс є повністю stateless, масштабується горизонтально, підтримує backpressure і гарантує, що користувачі бачать оновлення ≤ 1 с від моменту виникнення [25].

```
src/main/java/com.kpi.fict.aura.realtime/
│
├─ config/          → Kafka (Reactor Kafka), WebFlux/SSE, WebSocket, Redis, CORS
├─ security/        → ACL-фільтри
├─ dto/             → транспортні моделі для фронту
├─ kafka/           → приймання подій
├─ stream/          → Sinks (multicast/replay), fan-out, фільтрація за ACL
├─ api/             → SSE-контролер, WS-ендпоінти, REST для ручного replay
├─ util/            → утиліти
```

Рис. 3.13. Структура модулю обробки подій у реальному часі [25]

Базовий транспорт – Server-Sent Events (SSE), як найпростіший спосіб однонаправленого real-time стріму з мінімумом накладних витрат і чудовою сумісністю з проксі/балансувальниками. Для двосторонніх сценаріїв використовується STOMP-over-WebSocket.

Принцип роботи потоку реального часу:

- Risk-service публікує рішення HIGH_RISK_DETECTED у security-alerts.
- Realtime-service через Reactor Kafka споживає подію, застосовує ACL (чи має поточний користувач право бачити цей домен/тенант/користувача), та пушить її у in-memory Sinks.Many<AlertDto>.
- Клієнти, що підписані через /api/rt/alerts/stream (SSE) або /ws/alerts (WS), отримують елемент потоку.

```
@Configuration
class KafkaRealtimeConfig {

    @Bean
    ReceiverOptions<String, SecurityAlert>
receiverOptions(@Value("${kafka.bootstrap}") String bootstrap) {
        return ReceiverOptions.<String, SecurityAlert>create(Map.of(
                ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrap,
                ConsumerConfig.GROUP_ID_CONFIG, "realtime-service",
                ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class,
                ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class,
                JsonDeserializer.TRUSTED_PACKAGES, "*",
                ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"
        )).subscription(List.of("security-alerts"));
    }

    @Bean
    Flux<SecurityAlert> alertsStream(ReceiverOptions<String, SecurityAlert>
options) {
        return KafkaReceiver.create(options)
                .receive()
                .map(rec -> {
                    rec.receiverOffset().acknowledge();
                    return rec.value();
                })
                .publish()
                .refCount(1);
    }
}
```

Рис. 3.14. Витяг з коду приймання подій і фан-аут у гарячий стрім

```
@Component
@RequiredArgsConstructor
class JwtWebFilter implements WebFilter {

    private final JwtVerifier jwtVerifier;

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
        String auth =
exchange.getRequest().getHeaders().getFirst(HttpHeaders.AUTHORIZATION);
        if (auth != null && auth.startsWith("Bearer ")) {
            return jwtVerifier.authenticate(auth.substring(7))
                    .flatMap(p -> chain.filter(exchange.mutate()
                            .principal(Mono.just(p)).build()));
        }
        return chain.filter(exchange);
    }
}
```

Рис. 3.15. JWT-фільтр для SSE/REST

3.4. Брокер повідомлень

Брокер повідомлень є “кровоносною системою” усієї платформи: через нього aura-auth публікує події автентифікації, risk-service споживає їх і повертає рішення/алерти, а realtime-service розсилає критичні сповіщення клієнтам. Було обрано Apache Kafka як базовий транспорт подій через гарантовану горизонтальну масштабованість, порядок повідомлень у межах ключа (per-user/per-tenant), низьку затримку та зрілий інструментарій спостережності. Для збереження локального порядку подій користувача ключем партиціювання виступає userId**.**

Події поділяються за функціональними каналами:

- auth-events – оперативні події автентифікації (реєстрація, логін, OTP-верифікація, робота з пристроями).
- security-alerts – підвищений ризик/інциденти, що потребують реакції у реальному часі.

Трафік між сервісами та Kafka захищений SASL/SCRAM. На боці кластера застосовуються ACL:

- aura-auth: WRITE у auth-events;
- risk-service: READ із auth-events, WRITE у security-alerts;
- realtime-service: READ із security-alerts.

Жодних wildcard-прав. Доступ до брокерів сегментовано мережевими політиками (Kubernetes NetworkPolicy/VPC ACL). Секрети (логін/пароль SASL, сертифікати) зберігаються у Vault, а не в application.yml.

3.5. Сховище даних і кеш

PostgreSQL використовується як центральне сховище для сервісів aura-auth та risk-service. Вона забезпечує транзакційність, зовнішні ключі, підтримку JSONB і ефективну індексацію для пошуку за факторами ризику. База підтримує реплікацію з одним primary і двома standby вузлами для HA, а також logical

replication між сервісами для побудови аналітичних звітів без навантаження основних таблиць.

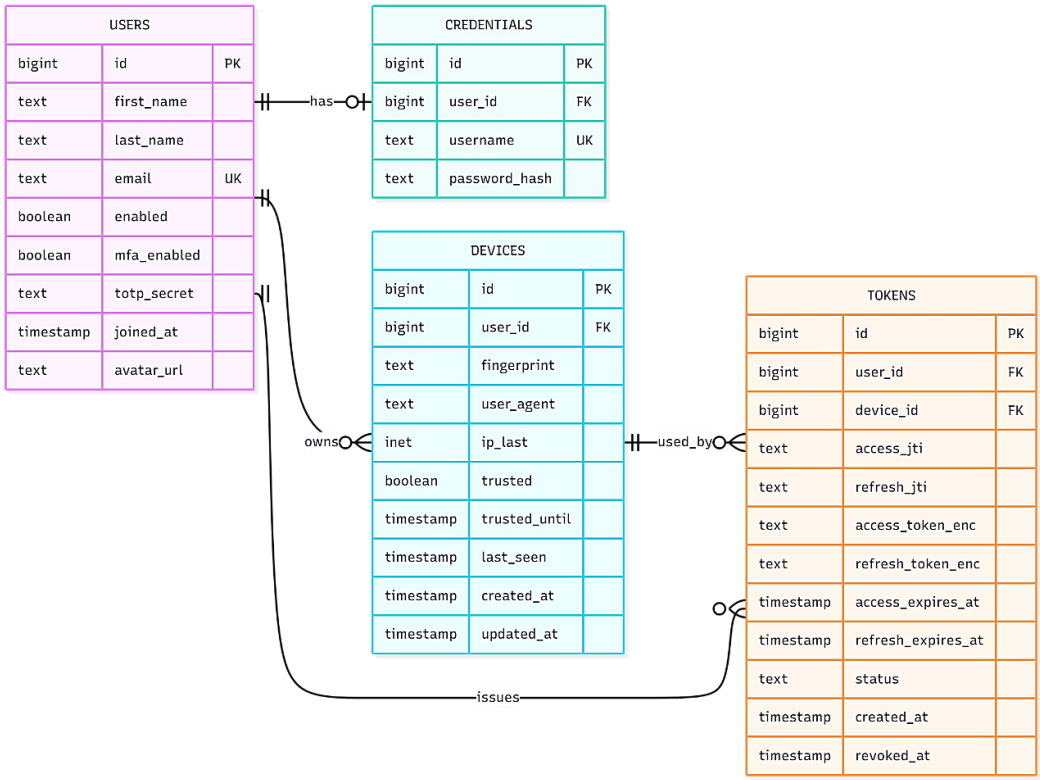

Рис. 3.16. ER-діаграма модулю автентифікації

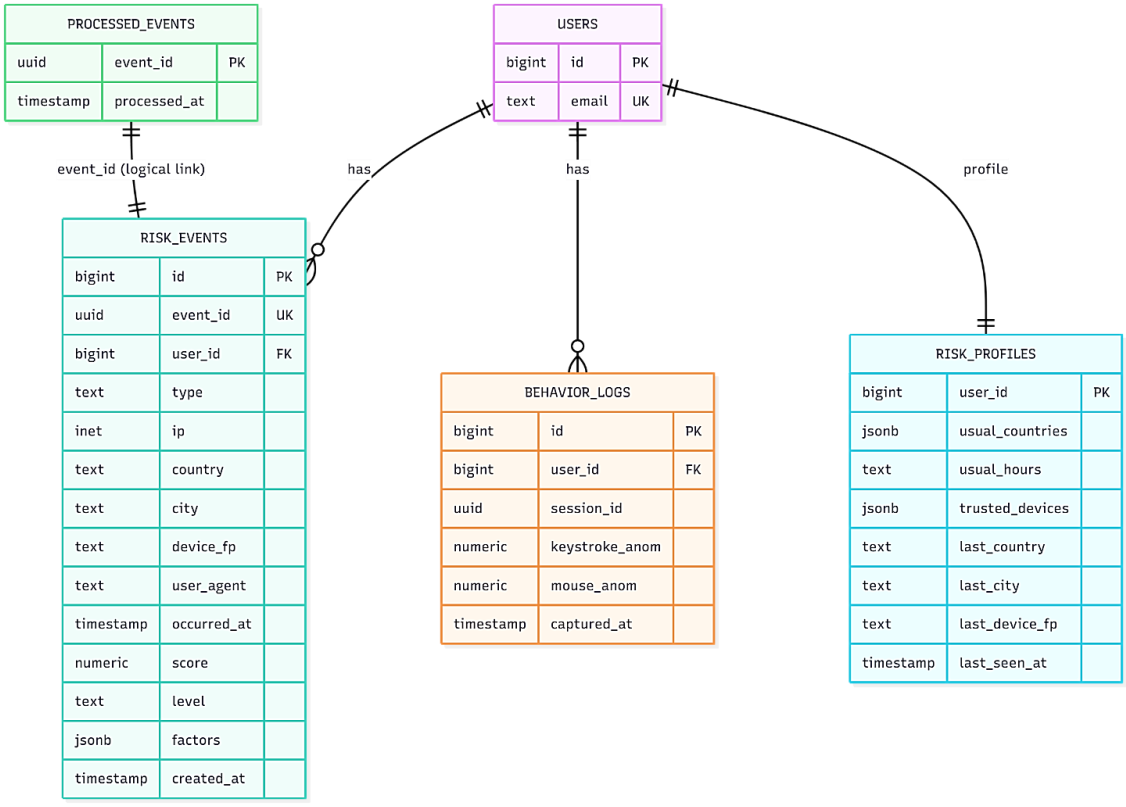

Рис. 3.17. ER-діаграма модулю оцінки ризиків

Redis використовується у всіх модулях для забезпечення миттєвого доступу до динамічних даних, які не потребують довгострокового зберігання.

Конфігурація Redis здійснюється через Spring Data Redis із пулом Lettuce та шифруванням транспортного каналу TLS:

```
spring:
  data:
    redis:
      host: ${REDIS_HOST}
      port: 6379
      ssl: true
      timeout: 2s
      lettuce:
        pool:
          max-active: 8
          max-idle: 4
```

Рис. 3.18. Конфігурація кешу Spring Data Redis

Кожен з трьох мікросервісів (auth-, risk-, realtime-service) мають інтегрований Redis у корисних для його застосування місцях, наприклад:

— Збереження TOTP-коду. Якщо користувач не введе код за TTL коду (30 секунд), запис автоматично видаляється.

```
@Service
@RequiredArgsConstructor
public class OtpCacheService {
    private final RedisTemplate<String, Object> redisTemplate;
    private static final Duration TTL = Duration.ofSeconds(60);

    public void saveOtp(String email, String otpCode) {
        redisTemplate.opsForValue().set("otp:" + email, otpCode, TTL);
    }

    public boolean verifyOtp(String email, String code) {
        String cachedCode = (String) redisTemplate.opsForValue().get("otp:" +
email);
        if (cachedCode != null && cachedCode.equals(code)) {
            redisTemplate.delete("otp:" + email);
            return true;
        }
        return false;
    }
}
```

Рис. 3.19. Використання кешу для верифікації TOTP

— Кешування профілю ризику. Профіль зберігається на 15 хвилин, що знижує навантаження на реляційну БД. Коли користувач проявляє нову поведінку, запис інвалідується.

```java
@Service
@RequiredArgsConstructor
public class RiskProfileCache {

    private final RedisTemplate<String, RiskProfile> redisTemplate;
    private final RiskProfileRepository repository;

    private static final Duration TTL = Duration.ofMinutes(15);

    public RiskProfile getOrLoad(Long userId) {
        String key = "risk:profile:" + userId;
        RiskProfile cached = redisTemplate.opsForValue().get(key);
        if (cached != null) return cached;

        RiskProfile fromDb = repository.findByUserId(userId)
                .orElse(new RiskProfile(userId));
        redisTemplate.opsForValue().set(key, fromDb, TTL);
        return fromDb;
    }
}
```

Рис. 3.20. Кешування профілю ризику

— Стрічка останніх подій. Це дозволяє відображати на фронтенді останні дії користувачів у реальному часі без складних SQL-запитів.

```java
@Service
@RequiredArgsConstructor
public class RealtimeEventBuffer {

    private final RedisTemplate<String, String> redisTemplate;
    private static final String KEY = "alerts:recent";
    private static final int MAX_SIZE = 1000;

    public void pushEvent(String eventJson) {
        redisTemplate.opsForList().leftPush(KEY, eventJson);
        redisTemplate.opsForList().trim(KEY, 0, MAX_SIZE - 1);
    }

    public List<String> getRecentEvents() {
        return redisTemplate.opsForList().range(KEY, 0, MAX_SIZE - 1);
    }
}
```

Рис. 3.21. Кешування останніх подій

Всі конфіденційні поля шифруються на рівні ORM за допомогою AES-256 через реалізацію AttributeConverter:

```
@Converter
@Component
public class StringEncryptionConverter implements AttributeConverter<String,
String> {
    private final EncryptionService encryptionService;

    @Override
    public String convertToDatabaseColumn(String attribute) {
        return attribute != null ? encryptionService.encrypt(attribute) : null;
    }

    @Override
    public String convertToEntityAttribute(String dbData) {
        return dbData != null ? encryptionService.decrypt(dbData) : null;
    }
}
```

Рис. 3.22. Конвертер шифрованих даних ORM-БД

Клас EncryptionService, що реалізує (де-)шифрування даних:

```
@Service
public class EncryptionService {
    private final String algorithm;
    private final SecretKey secretKey;

    public String encrypt(String data) {
        try {
            Cipher cipher = Cipher.getInstance(algorithm);
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            return
Base64.getEncoder().encodeToString(cipher.doFinal(data.getBytes()));
        } catch (Exception ex) {
            throw new EncryptionException(ex.getMessage());
        }
    }

    public String decrypt(String encryptedData) {
        try {
            Cipher cipher = Cipher.getInstance(algorithm);
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            return new
String(cipher.doFinal(Base64.getDecoder().decode(encryptedData)));
        } catch (Exception ex) {
            throw new EncryptionException(ex.getMessage());
        }
    }
}
```

Рис. 3.23. Клас (де-)шифрування даних

3.6. Система моніторингу та спостереження

Система моніторингу забезпечує повну прозорість стану мікросервісів, брокерів повідомлень, баз даних, кешів і мережевих компонентів.

Мета – виявляти відхилення ще до того, як вони переростуть у відмови, а також мати змогу швидко визначити причину інциденту (root cause analysis).

Для цього використано комплексний стек спостереження:

- Prometheus – збір і зберігання метрик;
- Grafana – візуалізація метрик і побудова дашбордів;
- Loki – централізований збір логів;
- Micrometer — шар інтеграції Prometheus зі Spring Boot.

Всі сервіси експортують метрики через ендпоінт /actuator/prometheus. Prometheus збирає їх з інтервалом 15 секунд, зберігає у власному таймсерійному сховищі, а Grafana відображає у вигляді інтерактивних панелей.

Технічні метрики системи:

a) aura-auth

- auth_requests_total{status="success|failed"} – кількість успішних/невдалих логінів;
- mfa_verification_time_seconds – середній час перевірки 2FA;
- jwt_token_refresh_count_total – кількість оновлень токенів;
- otp_redis_latency_ms – затримка доступу до Redis;
- vault_secret_fetch_latency_ms – час отримання секретів із Vault.

b) risk-service

- risk_event_processing_time_seconds – час обробки події ризику;
- risk_kafka_consume_lag – затримка між надсиланням і обробкою Kafka-повідомлення;
- cache_hits_total / cache_misses_total – ефективність кешування профілів ризику;
- anomaly_behavior_detected_total – кількість аномальних поведінкових подій.

c) realtime-service

- websocket_active_sessions_total – кількість активних WS-з’єднань;

- event_broadcast_latency_ms – затримка доставки подій;
- event_throughput_per_second – кількість відправлених подій у секунду;
- redis_pubsub_message_rate – частота повідомлень у Redis pub/sub.

Конфігурація Prometheus і Spring Boot:

```
management:
  endpoints:
    web:
      exposure:
        include: prometheus,health,info
  metrics:
    tags:
      application: aura-auth
      environment: prod
```

Рис. 3.24. Spring boot healthcheck конфігурація

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'aura-auth'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['aura-auth:8080']
  - job_name: 'risk-service'
    static_configs:
      - targets: ['risk-service:8081']
  - job_name: 'realtime-service'
    static_configs:
      - targets: ['realtime-service:8082']
```

Рис. 3.25. Prometheus конфігурація

У Grafana створено набір дашбордів:

- Authentication Dashboard – успішні/невдалі входи, OTP, токени, Redis latency;
- Risk Dashboard – розподіл ризиків (LOW/MEDIUM/HIGH), середній бал, затримка Kafka;
- System Health – CPU, heap memory, response time, open connections.

Для кожного дашборду задано порогові кольори (зелений, жовтий, червоний), що дозволяє миттєво оцінити стан системи.

Всі сервіси мають спільну конфігурацію для Promtail, який пересилає журнали в Loki:

```yaml
server:
  http_listen_port: 9080
positions:
  filename: /tmp/positions.yaml
clients:
  - url: http://loki:3100/loki/api/v1/push
scrape_configs:
  - job_name: 'aura-logs'
    static_configs:
      - targets: [localhost]
        labels:
          job: aura-auth
          __path__: /var/log/aura/*.log
```

Рис. 3.26. Налаштування пересилки журналів у Loki

3.7. Реалізація користувацького інтерфейсу

Фронтенд є центральним інтерфейсом взаємодії користувача із системою. Його роль – забезпечити зручну, швидку та безпечну роботу з функціоналом реального часу, автентифікації й моніторингу. Він виступає клієнтом для REST API сервісів (aura-auth, risk-service) та WebSocket-з'єднання (realtime-service).

```
frontend/
│
├─ components/
│   ├─ auth/            → форми логіну, MFA, реєстрації
│   ├─ dashboard/       → панелі з ризиками, користувачами, подіями
│   ├─ charts/          → графіки ризиків і активності
│   ├─ notifications/   → push-сповіщення в реальному часі
├─ store/               → стан (Zustand)
├─ services/            → REST і WebSocket клієнти
├─ hooks/               → власні хуки для аутентифікації, ризиків, подій
├─ pages/
   ├─ login.tsx
   ├─ register.tsx
   ├─ dashboard.tsx
   ├─ settings.tsx
```

Рис. 3.27. Структура модулю користувацького інтерфейсу

Основні функціональні потоки:

а) Реєстрація користувача

- користувач вводить email і пароль;
- верифікує email;

- отримує QR-код для підключення TOTP;
- після підтвердження система створює профіль ризику.

b) Авторизація

- користувач вводить логін → отримує оцінку ризику;
- якщо ризик LOW → прямий вхід;
- якщо MEDIUM/HIGH → вимога step-up 2FA.

c) Панель моніторингу

- після входу користувач бачить дашборд графіків аналітики у реальному часі через WebSocket. Як приклад стрімінгу даних, на графіках відображатиметься інформація про споживання серверних ресурсів, поточну кількість активних сесій та кількість ризикових подій.

d) Налаштування безпеки

- вибір політики застосування MFA;
- перегляд довірених пристроїв та керування ними.

3.8. Інфраструктура та оркестрація

Основна ідея – immutable infrastructure: кожен мікросервіс упаковується в контейнер із власними залежностями, що гарантує повторюваність середовищ.

Кожен мікросервіс має власний Dockerfile, який складається за принципом multi-stage build для зменшення розміру образу та підвищення безпеки. Для локального розгортання створено спільний docker-compose.yml, який оркеструє всі сервіси, бази даних, кеші та інструменти моніторингу:

```yaml
version: '3.9'
services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  kafka:
    image: bitnami/kafka:3.7
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CFG_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"

  zookeeper:
    image: bitnami/zookeeper:3.9
    environment:
      ALLOW ANONYMOUS LOGIN: "yes"

  vault:
    image: hashicorp/vault:1.17
    cap_add:
      - IPC_LOCK
    ports:
      - "8200:8200"
    environment:
      VAULT_ADDR: http://0.0.0.0:8200

  prometheus:
    image: prom/prometheus
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"

  aura-auth:
    build: ./aura-auth
    environment:
      SPRING PROFILES ACTIVE: docker
    depends_on:
      - postgres
      - redis
      - kafka
      - vault
    ports:
      - "8080:8080"

  risk-service:
    build: ./risk-service
    depends_on:
      - postgres
      - kafka
      - redis
    ports:
      - "8081:8081"

  realtime-service:
    build: ./realtime-service
    depends_on:
      - redis
      - kafka
    ports:
       - "8082:8082"
```

Рис. 3.31. Загальний docker-compose.yml

У цьому складі система може бути запущена командою docker compose up --build на локальній машині. Prometheus, Grafana та Vault працюють у тих самих мережах, забезпечуючи повноцінне середовище розробки.

У продакшн-середовищі кожен сервіс розгортається у Kubernetes-кластері, де забезпечується [22]:

- автоматичне масштабування;
- балансування навантаження через Ingress;
- зберігання конфігурацій через ConfigMap і Secret;
- самовідновлення при збоях (ReplicaSet + Liveness Probe).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aura-auth
spec:
  replicas: 3
  selector:
    matchLabels:
      app: aura-auth
  template:
    metadata:
      labels:
        app: aura-auth
    spec:
      containers:
        - name: aura-auth
          image: registry.local/aura-auth:latest
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
                name: aura-config
          livenessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
```

Рис. 3.32. aura-auth-deployment.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: aura-auth
spec:
  type: ClusterIP
  selector:
    app: aura-auth
  ports:
    - port: 8080
      targetPort: 8080
```

Рис. 3.33. aura-auth-service.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: aura-ingress
spec:
  rules:
    - host: aura.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: aura-auth
                port:
                  number: 8080
```

Рис. 3.34. aura-ingress.yaml

Використовується GitLab CI/CD pipeline, який автоматично збирає, тестує й деплоїть контейнери. CI/CD-процес забезпечує автоматизацію всіх етапів – від коміту до продакшену, з контролем версій та ізоляцією середовищ.

```yaml
stages:
  - build
  - deploy

build:
  stage: build
  image: maven:3.9-eclipse-temurin-21
  script:
    - mvn clean package -DskipTests
    - docker build -t registry.local/aura-auth:$CI_COMMIT_SHORT_SHA ./aura-auth
  only:
    - main

deploy:
  stage: deploy
  image: bitnami/kubectl:1.30
  script:
    - kubectl apply -f k8s/
  only:
    - main
```

Рис. 3.35. Реалізація CI/CD

3.9. Тестування системи

На етапі тестування була проведена перевірка працездатності основних компонентів системи та користувацьких сценаріїв, реалізованих у веб-інтерфейсі. Метою тестування було оцінити стабільність роботи системи в умовах високого навантаження, коректність механізмів автентифікації, реакцію на ризикові події, а

також виявити можливі розбіжності між очікуваними та фактичними результатами. В таблиці 3.2 представлено процес тестування системи.

Таблиця 3.2

Тестування системи

Тест-кейс	Опис тесту	Очікуваний результат	Фактичний результат
Реєстрація користувача через email	Заповнити форму реєстрації, підтвердити OTP-код із пошти, встановити пароль	Користувач перейшов до підключення 2FA за допомогою TOTP	Користувач перейшов до підключення 2FA за допомогою TOTP. OTP-код на підтвердження пошти доставляється із затримкою 3–5 сек
Увімкнення	Вписати логін/пароль, внести TOTP-код із застосунку	Користувач створений у базі даних, надсилається JWT-токен. Виконується вхід на головний дашборд	Користувач створений, JWT надісланий. Вхід на дашборд виконаний
Авторизація через OAuth2	Виконати вхід за допомогою облікового запису Google	Користувач перенаправлений на панель після успішної автентифікації. Після чого, якщо користувач є новим – результат аналогічний тест-кейсу з реєстрацією	Авторизація/реєстрація успішна, результат сходиться з очікуваним
Оцінка ризику входу	Увійти з нового пристрою, змінити часовий пояс	Рівень ризику = HIGH, застосувати step-up автентифікацію на віджети з чутливими даними	В деяких випадках зміна часового поясу призводить до рівня MEDIUM. Можливо, слід перерахувати ваги факторів ризику

Таблиця 3.2 (продовження)

Тестування системи

Тестування реального часу	Відкрити dashboard, здійснити логін з іншого браузера	Подія “new login” з’являється на панелі впродовж 1–2 сек	Подія з’являється через ~3 сек, часом дублюється
Довірити пристрій	Увімкнути опцію «Довіряти цьому пристрою»	Запис у списку Trusted	Запис у списку Trusted
Видалити пристрій	Видалити пристрій зі списку	Пристрій зник з переліку. Повторний вхід призведе до підвищення рівня ризку	Пристрій зник з переліку. Повторний вхід призвів до підвищення рівня ризку
Робота з Redis Cache	Зробити 5 послідовних логінів	Кеш оновлюється, latency < 10 мс	Кеш оновлюється, latency < 10 мс
Отримання секретів із Vault	Виконати авторизацію сервісу через AppRole	Система отримує токен доступу, без помилок	Тест успішний, конфігурації читаються коректно
Відновлення після збою (Failover)	Зупинити один із інстансів auth-service	Запит автоматично перенаправляється на резервний вузол	Перенаправлення працює

На рисунках 3.36-3.52 представлено скріншоти тестування користувацького інтерфейсу.

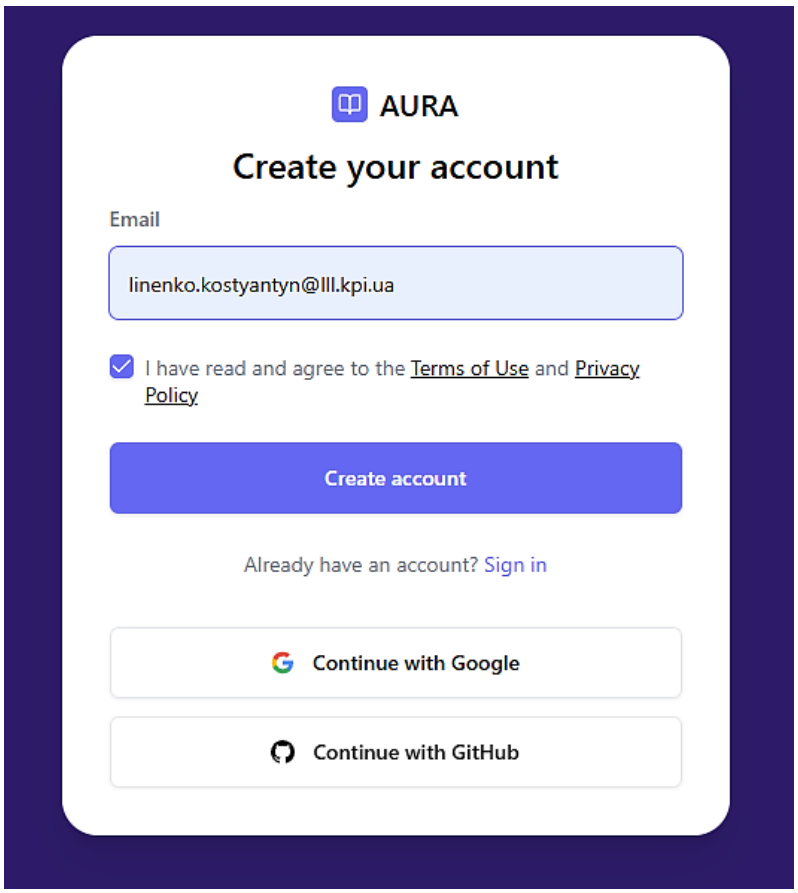

Рис. 3.36. Форма реєстрації через email

Користувач створює акаунт, вводячи свою електронну адресу для подальшої ідентифікації та підтвердження.

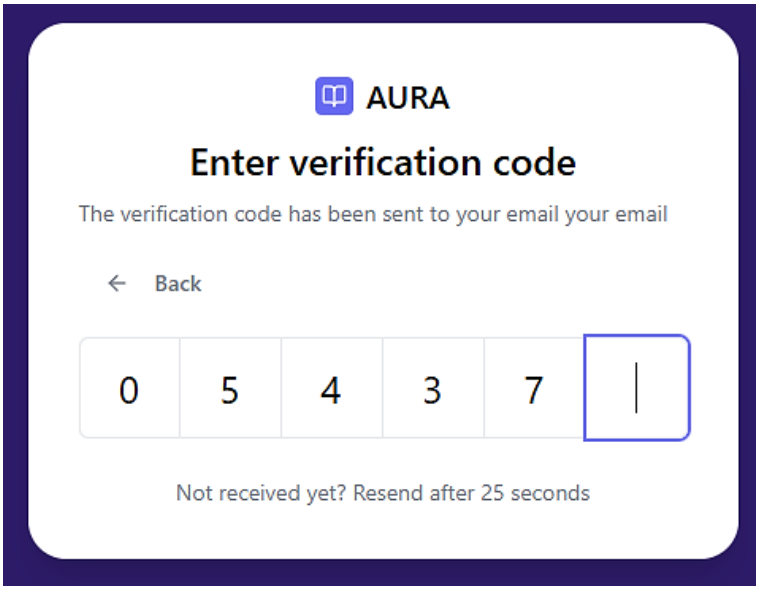

Рис. 3.37. Підтвердження email

На пошту надсилається secure-посилання з кодом підтвердження реєстрації. Код валідний 5 хвилин, генерувати новий код користувач може раз на хвилину.

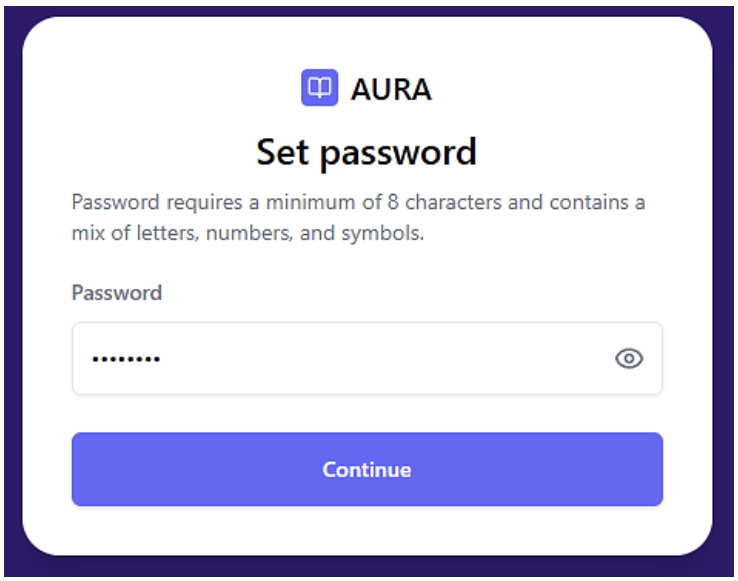

Рис. 3.38. Форма встановлення пароля

Після підтвердження email користувач задає свій основний пароль, який надалі застосовується для входу.

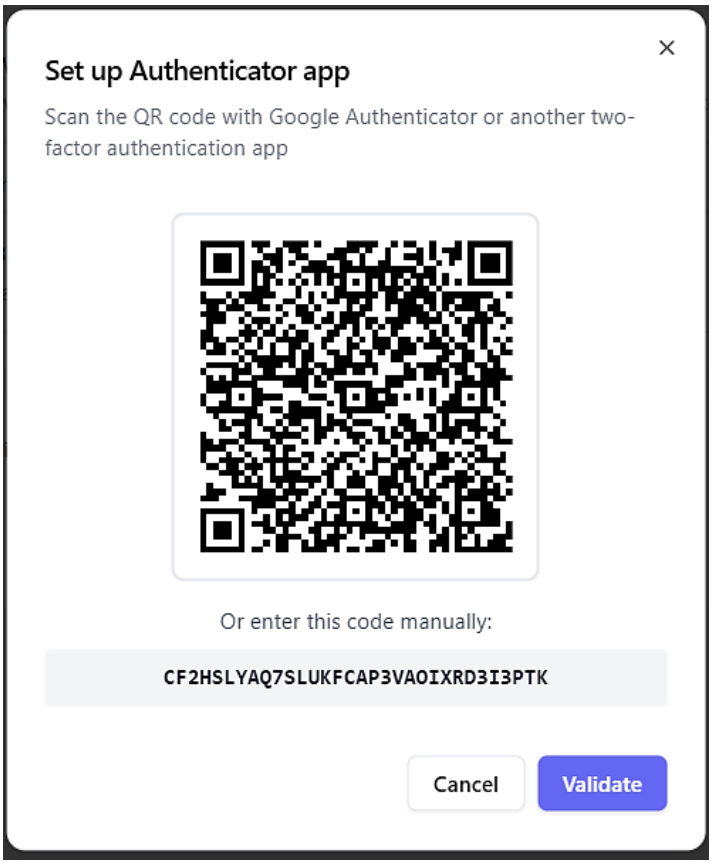

Рис. 3.39. QR-код із замаскованим секретом

Користувач активує двофакторну автентифікацію, скануючи QR-код у застосунку Google Authenticator або іншому TOTP-клієнті.

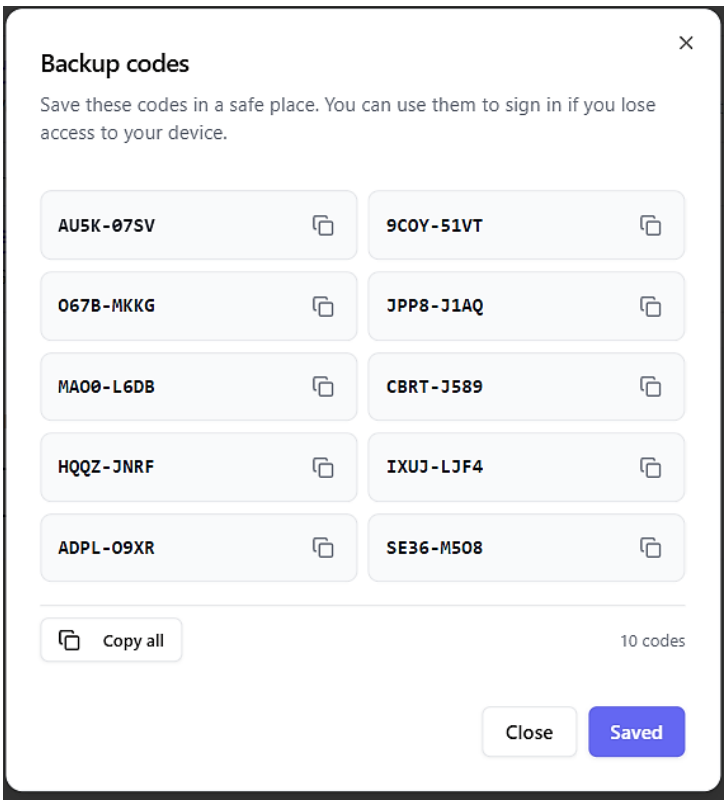

Рис. 3.40. Коди відновлення

Система може згенерувати набір резервних кодів, які дозволяють увійти в акаунт у разі втрати доступу до пристрою з TOTP. Ці коди є одноразовими.

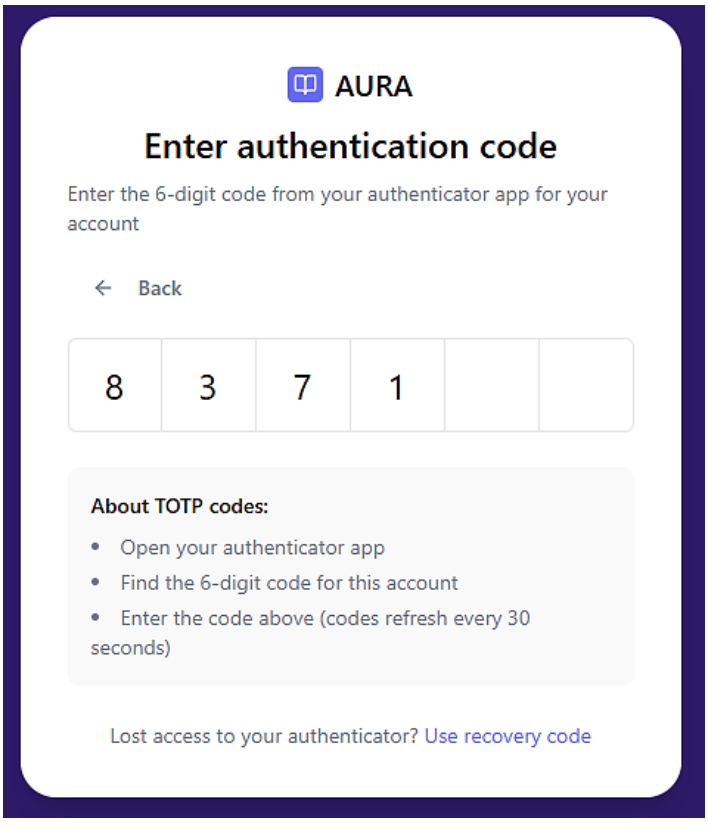

Рис. 3.41. Екран введення TOTP

Для завершення автентифікації користувач вводить тимчасовий одноразовий пароль із застосунку двофакторної автентифікації.

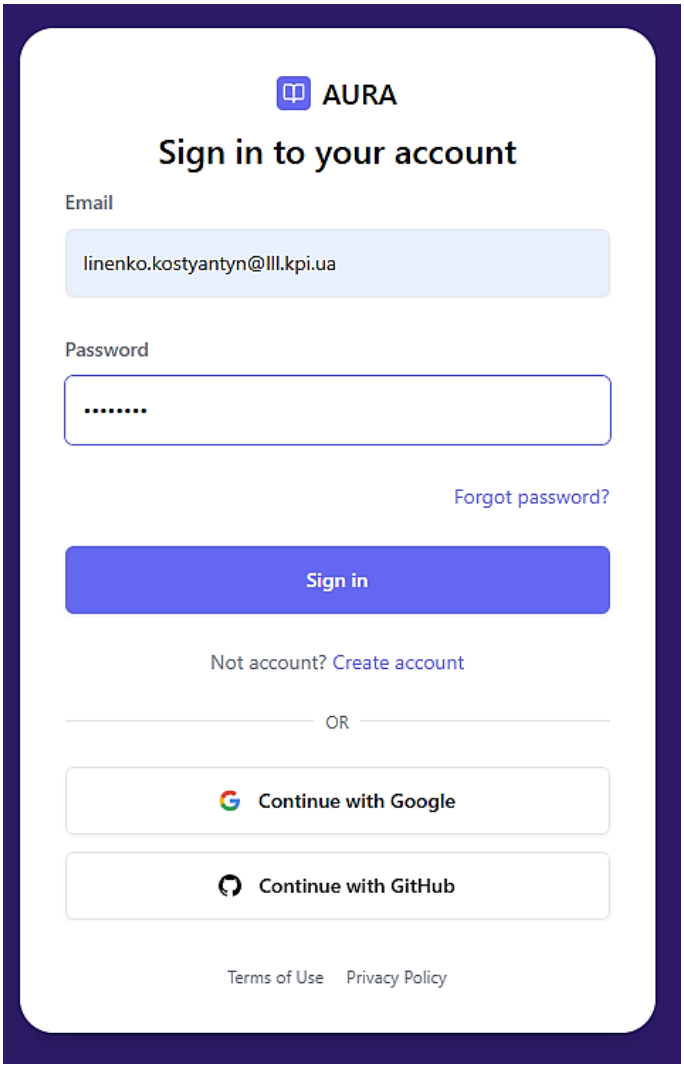

Рис. 3.42. Форма входу

Стандартна форма логіну: email + пароль, яка запускає подальші кроки адаптивної автентифікації залежно від налаштувань безпеки та рівня ризику.

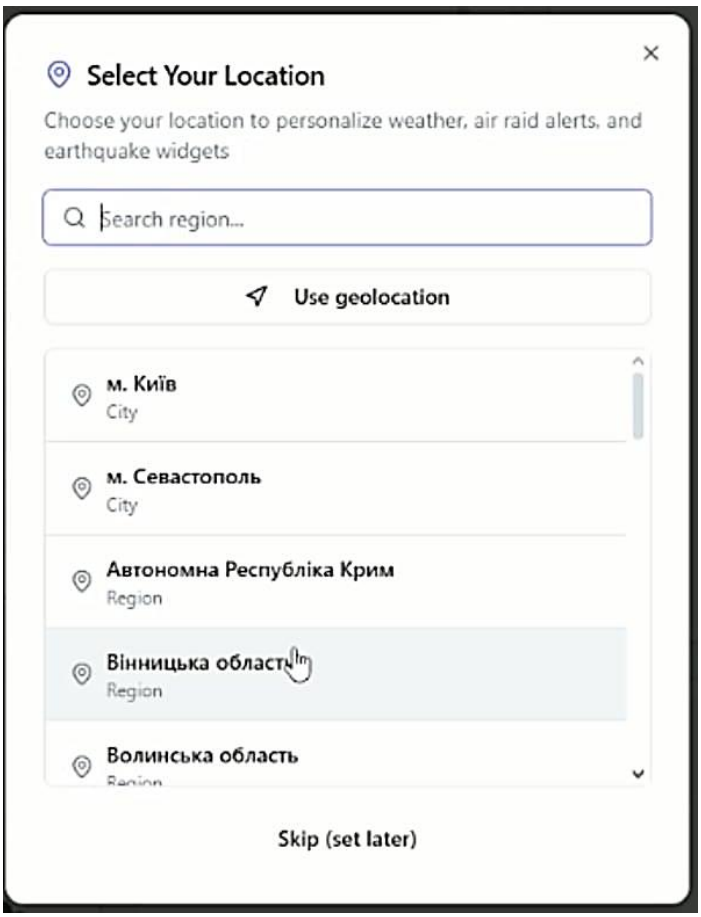

Рис. 3.43. Вибір геолокації після реєстрації

Після підтвердження акаунту з'являється діалог вибору локації, який визначає роботу віджетів, що залежать від геопозиції.

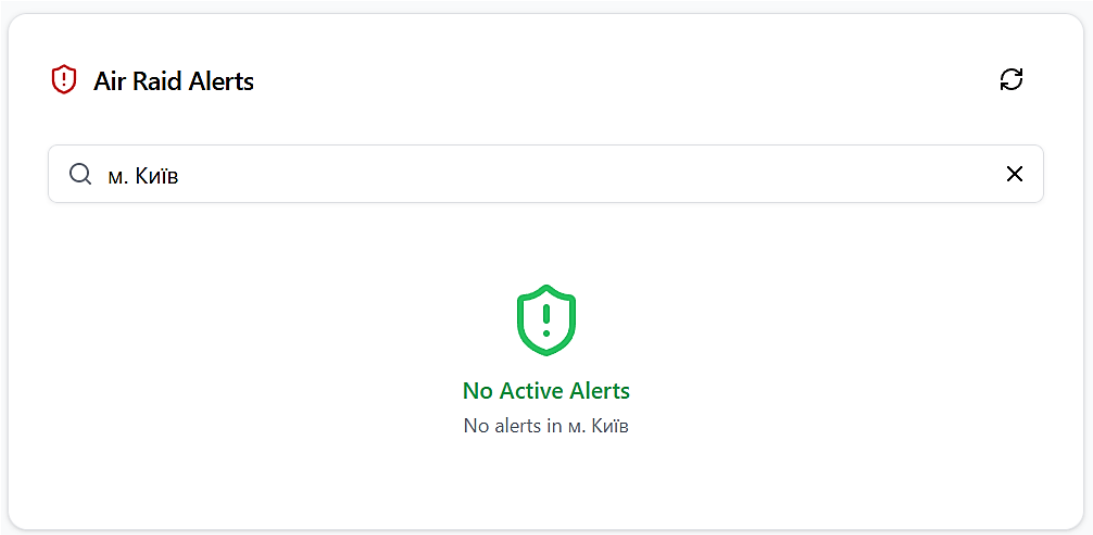

Рис. 3.44. Віджет повітряних тривог у поточному регіоні

Віджет показує активну інформацію про повітряну тривогу для обраної області або міста, отриману з alerts.in.ua.

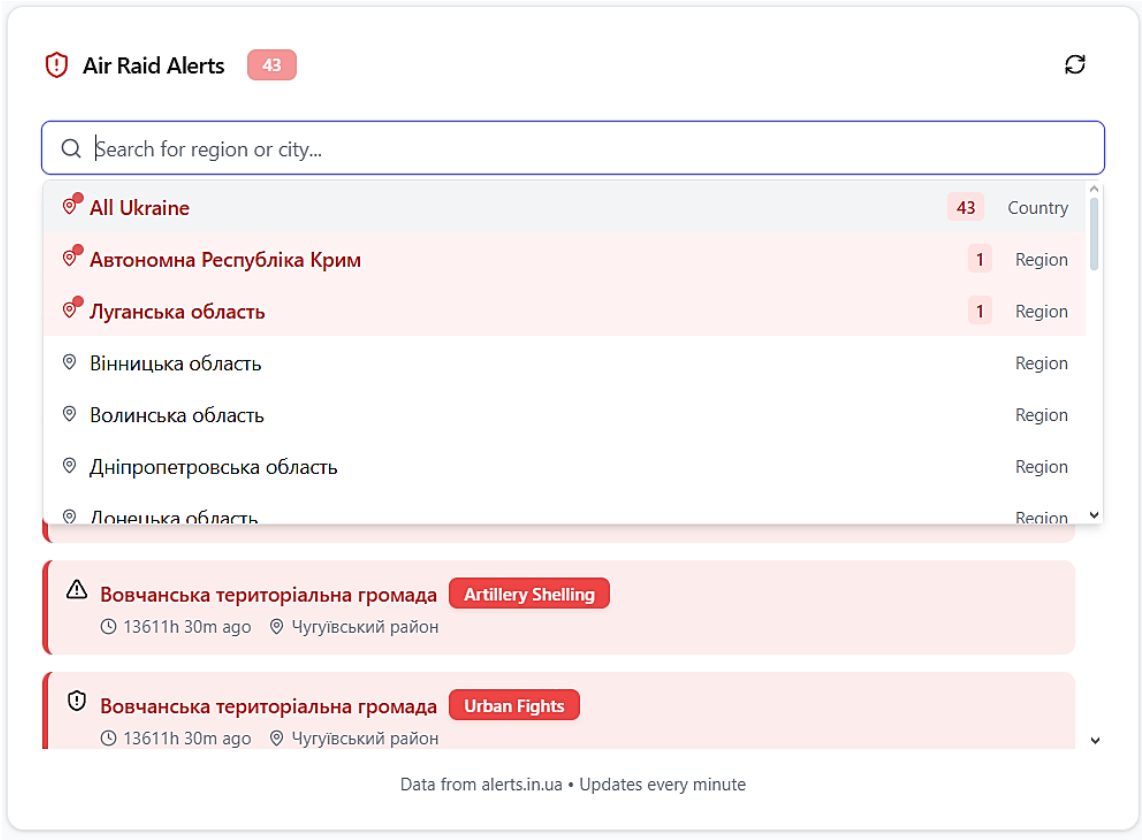

Рис. 3.45. Вибір регіонів для віджету повітряних тривог

Користувач може обрати іншу область, щоб переглядати стан тривог у будь-якому регіоні України.

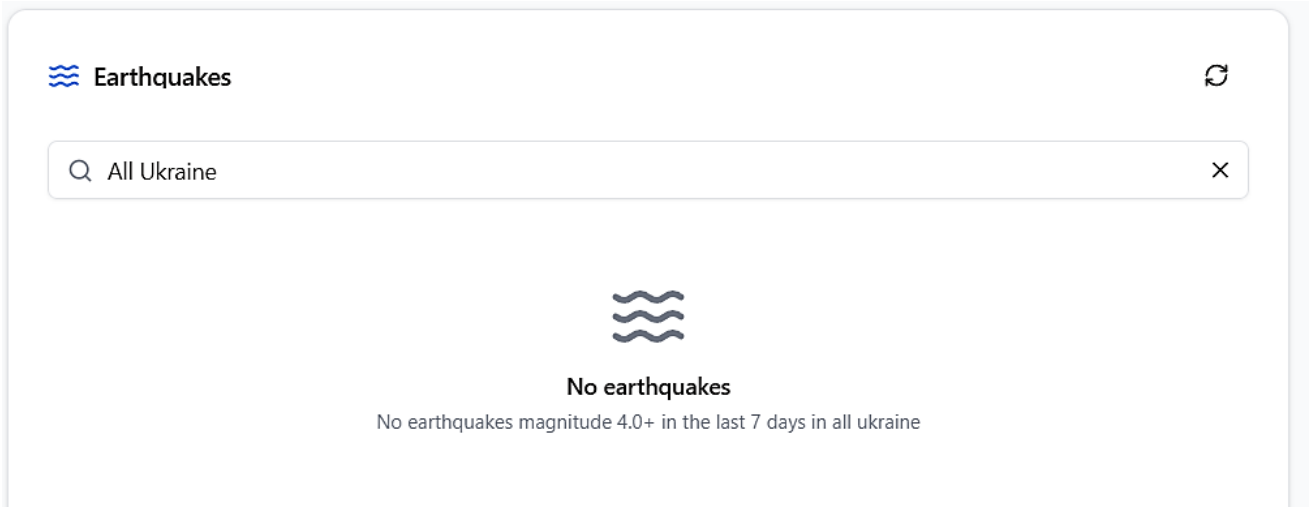

Рис. 3.46. Віджет землетрусів

Реальний-часовий віджет, що відображає останні зафіксовані землетруси у світі із вказанням інтенсивності.

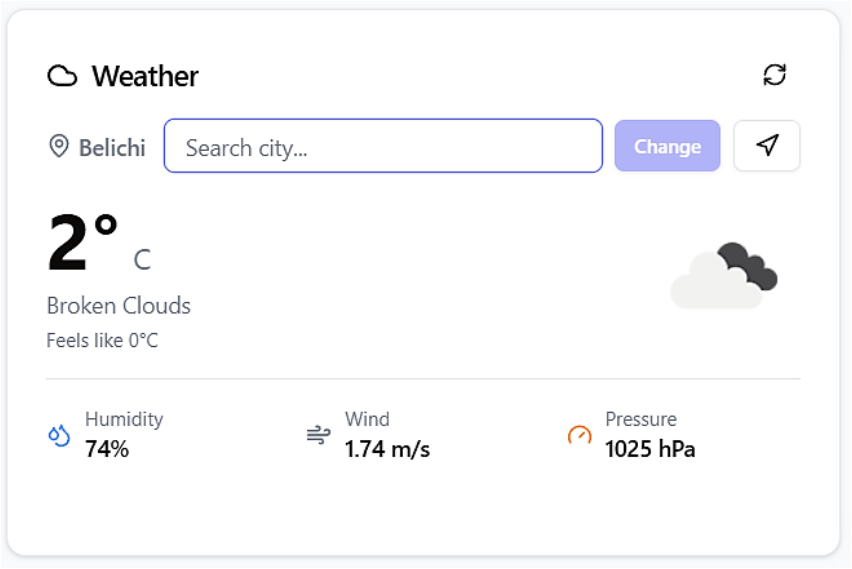

Рис. 3.47. Віджет погоди

Віджет показує поточну температуру, опади, вітер і тиск за обраною локацією.

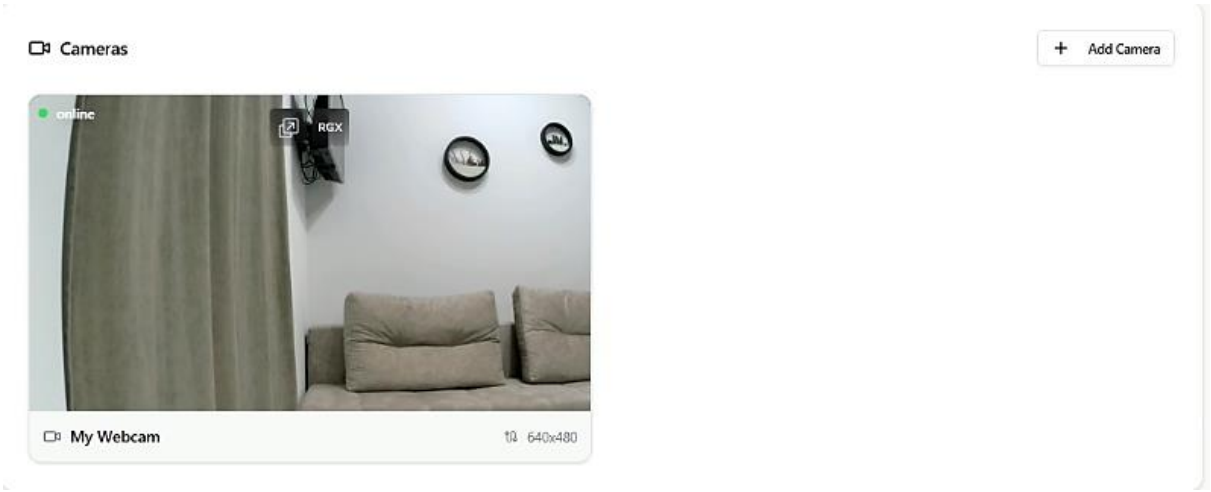

Рис. 3.48. Віджет камер відеоспостереження

Модуль відображає трансляції з різних камер, доступних у мережі, у режимі реального часу.

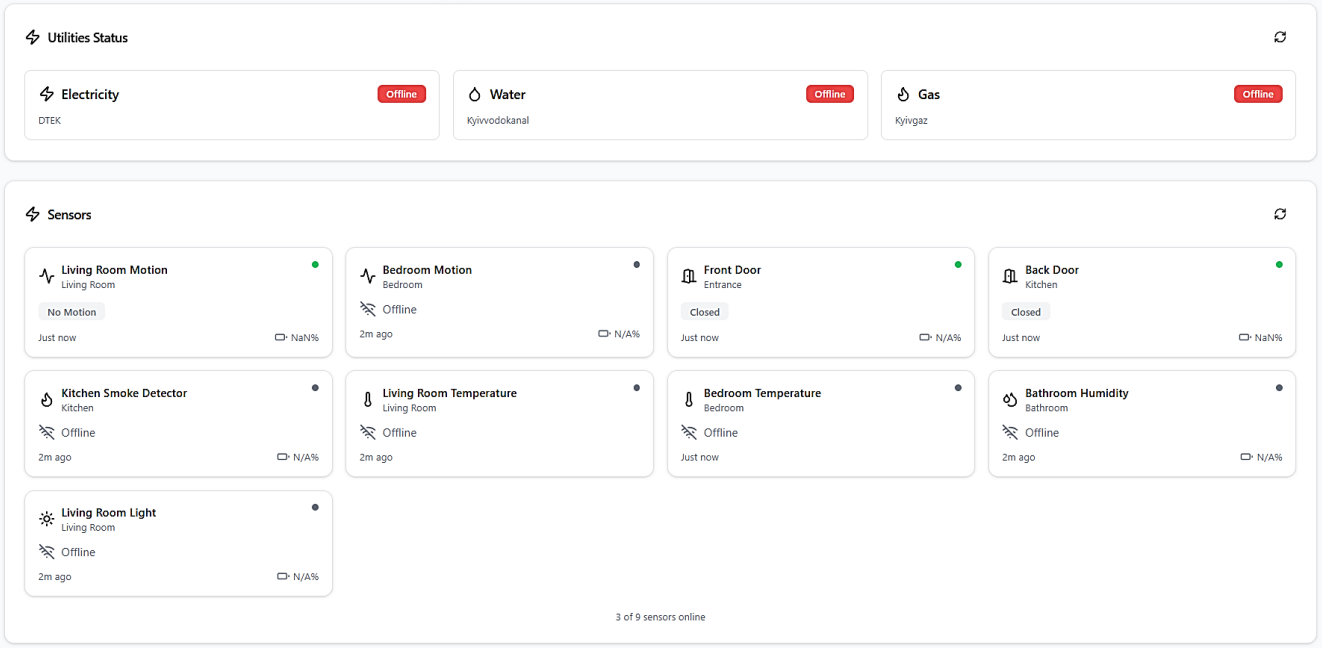

Рис. 3.49. Віджети електроенергії, газу, води та інших датчиків

Застосунок готовий приймати ці дані, але віджети залишаються неактивними через відсутність публічних API для підключення.

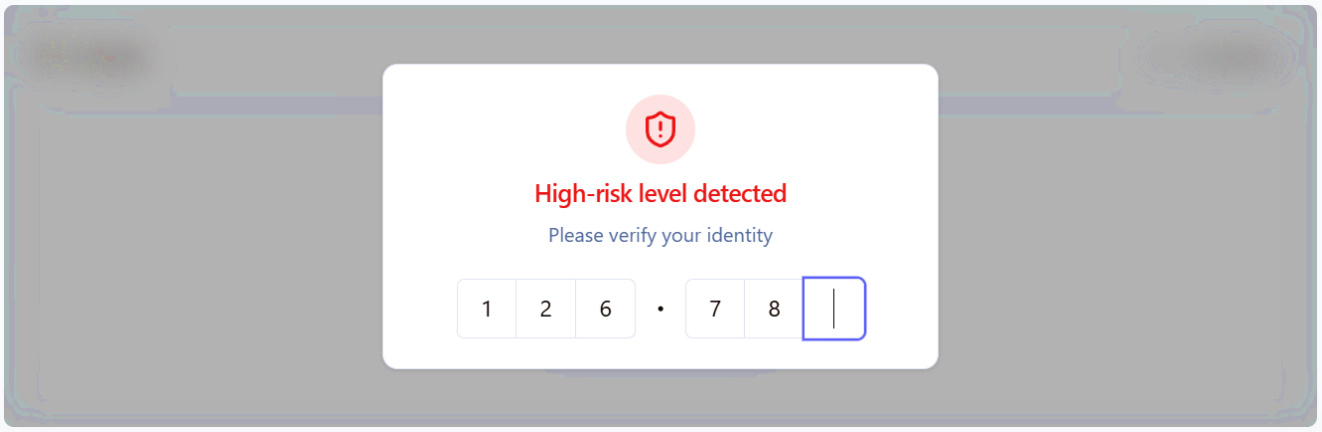

Рис. 3.50. Виявлення категорії ризику HIGH з блокуванням функціоналу

Під час використання системи тестовими користувачами було виявлено що найбільш вагомим фактором виявився вхід з нового девайсу, що недаремно, адже його абсолютна вага найбільш з-поміж інших – 30/100.

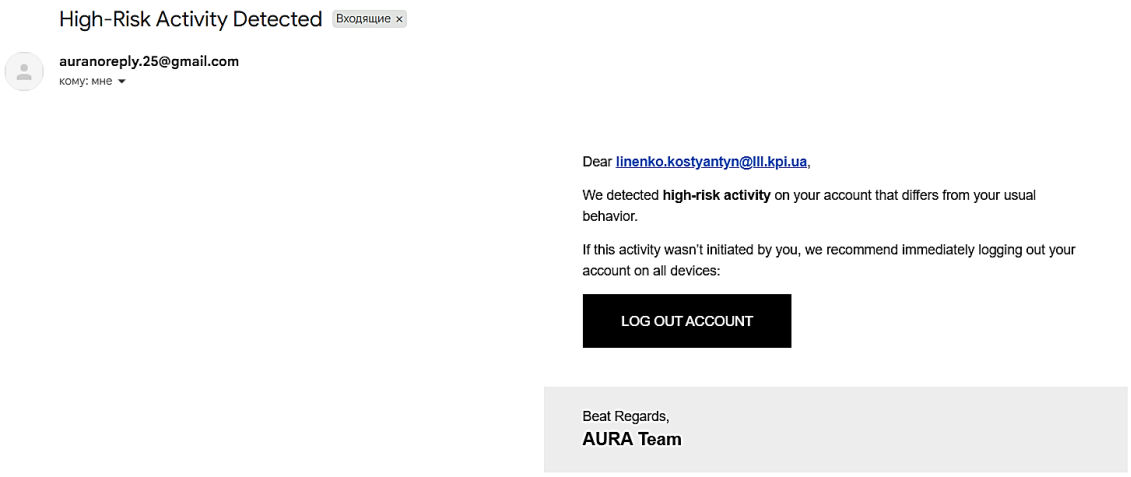

Рис. 3.51. Email-повідомлення користувачу про виявлення HIGH-ризикової активності

При виявленні HIGH-ризикової активності на пошту користувача приходить повідомлення з можливість відключити пристрій, на котрому спостерігається нетипова активність.

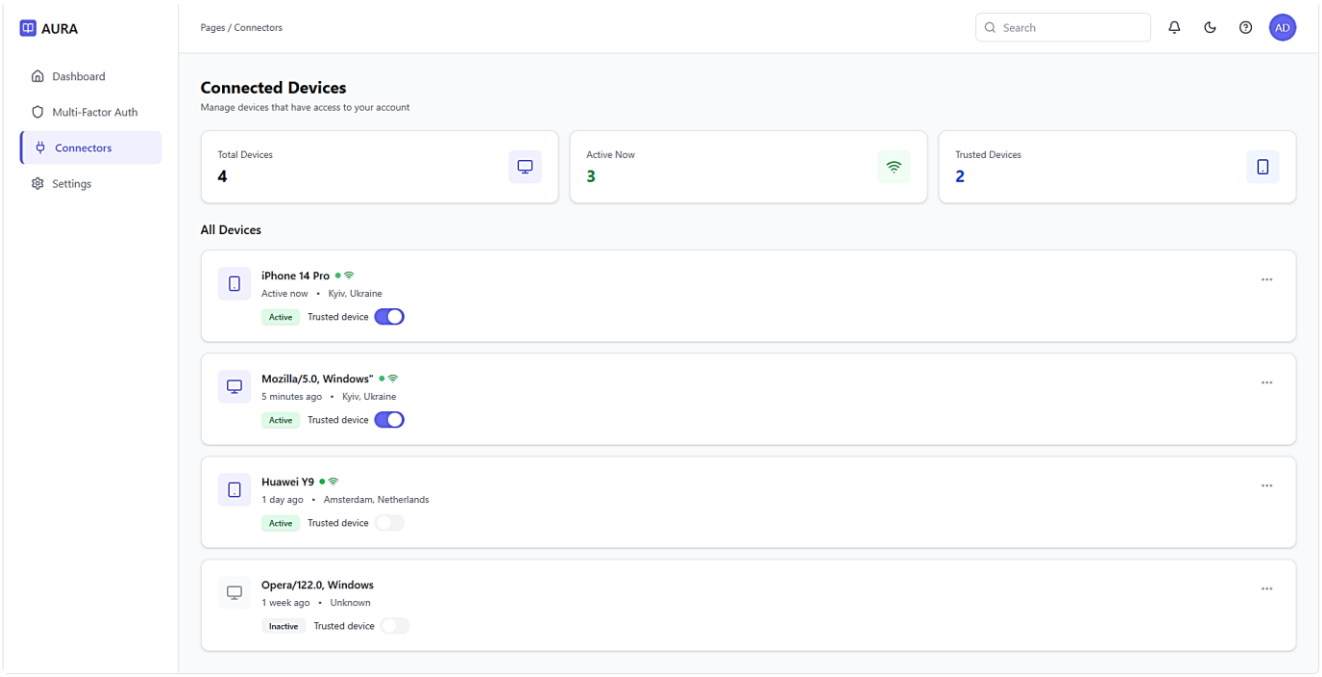

Рис. 3.52. Список підключених пристроїв

Є можливість керування підключеними до акаунту пристроями. На даний момент реалізована лише опція «Trusted device», що вимикає двохфакторну автентифікацію для цього пристрою – за умови, що механізм адаптивної автентифікації не виявить аномалій.

На рисунках 3.53-3.56 представлені графіки моніторингу системи в середовищі Grafana.

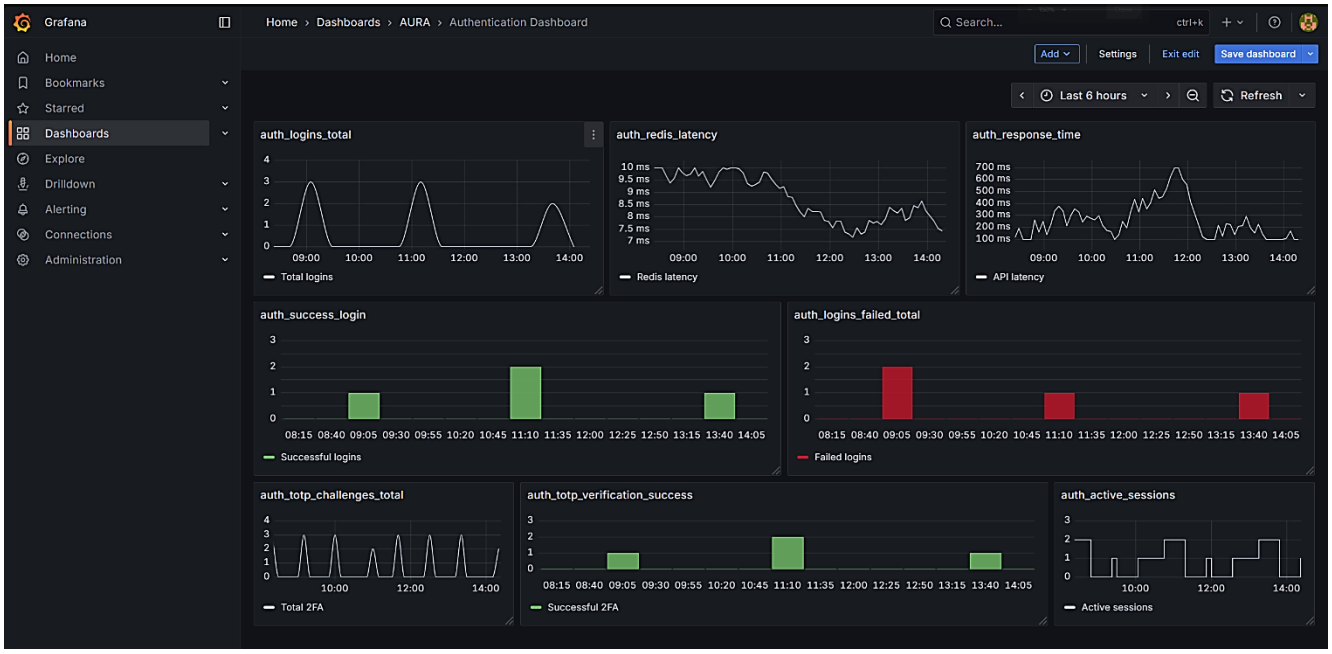

Рис. 3.53. Дашборд модулю автентифікації

Затримки Redis коливаються приблизно в межах 7-9,5 мс – це є нормальними значеннями для низького навантаження в локальному середовищі. Середній час відповіді API здебільшого 150-300 мс один помітний пік 600-700 мс.

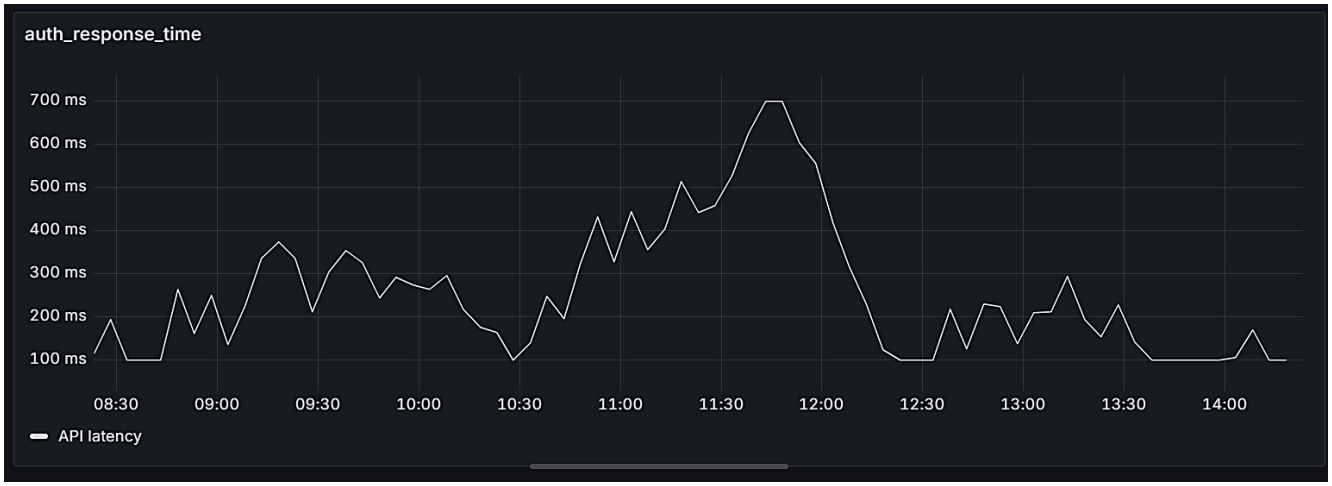

Рис. 3.54. Графік затримки відповіді API сервісу

Під час роботи було навмисно зупинено один контейнер Docker з auth-service, а балансувальник Nginx перенаправляв запити на резервний екземпляр. На графіку видно короткочасний сплеск затримки до ≈700ms під час перемикання, після чого стабілізувалася.

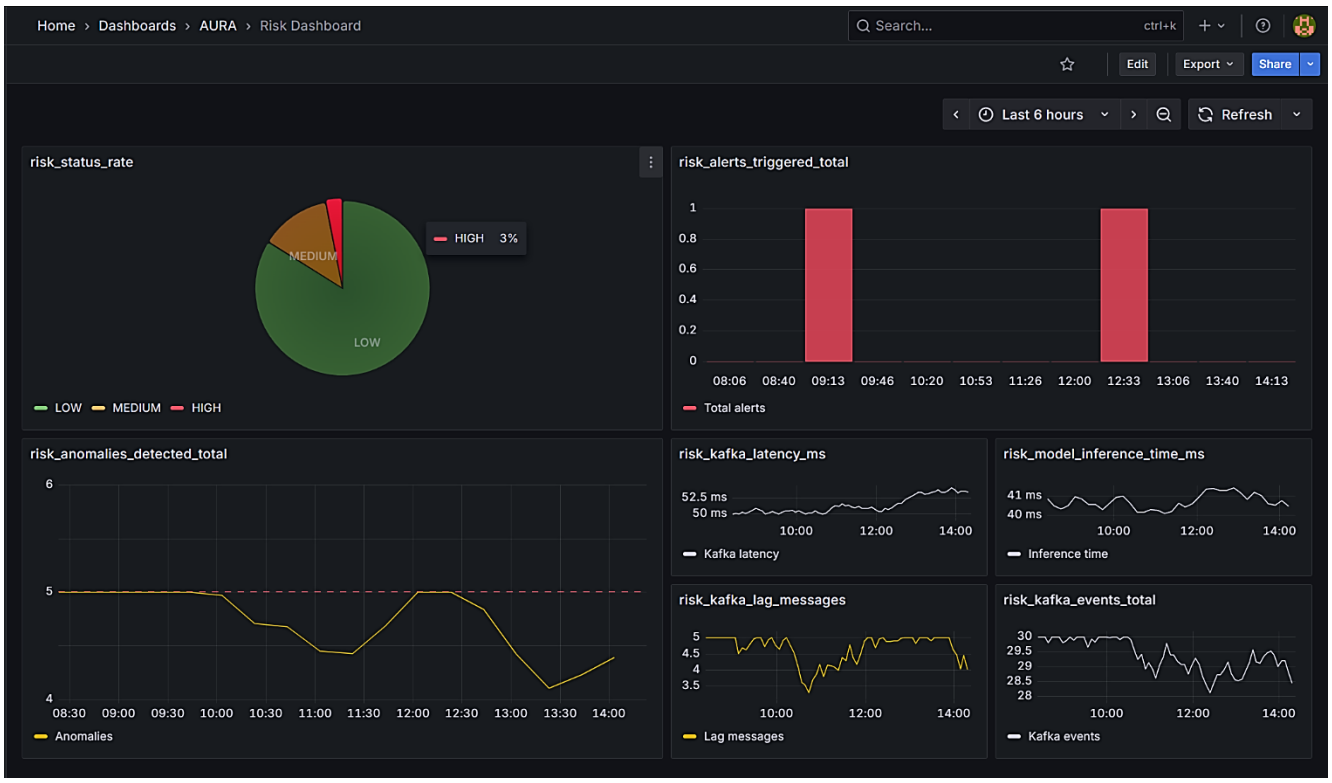

Рис. 3.55. Дашборд модулю оцінки ризику

Згідно з графіком risk_status_rate, більшість активних користувацьких сесій підпадали під категорію ризику LOW. Частки MEDIUM та HIGH відображають навмисні дії, щоб збільшити коефіцієнти ризикових факторів, зокрема: була спроба входу через VPN, зміна пристрою, та збільшена кількість невдалих спроб автентифікації.

Графік risk_alerts_triggered_total відображає фактичну кількість відправлених повідомлень на email про підозрілий вхід – ця кількість співпадає з кількістю обчислень алгоритмом коефіцієнтів ризику, коли категорія стала HIGH.

Графік risk_anomaloes_detected_total відображає кількість «спійманих» факторів ризику, коефіцієнти яких є більшими за середні показники поточних користувачів. Поріг виставлений як 5 для локального середовища.

Затримки Kafka коливаються в діапазоні 50-55 мс, без різких стрибків. Це нижче типового бюджету 100 мс, тому брокер повідомлень не є вузьким місцем у системі.

Рис. 3.56. Дашборд для моніторингу здоров’я системи

На графіку system_cpu_usage_percent видно піки навантаження: це може бути пов’язано піковими бізнес-подіями, такими, як батч-запити та переобчислення ризиків. У зв’язку з тим, що heap на графіку system_memory_heap_used_mb є досить вільним, можна подумати над кешуванням більшого набору даних задля покращення балансу CPU/RAM.

На графіку system_http_codes домінують 2xx коди, що свідчить про добре розроблене API, хоча іноді зустрічаються 4xx (проблеми клієнта) та 5xx (проблеми серверу) коди в поодиноких випадках. Як висновок, потрібно виконати максимально можливе покриття функціоналу тестами, особливо, у вироджених випадках.

Висновки до розділу 3

Було здійснено практичну реалізацію основних компонентів відмовостійкої системи реального часу AURA, яка поєднує в собі архітектурні принципи високої доступності та адаптивної автентифікації користувачів.

На основі попередньо спроєктованої архітектури розроблено набір мікросервісів, що взаємодіють через брокер повідомлень Kafka. Сервіс aura-auth відповідає за автентифікацію, авторизацію та керування токенами доступу, реалізує підтримку двофакторної автентифікації і інтеграцію з OAuth2-провайдерами. У ньому також передбачено механізм “step-up authentication”, який активується залежно від ризикового профілю користувача.

Сервіс aura-risk виконує обчислення ризиків у реальному часі на основі контекстних і поведінкових факторів. Розроблено алгоритм оцінки ризику, що класифікує рівень довіри і передає події далі у Kafka для реакції системи без затримки.

Застосунок інтегрує Redis як кеш для швидкого зберігання короткочасних даних сесій і контекстів ризику, а також Vault для безпечного збереження секретів і конфігурацій доступу.

Особливу увагу приділено відмовостійкості: конфігурація HA досягається за рахунок кластеризації брокера Kafka, балансування навантаження між екземплярами сервісів і механізмів автоматичного відновлення у випадку збою одного з вузлів.

Було також реалізовано клієнтський веб-інтерфейс для демонстрації потокового оновлення даних. Дашборди Grafana використовувалися для візуалізації ключових метрик у реальному часі.

У процесі тестування підтверджено стабільність роботи системи при відмові окремих компонентів. Це доводить, що розроблена архітектура відповідає вимогам систем реального часу та забезпечує необхідний рівень масштабованості й надійності.

РОЗДІЛ 4
РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

4.1. Опис ідеї проєкту

Проєкт «Відмовостійка система реального часу з адаптивною автентифікацією» пропонується для реалізації системи, що поєднує високу швидкість обробки даних із динамічним рівнем безпеки, який автоматично підлаштовується під поточну оцінку ризику користувацької активності. Основна інновація полягає у зберіганні оптимального рівня автентифікації без погіршення продуктивності та відмовостійкості системи.

Стартап орієнтований на створення універсальної архітектури, що може бути впроваджена у високонавантажених критичних середовищах – від транспортних і промислових систем до фінансових сервісів та державних інформаційних платформ.

Опис ідеї стартапу-проєкту наведений у таблиці 4.1.

Таблиця 4.1

Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Відмовостійка система реального часу з адаптивною автентифікацією	1) Транспорт і промисловість 2) Енергетика 3) Фінансово-банківський сектор 4) Державні інформаційні системи 5) Сервісі з мільйонами користувачів	Безперервність роботи навіть при збоях, зниження ризику несанкціонованого доступу, швидкість відгук системи

Аналіз потенційних технічно-економічних переваг ідеї порівняно із концепціями конкурентів наведено в таблиці 4.2.

Таблиця 4.2

Визначення сильних, слабких та нейтральних характеристик ідеї проєкту

№ п/п	Технічно-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів				W	N	S
		Мій проект	AWS IoT Core	Azure RTOS	Okta			
1	Відмостійкість	Балансування, реплікація даних	Резерування	Не призначений для масштабних веб-навантажень	Механізми відсутні		+	
2	Адаптивна автентифікація	Автентичний аналізатор контексту	Відсутня	Відсутня	Адаптивна MFA			+
3	Продуктивність у реальному часі	Висока	Висока	Висока	Низька		+	
4	Масштабованість	Висока	Висока, але ручне керування вузлами	Обмежене апаратними ресурсами	Середня		+	
5	Вартість впровадження	Висока на початку	Середня	Низька	Середня	+		

4.2. Технологічний аудит ідеї проєкту

У межах аудиту в таблиці 4.3 розглянуто ключові ідеї, підходи до їх реалізації, а також оцінено доступність відповідних технологій.

Таблиця 4.3

Технологічна здійсненність ідеї проєкту

№ п/п	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Модуль автентифікації та авторизації	Існуючі приклади реалізації підходу	Наявна	Так
2	Модуль оцінки ризиків	Статистичний аналіз, моделі класифікації	Необхідно розробити	Частково
3	Модуль обробки подій у реальному часі	Потокова обробка даних, черги повідомлень, асинхронна комунікація	Наявна	Так
4	Брокер повідомлень	Протоколи публікації/відписки	Наявна	Так
5	Сховище даних і кеш	(не-)реляційні БД	Наявна	Так
6	Система моніторингу та спостереження	Збір метрик, логування, візуалізація	Наявна	Так
7	Інфраструктура та оркестрація	Контейнеризація, CI/CD процеси	Наявна	Так
8	Програмний продукт, що включає в себе всі розроблені модулі	Програмні засоби розробки та описані вище підходи	Необхідно розробити	Частково
Обрані технології реалізації ідеї продукту: Java, Spring Boot/Webflux/Security, React, Apache Kafka, PostgreSQL, Redis, HashiCorp Vault, Prometheus, Grafana, Docker, Kubernetes, Gitlab CI/CD				

Можна зробити висновок, що реалізація цієї ідеї можлива і включає в себе розробку нових модулів та використання сучасних технологій.

4.3. Аналіз ринкових можливостей запуску стартап-проєкту

Метою аналізу ринкових можливостей є оцінка поточного стану галузі, визначення потенціалу розвитку, рівня конкуренції та перспективи комерціалізації проєкту. Ринковий аналіз дозволяє з'ясувати, наскільки ідея стартапу відповідає сучасним потребам користувачів і чи є економічно доцільною її реалізація.

У таблиці 4.4 відображено попередню характеристику потенційного ринку стартап-проекту.

Таблиця 4.4

Попередня характеристика потенційного ринку стартап-проєкту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців	4
2	Загальний обсяг продаж, грн/ум.од	≥ 50 млрд $/рік (у світі)
3	Динаміка ринку (якісна оцінка)	Стабільна з тенденцією до зростання 10-15%/рік
4	Наявність обмежень для входу (вказати характер обмежень)	Відсутні
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	20-30%

Ринок відмовостійких систем та рішень для автентифікації є стабільним і привабливим для виходу стартап-проєкту. Конкуренція помірна, але більшість гравців зосереджені на окремих аспектах — або відмовостійкості, або безпеці. Це відкриває нішу для інтегрованих рішень, які поєднують обидва фактори. Відсутність серйозних регуляторних обмежень, відносно висока рентабельність та

зростаюча потреба у таких системах формують сприятливі умови для комерціалізації розробленої технології.

У таблиці 4.5 наводяться визначені потенційні клієнти, а також їх характеристика.

Таблиця 4.5

Характеристика потенційних клієнтів стартап-проєкту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних груп клієнтів	Вимоги споживачів до товару
1	Забезпечення безперервної роботи критичних систем	Державні установи, оборонний сектор, енергетичні компанії	Орієнтовані на довгострокову стабільність, надійність та сертифікацію безпеки; закупівлі через державні тендери	Висока відмовостійкість, аудит дій користувачів, підтримка стандартів безпеки
2	Підвищення рівня безпеки користувацьких доступів	Банківські та фінансові установи, страхові компанії	Орієнтовані на мінімізацію ризику шахрайства та адаптивну автентифікацію	Підтримка багатофакторної автентифікації, швидкий відгук системи
3	Моніторинг і реакція на події в реальному часі	Транспортні компанії, системи «розумного міста», служби реагування	Потребують масштабованих рішень з низькою затримкою і високою швидкістю обробки подій	Реакція системи <100 мс, безперервна доступність, гнучка система сповіщень
4	Гнучка інтеграція у бізнес-процеси	Платформи з мільйонами користувачів	Орієнтовані на масштабованість, UX і швидкість	API-доступ, низька затримка

Потенційна клієнтська база стартапу є широкою та включає як державний, так і приватний сектори. Найбільш перспективними сегментами є фінансові організації, енергетичні компанії та державні структури, для яких критично важливі надійність і безпека системи. Ключовим фактором успіху є адаптація продукту під специфічні вимоги кожного сегмента, забезпечення високого рівня відмовостійкості, гнучкої автентифікації та простоти інтеграції з існуючою інфраструктурою.

У таблиці 4.6 наведені фактори загроз, що перешкоджають ринковому впровадженню.

Таблиця 4.6

Фактори загроз

№ п/п	Фактор	Зміг загрози	Можлива реакція компанії
1	Технологічні зміни	Поява нових технологій або стандартів, що роблять існуючі рішення менш ефективними або застарілими	Постійний моніторинг тенденцій, оновлення архітектури, використання модульного підходу для швидкої адаптації
2	Конкуренція	Вихід на ринок сильних гравців із подібними рішеннями, які мають більші ресурси	Формування унікальних конкурентних переваг, активне просування через партнерські програми
3	Кіберзагрози	Атаки на систему з метою компрометації даних або порушення стабільності роботи	Впровадження багаторівневого захисту, постійне тестування на вразливості, аудит безпеки
4	Нестабільність ринку	Зміна попиту через економічну ситуацію або скорочення фінансування цифрової трансформації	Диверсифікація ринкових сегментів, орієнтація на критичну інфраструктуру, урядові замовлення
5	Кадрові ризики	Нестача висококваліфікованих фахівців у сфері DevSecOps і кібербезпеки	Інвестування у навчання персоналу, співпраця з технічними університетами

У таблиці 4.7 наведені фактори можливостей, що сприяють ринковому впровадженню.

Таблиця 4.7

Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання попиту на безпечні системи	Підвищення уваги до кібербезпеки через часті кібератаки на бізнес і державні установи	Активне позиціонування продукту як рішення з підвищеною безпекою та відмовостійкістю
2	Розвиток хмарних технологій	Масовий перехід компаній до гібридних та мультихмарних інфраструктур	Інтеграція продукту з популярними хмарними платформами (AWS, Azure, GCP), забезпечення сумісності
3	Підтримка державних і міжнародних програм цифровізації	Державні ініціативи з підтримки інноваційних IT-рішень	Участь у грантових програмах, публічних тендерах, отримання статусу «національного IT-рішення»
4	Зростання ринку рішень у реальному часі	Попит на системи, що забезпечують обробку даних без затримок, зростає	Розширення функціональності для різних галузей, розробка SDK/API для інтеграції
5	Поширення DevOps та автоматизації	Компанії шукають інструменти, що легко інтегруються у CI/CD процеси	Оптимізація архітектури для безшовної інтеграції

У таблиці 4.8 наведено загальні риси конкуренції на ринку.

Таблиця 4.8

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції - олігополія з елементами монополістичної конкуренції	На ринку домінують великі гравці, проте існують нішеві можливості для інноваційних рішень	Створення унікальної ціннісної пропозиції, орієнтація на вузькі ринкові сегменти маркетингові кампанії на ключові сегменти
За рівнем конкурентної боротьби - середня до високої	Великі компанії мають значні ресурси, але інноваційні стартапи можуть швидко завоювати ніш	Інвестування у R&D, швидке тестування MVP, гнучке позиціонування продукту, маркетингові кампанії на ключові сегменти
За галузевою ознакою - IT, кібербезпека, хмарні сервіси	Активно розвиваються напрямки обробки подій у реальному часі та безпеки доступу	Фокус на інтеграції з галузевими стандартами, дотримання норм безпеки та відмовостійкості, сертифікація продукту
Конкуренція за видами товарів - пошукові системи, сервіси автентифікації	Стартапи можуть пропонувати комплексні рішення, що поєднують функціонал кількох продуктів	Об’єднання функцій у комплексне рішення, виділення унікальних модулів за швидкістю

Таблиця 4.8 (продовження)

Ступеневий аналіз конкуренції на ринку

За характером конкурентних переваг - великі гравці мають масштабованість і фінансові ресурси, стартапи – інноваційність та гнучкість	Основна боротьба за клієнта ведеться через цінність, функціонал і унікальні можливості продукту	Акцент на унікальні особливості системи: адаптивна автентифікація, висока відмовостійкість, модульна архітектура
За інтенсивністю - середня	Зростає із появою нових технологій і підвищенням потреб у безпечних реальних системах	Постійний моніторинг ринку, аналіз конкурентних пропозицій, швидка адаптація стратегії, партнерство з ключовими клієнтами

У таблиці 4.9 наведено більш детальні риси конкуренції на ринку за М. Портером.

Таблиця 4.9

Аналіз конкуренції за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари замінники
	1	2	3	4	5
	AWS IoT Core, Azure RTOS, Okta	Нові стартапи у сфері кібербезпеки та IoT, які можуть розробити аналогічні рішення	Постачальники хмарних платформ (AWS, Azure, GCP), постачальники апаратного забезпечення для обробки подій	Державні структури, банки, енергетика, транспорт, великі IT-компанії	Існуючі платформи автентифікації та обробки даних у реальному часі (частково покривають потребу Keycloak, Firebase Auth тощо)
Висновки	Конкуренція висока; необхідно виділятися унікальною комбінацією відмовостійкості та адаптивної автентифікації	Слід швидко виводити MVP на ринок, створювати бар'єри для входу	Потреба у надійних партнерських угодах і диверсифікації постачальників, щоб зменшити залежність	Орієнтація на потреби клієнта, адаптація продукту під різні галузі	Необхідність позиціонування продукту як більш комплексного та надійного рішення, що поєднує безпеку і відмовостійкість

На основі вище наведених таблиць виведено фактори конкурентоспроможності, та наведено у таблиці 4.10.

Таблиця 4.10

Обгрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обгрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проєктів значущим)
1	Інновації	Використання новітніх технологій для адаптивної автентифікації та обробки подій у реальному часі дозволяє виділити продукт серед конкурентів та задовольняє зростаючий попит на високотехнологічні рішення.
2	Цінова політика	Гнучкі моделі ліцензування та SaaS-підхід роблять продукт доступним для різних сегментів ринку – від стартапів до великих корпорацій, забезпечуючи конкурентну перевагу у вартості та співвідношенні ціна/якість.
3	Можливість використання системи для створення власних продуктів	Модульна архітектура та API дозволяють клієнтам будувати на базі системи власні рішення та сервіси, що збільшує привабливість для корпоративних та державних користувачів.
4	Інтеграція з існуючими системами	Підтримка стандартних протоколів та сумісність із популярними платформами зменшує бар'єри впровадження та скорочує час інтеграції, що важливо для клієнтів із розвиненою IT-інфраструктурою.
5	Безпека та відповідність стандартам	Дотримання міжнародних стандартів безпеки (ISO/IEC 27001, NIST, GDPR) підвищує довіру користувачів, дозволяє працювати у регульованих сферах і є ключовим фактором для залучення державних та фінансових клієнтів.

Порівняльний аналіз сильних та слабких сторін системи збору та агрегації даних для аналізу вакансій відображено у таблиці 4.11.

Таблиця 4.11

Порівняльний аналіз сильних та слабких сторін модуля аналітичного опрацювання текстової інформації

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з системою збору та агрегації даних для аналізу вакансій						
			-3	-2	-1	0	+1	+2	+3
1	Інновації	20			+				
2	Цінова політика	15		+					
3	Можливість використання системи для створення власних продуктів	20			+				
4	Адаптація системи до різних ринків	15					+		

Враховуючи вище наведені таблиці зробимо висновок ринкового аналізу, та сформуємо його у вигляді SWOT- аналізу (таблиця 4.12).

Таблиця 4.12

SWOT-аналіз стартап-проєкту

Сильні сторони	Слабкі сторони
Інноваційне поєднання відмовостійкої архітектури з адаптивною (ризик-базованою) автентифікацією, що підвищує безпеку без погіршення UX	Висока складність реалізації та налаштування компонентів системи (risk-engine, failover, брокер подій)
Можливості	Загрози
Використання у критичних галузях – фінтех, e-gov, телемедицина	Кібератаки, експлуатація вразливостей або збої інфраструктури можуть знизити ефективність системи

На основі SWOT-аналізу визначено альтернативи ринкового впровадження стартап-проекту у таблиці 4.13.

Таблиця 4.13

Альтернативи ринкового впровадження стартап-проєкту

№ п/п	Альтернатива (орієнтований комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Розробка системи з основним функціональним забезпеченням	Висока	5 місяців
2	Пошук замовників системи	Середня	6 місяців
3	Покращення основних можливостей системи на основі відгуків користувачів	Висока	7 місяців

Обрано першу альтернативу, тому що отримання ресурсів є більш простим та ймовірним, а строки реалізації – більш стислими.

4.4. Розроблення ринкової стратегії проєкту

Для визначення ринкової стратегії потрібно спочатку визначити стратегії охоплення ринку у таблиці 4.14.

Таблиця 4.14

Вибір цільових потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтований попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу в сегмент
1	Fintech	Висока	Висока	Висока	Низька
2	E-commerce	Середня	Висока	Середня	Середня
3	E-government	Висока	Середня	Середня	Низька
4	E-health	Середня	Середня	Низька	Середня
5	SaaS	Висока	Середня	Середня	Висока
6	Cloud services	Висока	Висока	Висока	Середня
Які цільові групи обрано: фінтех-сектор як найбільш перспективний напрям, що максимально продемонструє практичну користь проєкту					

При виборі одного сегменту використаємо стратегію диференційованого маркетингу та визначимо її базову стратегію розвитку (таблиця 4.15).

Таблиця 4.15

Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Розробка системи з основним функціональним забезпеченням	Диференційований маркетинг	Універсальність системи. Можливість використання розробленої системи як основи для створення власних продуктів	Стратегія диференціації

У таблиці 4.16 описано стратегію базової конкурентної поведінки.

Таблиця 4.16

Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проєкт «першопроходьцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки
1	Частково	Переважно шукати нових споживачів	Так, стандартні функції автентифікації	Інноваційна диференціація

Враховуючи описані вище стратегій визначимо стратегію позиціонування продукту (таблиця 4.17).

Таблиця 4.17

Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію власного проєкту
1	Універсальність системи	Інноваційна диференціація	Сучасні патерни побудова системи та її документуванн	Масштабованість, відмовостійкість, безперебійність, висока безпека
2	Можливість використання системи як основи для створення власних продуктів	Розширення клієнтської бази	Open-source проєкт з API щодо підключення власних сервісів-джерел інформації	

4.5. Розроблення маркетингової програми стартап-проєкту

Для формування маркетингової концепції продукту, спершу треба визначити результати аналізу конкурентоспроможності продукту (таблиця 4.18).

Таблиця 4.18

Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Універсальність системи	Забезпечує можливість адаптації до різних сфер використання (фінтех, e-commerce, держсектор) без зміни базової архітектури.	Гнучка модульна структура; простота інтеграції; підтримка кількох сценаріїв автентифікації.
2	Можливість використання системи для створення власних продуктів	Дає змогу клієнтам на базі системи швидко розробляти та впроваджувати власні рішення без значних витрат часу й ресурсів.	Відкрите API; масштабована архітектура; можливість кастомізації під конкретні потреби клієнта.

Певну популярність отримала модель маркетингу, що складається з трьох рівнів:

— споживачі;

— інструменти маркетингу;

— політика комунікацій.

У таблиці 4.19 наведена трирівнева маркетингова модель товару.

Таблиця 4.19

Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
Товар за задумом	Відмовостійка система реального часу з адаптивною автентифікацією		
Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Швидкодія Ефективність Користувацький інтерфейс	Нм Нм Нм	Тх/Тл/Е Тх/Тл Е
	Якість: реалізований веб-сервіс та модулі автентифікації/авторизації, оцінки ризику, обробки подій в реальному часі		
	Пакування: Docker-контейнери, мікросервісна архітектура, API-документація		
	Марка: SmartAuth RT		
Товар із підкріпленням	Програмний продукт		
	Програмний продукт		

Вартість на рекламу у товарі визначається типом реклами кількістю її відображення. У таблиці 4.20 наведено ціни відповідно до кількості реклами.

Таблиця 4.20

Визначення меж встановлення ціни

№ п/п	Рівень цін на товари замінники, тис. грн/програмний продукт	Рівень цін на товари аналоги, тис. грн/програмний продукт	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	-	20-35 тис.грн/міс	Середній-високий	Нижня межа: 15 тис.грн/міс Верхня межа: 40 тис.грн/мис

У таблиці 4.21 наведена оптимальна система збуту.

Таблиця 4.21

Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Оплата рекламної кампанії	-	До приватних підприємств, яким необхідне просування власних вакансій у системі	Веб-сайт, а також рекомендації між споживачами

У таблиці 4.22 наведено концепцію маркетингових комунікацій.

Таблиця 4.22

Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Цільові клієнти приймають рішення раціонально, орієнтуючись на надійність, безпеку та окупність інвестицій. Вони очікують чітких технічних доказів ефективності, сертифікацій	Професійні конференції та IT-форуми, тематичні онлайн-спільноти, технічні блоги, PR-статті у профільних виданнях, targeted реклама у B2B-сегменті.	Гнучкість інтеграції та масштабування, інтелектуальна адаптивна автентифікація, прозорий моніторинг стану системи	Підкреслити унікальність системи, сформувати довіру через експертизу та демонстрацію реальних сценаріїв, заохотити до тестування прототипу	Безпека без компромісів і затримок

Висновки до розділу 4

У цьому розділі була визначена актуальність стартапу, що розробляється, окреслені основні конкуренти та підтверджена технологічна здійсненність проекту.

Було проведено детальний аналіз ринкових можливостей запуску стартап-проекту та розроблена маркетингова стратегія, включно з позиціонуванням, комунікаційною політикою та визначенням меж встановлення ціни.

На даний момент проект має конкурентів, проте вони не володіють усіма функціями та можливостями, що реалізовані у даній системі. Універсальність системи та можливість використання її як основи для створення власних продуктів виступають ключовими характеристиками конкурентоспроможності продукту.

Реалізація продукту є доцільною, оскільки доведена його технологічна здійсненність, відмовостійкість і рентабельність.

Отже, можна зробити висновок, що стартап має високий потенціал успіху за умови запуску продукту з основними функціями та мінімальною вартістю для виходу на ринок, а також ефективного використання маркетингових комунікацій для залучення цільової аудиторії.

ВИСНОВКИ

Магістерська дисертація мала на меті створення відмовостійкої та безпечної системи для обробки даних у реальному часі, що здатна ефективно обслуговувати тисячі користувачів у поточний момент часу, забезпечувати адаптивну автентифікацію та швидку реакцію на події. Основні цілі включали розробку масштабованої мікросервісної архітектури, впровадження механізмів гарантії стабільної роботи при пікових навантаженнях або виходу з ладу деяких вузлів системи.

Архітектура системи включає кілька ключових компонентів, що інтегруються між собою. API Gateway забезпечує єдину точку входу, маршрутизацію запитів, балансування навантаження та підтримку WebSocket/SSE протоколів. Сервіс автентифікації реалізує stateless-підхід через JWT-токени, двохфакторну автентифікацію (TOTP) і step-up MFA, інтегруючись із сервісом оцінки ризиків, який аналізує поведінкові та контекстні фактори і визначає рівень загрози. Сервіс обробки подій у реальному часі транслює повідомлення клієнтам за моделлю publish-subscribe, підписуючись на брокер повідомлень Kafka – інструмент для асинхронного обміну даних, який забезпечує високу масштабованість та відмовостійкість.

Сховище даних і кешування поєднують PostgreSQL та Redis, забезпечуючи консистентне збереження даних, швидке обслуговування токенів і профілів ризику. HashiCorp Vault централізовано зберігає секрети, ключі та конфігурації з контролем доступу та ротацією. Observability модуль з використанням Prometheus та Grafana дозволяє відстежувати метрики, логувати події та швидко реагувати на аномалії, межі яких можна гнучко налаштувати. Контейнеризація та оркестрація через Docker та Kubernetes забезпечує автоматичне масштабування, rolling updates та self-healing. Користувацький інтерфейс був реалізований фреймворком React.js, продемонстрував мінімальні затримки та добре продуманий UI/UX дизайн.

Для тестування системи було обрано низку ключових метрик моніторингу: середній відсоток коефіцієнту ризику, активність користувачів та брокера

повідомлень, статистику (не-)успішних TOTP-перевірок. Тестові сценарії показали, що найбільш значущим фактором ризику є вхід з нового пристрою. Затримки Redis коливаються в межах 7-10 мс, що є нормою для локального середовища при низько-середньому навантаженні, а середній час відповіді API здебільшого складав 150-300 мс з піками 600-700 мс під час балансування навантаження між репліками сервісу автентифікації. Балансувальник Nginx коректно перенаправляв запити на резервний контейнер, забезпечуючи стабілізацію після короткочасного сплеску. На графіку HTTP-кодів домінував 2xx, що підтверджує якісну реалізацію API, хоча поодинокі 4xx та 5xx сигналізують про необхідність додаткового тестування вироджених сценаріїв.

У четвертому розділі було підтверджено актуальність розроблюваного стартапу, проведено аналіз ринку та конкурентів, розроблено маркетингову стратегію з позиціонуванням, комунікацією та ціноутворенням. Незважаючи на наявність конкурентів, запропонована система вирізняється універсальністю та можливістю створення на її основі власних продуктів, що підвищує її конкурентоспроможність. Технологічна здійсненність, відмовостійкість та рентабельність продукту доведені, що дозволяє зробити висновок про високий потенціал успіху стартапу за умови запуску ключових функцій з мінімальною початковою вартістю та ефективного використання маркетингових комунікацій для залучення цільової аудиторії.

Таким чином, розроблена система демонструє високу надійність та відмовостійкість, мінімальні затримки обробки подій, масштабованість без простоїв, швидко працюючу багаторівневу безпеку з адаптивною автентифікацією та основу для подальшого розвитку й інтеграції нових сервісів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. MIT Lincoln Laboratory. (n.d.). SAGE: Semi-Automatic Ground Environment. Retrieved from https://www.ll.mit.edu/.

2. Amazon Web Services. (n.d.). AWS Well-Architected Framework: Reliability Pillar. Retrieved from https://docs.aws.amazon.com/.

3. David Temoshok et al. Digital Identity Guidelines: Authentication and Lifecycle Management. 2025. NIST Special Publication 800-63B.

4. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Incorporated, 2017. 616 p.

5. L. Bilge, T. Dumitras. Before We Knew It: An Empirical Study of Zero Day Attacks In The Real World.

6. Newman S. Building Microservices. O'Reilly Media, Incorporated, 2015.

7. Офіційна документація (потоки, exactly-once семантика, конвеєри подій). Apache Kafka.

8. RabbitMQ Documentation | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/docs.

9. Іваненко, О. П. *Основи відмовостійких систем та кластеризації*. Київ: Видавництво «ТехноПрогрес», 2021. – 56 с.

10. Мельник, Д. С. *Інфраструктурні технології та керування навантаженням у розподілених системах*. Львів: Видавничий центр «СофтСистеми», 2022. – 94 с.

11. Nygard M. T. Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers) (Pragmatic Programmers). Pragmatic Bookshelf, 2007. 326 p.

12. Microsoft M. J. JSON Web Token (JWT). 2015. RFC 7519.

13. Verisign, Inc D. M. Time-Based One-Time Password Algorithm. 2011. RFC 6238.

14. A. Monrose, A. Rubin. Keystroke Dynamics as a Biometric for User Identification and Authentication.

15. Multifactor Authentication Cheat Sheet. https://cheatsheetseries.owasp.org.

16. National Institute of Standards and Technology (NIST). Digital Identity Guidelines. NIST Special Publication 800-63-3. URL: https://pages.nist.gov/800-63-3/.

17. Pedregosa F. та ін. Scikit-learn: Machine Learning in Python. URL: https://scikit-learn.org/.

18. API gateways - Azure Architecture Center. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/gateway.

19. V. Bertocci. JWT Profile for OAuth 2.0 Access Tokens. 2021. RFC 9068.

20. Part 1: Apache Kafka for beginners - What is Apache Kafka? - CloudKarafka, Apache Kafka. CloudKarafka. URL: https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html.

21. Kubernetes Documentation. Ingress Controllers. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

22. Creating Highly Available Clusters with kubeadm. Kubernetes. Https://kubernetes.io/docs.

23. Ліненко К. М. GitHub - KonstantinLi/aura-auth. URL: https://github.com/KonstantinLi/aura-auth.

24. Ліненко К. М. GitHub - KonstantinLi/aura-risk. URL: https://github.com/KonstantinLi/aura-risk.

25. Ліненко К. М. GitHub - KonstantinLi/aura-realtime. URL: https://github.com/KonstantinLi/aura-realtime.

ДОДАТОК 1

Частина програмного коду

Сервіс автентифікації та авторизації

```
public class UserService {

   private final UserMapper userMapper;
   private final ImageService imageService;
   private final UserRepository userRepository;
   private final CredentialsRepository credentialsRepository;

   @Transactional(readOnly = true)
   public boolean isUserEnabled(String username) {
      Optional<User> user = userRepository.findByEmail(username);
      return user.isPresent() && user.get().isEnabled();
   }

   @Transactional(readOnly = true)
   public boolean isUserRegistered(String username) {
      return userRepository.findByEmail(username).isPresent();
   }

   @Transactional(readOnly = true)
   public Optional<UserSecurityDto> findByUsername(String username) {
      Optional<Credentials> optionalCredentials =
credentialsRepository.findByUsername(username);
      return optionalCredentials.map(userMapper::toUserSecurityDto);
   }

   @Transactional
   public void setUserLogo(Long userId, UploadImageDto image, double quality) {
      String avatarUrl = image.content() != null ? imageService.uploadImage(userId, image,
quality) : null;
      userRepository.findByIdRequired(userId).setAvatarUrl(avatarUrl);
   }

   @Transactional
   public UserSecurityDto saveUser(SaveUserDto saveUserDto) {
      Optional<User> optionalUser = userRepository.findByEmail(saveUserDto.getEmail())
            .map(user -> userMapper.updateRegisteredUser(user, saveUserDto));
      Credentials credentials =
optionalUser.map(credentialsRepository::findByUser).filter(Optional::isPresent)
            .map(optionalCredentials -> userMapper.updateCredentials(optionalCredentials.get(),
saveUserDto))
            .orElse(userMapper.toCredentials(saveUserDto));
      if (optionalUser.isEmpty()) {
         credentials.setUser(userRepository.save(userMapper.toUser(saveUserDto)));
         credentials = credentialsRepository.save(credentials);
```

```
        }
        return userMapper.toUserSecurityDto(credentials);
    }

    @Transactional(readOnly = true)
    public Optional<UserSecurityDto> getUserSecurityInfoById(Long id) {
        return userRepository.findById(id).map(userMapper::toUserSecurityDto);
    }

}
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserService userService;
    private final UserMapper securityMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return userService.findByUsername(username)
                .map(securityMapper::toAuthenticatedUser)
                .orElseThrow(() -> new UsernameNotFoundException(username));
    }

}

public class SecurityService {

    private final JwtService jwtService;
    private final UserMapper userMapper;
    private final UserService userService;
    private final PasswordEncoder passwordEncoder;
    private final TokenRepository tokenRepository;
    private final SecurityValidator securityValidator;

    @Transactional
    public Long saveUser(SignUpRequest signUpRequest) {
        securityValidator.validateUserRegistration(signUpRequest);
        UserSecurityDto savedUser =
userService.saveUser(userMapper.toSaveUserDto(signUpRequest, passwordEncoder));
        userService.notifyUserConfirmation(savedUser);
        return savedUser.id();
    }

    @Transactional
    public AuthTokensResponse generateAuthTokens(Long authenticatedUserId) {
        var token = jwtService.generateToken(authenticatedUserId);
        var refreshToken = jwtService.generateRefreshToken(authenticatedUserId);
        saveTokenData(authenticatedUserId, token, refreshToken);
        return new AuthTokensResponse(token, refreshToken);
    }
```

```
    private void saveTokenData(Long userId, String accessToken, String refreshToken) {
        Token token = tokenRepository.findByUserId(userId).orElse(new Token());
        token.setToken(accessToken);
        token.setRefreshToken(refreshToken);
        token.setUserId(userId);
        tokenRepository.save(token);
    }

}

public class JwtService {

    private final SecretKey secretKey;
    private final long tokenExpirationMilliseconds;
    private final long refreshTokenExpirationMilliseconds;

    public JwtService(@Value("${application.security.token-secret-key}") String tokenSecretKey,
                      @Value("${application.security.token-expiration}") long
tokenExpirationMilliseconds,
                      @Value("${application.security.refresh-token-expiration}") long
refreshTokenExpirationMilliseconds) {
        this.secretKey = Keys.hmacShaKeyFor(Decoders.BASE64.decode(tokenSecretKey));
        this.tokenExpirationMilliseconds = tokenExpirationMilliseconds;
        this.refreshTokenExpirationMilliseconds = refreshTokenExpirationMilliseconds;
    }

    public boolean isTokenNotAcceptable(String token) {
        try {
            Claims jwtClaims = extractAllClaims(token);
            return jwtClaims.getExpiration().toInstant().isBefore(Instant.now()) ||
jwtClaims.getSubject() == null;
        } catch (JwtException | IllegalArgumentException ex) {
            return true;
        }
    }

    public String generateToken(Long authenticatedUserId) {
        return createToken(authenticatedUserId, tokenExpirationMilliseconds);
    }

    public String generateRefreshToken(Long authenticatedUserId) {
        return createToken(authenticatedUserId, refreshTokenExpirationMilliseconds);
    }

    private String createToken(Long authenticatedUserId, long tokenExpirationMilliseconds) {
        return Jwts.builder()
                .subject(authenticatedUserId.toString())
                .issuedAt(Date.from(Instant.now()))
                .expiration(Date.from(Instant.now().plusMillis(tokenExpirationMilliseconds)))
                .signWith(secretKey, Jwts.SIG.HS256)
                .compact();
```

```
    }

    public String extractSubject(String token) {
        return extractAllClaims(token).getSubject();
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser()
                .verifyWith(secretKey)
                .build()
                .parseSignedClaims(token)
                .getPayload();
    }

}

public class EncryptionService {

    private final String algorithm;
    private final SecretKey secretKey;

    public EncryptionService(
            @Value("${application.security.encryption-secret-key}") String secretKey,
            @Value("${application.security.encryption-algorithm}") String algorithm) {
        this.algorithm = algorithm;
        this.secretKey = new SecretKeySpec(Base64.getDecoder().decode(secretKey), algorithm);
    }

    public String encrypt(String data) {
        try {
            Cipher cipher = Cipher.getInstance(algorithm);
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            return Base64.getEncoder().encodeToString(cipher.doFinal(data.getBytes()));
        } catch (Exception ex) {
            throw new EncryptionException(ex.getMessage());
        }
    }

    public String decrypt(String encryptedData) {
        try {
            Cipher cipher = Cipher.getInstance(algorithm);
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            return new String(cipher.doFinal(Base64.getDecoder().decode(encryptedData)));
        } catch (Exception ex) {
            throw new EncryptionException(ex.getMessage());
        }
    }

    public String generateKey() {
        try {
            KeyGenerator generator = KeyGenerator.getInstance("AES");
```

```
            generator.init(256);
            SecretKey secretKey = generator.generateKey();
            return Base64.getEncoder().encodeToString(secretKey.getEncoded());
        } catch (NoSuchAlgorithmException ex) {
            throw new RuntimeException(ex);
        }
    }

}

public class OAuth2AuthenticationHandler extends BaseHandler {

    private static final String SIGN_IN_URI = "/authentication/sign-in";
    private static final String WELCOME_PAGE_URI = "/authentication/welcome";

    private final String clientHost;
    private final UserMapper userMapper;
    private final List<ExternalUserProcessor> externalUserProcessors;

    public OAuth2AuthenticationHandler(UserMapper userMapper, UserService userService,
            SecurityService securityService, List<ExternalUserProcessor> externalUserProcessors,
            @Value("${application.host}") String host, @Value("${spring.profiles.active:local}") String
profile) {
        super(userService, securityService);
        this.userMapper = userMapper;
        this.externalUserProcessors = externalUserProcessors;
        this.clientHost = !profile.equals("local") ? host : host.replace("8080", "3001");
    }

    private ExternalUserProcessor defineExternalUserProcessor(OidcUser externalUser) {
        return externalUserProcessors.stream().filter(processor ->
processor.canProcess(externalUser)).findAny()
                .orElseThrow(() -> new
ExternalUserProcessorNotFoundException(externalUser.getIssuer().toString()));
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response,
            Authentication authentication) throws IOException {
        OidcUser externalUser = (OidcUser) authentication.getPrincipal();
        OAuth2UserDto externalUserDto = defineExternalUserProcessor(externalUser)
                .extractExternalUser(externalUser, authentication);
        AuthenticatedUser user = userService.findByUsername(externalUserDto.email())
                .map(userMapper::toAuthenticatedUser).orElse(null);
        if (user != null) {
            setUpResponse(user.getId(), clientHost + WELCOME_PAGE_URI, response);
        }
        try {
            Long userId =
securityService.saveUser(userMapper.toExternalSignUpRequest(externalUserDto));
```

```
            userService.setUserLogo(userId, new UploadImageDto(externalUserDto.image(),
"image/png", null), 0.3);
            setUpResponse(userId, clientHost + WELCOME_PAGE_URI, response);
        } catch (Exception ex) {
            log.debug("OAuth2 registration failed: {}", ex.getMessage(), ex);
            setUpResponse(null, clientHost + SIGN_IN_URI, response);
        }
    }

    private void setUpResponse(Long userId, String redirect, HttpServletResponse response)
throws IOException {
        if (userId != null) {
            AuthTokensResponse authTokensResponse =
securityService.generateAuthTokens(userId);
            response.addCookie(createCookie("accessToken", authTokensResponse.token()));
            response.addCookie(createCookie("refreshToken",
authTokensResponse.refreshToken()));
        }
        response.sendRedirect(redirect);
    }

    private Cookie createCookie(String name, String value) {
        var cookie = new Cookie(name, value);
        cookie.setPath("/");
        cookie.setSecure(true);
        cookie.setHttpOnly(true);
        return cookie;
    }

}

public class UsernamePasswordAuthenticationHandler extends BaseHandler {

    private final ObjectMapper objectMapper;

    public UsernamePasswordAuthenticationHandler(SecurityService securityService,
UserService userService, ObjectMapper objectMapper) {
        super(userService, securityService);
        this.objectMapper = objectMapper;
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication)
            throws IOException {
        Long userId = ((AuthenticatedUser) authentication.getPrincipal()).getId();
        AuthTokensResponse authTokensResponse = securityService.generateAuthTokens(userId);

response.getOutputStream().write(objectMapper.writeValueAsBytes(authTokensResponse));
    }
```

```
}

public class AuthConfiguration {

   @Bean
   public RestTemplate restTemplate() {
      SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFactory();
      factory.setConnectTimeout(5000);
      factory.setReadTimeout(5000);
      return new RestTemplate(factory);
   }

}

public class SecurityConfiguration implements WebMvcConfigurer {

   private final String host;
   private final String profile;

   public SecurityConfiguration(
         @Value("${application.host}") String host,
         @Value("${spring.profiles.active:local}") String profile) {
      this.host = host;
      this.profile = profile;
   }

   @Override
   public void addCorsMappings(CorsRegistry registry) {
      List<String> allowedOrigins = new ArrayList<>(List.of(host));
      if (!profile.equals("prod")) allowedOrigins.add("http://localhost:3001");
      registry.addMapping("/**")
            .allowedOrigins(allowedOrigins.toArray(new String[0]))
            .allowedMethods("*")
            .allowedHeaders("*")
            .exposedHeaders("*")
            .allowCredentials(true)
            .maxAge(900);
   }

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http, OAuth2AuthenticationHandler
oAuth2AuthenticationHandler,
         UsernamePasswordAuthenticationHandler usernamePasswordAuthenticationHandler,
         UserDetailsService userDetailsService, JwtAuthFilter jwtAuthFilter,
         ClearTokensLogoutHandler logoutHandler) throws Exception {

      http.sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
            .csrf(AbstractHttpConfigurer::disable)
            .cors(withDefaults())
```

```
            .addFilterBefore(jwtAuthFilter, LogoutFilter.class)
            .authorizeHttpRequests(requestMatcherRegistry -> requestMatcherRegistry
                .anyRequest().authenticated())
            .oauth2Login(oAuth2LoginConfigurer -> oAuth2LoginConfigurer
                .successHandler(oAuth2AuthenticationHandler)
                .failureHandler(oAuth2AuthenticationHandler))
            .formLogin(formLoginConfigurer -> formLoginConfigurer
                .loginProcessingUrl("/login")
                .successHandler(usernamePasswordAuthenticationHandler)
                .failureHandler(usernamePasswordAuthenticationHandler))
            .userDetailsService(userDetailsService)
            .logout(logoutConfigurer -> logoutConfigurer
                .logoutRequestMatcher(new AndRequestMatcher(
                    new
RequestHeaderRequestMatcher(JwtAuthFilter.AUTHORIZATION_HEADER_NAME),
                    new AntPathRequestMatcher("/logout", HttpMethod.POST.name())
                ))
                .addLogoutHandler(logoutHandler)
                .logoutSuccessHandler(logoutHandler));

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
public class JwtAuthFilter extends OncePerRequestFilter {

    public static final String AUTHORIZATION_TOKEN_PREFIX = "Bearer ";
    public static final String AUTHORIZATION_HEADER_NAME = "Authorization";
    private static final String USER_ID_NOT_FOUND_MESSAGE_TEMPLATE = "User with id %s not
found";

    private final JwtService jwtService;
    private final UserService userService;
    private final UserMapper securityMapper;
    private final ObjectMapper objectMapper;

    @Override
    protected void doFilterInternal(HttpServletRequest request, @NonNull HttpServletResponse
response,
            @NonNull FilterChain filterChain) throws ServletException, IOException {
        String requestUri = request.getRequestURI();
        log.debug("Processing authentication for request: {}", requestUri);

        String token = Optional.ofNullable(request.getHeader(AUTHORIZATION_HEADER_NAME))
            .orElseGet(() -> Optional.ofNullable(getCookie(request, "accessToken"))
                .map(AUTHORIZATION_TOKEN_PREFIX::concat).orElse(null));
```

```
        if (jwtService.isTokenNotAcceptable(token)) {
            log.debug("Unauthorized access attempt - Missing or invalid Authorization header in
request to {}", requestUri);
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return;
        }

        Long userId = Long.valueOf(jwtService.extractSubject(token));
        log.debug("Extracted user ID {} from JWT for request to {}", userId, requestUri);
        AuthenticatedUserInfo authenticatedUser;
        try {
            authenticatedUser = loadUserById(userId);
        } catch (DisabledException ex) {
            setupResponse(response, new ErrorResponse(ex.getMessage()),
HttpServletResponse.SC_FORBIDDEN);
            return;
        } catch (Exception ex) {
            setupResponse(response, new ErrorResponse(ex.getMessage()),
HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            return;
        }

        UsernamePasswordAuthenticationToken authentication =
UsernamePasswordAuthenticationToken
                .authenticated(authenticatedUser, null, Collections.emptyList());
        authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authentication);

        log.debug("Successfully authenticated user {} for request to {}", userId, requestUri);
        filterChain.doFilter(request, response);
    }

    private AuthenticatedUserInfo loadUserById(Long id) {
        log.debug("Loading user by ID: {}", id);
        UserSecurityDto userSecurityDto =
userService.getUserSecurityInfoById(id).orElseThrow(() -> {
            log.error("User not found with ID: {}", id);
            return new
BadCredentialsException(USER_ID_NOT_FOUND_MESSAGE_TEMPLATE.formatted(id));
        });
        if (!userSecurityDto.enabled()) {
            log.debug("Attempt to authenticate disabled user with ID: {}", id);
            throw new DisabledException("User with id %s is disabled".formatted(id));
        }
        return securityMapper.toAuthenticatedUserInfo(userSecurityDto);
    }

    private String getCookie(HttpServletRequest request, String cookieName) {
        return Optional.ofNullable(request.getCookies()).flatMap(cookies ->
Arrays.stream(cookies)
                .filter(cookie ->
```

```
cookieName.equals(cookie.getName())).findAny().map(Cookie::getValue))
            .orElse(null);
    }

    private void setupResponse(HttpServletResponse response, Object payload, int code) throws
IOException {
        response.setStatus(code);
        response.setCharacterEncoding("UTF-8");
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        response.getWriter().write(objectMapper.writeValueAsString(payload));
    }

}
```

Сервіс оцінки ризиків

```
public class RiskEvaluator {


    public enum RiskLevel {
        LOW, MEDIUM, HIGH
    }


    public enum RiskFactor {
        GEO, TIME, DEVICE, FAILURES, BEHAVIOR
    }


    private final Map<RiskFactor, Double> weights;


    public RiskEvaluator() {
        weights = Map.of(
                RiskFactor.GEO, 0.20,
                RiskFactor.TIME, 0.10,
                RiskFactor.DEVICE, 0.30,
                RiskFactor.FAILURES, 0.20,
                RiskFactor.BEHAVIOR, 0.20
        );
    }


    public RiskResult score(EnrichedAuthEvent event) {
```

```
    Map<RiskFactor, Double> factorScores = new EnumMap<>(RiskFactor.class);
    factorScores.put(RiskFactor.GEO, geoScore(event));
    factorScores.put(RiskFactor.TIME, timeScore(event));
    factorScores.put(RiskFactor.DEVICE, deviceScore(event));
    factorScores.put(RiskFactor.FAILURES, failuresScore(event));
    factorScores.put(RiskFactor.BEHAVIOR, behaviorScore(event));

    double totalScore = factorScores.entrySet().stream()
            .mapToDouble(e -> e.getValue() * weights.getOrDefault(e.getKey(), 0.0))
            .sum() * 100.0;

    RiskLevel level = classifyRisk(totalScore);

    return new RiskResult(
            event.userId(),
            totalScore,
            level,
            factorScores
    );
}

private RiskLevel classifyRisk(double score) {
    if (score < 30) return RiskLevel.LOW;
    else if (score < 60) return RiskLevel.MEDIUM;
    else return RiskLevel.HIGH;
}

private double geoScore(EnrichedAuthEvent e) {
    Optional<GeoLocation> lastGeo = e.lastGeo();
    if (lastGeo.isEmpty()) return 0.1;

    boolean countryChanged = !e.geo().country().equals(lastGeo.get().country());
    boolean impossibleTravel = e.travelTimeHours() < e.minTravelTimeHours();
    return countryChanged ? (impossibleTravel ? 1.0 : 0.6) : 0.0;
```

```
    }

    private double timeScore(EnrichedAuthEvent e) {
        return e.isUnusualHour() ? 0.7 : 0.0;
    }

    private double deviceScore(EnrichedAuthEvent e) {
        return e.isKnownDevice() ? 0.0 : 0.8;
    }

    private double failuresScore(EnrichedAuthEvent e) {
        int attempts = e.failedAttemptsLast15m();
        return Math.min(1.0, attempts / 5.0);
    }

    private double behaviorScore(EnrichedAuthEvent e) {
        Optional<BehaviorMetrics> behavior = e.behavior();
        if (behavior.isEmpty()) return 0.15;

        BehaviorMetrics b = behavior.get();
        return clamp(0.0, 1.0, 0.5 * b.keystrokeAnomaly() + 0.5 * b.mouseAnomaly());
    }

    private double clamp(double min, double max, double value) {
        return Math.max(min, Math.min(max, value));
    }
}
```

Сервіс обробки подій у реальному часі

```
class KafkaRealtimeConfig {

    @Bean
    ReceiverOptions<String, SecurityAlert> receiverOptions(@Value("${kafka.bootstrap}")
                                        String bootstrap) {
        return ReceiverOptions.<String, SecurityAlert>create(Map.of(
            ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrap,
```

```
            ConsumerConfig.GROUP_ID_CONFIG, "realtime-service",
            ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class,
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class,
            JsonDeserializer.TRUSTED_PACKAGES, "*",
            ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest"
        )).subscription(List.of("security-alerts"));
    }

    @Bean
    Flux<SecurityAlert> alertsStream(ReceiverOptions<String, SecurityAlert> options) {
        return KafkaReceiver.create(options)
                .receive()
                .map(rec -> {
                    rec.receiverOffset().acknowledge(); // at-least-once: ack після
                    обробки/буферизації
                    return rec.value();
                })
                .publish()
                .refCount(1);
    }
}

class AlertsFanout {

    private final Sinks.Many<SecurityAlert> sink =
            Sinks.many().multicast().onBackpressureBuffer();

    AlertsFanout(Flux<SecurityAlert> alertsStream) {
        alertsStream.subscribe(alert -> sink.tryEmitNext(alert));
    }

    public Flux<SecurityAlert> streamFor(UserPrincipal principal, @Nullable String tenantId)
    {
        return sink.asFlux().filter(a -> hasAccess(principal, a, tenantId))
                .onBackpressureLatest();
    }

    private boolean hasAccess(UserPrincipal p, SecurityAlert a, String tenantId) {
        if (p.isAdmin()) {
            return tenantId == null || tenantId.equals(a.tenantId());
        }
        return a.userId() != null && a.userId().equals(p.userId());
    }

    private boolean hasAccess(UserPrincipal p, SecurityAlert a, String tenantId) {
        if (p.isAdmin()) {
            return tenantId == null || tenantId.equals(a.tenantId());
        }
        return a.userId() != null && a.userId().equals(p.userId());
    }
```

```
}

class AlertsSseController {

   private final AlertsFanout fanout;

   @GetMapping(path = "/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
   public Flux<ServerSentEvent<AlertDto>> stream(@AuthenticationPrincipal
                                UserPrincipal principal, @RequestParam(required = false) String
tenantId) {
      return fanout.streamFor(principal, tenantId)
            .map(AlertDto::from)
            .map(dto -> ServerSentEvent.builder(dto).event("alert").build());
   }

class JwtWebFilter implements WebFilter {

   private final JwtVerifier jwtVerifier;

   @Override
   public Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain) {
      String auth =
            exchange.getRequest().getHeaders().getFirst(HttpHeaders.AUTHORIZATION);
      if (auth != null && auth.startsWith("Bearer ")) {
         return jwtVerifier.authenticate(auth.substring(7))
               .flatMap(p -> chain.filter(exchange.mutate()
                     .principal(Mono.just(p)).build()));
      }
      return chain.filter(exchange);
   }
}
```

Конфігурація сховища даних та кешу

```
spring:
      data:
      redis:
      host: ${REDIS_HOST}
      port: 6379
      ssl: true
      timeout: 2s
      lettuce:
      pool:
      max-active: 8
      max-idle: 4

public class OtpCacheService {
   private final RedisTemplate<String, Object> redisTemplate;
   private static final Duration TTL = Duration.ofSeconds(60);
```

```
    public void saveOtp(String email, String otpCode) {
        redisTemplate.opsForValue().set("otp:" + email, otpCode, TTL);
    }

    public boolean verifyOtp(String email, String code) {
        String cachedCode = (String) redisTemplate.opsForValue().get("otp:" + email);
        if (cachedCode != null && cachedCode.equals(code)) {
            redisTemplate.delete("otp:" + email);
            return true;
        }
        return false;
    }
}

public class RiskProfileCache {

    private final RedisTemplate<String, RiskProfile> redisTemplate;
    private final RiskProfileRepository repository;

    private static final Duration TTL = Duration.ofMinutes(15);

    public RiskProfile getOrLoad(Long userId) {
        String key = "risk:profile:" + userId;
        RiskProfile cached = redisTemplate.opsForValue().get(key);
        if (cached != null) return cached;

        RiskProfile fromDb = repository.findByUserId(userId)
                .orElse(new RiskProfile(userId));
        redisTemplate.opsForValue().set(key, fromDb, TTL);
        return fromDb;
    }
}

public class RealtimeEventBuffer {

    private final RedisTemplate<String, String> redisTemplate;
    private static final String KEY = "alerts:recent";
    private static final int MAX_SIZE = 1000;

    public void pushEvent(String eventJson) {
        redisTemplate.opsForList().leftPush(KEY, eventJson);
        redisTemplate.opsForList().trim(KEY, 0, MAX_SIZE - 1);
    }

    public List<String> getRecentEvents() {
        return redisTemplate.opsForList().range(KEY, 0, MAX_SIZE - 1);
    }
}
```

Конфігурація систем збору метрик та моніторингу

application.yml

```
management:
    endpoints:
    web:
    exposure:
    include: prometheus,health,info
    metrics:
    tags:
    application: aura-auth
    environment: prod
```

prometheus.yml

```
global:
    scrape_interval: 15s
    scrape_configs:
    - job_name: 'aura-auth'
    metrics_path: '/actuator/prometheus'
    static_configs:
    - targets: ['aura-auth:8080']
    - job_name: 'risk-service'
    static_configs:
    - targets: ['risk-service:8081']
    - job_name: 'realtime-service'
    static_configs:
    - targets: ['realtime-service:8082']
```

loki.yml

```
server:
    http_listen_port: 9080
    positions:
    filename: /tmp/positions.yaml
    clients:
    - url: http://loki:3100/loki/api/v1/push
    scrape_configs:
    - job_name: 'aura-logs'
    static_configs:
    - targets: [localhost]
    labels:
    job: aura-auth
    __path__: /var/log/aura/*.log
```

Конфігурація інфраструктури та оркестрації

aura-auth/Dockerfile

```
FROM maven:3.9-eclipse-temurin-21 AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline -B
COPY src ./src
RUN mvn clean package -DskipTests
```

```
FROM eclipse-temurin:21-jre
WORKDIR /app
COPY --from=builder /app/target/aura-auth.jar app.jar
ENTRYPOINT ["java","-jar","app.jar"]
```

Risk-service/Dockerfile

```
FROM maven:3.9-eclipse-temurin-21 AS builder
WORKDIR /risk
COPY . .
RUN mvn clean package -DskipTests

FROM eclipse-temurin:21-jre
WORKDIR /risk
COPY --from=builder /risk/target/risk-service.jar .
ENTRYPOINT ["java","-jar","risk-service.jar"]
```

realtime-service/Dockerfile

```
FROM eclipse-temurin:21-jre
WORKDIR /realtime
COPY target/realtime-service.jar .
EXPOSE 8082
ENTRYPOINT ["java","-jar","realtime-service.jar"]
```

docker-compose.yml

```
version: '3.9'

services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: root
      POSTGRES_PASSWORD: test
      POSTGRES_DB: scaler
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  kafka:
    image: bitnami/kafka:3.7
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CFG_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
```

```
  zookeeper:
    image: bitnami/zookeeper:3.9
    environment:
      ALLOW_ANONYMOUS_LOGIN: "yes"

  vault:
    image: hashicorp/vault:1.17
    cap_add:
      - IPC_LOCK
    ports:
      - "8200:8200"
    environment:
      VAULT_DEV_ROOT_TOKEN_ID: dev-root-token
      VAULT_ADDR: http://0.0.0.0:8200

  prometheus:
    image: prom/prometheus
    volumes:
      - ./infra/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    ports:
      - "3000:3000"

  aura-auth:
    build: ./aura-auth
    environment:
      SPRING_PROFILES_ACTIVE: docker
    depends_on:
      - postgres
      - redis
      - kafka
      - vault
      ports:
      - "8080:8080"

  risk-service:
    build: ./risk-service
    depends_on:
      - postgres
      - kafka
```

aura-auth-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aura-auth
```

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: aura-auth
    template:
      metadata:
        labels:
          app: aura-auth
      spec:
        containers:
          - name: aura-auth
          image: registry.local/aura-auth:latest
          ports:
            - containerPort: 8080
          envFrom:
            - configMapRef:
            name: aura-config
            - secretRef:
            name: aura-secrets
          livenessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
```

aura-auth-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: aura-auth
spec:
  type: ClusterIP
  selector:
    app: aura-auth
  ports:
    - port: 8080
    targetPort: 8080
```

aura-ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: aura-ingress
spec:
  rules:
    - host: aura.local
    http:
      paths:
        - path: /
```

```
            pathType: Prefix
            backend:
              service:
                name: aura-auth
                port:
                  number: 8080
```

.gitlab-ci.yml

```
stages:
  - build
  - deploy

build:
  stage: build
  image: maven:3.9-eclipse-temurin-21
  script:
    - mvn clean package -DskipTests
    - docker build -t registry.local/aura-auth:$CI_COMMIT_SHORT_SHA ./aura-auth
  only:
    - main

deploy:
  stage: deploy
  image: bitnami/kubectl:1.30
  script:
    - kubectl apply -f k8s/
  only:
    - main
```

Користувацький інтерфейс

```
interface AuthLayoutProps {
 children: ReactNode
 title: string
 subtitle?: string
}

export function AuthLayout({ children, title, subtitle }: AuthLayoutProps) {
 return (
  <div className="min-h-screen flex items-center justify-center bg-[#2D1B69] p-4">
   <div className="w-full max-w-md bg-white rounded-3xl p-8 shadow-xl">
    <div className="flex items-center justify-center flex-col">
     <h1 className="text-2xl font-semibold mb-2 text-center">{title}</h1>
    </div>
    {subtitle && <p className="text-sm text-muted-foreground mb-4">{subtitle}</p>}
    {children}
   </div>
  </div>
```

```
  )
}

interface AuthLayoutProps {
  children: ReactNode
  title: string
  subtitle?: string
}

export function AuthLayout({ children, title, subtitle }: AuthLayoutProps) {
  return (
    <div className="min-h-screen flex items-center justify-center bg-[#2D1B69] p-4">
      <div className="w-full max-w-md bg-white rounded-3xl p-8 shadow-xl">
        <div className="flex items-center justify-center flex-col">
          <h1 className="text-2xl font-semibold mb-2 text-center">{title}</h1>
        </div>
        {subtitle && <p className="text-sm text-muted-foreground mb-4">{subtitle}</p>}
        {children}
      </div>
    </div>
  )
}


interface DividerProps {
  text?: string
  className?: string
}

export function Divider({ text = 'or', className = '' }: DividerProps) {
  return (
    <div className={`my-6 relative ${className}`}>
      <div className="absolute inset-0 flex items-center">
        <div className="w-full border-t border-border" />
      </div>
      <div className="relative flex justify-center text-xs uppercase">
        <span className="bg-white px-2 text-muted-foreground">{text}</span>
      </div>
    </div>
  )
}


interface LogoProps {
  className?: string
}

export function Logo({ className = '' }: LogoProps) {
  return (
    <div className={`flex items-center gap-2 ${className}`}>
      <div className="w-6 h-6 bg-[hsl(var(--primary))] rounded flex items-center justify-
```

```
center">
      <BookOpen className="w-4 h-4 text-white" />
    </div>
    <span className="text-xl font-semibold">AURA</span>
   </div>
  )
}


interface PageRoute {
  path: string
  label: string
  group?: 'auth' | 'main'
}

const PAGE_ROUTES: PageRoute[] = [
  { path: ROUTES.SIGN_IN, label: 'Sign In', group: 'auth' },
  { path: ROUTES.SIGN_UP, label: 'Sign Up', group: 'auth' },
  { path: ROUTES.VERIFY_CODE, label: 'Verify Code', group: 'auth' },
  { path: ROUTES.VERIFY_TOTP, label: 'Verify TOTP', group: 'auth' },
  { path: ROUTES.SCAN_QR_CODE, label: 'Scan QR Code', group: 'auth' },
  { path: ROUTES.SET_PASSWORD, label: 'Set Password', group: 'auth' },
  { path: ROUTES.DASHBOARD, label: 'Dashboard', group: 'main' },
]

export function PageNavigator() {
  const navigate = useNavigate()
  const location = useLocation()
  const [open, setOpen] = useState(false)

  const handleNavigate = (path: string) => {
    navigate(path)
    setOpen(false)
  }

  const authPages = PAGE_ROUTES.filter((route) => route.group === 'auth')
  const mainPages = PAGE_ROUTES.filter((route) => route.group === 'main')

  return (
    <div className="fixed top-4 left-4 z-50">
      <DropdownMenu open={open} onOpenChange={setOpen}>
        <DropdownMenuTrigger asChild>
          <button
            className="flex items-center justify-center w-10 h-10 rounded-full bg-primary text-
primary-foreground shadow-lg hover:bg-primary/90 transition-all hover:scale-105
focus:outline-none focus:ring-2 focus:ring-primary focus:ring-offset-2"
            aria-label="Navigation menu"
          >
            <Menu className="h-5 w-5" />
          </button>
        </DropdownMenuTrigger>
```

```
      <DropdownMenuContent align="start" className="w-56">
       <DropdownMenuLabel>Navigation</DropdownMenuLabel>
       <DropdownMenuSeparator />

       <DropdownMenuLabel className="text-xs text-muted-foreground font-normal">
        Authentication
       </DropdownMenuLabel>
       {authPages.map((route) => (
        <DropdownMenuItem
         key={route.path}
         onClick={() => handleNavigate(route.path)}
         className={location.pathname === route.path ? 'bg-accent text-accent-foreground' : ''}
        >
         <span className="flex items-center gap-2">
          {location.pathname === route.path && (
           <span className="w-1.5 h-1.5 rounded-full bg-primary" />
          )}
          {route.label}
         </span>
        </DropdownMenuItem>
       ))}

       <DropdownMenuSeparator />

       <DropdownMenuLabel className="text-xs text-muted-foreground font-normal">
        Main Pages
       </DropdownMenuLabel>
       {mainPages.map((route) => (
        <DropdownMenuItem
         key={route.path}
         onClick={() => handleNavigate(route.path)}
         className={location.pathname === route.path ? 'bg-accent text-accent-foreground' : ''}
        >
         <span className="flex items-center gap-2">
          {location.pathname === route.path && (
           <span className="w-1.5 h-1.5 rounded-full bg-primary" />
          )}
          {route.label}
         </span>
        </DropdownMenuItem>
       ))}
      </DropdownMenuContent>
     </DropdownMenu>
    </div>
  )
}


export function RootLayout() {
  return (
    <NuqsAdapter>
```

```
    <PageNavigator />
    <Outlet />
   </NuqsAdapter>
  )
}


type SidebarProps = {
  selected: SidebarKey
  onSelect: (key: SidebarKey) => void
}

export default function Sidebar({ selected, onSelect }: SidebarProps): JSX.Element {
  const items: Array<{ key: SidebarKey; label: string; icon: React.ReactNode }> = [
   { key: 'dashboard', label: 'Dashboard', icon: <HomeIcon size={20} /> },
   { key: 'mfa', label: 'Multi-factor auth', icon: <Shield size={20} /> },
   { key: 'connectors', label: 'Connectors', icon: <Plug size={20} /> },
   { key: 'settings', label: 'Settings', icon: <SettingsIcon size={20} /> },
  ]

  return (
   <aside className="w-64 bg-white p-6 flex flex-col border-r">
    <div className="mb-8">
     {' '}
     <Logo />
    </div>

    <nav className="flex-1 space-y-1">
     {items.map((item) => {
      const isActive = selected === item.key
      const base =
       'flex items-center gap-3 px-4 py-3 rounded-lg cursor-pointer whitespace-nowrap
capitalize w-full'
      const inactive = 'text-gray-500 hover:bg-gray-50'
      const active = 'text-primary bg-primary-50 font-medium border-l-4 border-primary '
      return (
       <button
        key={item.key}
        type="button"
        className={`${base} ${isActive ? active : inactive}`}
        onClick={() => onSelect(item.key)}
       >
        {item.icon}
        {item.label}
       </button>
      )
     })}
    </nav>
   </aside>
  )
}
```

```
interface CoreProps {
  factors?: typeof MFA_FACTORS
  backupFactor?: typeof MFA_BACKUP_FACTOR
  policyOptions?: typeof MFA_POLICY_OPTIONS
  onFactorToggle?: (id: string, enabled: boolean) => void
  onPolicyChange?: (value: string) => void
}

export default function Core({
  factors = MFA_FACTORS,
  backupFactor = MFA_BACKUP_FACTOR,
  policyOptions = MFA_POLICY_OPTIONS,
  onFactorToggle,
  onPolicyChange,
}: CoreProps = {}): JSX.Element {
  const handleFactorToggle = (id: string, enabled: boolean) => {
    console.log(`Factor ${id} toggled to ${enabled}`)
    onFactorToggle?.(id, enabled)
  }

  const handlePolicyChange = (value: string) => {
    console.log(`Policy changed to ${value}`)
    onPolicyChange?.(value)
  }

  return (
    <div className="w-full space-y-6">
      <div>
        <h1 className="text-2xl font-bold text-gray-900">Multi-Factor Authentication</h1>
        <p className="text-sm text-gray-500 mt-1">
          Secure your account with multiple authentication methods
        </p>
      </div>

      <div className="flex flex-col w-full items-start gap-4">
        <FactorsSection
          factors={factors}
          backupFactor={backupFactor}
          onFactorToggle={handleFactorToggle}
        />
        <PolicySection options={policyOptions} onPolicyChange={handlePolicyChange} />
      </div>
    </div>
  )
}


interface FactorCardProps {
  factor: MFAFactor | MFABackupFactor
```

```
  onToggle?: (id: string, enabled: boolean) => void
}

export function FactorCard({ factor, onToggle }: FactorCardProps): JSX.Element {
  const hasInfo = 'hasInfo' in factor && factor.hasInfo

  const handleToggle = (checked: boolean) => {
    onToggle?.(factor.id, checked)
  }

  return (
    <div className="flex w-full items-center gap-6 px-6 py-4 rounded-xl border border-solid
border-border">
      <div className="flex flex-col items-start justify-center gap-1 flex-1">
        <div className="inline-flex items-center gap-3">
          <img className="w-5 h-5" alt={factor.title} src={factor.icon} />
          <div className="inline-flex items-center gap-1">
            <span className="text-sm font-medium text-gray-900 whitespace-
nowrap">{factor.title}</span>
            {hasInfo && <AlertCircleIcon className="w-4 h-4 text-gray-500" />}
          </div>
        </div>
        <p className="self-stretch text-sm text-gray-500">{factor.description}</p>
      </div>
      <Switch defaultChecked={factor.enabled} onCheckedChange={handleToggle} />
    </div>
  )
}


interface FactorsSectionProps {
  factors: MFAFactor[]
  backupFactor: MFABackupFactor
  onFactorToggle?: (id: string, enabled: boolean) => void
}

export function FactorsSection({
  factors,
  backupFactor,
  onFactorToggle,
}: FactorsSectionProps): JSX.Element {
  return (
    <Card className="w-full bg-white rounded-xl border-0 shadow-none">
      <CardContent className="flex gap-[54px] px-8 py-6">
        <SectionHeader
          title="FACTORS"
          description="Users need to verify one of the enabled factors for 2-step verification."
          learnMoreLink="#"
        />

        <div className="flex flex-col gap-4 flex-1">
```

```
        <div className="flex flex-col items-start gap-2 w-full">
         <div className="flex flex-col items-start gap-[3px] w-full">
          <Label className="text-sm font-medium text-gray-900">Multi-factors</Label>
         </div>

         {factors.map((factor) => (
          <FactorCard key={factor.id} factor={factor} onToggle={onFactorToggle} />
         ))}
        </div>

        <div className="flex flex-col items-start gap-2 w-full">
         <p className="self-stretch text-sm text-gray-500">
          When users can't verify the above MFA factors, use the backup option.
         </p>
         <FactorCard factor={backupFactor} onToggle={onFactorToggle} />
        </div>
       </div>
      </CardContent>
     </Card>
    )
}


const tabItems = [
  { value: 'desktop', label: 'Desktop', active: true },
  { value: 'mobile', label: 'Mobile', active: false },
]

export const SignInFrameSection = (): JSX.Element => {
  return (
    <section className="flex flex-col h-[823px] rounded-xl border text-card-foreground shadow
col-span-7 p-6 bg-white overflow-visible">
      <header className="w-full bg-white">
       <div className="flex items-center justify-between">
        <div className="text-xs font-bold text-gray-500 uppercase tracking-wider">
         SIGN IN PREVIEW
        </div>

        <div className="flex items-center gap-2">
         <Button
          variant="outline"
          size="icon"
          className="h-auto p-2 bg-white rounded-lg border-input"
         >
          <SunIcon className="w-4 h-4" />
         </Button>

         <Button
          variant="outline"
          className="h-auto w-[100px] justify-between px-2 py-1.5 rounded-lg border-input"
         >
```

```
          <span className="flex-1 text-left text-sm text-gray-900 overflow-hidden text-ellipsis
whitespace-nowrap">
            English
          </span>
          <ChevronDownIcon className="w-5 h-5 flex-shrink-0" />
        </Button>

        <Button
          variant="outline"
          className="h-auto gap-1.5 px-3 py-1.5 rounded-lg border-input"
        >
          <span className="text-sm text-gray-900">Live preview</span>
          <ExternalLinkIcon className="w-4 h-4" />
        </Button>
      </div>
    </div>

    <Tabs defaultValue="desktop" className="w-full px-[18px] mt-[26px]">
      <TabsList className="h-auto bg-transparent p-0 gap-4 border-b-0">
        {tabItems.map((tab) => (
          <TabsTrigger
            key={tab.value}
            value={tab.value}
            className={`h-auto flex flex-col items-start gap-1 px-1.5 py-0.5 rounded data-
[state=active]:bg-transparent data-[state=inactive]:bg-transparent relative ${
              tab.active
                ? 'after:absolute after:w-full after:h-[2px] after:bg-primary after:bottom-0 after:left-0'
                : ''
            }`}
          >
            <span
              className={`text-sm font-medium ${tab.active ? 'text-primary' : 'text-gray-600'}`}
            >
              {tab.label}
            </span>
          </TabsTrigger>
        ))}
      </TabsList>
    </Tabs>
  </header>

  <main className="flex-1 bg-gray-100 flex items-center justify-center">
    <div className="flex flex-col items-center gap-[19px]">
      <img src="/card.png" alt="Sign in preview" width={500} height={530} />
    </div>
  </main>
 </section>
)
}

export const SignInCardSection = (): JSX.Element => {
```

```
 const signUpAuthOptions = [
  { id: 'password', label: 'Create your password' },
  { id: 'verify', label: 'Verify at sign up' },
 ]

 return (
  <section className="flex flex-col w-full items-start gap-3">
   <Card className="w-full bg-white rounded-xl border-0 shadow-none">
    <CardContent className="rounded-xl border text-card-foreground shadow col-span-7 p-6
bg-white overflow-visible">
     <div className="flex flex-col items-start gap-6">
      <div className="text-xs font-bold text-gray-500 uppercase tracking-wider">SIGN
UP</div>

      <div className="w-full flex flex-col items-start gap-2">
       <div className="flex flex-col items-start gap-[3px] w-full">
        <div className="flex w-full items-center gap-[15px]">
         <div className="text-sm font-medium text-gray-900">Sign up identifier</div>
        </div>
        <div className="text-sm text-gray-500">
         Sign up identifier is mandatory information for account creation and has to be
         included in sign in.
        </div>
       </div>

       <div className="flex flex-col items-start w-full rounded-lg">
        <div className="flex items-center gap-1 pl-3 pr-2 py-2 w-full rounded-lg border
border-solid border-input">
         <div className="flex-1 text-sm text-gray-900">Email address</div>
         <ArrowRightIcon className="w-5 h-5 text-gray-900" />
        </div>
       </div>
      </div>

      <div className="w-full flex flex-col items-start gap-2">
       <div className="flex flex-col items-start gap-[3px]">
        <div className="flex w-full items-start gap-3">
         <div className="flex items-center gap-[15px] flex-1">
          <div className="text-sm font-medium text-gray-900">Sign up
authentication</div>
         </div>
        </div>
        <div className="text-sm text-gray-500">
         Users will go through all checked options as required steps in flow
        </div>
       </div>

       <div className="flex flex-col w-full items-start gap-3">
        {signUpAuthOptions.map((option) => (
         <div key={option.id} className="flex items-start gap-2">
          <Checkbox id={option.id} defaultChecked className="w-5 h-5 mt-0.5" />
```

```
            <label htmlFor={option.id} className="text-sm text-gray-900 cursor-pointer">
             {option.label}
            </label>
           </div>
          ))}
         </div>
        </div>
       </div>
      </CardContent>
     </Card>

     <Card className="w-full bg-white rounded-xl border-0 shadow-none">
      <CardContent className="rounded-xl border text-card-foreground shadow col-span-7 p-6
bg-white overflow-visible">
       <div className="flex flex-col items-start gap-6">
        <div className="text-xs font-bold text-gray-500 uppercase tracking-wider">SIGN
IN</div>

        <div className="flex flex-col items-start gap-2 w-full">
         <div className="flex flex-col items-start gap-[3px] w-full">
          <div className="text-sm font-medium text-gray-900">
           Sign in identifier and authentication
          </div>
          <div className="text-sm text-gray-500">
           Users can sign in using any of the options available. Adjust the layout by drag
           and dropping below options.
          </div>
         </div>

         <div className="flex flex-col items-start gap-3 w-full">
          <div className="flex items-center justify-center gap-2 w-full">
           <div className="flex items-center gap-[3px] px-2 py-3 flex-1 bg-gray-100 rounded-
lg">
            <div className="flex items-start gap-1">
             <GripVerticalIcon className="w-5 h-5 text-gray-500" />
             <div className="text-sm font-medium text-gray-900 w-[130px]">
              Email address
             </div>
            </div>
            <div className="flex items-start justify-between flex-1">
             <div className="flex items-start gap-2">
              <Checkbox defaultChecked className="w-5 h-5" />
              <div className="text-sm text-gray-900">Password</div>
             </div>
             <img
              className="w-5 h-5"
              alt="Arrows connection"
              src="/arrows---connection.svg"
             />
             <div className="flex items-start gap-2">
              <Checkbox defaultChecked className="w-5 h-5" />
```

```
              <div className="text-sm text-gray-900">Verification code</div>
             </div>
            </div>
           </div>
           <button className="w-5 h-5 flex items-center justify-center">
            <MinusIcon className="w-5 h-5 text-gray-900" />
           </button>
          </div>
          <button className="flex items-center gap-1 rounded">
           <PlusIcon className="w-4 h-4 text-primary" />
           <div className="text-sm font-medium text-primary">Add another</div>
          </button>
         </div>
        </div>
       </div>
      </CardContent>
    </Card>

    <Card className="w-full bg-white rounded-xl border-0 shadow-none">
     <CardContent className="rounded-xl border text-card-foreground shadow col-span-7 p-6
bg-white overflow-visible">
      <div className="flex flex-col items-start gap-6">
       <div className="text-xs font-bold text-gray-500 uppercase tracking-wider">
        SOCIAL SIGN IN
       </div>

       <div className="flex flex-col items-start gap-3 w-full rounded-2xl">
        <div className="flex flex-col items-start gap-6 w-full">
         <div className="flex flex-col items-start gap-2 w-full">
          <div className="flex flex-col items-start gap-[3px] w-full">
           <div className="flex flex-col items-start gap-[3px]">
            <div className="flex items-start gap-3 w-full">
             <div className="flex-1 text-sm font-medium text-gray-900">
              Social sign in
             </div>
            </div>
           </div>
           <div className="text-sm text-gray-500">
            Depending on the mandatory identifier you set up, your user may be asked to
            provide an identifier when signing up via social connector.
           </div>
          </div>

          <div className="flex items-center justify-center gap-2 w-full">
           <div className="flex items-center justify-between px-2 py-3 flex-1 bg-gray-100
rounded-lg">
            <GripVerticalIcon className="w-5 h-5 text-gray-500" />
            <div className="flex items-start gap-3 rounded-[5px]">
             <GoogleIcon />
             <div className="text-sm font-medium text-gray-900">Google</div>
            </div>
```

```
              <div className="flex-1" />
             </div>
             <button className="w-5 h-5 flex items-center justify-center">
              <MinusIcon className="w-5 h-5 text-gray-900" />
             </button>
            </div>

            <button className="flex items-center gap-1 rounded">
             <PlusIcon className="w-4 h-4 text-primary" />
             <div className="text-sm font-medium text-primary">Add another</div>
            </button>
           </div>
          </div>

          <div className="text-sm">
           <span className="text-gray-500">Not in the list? </span>
           <span className="text-primary cursor-pointer">Set up</span>
           <span className="text-gray-500"> other social connectors now.</span>
          </div>
         </div>
        </div>
       </CardContent>
      </Card>

      <Card className="w-full bg-white rounded-xl border-0 shadow-none">
       <CardContent className="rounded-xl border text-card-foreground shadow col-span-7 p-6
bg-white overflow-visible">
        <div className="flex flex-col items-start gap-6">
         <div className="text-xs font-bold text-gray-400 uppercase tracking-wider">
          ADVANCED OPTIONS
         </div>

         <div className="flex flex-col items-start gap-1 w-full">
          <div className="flex flex-col items-start gap-1 w-full">
           <div className="flex items-center gap-1 w-full">
            <div className="text-sm font-medium text-gray-900">Enable enterprise SSO</div>
           </div>
           <div className="flex flex-col items-start justify-center gap-1 w-full">
            <div className="flex items-center gap-6 p-4 w-full rounded-lg border border-solid
border-border">
             <div className="flex-1 font-en-body-body2 font-[number:var(--en-body-body2-
font-weight)] text-gray-900 text-[length:var(--en-body-body2-font-size)] tracking-[var(--en-
body-body2-letter-spacing)] leading-[var(--en-body-body2-line-height)] [font-style:var(--en-
body-body2-font-style)]">
              Enable users to sign into the application using Single Sign-On with their
              enterprise identities.
             </div>
             <Switch defaultChecked />
            </div>
           </div>
          </div>
```

```
          <div className="text-sm">
            <span className="text-gray-500">Go to &amp;quot;</span>
            <span className="text-primary cursor-pointer">Enterprise SSO</span>
            <span className="text-gray-500">
              &amp;quot; section to set up more enterprise connectors.
            </span>
          </div>
        </div>

        <div className="flex flex-col items-start gap-1 w-full">
          <div className="flex items-center gap-1 w-full">
            <div className="text-sm font-medium text-gray-900">Enable user
registration</div>
          </div>
          <div className="flex flex-col items-start justify-center gap-1 w-full">
            <div className="flex items-center gap-6 p-4 w-full rounded-lg border border-solid
border-border">
              <div className="flex-1 font-en-body-body2 font-[number:var(--en-body-body2-font-
weight)] text-gray-900 text-[length:var(--en-body-body2-font-size)] tracking-[var(--en-body-
body2-letter-spacing)] leading-[var(--en-body-body2-line-height)] [font-style:var(--en-body-
body2-font-style)]">
                Enable or disallow user registration. Once disabled, users can still be added in
                the admin console but users can no longer establish accounts through the sign-in
                UI.
              </div>
              <Switch defaultChecked />
            </div>
          </div>
        </div>
      </div>
    </CardContent>
  </Card>
 </section>
)
}

export default function SignInExperiencePage(): JSX.Element {
 return (
  <div className="w-full space-y-6">
   {/* Header */}
   <div>
    <h1 className="text-2xl font-bold text-gray-900">Sign-In Experience</h1>
    <p className="text-sm text-gray-500 mt-1">
     Customize the sign-in interface and authentication methods for your users
    </p>
   </div>

   {/* Content */}
   <div className="flex w-full items-start relative gap-4">
    <div className="flex-1 relative">
```

```
        <SignInCardSection />
       </div>
       <div className="flex-1 relative">
        <SignInFrameSection />
       </div>
     </div>
    </div>
  )
}

import { useEffect, useRef, useState } from "react";

export function useWebSocket(url) {
  const ws = useRef(null);
  const reconnectTimer = useRef(null);

  const [isConnected, setIsConnected] = useState(false);
  const [messages, setMessages] = useState([]);

  const connect = () => {
    ws.current = new WebSocket(url);

    ws.current.onopen = () => {
      console.log("WS connected");
      setIsConnected(true);
    };

    ws.current.onmessage = (event) => {
      console.log("WS message:", event.data);
      setMessages(prev => [...prev, event.data]);
    };

    ws.current.onerror = (e) => {
      console.error("WS error:", e);
    };

    ws.current.onclose = () => {
      console.warn("WS closed. Reconnecting in 3s...");
      setIsConnected(false);

      reconnectTimer.current = setTimeout(() => {
        connect();
      }, 3000);
    };
  };

  const sendMessage = (msg) => {
    if (ws.current && ws.current.readyState === WebSocket.OPEN) {
      ws.current.send(msg);
    } else {
      console.warn("WS not connected, cannot send");
```

```
  }
 };

 useEffect(() => {
  connect();

  return () => {
   if (reconnectTimer.current) clearTimeout(reconnectTimer.current);
   ws.current?.close();
  };
  // eslint-disable-next-line
 }, []);

 return {
  isConnected,
  messages,
  sendMessage
 };
}

import { useState } from "react";
import { useWebSocket } from "./useWebSocket";

export default function ChatComponent() {
 const { isConnected, messages, sendMessage } = useWebSocket("ws://localhost:8080/ws");
 const [input, setInput] = useState("");

 const handleSend = () => {
  sendMessage(input);
  setInput("");
 };

 return (
  <div style={{ padding: 20 }}>
   <h2>WebSocket Chat</h2>

   <p>
    Status:{" "}
    <b style={{ color: isConnected ? "green" : "red" }}>
     {isConnected ? "Connected" : "Disconnected"}
    </b>
   </p>

   <div
    style={{
     border: "1px solid #ccc",
     padding: 10,
     marginBottom: 10,
     height: 150,
     overflowY: "auto",
    }}
```

```
      >
        {messages.map((msg, i) => (
          <div key={i}>{msg}</div>
        ))}
      </div>

      <input
        style={{ padding: 5, width: "70%" }}
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Enter message"
      />
      <button onClick={handleSend} style={{ padding: 5, marginLeft: 10 }}>
        Send
      </button>
    </div>
  );
}
```