Міністерство освіти і науки України

Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерної інженерії та охоронних систем

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ХМАРНА ІНТЕЛЕКТУАЛЬНА ІНФОРМАЦІЙНА СИСТЕМА МОНІТОРИНГУ БЕЗПЕКИ ОБ'ЄКТА З ВІДЕОАНАЛІТИКОЮ ТА ПРОГНОЗУВАННЯМ РИЗИКІВ

CLOUD-BASED INTELLIGENT INFORMATION SYSTEM FOR FACILITY SECURITY MONITORING WITH VIDEO ANALYTICS AND RISK PREDICTION

спеціальність 126 Інформаційні системи та технології
(шифр і назва спеціальності)

освітня програма «Інформаційні системи та технології охорони і безпеки»
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи ІСТО-41
МАЛЬЧЕВСЬКИЙ Антон Віталійович
_____
(підпис)

Керівник:
к.т.н., доцент
КОСТЮЧКО Сергій Миколайович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
к.т.н., доцент
ТЕРЛЕЦЬКИЙ Тарас Володимирович
_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет: *комп'ютерних та інформаційних технологій*
Кафедра: *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *126 Інформаційні системи та технології*
Освітня програма: «*Інформаційні системи та технології охорони і безпеки*»

ЗАТВЕРДЖУЮ
Завідувач кафедри КІБ
к.т.н., доцент Терлецький Т. В.
_____
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

МАЛЬЧЕВСЬКОГО Антона Віталійовича
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: *Хмарна інтелектуальна інформаційна система моніторингу безпеки об'єкта з відеоаналітикою та прогнозуванням ризиків (комплексна робота з Ковш Ю. Г.).*

Керівник роботи: *к.т.н., доцент Костючко Сергій Миколайович*
затверджені наказом закладу вищої освіти від «16» грудня 2026 р. № 529/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи: «30» травня 2026 р.

3. Вихідні дані до роботи: Джерелами для виконання роботи слугують вітчизняна та зарубіжна науково-технічна література, публікації у фахових періодичних виданнях за темою дослідження, а також спеціалізовані електронні ресурси технічного спрямування.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *Анотація. Вступ. Розділ 1. Аналітичний огляд стану предметної області (архітектура серверних систем моніторингу безпеки, хмарні платформи для обробки відеоданих, Інтелектуальні методи відеоаналізу в системах безпеки, Аналіз програмних рішень та AI-моделей для виявлення ризиків, формування завдання та вимог до проектованої системи) Розділ 2. Обґрунтування вибору засобів та методів реалізації (обгрунтування вибору серверної архітектури, обгрунтування вибору AI-моделей для відеоаналітики, обрунтування обгрунтування мотодів обробки та зберігання подій, вибір технології прогнозування ризиків Розділ 3. Практична реалізація (загальна архітектура серверної частини системи, проектування та реалізація модуля відеоаналітики, реалізація логіки формування подій безпеки, реалізація модуля прогнузування та оцінювання ризиків, тестування серверної частини системи) Загальні висновки та рекомендації.Список використаних джерел. Додатки.*

5. Перелік графічного (ілюстративного) матеріалу Презентація на 15 слайдах

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Розділ 1 Аналітичний огляд стану предметної області	*Костючко С. М.*		
Розділ 2 Обґрунтування вибору засобів та методів реалізації	*Костючко С. М.*		
Розділ 3 Практична реалізація	*Костючко С. М.*		
Загальні висновки та рекомендації	*Костючко С. М.*		
Нормоконтроль	*Кайдик О. Л.*		
Гарант ОП	*Терлецький Т. В.*		
Показник запозичень тексту			
Академічна доброчесність	*Кайдик О. Л.*		

7. Дата видачі завдання: «16» грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1.	Обґрунтування теми	До 12.12.2025 р.	
2.	Огляд літератури із досліджуваної проблеми	До 12.12.2025 р.	
3.	Розділ 1 Аналітичний огляд стану предметної області	До 28.02.2026 р.	
4.	Розділ 2 Обґрунтування вибору засобів та методів реалізації	До 31.03.2026 р.	
5	Розділ 3 Практична реалізація	До 30.04.2026 р.	
6.	Загальні висновки та рекомендації	До 16.05.2026 р.	
7.	Формування списку використаних джерел	До 20.05.2026 р.	
8.	Формування додатків.	До 20.05.2026 р.	
9.	Формування презентації за темою кваліфікаційної роботи	До 20.05.2026 р.	
10.	Нормоконтроль	До 21.05.2026 р.	
11.	Інструментальна перевірка на академічний плагіат	До 22.05.2026 р.	
12.	Представлення кваліфікаційної роботи бакалавра до захисту	До 02.06.2026 р.	

Здобувач вищої освіти _____ (*Мальчевський А. В.*)
(підпис)

Керівник кваліфікаційної роботи _____ (*Костючко С. М.*)
(підпис)

АНОТАЦІЯ

Мальчевський А. В. Хмарна інтелектуальна інформаційна система моніторингу безпеки об’єкта з відеоаналітикою та прогнозуванням ризиків. Рукопис.

Кваліфікаційна робота бакалавра ОП «Інформаційні системи та технології охорони і безпеки» Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційну роботу присвячено розробці хмарної мікросервісної системи інтелектуального моніторингу безпеки об’єкта з відеоаналітикою та прогнозуванням ризиків на основі Python-екосистеми. У роботі проведено аналіз сучасних архітектур серверних систем відеоспостереження, хмарних платформ обробки відеоданих, методів комп’ютерного зору та відеоаналітики, а також програмних рішень для виявлення та оцінювання безпекових ризиків. Обґрунтовано вибір мікросервісної хмарно-периферійної архітектури та технологічного стека: асинхронного веб-фреймворку FastAPI, брокера повідомлень RabbitMQ, реляційної СУБД PostgreSQL, системи контейнеризації Docker. Розроблено чотирикомпонентну серверну архітектуру, що включає сервіс вилучення кадрів, аналітичний AI-сервіс, основний серверний сервіс та клієнтський застосунок. Реалізовано багатоетапний конвеєр відеоаналітики: адаптивне виявлення руху на основі експоненціального ковзного середнього, детектування об’єктів нейронною мережею YOLOv8, Intersection over Union - трекінг (IoU) з прогнозуванням позиції об’єктів між кадрами та аналіз зональної приналежності алгоритмом ray casting. Розроблено гібридний рушій оцінювання ризиків із вісьмома типами подій, чотирирівневою шкалою ризику та механізмом часових розкладів. Тестування на трьох рівнях.

Ключові слова: мікросервісна архітектура, відеоаналітика, YOLOv8, об’єктне відстеження, оцінювання ризиків, хмарні технології, FastAPI, RabbitMQ, Docker, ray casting.

ANNOTATION

Malchevsky A. Cloud-based intelligent information system for facility security monitoring with video analytics and risk prediction. Manuscript.

Bachelor’s qualification work EP «Information systems and technologies for security and safety» Lutsk National Technical University. Lutsk, 2026.

The qualification work is devoted to the development of a cloud-based microservice system for intelligent monitoring facility security with video analytics and risk prediction based on the Python ecosystem. The paper analyzes modern architectures of video surveillance server systems, cloud video data processing platforms, computer vision and video analytics methods, as well as software solutions for detecting and assessing security risks. The choice of a microservice cloud-peripheral architecture and technological stack is justified: asynchronous web framework FastAPI, message broker RabbitMQ, relational DBMS PostgreSQL, containerization system Docker. A four-component server architecture was developed, including a frame extraction service, an analytical AI service, a main server service, and a client application. A multi-stage video analytics pipeline was implemented: adaptive motion detection based on exponential moving average, object detection using a YOLOv8 neural network, IoU tracking with prediction of object positions between frames, and analysis of zonal affiliation using a ray casting algorithm. A hybrid risk assessment engine with a higher event type, a four-level risk scale, and a timeline mechanism has been developed. Testing at three levels.

Keywords: microservice architecture, video analytics, YOLOv8, object tracking, risk assessment, cloud technologies, FastAPI, RabbitMQ, Docker, ray casting.

ЗМІСТ

ВСТУП

Зростання кількості загроз фізичній безпеці об’єктів, розширення периметрів охорони та обмежені можливості ручного відеоспостереження зумовлюють критичну потребу у впровадженні інтелектуальних автоматизованих систем моніторингу. Традиційні системи відеоспостереження вимагають постійної присутності операторів для аналізу відеопотоку. Це призводить до впливу «людського фактора» та швидкої втоми персоналу, що різко знижує ефективність оперативного реагування на загрози та неминуче призводить до пропуску інцидентів.

Сучасні методи глибокого навчання, зокрема алгоритми сімейства YOLO, дозволяють виявляти та класифікувати об’єкти у відеопотоці в реальному часі з високою точністю. Однак промислове впровадження таких рішень стикається з низкою проблем: складнощами масштабованості при обробці великих обсягів даних, забезпеченням відмовостійкості, прозорістю прийняття рішень та специфікою розгортання в хмарних і гібридних середовищах. Необхідність розробки комплексної, готової до розгортання серверної платформи, що інтегрує сучасну відеоаналітику, підсистему оперативного сповіщення та інтелектуальний рушій оцінювання ризиків у єдиному відмовостійкому програмному комплексі, визначає високу актуальність даної кваліфікаційної роботи.

Об’єкт дослідження – процес автоматизованого відеоаналізу для виявлення, класифікації та оцінювання загроз фізичній безпеці на промислових і комерційних об’єктах.

Предмет дослідження – методи та засоби реалізації серверної частини хмарної інтелектуальної системи моніторингу безпеки з відеоаналітикою.

Мета кваліфікаційної роботи – розробка хмарної мікросервісної системи моніторингу безпеки об’єкта з відеоаналітикою на базі нейронної мережі YOLOv8 та гібридним рушієм оцінювання і прогнозування ризиків, що забезпечує автоматичне виявлення подій безпеки в режимі реального часу.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД СТАНУ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Архітектура серверних систем моніторингу безпеки

Серверна частина системи відеоспостереження відповідає за приймання відеопотоків від камер, їхню обробку, надійне зберігання та надання доступу операторам або суміжним системам. З розвитком цифрових технологій та підвищенням роздільної здатності камер обсяг відеоданих суттєво зріс, що спонукало архітекторів систем переглядати усталені підходи до побудови серверної частини [1].

На початку розвитку IP-відеоспостереження домінувала централізована клієнт-серверна модель. Всі камери передавали відео на один сервер, який паралельно виконував декодування, запис на диски, управління правами та резервне копіювання. Простота такого підходу є очевидною, але при зростанні кількості камер виникають серйозні проблеми з масштабованістю та відмовостійкістю – вихід із ладу єдиного сервера паралізує всю систему, що неприйнятно для об’єктів з підвищеними вимогами до безпеки [2, 3].

Прагнення підвищити надійність систем призвело до поширення розподілених архітектур. У них обробка відеоданих рознесена між кількома серверами, що дозволяє автоматично балансувати навантаження та нарощувати потужність шляхом додавання нових вузлів [4].

Логічним продовженням цього напрямку стала мікросервісна архітектура, яка передбачає декомпозицію великого застосунку на набір невеликих незалежних сервісів. Для системи відеоспостереження це означає виокремлення окремих модулів для приймання відео, роботи з базою даних, аналітики та автентифікації. Кожен компонент можна розгортати, масштабувати та оновлювати незалежно від інших [5].

Паралельно зі змінами в архітектурі серверів відбувався розвиток функцій інтелектуальної обробки відео. Сучасні системи здатні автоматично виявляти рухомі об’єкти, розпізнавати обличчя, класифікувати поведінку та формувати

структуровані події безпеки – задачі, для вирішення яких раніше потрібні були спеціально навчені оператори [6].

Продуктивна обробка нейромережевих алгоритмів вимагає значних обчислювальних ресурсів. Тому сучасні сервери відеоаналітики часто оснащуються графічними прискорювачами GPU архітектури CUDA, які завдяки масовому паралелізму добре пристосовані до матричних операцій, що лежать в основі роботи згорткових нейронних мереж [7].

1.2 Хмарні платформи для обробки відеоданих

Хмарні технології суттєво змінили підхід до побудови інфраструктури відеоспостереження. Замість придбання та обслуговування власного обладнання організації отримали можливість орендувати обчислювальні ресурси в міру потреби, переводячи капітальні витрати на операційні [8].

Провідні хмарні провайдери пропонують спеціалізовані сервіси для відеоаналітики. Amazon Web Services надає Kinesis Video Streams для прийому та буферизації відеопотоків у реальному часі та Rekognition Video для розпізнавання облич, об’єктів і сцен без необхідності розгортання власних моделей. Google Cloud Video Intelligence API забезпечує автоматичне маркування об’єктів, визначення часових сегментів та виявлення явного вмісту. Microsoft Azure Video Analyzer поєднує крайову обробку з хмарним зберіганням, дозволяючи розгортати модулі аналітики безпосередньо на периферійних пристроях [9].

Технологічним фундаментом хмарних систем є контейнеризація та оркестрація. Docker забезпечує ізоляцію сервісів у переносних контейнерах із усіма залежностями, а Kubernetes автоматично керує їхнім розгортанням, масштабуванням та самовідновленням у кластері [10, 11].

Окремої уваги заслуговує концепція крайових обчислень (edge computing) у контексті відеоспостереження. Передача необробленого відеопотоку до хмари вимагає значної пропускної здатності мережі та вносить затримку, неприйнятну

для систем безпеки реального часу. Тому сучасні рішення виконують попередню фільтрацію та детекцію безпосередньо на периферійних пристроях – таких як NVIDIA Jetson Orin або Intel NUC – і передають до хмари лише структуровані метадані подій. Це скорочує мережевий трафік на 95–98% порівняно з передачею повного відеопотоку.

Порівнюючи локальні та хмарні рішення, слід визнати переваги обох підходів. Хмара забезпечує гнучкість масштабування та скорочення початкових інвестицій, тоді як локальна інфраструктура пропонує повний контроль над даними та передбачувані затримки. Гібридний підхід дозволяє поєднати ці переваги: зберігати чутливі дані локально та задіювати хмарні ресурси для обчислювально інтенсивних задач. Порівняльний аналіз підходів до розгортання систем відеоспостереження наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз підходів до розгортання систем відеоспостереження

Критерій	Локальне розгортання	Хмарне розгортання	Гібридний підхід
Затримка обробки	Мінімальна (< 50 мс)	Висока (100–500 мс)	Низька на периферії
Початкові витрати	Високі	Мінімальні	Помірні
Масштабування	Обмежене апаратно	Необмежене	Гнучке
Приватність даних	Повний контроль	Залежить від провайдера	Чутливі дані локально
Відмовостійкість	Залежить від обладнання	Висока (SLA 99,9%)	Висока
Обслуговування	Потребує персоналу	Мінімальне	Помірне

1.3 Інтелектуальні методи відеоаналізу в системах безпеки

Трансформація камер відеоспостереження зі звичайних реєстраторів у аналітичні інструменти стала можливою завдяки прогресу в галузі машинного навчання та комп’ютерного зору. Сучасні нейромережі навчаються розпізнавати образи не за жорстко заданими правилами, а за прикладами, що дозволяє досягти значно кращих результатів у реальних умовах, ніж класичні алгоритми обробки зображень [12, 13].

Для виявлення об’єктів у відеопотоці реального часу найбільшого поширення набули алгоритми сімейства YOLO. Їхня ключова особливість – трактування детекції як задачі регресії: мережа за один прохід одночасно передбачає координати обмежувальних рамок та класи об’єктів, що обумовлює високу швидкодію при прийнятній точності [14].

Альтернативою є архітектура Single Shot MultiBox Detector (SSD), яка також виконує детекцію за один прохід, але будує передбачення на ознаках різного масштабу. Порівняно з двоетапними детекторами типу Faster R-CNN, обидва підходи виграють за швидкістю за незначних втрат у точності.

Виявлення об’єктів на окремих кадрах вирішує лише частину задачі. Щоб відстежувати конкретний об’єкт упродовж часу та аналізувати його траєкторію, застосовуються алгоритми багатоцільового відстеження (Multi-Object Tracking). Ключовим механізмом зіставлення детекцій між кадрами є метрика перетину над об’єднанням (IoU), яка оцінює ступінь збігу обмежувальних рамок [15].

Біометрична ідентифікація через розпізнавання облич доповнює детекцію верифікацією особистості. Система порівнює виявлене обличчя з базою облікових записів і надає доступ лише авторизованим особам. Проте застосування таких систем ставить питання захисту приватності та відповідності принципам пояснюваного штучного інтелекту [16].

Аналіз поведінки є найскладнішим напрямком сучасної відеоаналітики. На відміну від статичної класифікації, рекурентні архітектури та відеотрансформери аналізують послідовності кадрів, щоб виявляти поведінкові паттерни – незвичайні траєкторії руху, тривале перебування в забороненій зоні чи раптові зміни напрямку [17].

У поєднанні ці методи дозволяють перетворити сирий відеопотік на структурований журнал подій: хто, де і як довго перебував у кадрі. Сучасні системи здатні в реальному часі формувати сповіщення про відхилення від норми, суттєво скорочуючи час реакції персоналу на потенційні загрози. Для оператора це є якісним переходом від реактивного перегляду запису до проактивного моніторингу ситуації.

1.4 Аналіз програмних рішень та AI-моделей для виявлення ризиків

Ринок систем управління відеоспостереженням охоплює широкий спектр рішень: від великих комерційних платформ типу Milestone XProtect чи Genetec Security Center з розвиненою екосистемою плагінів до хмарних сервісів та відкритих проектів. Вибір залежить від масштабу об’єкта, вимог до інтеграції та бюджету.

Відкриті системи на кшталт ZoneMinder чи Frigate NVR надають повний контроль над кодовою базою та дозволяють налаштовувати систему під специфічні потреби без ліцензійних обмежень. Frigate, зокрема, орієнтована на локальну нейромережеву аналітику з мінімальними апаратними вимогами та нативну інтеграцію з платформою Home Assistant [18].

Впровадження штучного інтелекту кардинально змінює роль систем відеоспостереження: із пасивного реєстратора вони перетворюються на проактивні системи попередження загроз. Замість фіксації вже скоєного інциденту, інтелектуальна система виявляє передвісники небезпеки та сповіщає операторів завчасно [19].

Водночас машинне навчання має практичні обмеження. Найістотніше з них – проблема хибних тривог: модель може помилково класифікувати нешкідливий об’єкт як загрозу. У середовищах з динамічним освітленням або нетиповими ракурсами хибні спрацьовування можуть суттєво знизити довіру операторів до системи.

Серйозну загрозу становлять адверсаріальні атаки на нейромережеві компоненти: зловмисники можуть використовувати спеціально сформовані візуальні артефакти для обходу детектора. Це вимагає комплексного підходу до захисту серверної інфраструктури [20].

Поряд із технічними аспектами, розробники та оператори систем відеоспостереження зобов’язані дотримуватися вимог законодавства про захист персональних даних. В Євросоюзі GDPR суворо регламентує умови збору,

зберігання та обробки біометричних даних, отримання інформованої згоди суб’єктів та визначення відповідальних осіб [21, 22].

1.5 Постановка завдань на кваліфікаційну роботу бакалавра

У межах даної роботи розробляється серверна система моніторингу безпеки на основі Python. Основною метою є створення програмного рішення для автоматизованого аналізу відеокадрів з метою виявлення потенційно небезпечних ситуацій та фіксації подій без постійного ручного контролю оператором.

Система повинна забезпечувати безперебійне приймання відеокадрів від клієнтських застосунків, їхню потокову обробку в режимі, максимально наближеному до реального часу, та автоматичне розпізнавання наперед заданих класів об’єктів. У разі виявлення цільової події система має автоматично формувати запис у базі даних та генерувати відповідне системне сповіщення для подальшої обробки або візуалізації в інтерфейсі користувача. Окремою вимогою є забезпечення можливості роботи з архівом, що включає інструменти швидкого пошуку інцидентів за часовими мітками або типами виявлених об’єктів.

Дані між вузлами системи передаються виключно через захищені криптографічні протоколи. Доступ надається виключно через JSON Web Token - автентифікацію (JWT) з рольовим розмежуванням прав. Облікові дані та ключі не зберігаються у відкритому вигляді в конфігураційних файлах чи змінних середовища.

Система використовує конвеєр безперервної інтеграції Continuous Integration (CI), який автоматично перевіряє якість коду, запускає тести та аналізує залежності на відомі вразливості. Успішна збірка завершується формуванням артефакту, готового до розгортання.

Система розрахована на обробку інтенсивних потоків відеоданих і може горизонтально масштабуватись без зміни програмного ядра. Вибір мови Python як основного інструменту розробки повністю задовольняє ці вимоги завдяки

наявності потужної екосистеми бібліотек для роботи з комп’ютерним зором та машинним навчанням. Це дозволяє гнучко реалізувати аналітичний модуль та забезпечує можливість контейнерного розгортання як на локальних серверах підприємства, так і в сучасних хмарних середовищах.

Виходячи із загальної мети та сформульованих вимог, визначаються такі конкретні завдання, які мають бути поетапно вирішені під час виконання роботи:

1) провести комплексний аналіз існуючих архітектур серверних систем моніторингу безпеки, визначивши їхні ключові переваги та функціональні обмеження;

2) обґрунтувати вибір технологічного стека для програмної реалізації системи інтелектуального відеоаналізу на основі екосистеми Python;

3) розробити архітектуру серверного застосунку з використанням високопродуктивного веб-фреймворку FastAPI для швидкої обробки відеокадрів та маршрутизації подій;

4) реалізувати аналітичний модуль детекції об’єктів на основі згорткової нейронної мережі архітектури YOLO, оптимізований для роботи в реальному часі;

5) спроєктувати реляційну схему бази даних та забезпечити надійне збереження метаданих про події з можливістю подальшого пошуку по архіву;

6) впровадити надійні механізми автентифікації (наприклад, на базі JWT-токенів) та розмежування прав доступу користувачів для мінімізації ризиків несанкціонованого втручання;

7) забезпечити кросплатформність та простоту розгортання спроєктованої системи за допомогою технологій контейнеризації, підготувавши її для ізольованого запуску в будь-якому середовищі.

РОЗДІЛ 2
ОБҐРУНТУВАННЯ ВИБОРУ ЗАСОБІВ ТА МЕТОДІВ РЕАЛІЗАЦІЇ

2.1 Обґрунтування вибору серверної архітектури

Серверна частина системи відеоспостереження з інтелектуальним аналізом повинна одночасно виконувати різнорідні задачі: декодування та маршрутизацію відеопотоків, нейромережеву аналітику, збереження подій у базі даних та обслуговування REST/WebSocket-запитів від клієнтів. Традиційна монолітна архітектура погано підходить для цих задач через неможливість незалежного масштабування компонентів із різним навантаженням.

Мікросервісний підхід вирішує цю проблему завдяки чіткому розподілу відповідальності між незалежними компонентами – принцип Single Responsibility, поширений з рівня класів на рівень сервісів. Це означає, що вузьке місце в AI-сервісі не впливає на роботу маршрутизатора відеопотоків, а оновлення бізнес-логіки не вимагає перезапуску аналітичного конвеєра. Структурну схему мікросервісної архітектури системи наведено на рисунку 2.1. Порівняльний аналіз монолітної та мікросервісної архітектур за ключовими критеріями наведено у таблиці 2.1.

Таблиця 2.1 – Порівняльний аналіз архітектурних підходів

Критерій	Монолітна архітектура	Мікросервісна архітектура
Відмовостійкість	Збій у модулі аналітики зупиняє запис відео та роботу веб-інтерфейсу	Ізольованість процесів: падіння аналітики не впливає на маршрутизацію відео
Масштабування	Вимагає збільшення потужності всього сервера (вертикальне)	Дозволяє клонувати лише високонавантажені вузли, наприклад AI-сервіс (горизонтальне)
Технологічний стек	Жорстка прив’язка до однієї мови програмування	Можливість використання оптимальних технологій для кожної підзадачі
Оновлення	Потребує повної зупинки системи	Дозволяє оновлювати окремі сервіси без переривання загального моніторингу

Окремої уваги заслуговує рішення виокремити відеошлюз у самостійний компонент на базі MediaMTX. IP-камери працюють за протоколом RTSP, але мікросервіси системи потребують доступу до окремих кадрів, а не до

безперервного потоку. MediaMTX приймає RTSP-потоки від камер та перетворює їх у формати, зручні для подальшої обробки, включно з HLS та WebRTC.

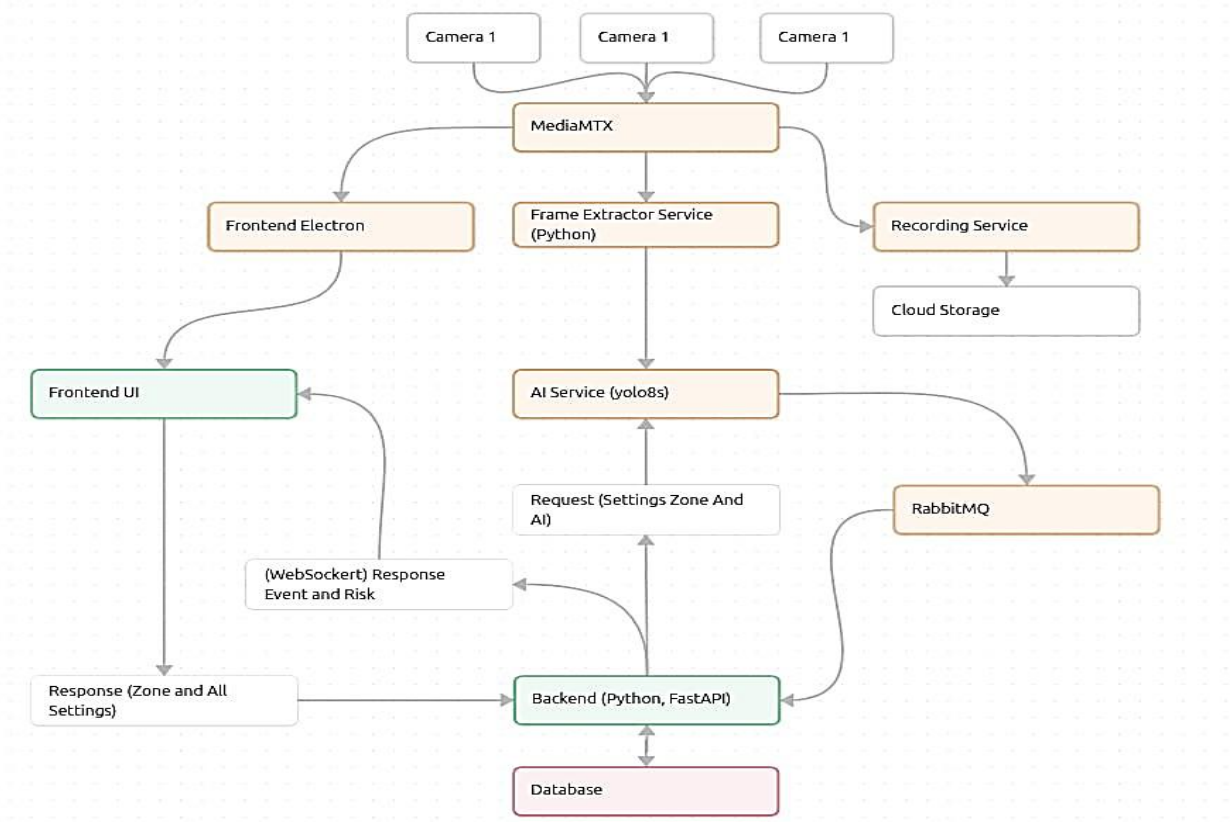

Рисунок 2.1 – Структурна схема мікросервісної архітектури системи

Відокремлення сервісу попередньої обробки кадрів також є архітектурно обґрунтованим. Декодування відеопотоку та виявлення руху є ресурсоємними операціями, які вигідно виконувати на периферійному вузлі поблизу джерел відео – це зменшує мережевий трафік та затримку передачі кадрів на аналіз [23].

Для реалізації API та AI-сервісу обрано FastAPI – фреймворк на основі стандарту ASGI, що нативно підтримує асинхронне програмування Python. Асинхронна модель виконання дозволяє одночасно обслуговувати велику кількість WebSocket-з’єднань та HTTP-запитів без блокування потоку при операціях введення-виведення [24].

Для міжсервісної взаємодії в асинхронній системі покладатися лише на синхронні REST-виклики недоцільно – при пікових навантаженнях черга до AI-

сервісу може зростати швидше, ніж він встигає обробляти запити. Брокер повідомлень RabbitMQ вирішує цю проблему, буферизуючи події між виробниками та споживачами та гарантуючи їхню доставку навіть при тимчасовому збої сервісу [25].

Вибір підсистеми зберігання даних ґрунтується на дворівневій моделі. Для довготривалого зберігання структурованих даних про події та конфігурацій зон обрано PostgreSQL – об’єктно-реляційну СУБД із широкими можливостями індексування та підтримкою типу JSONB для напівструктурованих метаданих. Для кешування конфігурацій із коротким терміном актуальності використовується in-memory сховище [26, 27].

Docker-контейнеризація ізолює кожен сервіс разом із його залежностями в переносний образ, усуваючи проблему несумісності середовищ і спрощуючи відтворюване розгортання. Docker Compose декларативно описує склад усіх сервісів, їхні залежності та порядок запуску, що дозволяє запустити весь стек однією командою.

Обраний технологічний стек – мікросервісна архітектура з асинхронним API-рівнем, брокером повідомлень та гібридним зберіганням даних – відповідає вимогам системи щодо пропускної здатності, відмовостійкості та незалежного масштабування окремих компонентів.

2.2 Обґрунтування вибору AI-моделей для відеоаналітики

Центральним завданням інтелектуального рівня є виявлення об’єктів у відеопотоці та відстеження їхніх траєкторій у часі. Від точності та швидкодії цих алгоритмів безпосередньо залежить якість сформованих подій безпеки, тому вибір моделі детекції є критично важливим архітектурним рішенням.

Для задачі детекції обрано сімейство YOLO, зокрема моделі YOLOv8n та YOLOv8s, що забезпечують прийнятний баланс між точністю та затримкою обробки. Обидві реалізують принцип одноетапної детекції: за один прохід через

нейронну мережу отримуємо координати всіх знайдених об’єктів та їхні класи [28].

Вибір детекційної моделі замість сегментаційної чи багатокадрової архітектури обумовлений вимогами до продуктивності. Детекційна модель повертає обмежувальні рамки та класи – достатньо для визначення факту присутності об’єкта та його розташування в кадрі. Сегментація та трекінг на рівні пікселів вимагають значно більших обчислювальних витрат без суттєвої переваги для задач зонального аналізу.

Порівняльний аналіз моделей сімейства YOLO за показниками обчислювальної складності та точності наведено в таблиці 2.2. Вибір версій nano та small пояснюється орієнтацією на розгортання без виділеного GPU, де необхідно забезпечити обробку 2-5 кадрів на секунду на CPU [29].

Таблиця 2.2 – Порівняльний аналіз швидкодії та точності моделей сімейства YOLO

Модель	Параметрів (млн)	FLOPs (млрд)	mAP 50-95	Цільове забезпечення
YOLO Nano	3,2	8,7	37,3	Вбудовані системи, CPU, Edge-сервери
YOLO Small	11,2	28,6	44,9	Офісні сервери, базові GPU
YOLO Medium	25,9	78,9	50,2	Потужні GPU-кластери

Використання попередньо навченої моделі суттєво скорочує час підготовки системи. Замість навчання з нуля, достатньо донавчити модель на вибірці конкретного об’єкта, що вимагає значно менших ресурсів для розмітки та обчислень.

Оскільки система аналізує безперервний відеопотік, для стабілізації результатів детекції застосовується алгоритм відстеження. Детектор може пропустити об’єкт на кількох кадрах через оклюзію або зміну освітлення, але трекер зберігає його ідентичність, спираючись на передбачену позицію.

Поєднання нейромережевої детекції та алгоритмічного трекінгу виправдане з огляду на продуктивність. Запуск важкої нейронної мережі для кожного кадру з метою трекінгу надмірно витратний; натомість легкий IoU-

алгоритм ефективно зіставляє детекції між послідовними кадрами за мінімальних обчислювальних витрат.

Перспективні архітектури на основі відеотрансформерів та 3D-згорток досягають вищої точності за рахунок аналізу часових послідовностей, однак їхня обчислювальна вимогливість унеможливлює розгортання без потужних GPU-кластерів.

Таким чином, одноетапна нейромережева детекція з сімейства YOLO у поєднанні з IoU-трекінгом є оптимальним вибором для системи безпекового моніторингу, що має функціонувати в умовах обмеженого апаратного забезпечення.

2.3 Обґрунтування методів обробки та зберігання подій

У розробленій системі інтелектуального відеоспостереження подія безпеки є центральною одиницею обміну даними між компонентами. Кожна подія несе вичерпну інформацію про виявлений інцидент і має бути надійно доставлена від AI-сервісу до бекенду для збереження та відображення оператору.

Середовище відеоаналітики генерує інтенсивний та нерівномірний потік подій: у спокійні години їх кількість мінімальна, а при виявленні одночасної активності від кількох камер – різко зростає. Пряма синхронна передача між сервісами в таких умовах призводить до накопичення затримок або втрати даних.

Брокер повідомлень RabbitMQ вирішує цю проблему, виступаючи буфером між виробниками та споживачами подій. Компонент Topic Exchange забезпечує гнучку маршрутизацію за шаблонами ключів, дозволяючи різним підписникам отримувати лише цікаві їм типи подій.

Формат даних у брокері суттєво впливає на ефективність системи. Замість необроблених зображень у черзі передаються лише структуровані метадані події, координати, клас об’єкта, рівень ризику та посилання на знімок. Це мінімізує навантаження на брокер та мережу.

Для зберігання подій безпеки вибір припав на PostgreSQL – об’єктно-реляційну СУБД із транзакційністю та широкими можливостями індексування. Реляційна схема дозволяє ефективно вибирати події за камерою, часовим діапазоном та рівнем ризику. Додатковою перевагою є нативна підтримка бінарного формату JSONB, що дозволяє реалізувати гібридну схему: фіксовані поля події зберігаються як реляційні стовпці з можливістю індексування, а змінна частина – у JSONB-полі без необхідності зміни схеми при додаванні нових типів подій [30].

PostgreSQL є обґрунтованим вибором для початкового розгортання та структурованого зберігання подій з можливістю складних реляційних запитів. Проте при масштабуванні до накопичення великих обсягів часових рядів варто розглянути міграцію на спеціалізовані time-series СУБД – TimescaleDB або InfluxDB – що забезпечують ефективніше стискання даних та нативні функції агрегації по часових вікнах.

Окремо слід підкреслити, що для ефективної системи моніторингу безпеки замало просто надійно архівувати події на жорсткий диск; їх необхідно миттєво доводити до відома чергового оператора. Традиційні підходи, засновані на періодичному опитуванні сервера клієнтським застосунком, створюють надлишкове мережеве навантаження, генерують сотні марних запитів до бази даних і вносять неприпустиму штучну затримку. Тому у системі імплементовано повнодуплексний протокол WebSocket. Одразу після того, як основний бекенд успішно здійснює запис події у базу даних, диспетчер з’єднань автоматично розсилає цей інформаційний об’єкт усім активним сесіям операторів. Це забезпечує реакцію графічного інтерфейсу в межах мілісекунд, що є критичним показником для оперативного реагування на загрози.

Дублювання подій є серйозною проблемою потокової відеоаналітики: без механізму дедуплікації система генерувала б повторювані сигнали тривоги з частотою, що дорівнює частоті кадрів. Реалізований TTL-кеш відстежує останній час спрацьовування для кожної комбінації камери, типу події та ідентифікатора треку.

Подієво-орієнтований підхід поширюється не лише на інциденти, а й на управління конфігураціями зон: зміни, внесені через API, публікуються в окрему чергу, що дозволяє AI-сервісу оновлювати поведінку без перезапуску.

Інтеграція брокера повідомлень для маршрутизації подієвого потоку, реляційної СУБД з підтримкою JSONB для довготривалого зберігання та механізму TTL-дедуплікації формує надійну та масштабовану підсистему обробки даних безпекової системи.

2.4 Обґрунтування вибору технологій прогнозування ризиків

Виявлення об’єкта в кадрі є лише першим кроком. Система безпеки повинна автоматично оцінювати ступінь небезпеки ситуації та надавати оператору інтерпретований результат, а не лише технічні факти про присутність об’єктів у зонах.

З огляду на жорсткі вимоги до затримки та необхідність пояснюваних рішень, покластися виключно на глибоке навчання для фінальної оцінки ризику є недоцільним. Нейромережа є «чорним ящиком» і не може надати детального пояснення прийнятого рішення, що критично важливо для систем безпеки.

Компонент на основі правил (rule-based engine) дозволяє кодифікувати досвід фахівців із безпеки у формалізованих умовах спрацьовування. Такі правила зрозумілі, перевіряються та можуть бути змінені оператором без втручання в нейромережеву частину системи.

Ключова перевага правилового підходу абсолютна пояснюваність рішень. У сучасній інженерії систем штучного інтелекту принцип пояснюваного AI набуває все більшого значення: оператор повинен розуміти, чому система сформувала той чи інший сигнал тривоги.

Разом із тим безпекові ситуації далеко не завжди мають бінарний характер. Для поступово наростаючих загроз скупчення людей, тривалого перебування в зоні більш адекватною є бальна модель, яка накопичує ризик у часі та генерує подію лише при перевищенні порогу.

Суттєвою складовою рушія ризиків є механізм просторово-часового контекстного аналізу. Одна й та сама подія – наприклад, людина у зоні RESTRICTED – має різний рівень небезпеки вдень у робочий час та вночі у вихідний день. Механізм розкладів дозволяє автоматично коригувати рівень ризику залежно від часового контексту без втручання оператора.

На поточному етапі розробки свідомо відмовлено від рекурентних та трансформерних архітектур для прогнозування поведінки. Такі моделі потребують великих навчальних наборів із розмітками поведінкових паттернів та значних обчислювальних ресурсів для інференсу, що не відповідає поточним апаратним умовам.

Практична цінність обраного гібридного підходу, заснованого на правилах та бальній оцінці, полягає також у його високій конфігурованості. Налаштування геометричних меж зон, часових розкладів, порогів спрацювання лічильників та вагових коефіцієнтів ризику може виконуватися адміністратором «на льоту», без необхідності зупинки системи або дороговартісного перенавчання нейронних мереж. Це дає змогу адаптувати одне й те саме програмне рішення до кардинально різних умов експлуатації: від режиму роботи закритого складського комплексу до відкритого паркінгу або офісного центру.

Для задач оцінювання та прогнозування ризику в системі відеоспостереження гібридний підхід, що поєднує детерміновані правила з накопичувальною бальною моделлю, є обґрунтованим вибором – він забезпечує пояснюваність рішень та стійкість до шуму у вхідних даних.

Обґрунтовано вибір мікросервісної хмарно-периферійної архітектури, що складається з чотирьох базових компонентів: екстрактора кадрів, AI-сервісу, бекенду та фронтенду, а також трьох інфраструктурних елементів: MediaMTX, RabbitMQ та PostgreSQL.

РОЗДІЛ 3
ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Загальна архітектура серверної частини системи

Проектування серверної частини системи безпекового моніторингу вимагало балансування між суперечливими вимогами: висока пропускна здатність для одночасної обробки відеопотоків від кількох камер, мінімальна затримка від виявлення об’єкта до відображення події та надійність, що дозволяє переживати збій окремих компонентів без переривання спостереження.

З огляду на ці вимоги, для реалізації системи обрано мікросервісну архітектуру з асинхронним міжсервісним зв’язком. Мікросервісний підхід дозволяє масштабувати найбільш навантажені компоненти незалежно, а асинхронна взаємодія через брокер повідомлень розриває тимчасові зв’язки між виробниками та споживачами даних.

Серверна частина системи, загальну архітектуру якої показано на рисунку 3.1, складається з чотирьох основних мікросервісів та трьох інфраструктурних компонентів. Кожен сервіс відповідає за чітко визначену функцію та взаємодіє з іншими лише через визначені інтерфейси.

Сервіс вилучення кадрів – периферійний компонент, що розгортається поблизу джерел відео. Він підключається до IP-камер через MediaMTX, відбирає кадри з достатньою руховою активністю та передає їх у AI-сервіс для аналізу.

AI-сервіс є центральним аналітичним компонентом. Він виконує нейромережеву детекцію об’єктів, підтримує їхнє відстеження між кадрами та перевіряє виявлені об’єкти на відповідність правилам зон. Результати аналізу публікуються у чергу RabbitMQ.

Основний серверний сервіс реалізує бізнес-логіку та виступає API-шлюзом. Він зберігає події в PostgreSQL, транслює їх клієнтам через WebSocket та обслуговує REST-запити для управління конфігурацією зон і перегляду журналу подій.

Клієнтська частина реалізована як веб-застосунок на базі Node.js, Vite та фреймворку Electron. Її головна мета – відображення відеопотоків, управління зонами та перегляд журналу подій безпеки.

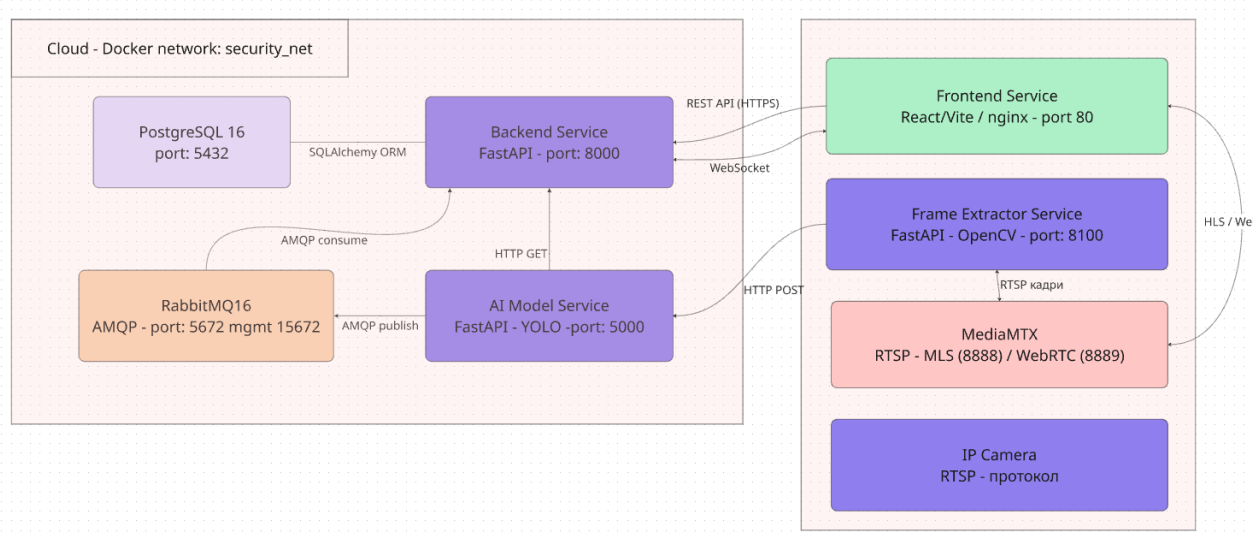

Рисунок 3.1 – Загальна архітектура серверної частини системи: діаграма мікросервісів із зазначенням протоколів взаємодії

Серед інфраструктурних компонентів ключову роль відіграють три елементи:

– MediaMTX сервер ретрансляції медіапотоків із підтримкою RTSP, HLS та WebRTC, який виступає центральним вузлом агрегації відеосигналів від різних камер;

– RabbitMQ брокер повідомлень на основі протоколу AMQP, що гарантує надійну асинхронну комунікацію між мікросервісами за патерном публікації та підписки;

– PostgreSQL реляційна система управління базами даних для постійного зберігання подій безпеки та конфігурацій зон.

Для реалізації серверних компонентів обрано Python 3.10 та FastAPI – фреймворк на основі стандарту ASGI із нативною асинхронною обробкою запитів. Висока продуктивність, автоматична генерація OpenAPI-документації та підтримка WebSocket роблять його підходящим для API-шлюзу системи.

Ключовими перевагами використання цього фреймворку для даного проєкту є:

– автоматична генерація OpenAPI-документації;

– нативна підтримка асинхронного програмування;

– висока продуктивність, що порівнянна з Node.js та Go.

Сервером застосунків обрано Uvicorn – ASGI-сервер з підтримкою HTTP/1.1, HTTP/2 та WebSocket. Він запускається у кількох робочих процесах для повного використання ядер процесора та забезпечує стабільну роботу FastAPI-застосунку під навантаженням. Повний технологічний стек серверної частини системи з версіями компонентів та їх призначенням наведено у таблиці 3.1.

Таблиця 3.1 – Технологічний стек серверної частини системи

Сервіс	Технологія	Версія	Призначення
Frame Extractor Service	Python, OpenCV, FastAPI, asyncio	Python 3.11, OpenCV 4.9	Підключення до RTSP-потоків, фільтрація руху, передача кадрів
AI Model Service	Python, Ultralytics YOLO, FastAPI, aio-pika	Python 3.11, YOLOv8 8.0	Детектування об’єктів, IoU-трекінг, аналіз зон, генерація подій
Backend Service	Python, FastAPI, SQLAlchemy 2.0, aio-pika	Python 3.11, FastAPI 0.110	REST API, збереження подій, WebSocket, споживання черги RabbitMQ
Frontend Service	Node.js, Vite, Electron	Node.js 20, Vite 5.0	Клієнтський застосунок, відображення відео та журналу подій
MediaMTX	Go (binary)	1.8.x	RTSP/HLS/WebRTC сервер ретрансляції відеопотоків
RabbitMQ	Erlang/OTP	3.12	Брокер повідомлень AMQP, Topic Exchange
PostgreSQL	C	16.x	Реляційна СУБД, зберігання подій та конфігурацій зон

Розгортання всіх сервісів організовано засобами Docker Compose, що декларативно описує склад стеку, змінні середовища, залежності між сервісами та умови запуску. Це дозволяє відтворити повне середовище на будь-якому хосту з встановленим Docker. Фрагмент Docker показано в лістингу 3.1

Лістинг 3.1 – Фрагмент `docker-compose.yml`: конфігурація сервісів backend та postgres, rabbitmq із залежностями

```
  postgres:
    image: postgres:16
    healthcheck:
      test: [“CMD-SHELL”, “pg_isready -U ${POSTGRES_USER:-postgres}”]
      interval: 5s
      retries: 5

  rabbitmq:
    image: rabbitmq:3-management-alpine
    healthcheck:
      test: [“CMD”, “rabbitmq-diagnostics”, “-q”, “ping”]
      interval: 10s
      retries: 5

  backend:
    build:
      context: ./cloud/backend_service/backend
    depends_on:
      postgres:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    ports:
      - “8000:8000”
    environment:
      DATABASE_URL:
postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@postgres:5432/${POSTGR
ES_DB}
      RABBITMQ_URL:
amqp://${RABBITMQ_USER}:${RABBITMQ_PASS}@rabbitmq:5672/
    networks:
      - security_net
```

Кінець лістингу 3.1

Для забезпечення правильного порядку запуску реалізовано перевірки готовності (health checks) між залежними сервісами. Бекенд та AI-сервіс стартують лише після отримання позитивної відповіді від PostgreSQL та RabbitMQ відповідно, що унеможливлює звернення до недоступних ресурсів.

Ключові параметри конфігурації AI-сервісу включають шлях до файлу моделі YOLOv8, порогові значення детектування (`DETECTION_CONFIDENCE = 0.4, DETECTION_IOU = 0.45`) та максимальну кількість об’єктів на кадр

(MAX_DETECTIONS = 50). Для модуля відстеження об’єктів передбачено налаштування максимального часу життя треку (TRACKER_MAX_AGE_SECONDS = 3.0), мінімальної кількості підтверджень (TRACKER_MIN_HITS = 2) та порогу перетину (TRACKER_IOU_THRESHOLD = 0.3). Окрім цього, визначаються URL-адреси зовнішніх інфраструктурних компонентів та час життя кешу конфігурації зон спостереження (ZONE_CACHE_TTL = 30). Своєю чергою, параметри сервісу вилучення кадрів (Frame Extractor) відповідають за оптимізацію обробки відеопотоку. Тут налаштовується цільова частота кадрів (DEFAULT_FPS = 2.0), ширина масштабування зображення (DEFAULT_RESIZE_WIDTH = 1280) та якість JPEG-стиснення (DEFAULT_JPEG_QUALITY = 95). Для забезпечення стабільності з’єднання передбачено параметр затримки повторного підключення до камери у разі обриву зв’язку (DEFAULT_RECONNECT_DELAY = 3).

Взаємодія між мікросервісами побудована на використанні двох незалежних каналів комунікації. Синхронний канал реалізовано через REST API (HTTP/JSON) і використовується переважно для операцій керування. До таких завдань належать виконання базових CRUD-операцій над зонами контролю через Backend API, отримання актуальної конфігурації зон AI-сервісом, а також виклик ендпоінту /api/v1/detect сервісом вилучення кадрів. Асинхронний канал побудовано на базі RabbitMQ Topic Exchange для обміну даними в реальному часі. Він забезпечує передачу потоку подій безпеки від AI-сервісу до Backend та розсилку сповіщень про зміни конфігурації зон у зворотному напрямку. Такий підхід забезпечує слабке зв’язування компонентів і відповідає архітектурним патернам розподілених систем.

3.2 Проектування та реалізація модуля відеоаналітики

Модуль відеоаналітики реалізований як багатоетапний конвеєр: кожен етап виконує строго визначену функцію та передає результати наступному. Така архітектура полегшує тестування та заміну окремих компонентів незалежно від

решти конвеєра. Загальну схему конвеєра обробки відеопотоку наведено на рисунку 3.2.

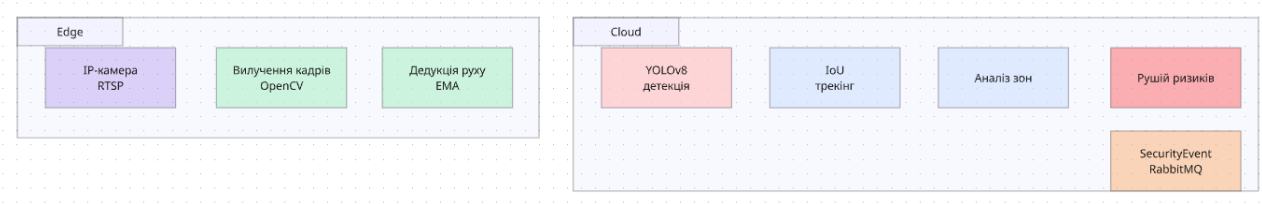

Рисунок 3.2 – Схема конвеєра обробки відеопотоку з позначенням рівня виконання коного етапу (периферія/хмара)

Сервіс вилучення кадрів розгортається на периферійному вузлі та встановлює RTSP-з’єднання з камерами через MediaMTX. Перед передачею кадри зменшуються до стандартної ширини для зниження навантаження на CPU при подальшій обробці наведено в лістингу 3.2.

Лістинг 3.2 – Клас MotionDetectorProcessor: ресайз кадру та виклик детектора руху

```
class MotionDetectorProcessor(IFrameProcessor):

    def __init__(self, motion_config: MotionConfig, resize_width: int) ->
None:
        self.resize_width = resize_width
        self._motion_detector =
MotionDetector(_schema_to_detector_config(motion_config))

    def process(
        self, frame: np.ndarray, current_time: float
    ) -> Tuple[bool, Optional[np.ndarray]]:
        processed = frame
        if self.resize_width > 0:
            h, w = processed.shape[:2]
            if w != self.resize_width:
                new_h = int(h * self.resize_width / w)
                processed = cv2.resize(processed, (self.resize_width,
new_h))

        should_send = (
            self._motion_detector.detect(processed, current_time)
            if self.motion_config_schema.enabled
```

```
            else True
        )
        return should_send, processed
```

Кінець лістингу 3.2

Для оптимізації обчислювальних ресурсів реалізовано модуль попередньої фільтрації кадрів на основі виявлення руху. Статичні кадри, де жоден піксель не змінився суттєво відносно моделі фону, відсіюються і не надсилаються на нейромережевий аналіз.

Виявлення руху відбувається шляхом обчислення різниці між поточним кадром та моделлю фону, застосування порогової бінаризації та морфологічних операцій для видалення шуму. Якщо площа зв'язних компонент перевищує встановлений поріг, кадр вважається таким, що містить рух, і передається на подальшу обробку. Робота алгоритму регулюється низкою параметрів: чутливістю детектування (`threshold_multiplier = 1.5`), розміром ядра розмиття (`blur_size = 7`), мінімальною кількістю послідовних кадрів із рухом для підтвердження (`min_consecutive_frames = 2`) та часом очікування після виявлення руху (`cooldown_seconds = 2.0`). Програмну реалізацію обчислення базового руху, а саме метод порівняння поточного кадру з моделлю фону на основі експоненційного ковзного середнього (EMA), наведено у лістингу 3.3.

Лістинг 3.3 – Метод `_compute_raw_motion`: порівняння кадру з моделлю фону (EMA)

```
def _compute_raw_motion(self, frame: np.ndarray) -> bool:
    cfg = self.config
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gray = cv2.GaussianBlur(gray, (cfg.blur_size, cfg.blur_size), 0)

    if self._background is None:
        self._background = gray.copy()
        return False

    diff = cv2.absdiff(self._background, gray)
    _, thresh = cv2.threshold(diff, cfg.diff_threshold, 255,
cv2.THRESH_BINARY)
```

```
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    thresh = cv2.dilate(thresh, kernel, iterations=cfg.dilate_iterations)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    total_area = 0
    significant = 0
    for c in contours:
        area = cv2.contourArea(c)
        if area < cfg.min_contour_area:
            continue
        hull_area = cv2.contourArea(cv2.convexHull(c))
        if hull_area > 0 and (area / hull_area) < cfg.min_solidity:
            continue
        total_area += area
        significant += 1

    return significant > 0 and total_area >= cfg.min_total_area
```

Кінець лістингу 3.3

Для контрольованого навантаження на downstream-сервіси реалізовано механізм обмеження частоти кадрів (frame throttling). Кожному процесору камери призначається цільова частота (FPS), і кадри, що надходять швидше за цільову частоту, пропускаються без обробки. Алгоритм порівнює часові мітки: якщо час від останньої обробки менший за (1.0 / target_fps), кадр пропускається. При виявленні руху таймер скидається, що дозволяє негайно обробити наступні кадри після події. За замовчуванням сервіс налаштований на 2 кадри на секунду для кожної камери – цей показник є компромісом між своєчасністю реагування та обчислювальним навантаженням.

Для детектування об’єктів у відеокадрах застосовується архітектура YOLOv8 від Ultralytics. Модель належить до класу одноетапних детекторів: вхідне зображення ділиться на сітку комірок, і мережа одночасно для всієї сітки передбачає обмежувальні рамки та класи об’єктів, що забезпечує низьку затримку інференсу порівняно з двоетапними підходами.

YOLOv8 підтримує виявлення широкого спектру об’єктів, зокрема людей та транспортних засобів, що відповідає основним сценаріям відеоспостереження. Архітектура мережі складається з трьох функціональних блоків: backbone

відповідає за витягування ознак із вхідного зображення, neck агрегує ознаки різних масштабів за допомогою механізму FPN, а head формує фінальні передбачення координат та класів об’єктів. Така структура забезпечує стійку роботу при різних масштабах об’єктів у кадрі. Архітектуру нейронної мережі YOLOv8 наведено на рисунку 3.3.

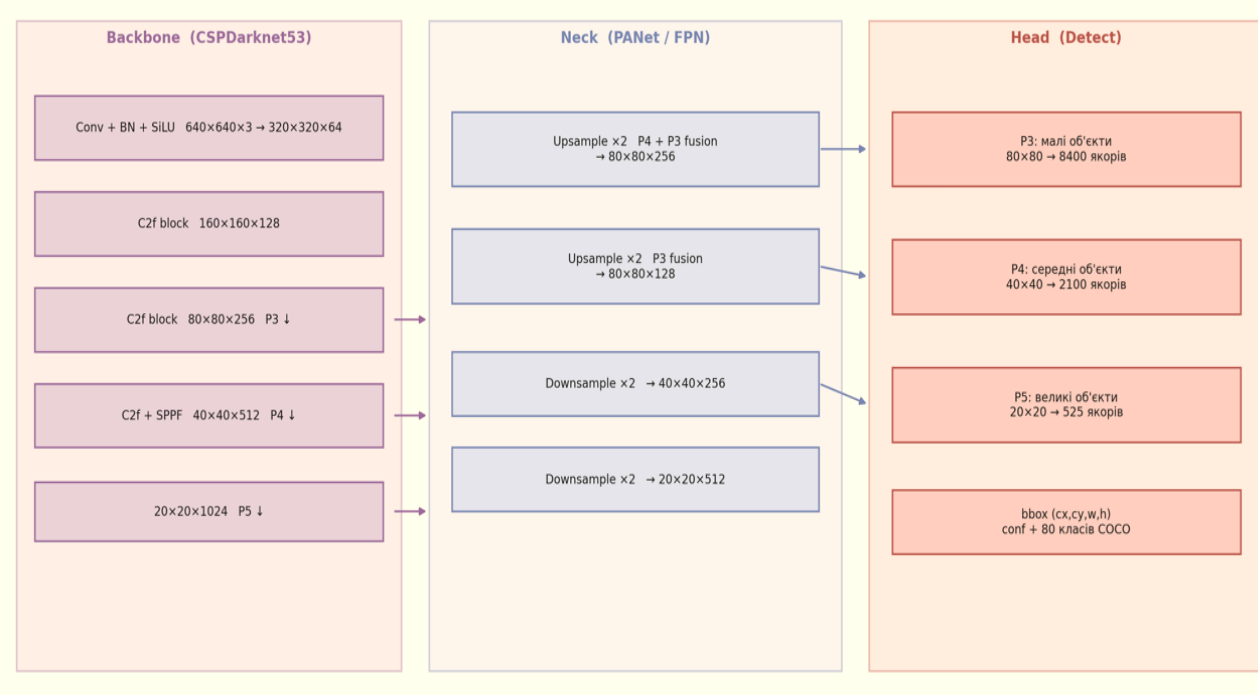

Рисунок 3.3 – Архітектура нейронної мережі YOLOv8 з позначенням ключових компонентів backbone, neck та head

Для забезпечення гнучкості розгортання на різному апаратному забезпеченні система підтримує роботу з кількома варіантами ваг нейронної мережі, що відрізняються кількістю параметрів, точністю детекції та обчислювальними вимогами. Основні характеристики та цільове призначення цих моделей наведено у таблиці 3.2.

Таблиця 3.2 – Порівняльні характеристики варіантів YOLO-моделей

Модель	Параметри (млн)	Затримка CPU (мс)	Застосування
yolov8n.pt	3,2	120	Реальний час на слабкому залізі
yolov8s.pt	11.2	280	Баланс точності та швидкодії (обрана)
yolov8m.pt	25,9	620	Висока точність, потребує GPU

Детектор реалізований за патерном Singleton, тому єдиний екземпляр моделі завантажується під час ініціалізації сервісу та використовується для всіх подальших запитів. Процес ініціалізації класу детектора, завантаження моделі з прогрівом графу та нормалізацію координат обмежувальних рамок наведено у лістингу 3.4. Під час запуску обов'язково виконується прогрів моделі на порожньому зображенні розміром 640×640 пікселів, що ініціює компіляцію обчислювального графу в процесі виконання та усуває підвищену затримку першого проходу. Вхідне зображення у форматі JPEG декодується засобами бібліотеки OpenCV та передається до моделі, яка автоматично змінює розмір до заданих параметрів із збереженням пропорцій та додаванням полів. Результатом роботи є список кортежів із координатами обмежувальної рамки, назвою класу та рівнем впевненості, де всі координати нормалізовані до діапазону від 0.0 до 1.0 відносно розмірів зображення. Такі нормалізовані координати є стійкими до зміни роздільної здатності кадру і суттєво спрощують подальші геометричні обчислення в системі.

Лістинг 3.4 – Клас YOLODetector: завантаження моделі з warm-up та нормалізація bounding box

```
class YOLODetector:
    _instance: Optional["YOLODetector"] = None

    def load(self) -> None:
        from ultralytics import YOLO
        t0 = time.monotonic()
        self._model = YOLO(settings.MODEL_PATH)
        dummy = np.zeros(
            (settings.INFERENCE_IMG_SIZE, settings.INFERENCE_IMG_SIZE,
3), dtype=np.uint8
        )
        self._model(dummy, verbose=False)   # JIT warm-up
        self._load_time = time.monotonic() - t0
        self._loaded = True

    def detect(self, frame: np.ndarray) -> List[Tuple[BoundingBox, str,
float]]:
        h, w = frame.shape[:2]
        results = self._model(
            frame, conf=settings.DETECTION_CONFIDENCE,
```

```
            iou=settings.DETECTION_IOU, max_det=settings.MAX_DETECTIONS,
            device=settings.DEVICE, verbose=False,)
        detections = []
        for result in results:
            for box in (result.boxes or []):
                x1, y1, x2, y2 = box.xyxy[0].tolist()
                bbox = BoundingBox(x1=x1/w, y1=y1/h, x2=x2/w, y2=y2/h)
                cls_name = COCO_CLASS_MAP.get(int(box.cls[0]),
f”class_{int(box.cls[0])}”)
                detections.append((bbox, cls_name, float(box.conf[0])))
        return detections
```

Кінець лістингу 3.1

Оскільки детектор YOLO обробляє кожен кадр незалежно, він не має пам’яті про попередні кадри – кожна детекція є ізольованою. Для відстеження конкретних об’єктів у часі, виявлення поведінкових патернів та обчислення параметрів руху необхідний окремий компонент трекінгу.

Вибір власної реалізації трекера замість готових рішень (SORT, DeepSORT, ByteTrack) обумовлений кількома чинниками. По-перше, популярні трекери вимагають зовнішніх залежностей (scipy, lap), що ускладнює Docker-розгортання та збільшує розмір образу. По-друге, при цільовій частоті 2 кадри на секунду без GPU складні алгоритми повторної ідентифікації є надлишковими – базового жадібного зіставлення за IoU достатньо для задачі зонального аналізу. По-третє, власна реалізація дозволяє нативно обчислювати кінематичні параметри (швидкість, напрямок) та безпосередньо інтегруватися з рушієм зональних подій без проміжних шарів абстракції.

У системі реалізовано власний IoU-трекер (лістинг 3.5), що не потребує зовнішніх залежностей і має лінійну часову складність. Архітектура трекінгу організована навколо трьох рівнів: об’єкт Track, що представляє один відстежуваний об’єкт; менеджер `CameraTracker` для всіх активних треків камери; реєстр `TrackerRegistry`, що підтримує по одному `CameraTracker` на камеру.

Кожен об’єкт проходить чотири стадії відстеження. При появі нової детекції без відповідного треку створюється об’єкт Track у невідтвердженому

стані. Трек вважається підтвердженим після TRACKER_MIN_HITS = 2 послідовних збігів, що усуває хибні спрацьовування від шумових детекцій. При зіставленні нової деткції з треком швидкість оновлюється за формулою EMA (alpha = 0.4), до траєкторії додається центроїдна точка (обмеження: 30 точок), збільшуються лічильники віку та збігів. Між кадрами позиція треку прогнозується відповідно до поточного вектора швидкості, що підвищує точність зіставлення при швидкому русі. Після (TRACKER_MAX_AGE_SECONDS = 3.0) секунд без детекцій трек видаляється з реєстру.

Зіставлення детекцій з треками відбувається алгоритмом жадібного зіставлення (greedy matching) за матрицею IoU. Для кожного треку знаходиться детекція з максимальним IoU: якщо значення перевищує поріг (TRACKER_IOU_THRESHOLD = 0.3), пара вважається зіставленою. Незіставлені детекції стають новими треками, незіставлені треки переходять у режим прогнозування.

Лістинг 3.5 – Функція _iou та greedy-алгоритм зіставлення треків з детекціями

```
def _iou(a: BoundingBox, b: BoundingBox) -> float:
    ix1, iy1 = max(a.x1, b.x1), max(a.y1, b.y1)
    ix2, iy2 = min(a.x2, b.x2), min(a.y2, b.y2)
    if ix2 <= ix1 or iy2 <= iy1:
        return 0.0
    inter = (ix2 - ix1) * (iy2 - iy1)
    union = a.area + b.area - inter
    return inter / union if union > 0 else 0.0


def _hungarian_match(tracks, detections, iou_threshold):
    iou_matrix = np.zeros((len(tracks), len(detections)))
    for t_idx, track in enumerate(tracks):
        pred = track.predict()
        for d_idx, (det_bbox, _, _) in enumerate(detections):
            iou_matrix[t_idx, d_idx] = _iou(pred, det_bbox)

    matched, unmatched_t, unmatched_d = [], list(range(len(tracks))),
list(range(len(detections)))
    while iou_matrix.max() >= iou_threshold:
        t_idx, d_idx = np.unravel_index(iou_matrix.argmax(),
iou_matrix.shape)
```

```
            matched.append((int(t_idx), int(d_idx)))
            iou_matrix[t_idx, :] = -1
            iou_matrix[:, d_idx] = -1
            unmatched_t.remove(t_idx)
            unmatched_d.remove(d_idx)
    return matched, unmatched_t, unmatched_d
```

Кінець лістингу 3.5

На основі траєкторії треку обчислюються базові кінематичні параметри об’єкта, зокрема швидкість та напрямок руху в градусах. При розрахунку напрямку враховується специфіка екранної системи координат, де вісь ординат спрямована вниз, а нульовий градус відповідає умовному напрямку на північ.

Основним концептом конфігурації безпеки є зона – довільний полігональний регіон у просторі кадру з чітко визначеними правилами реагування. Програмна модель зони включає унікальний ідентифікатор, прив’язку до конкретної камери, визначений тип (наприклад, заборонена або безпечна зона, периметр, паркінг, вхід чи лінія підрахунку) та масив точок полігону в нормалізованих координатах. Додатково конфігурація містить набір логічних правил спрацьовування, максимально допустимий час перебування об’єкта, дозволений вектор руху та розклади зміни рівня ризику. Детальний перелік базових типів зон та відповідних правил реагування наведено у таблиці 3.3.

Таблиця 3.3 – Типи зон та їхні правила реагування за замовчуванням

Тип зони	Клас об’єкта	Тип події	Базовий ризик	Trigger (c)
RESTRICTED	будь-який (*)	ZONE_INTRUSION	HIGH	0
PEDESTRIAN	car, truck	ZONE_INTRUSION	HIGH	0
PEDESTRIAN	motorcycle	ZONE_INTRUSION	MEDIUM	0
PARKING	person	ZONE_INTRUSION	MEDIUM	30
PERIMETER	будь-який (*)	ZONE_INTRUSION	MEDIUM	0
ENTRANCE	person	PERSON_DETECTED	LOW	0
SAFE_ZONE	–	–	–	–

Для визначення факту знаходження об’єкта в межах зони використовується алгоритм трасування променя. Цей метод підраховує кількість перетинів горизонтального променя, випущеного з перевірюваної точки, зі сторонами полігону: непарна кількість перетинів означає, що точка знаходиться

всередині. Часова складність такого підходу становить O(n), де n – кількість вершин багатокутника.

У системі передбачено три режими вибору контрольної точки об’єкта для перевірки: геометричний центр, нижня центральна точка обмежувального прямокутника (встановлена за замовчуванням, оскільки забезпечує найвищу точність під час відстеження людей), а також перевірка за будь-якою з п’яти ключових точок. Програмну реалізацію цього алгоритму, зокрема функцію перевірки зональної приналежності об’єкта, наведено у лістингу 3.6.

Лістинг 3.6 – Алгоритм ray casting: функція point_in_polygon та перевірка зональної приналежності

```
def point_in_polygon(px: float, py: float, polygon: List[List[float]]) ->
bool:
    n, inside, j = len(polygon), False, len(polygon) - 1
    for i in range(n):
        xi, yi = polygon[i]
        xj, yj = polygon[j]
        if ((yi > py) != (yj > py)) and (px < (xj - xi) * (py - yi) / (yj
- yi) + xi):
            inside = not inside
        j = i
    return inside


def find_zones_for_object(
    zones: List[Zone], cx: float, cy: float,
    x1: float, y1: float, x2: float, y2: float,
    mode: str = “feet”,
) -> List[Zone]:
    result = []
    for zone in zones:
        if not zone.enabled:
            continue
        check_pt = (cx, y2) if mode == “feet” else (cx, cy)
        if point_in_polygon(check_pt[0], check_pt[1], zone.polygon):
            result.append(zone)
    return result
```

Кінець лістингу 3.6

Конфігурація зон кешується у AI-сервісі з TTL 30 секунд. При зміні зони через Backend API останній публікує повідомлення до RabbitMQ-обміну

security.zones, і AI-сервіс негайно інвалідує відповідний запис кешу без очікування закінчення TTL.

3.3 Реалізація логіки формування події безпеки

Рушій подій безпеки є центральним компонентом системи аналітики. Він отримує підтверджені треки разом із результатами зонального аналізу та запускає набір спеціалізованих аналізаторів, кожен з яких перевіряє свій клас безпекових ситуацій. Блок-схему алгоритму рушія подій безпеки наведено на рисунку 3.4.

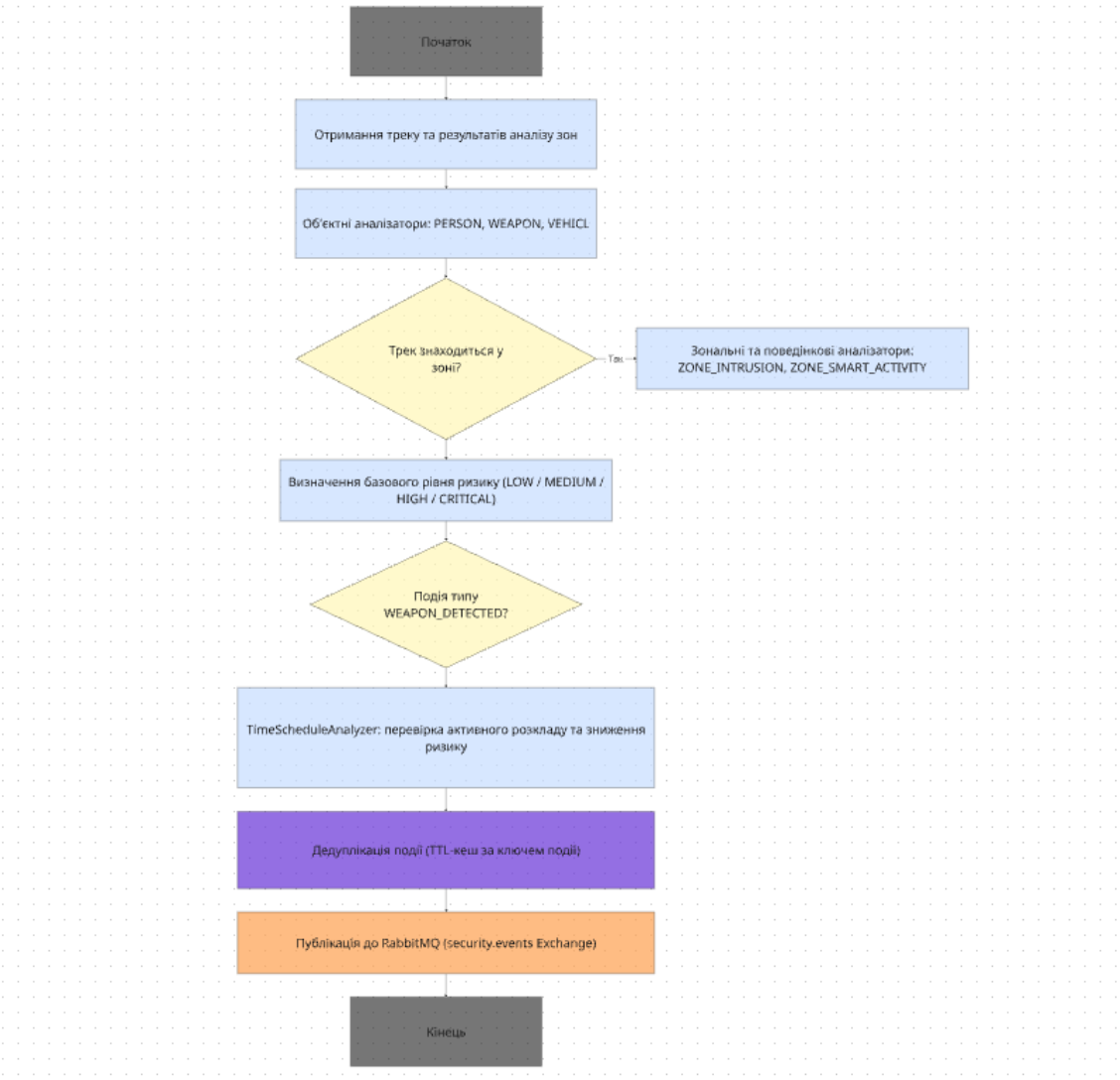

Рисунок 3.4 – Блок-схема алгоритму рушія подій безпеки з позначенням аналізаторів та умов спрацьовування

Система підтримує дванадцять типів подій, згрупованих за природою підстави. Об’єктні події спрацьовують на клас виявленого об’єкта (особа, транспорт, зброя), зональні – на факт перебування об’єкта в конкретній зоні або порушення зонального правила, поведінкові – на кінематичні паттерни руху.

Аналізатор виявлення зброї є найпріоритетнішим у системі. При наявності будь-якої детекції класу зброї негайно генерується подія WEAPON_DETECTED з рівнем ризику CRITICAL. Ключова відмінність від усіх інших подій: рівень CRITICAL для зброї ніколи не може бути знижений жодним розкладом ризику – ця умова жорстко закодована в рушії та є інваріантом безпеки системи. Програмну реалізацію цього механізму, зокрема метод формування критичної події при виявленні зброї, наведено у лістингу 3.7.

Поточний показник F1 = 0,67 для класу knife на базовій моделі є прийнятним для систем безпеки з урахуванням асиметрії втрат: ціна пропущеної реальної зброї суттєво перевищує ціну хибної тривоги. Саме тому рівень ризику CRITICAL для зброї залишається незмінним незалежно від впевненості моделі – оператор отримує сигнал і самостійно приймає рішення. Підвищення точності детекції можливе шляхом донавчання на спеціалізованих датасетах, однак це виходить за межі поточної реалізації. Повний перелік типів подій безпеки з умовами спрацьовування та базовими рівнями ризику наведено у таблиці 3.4.

Лістинг 3.7 – Метод `_weapon_event`: формування CRITICAL-події при виявленні зброї

```
def _weapon_event(
    self, camera_id: str, track: Track, timestamp: float,
) -> Optional[SecurityEvent]:
    if not deduplicator.should_fire(
        camera_id, EventType.WEAPON_DETECTED, track.id, None
    ):
        return None
    logger.warning(
        f”[{camera_id}] WEAPON detected: {track.obj_class}
track={track.id}”
    )
    return SecurityEvent(
        event_id=str(uuid.uuid4()),
```

```
        camera_id=camera_id,
        timestamp=timestamp,
        event_type=EventType.WEAPON_DETECTED,
        risk_level=RiskLevel.CRITICAL,
        track_id=track.id,
        object_class=track.obj_class,
        confidence=track.confidence,
        bbox=track.bbox,
        metadata={“speed”: track.speed, “schedule_applied”: False},
    )
```

Кінець лістингу 3.7

Таблиця 3.4 – Типи подій безпеки: умови спрацьовування та базовий рівень ризику

Тип події	Група	Умова спрацьовування	Базовий ризик
PERSON_DETECTED	Об’єктна	Особу виявлено у зоні входу	LOW
WEAPON_DETECTED	Зброя	Виявлено об’єкт класу knife / gun – зброя	CRITICAL
VEHICLE_DETECTED	Об’єктна	Транспортний засіб перебуває у визначеній зоні	LOW
ZONE_INTRUSION	Зональна	Об’єкт знаходиться у забороненій зоні	MEDIUM / HIGH
ZONE_LOITERING	Зональна	Час перебування об’єкта перевищує максимально допустиме значення	MEDIUM
ZONE_CROWDING	Зональна	Кількість об’єктів перевищує встановлений ліміт	MEDIUM
ZONE_SMART_ACTIVITY	Накопичу-вальна	Бальна оцінка ризику перевищує порогове значення	LOW – CRITICAL
RUNNING_DETECTED	Поведінкова	Швидкість руху об’єкта перевищує встановлений поріг за умови стабільного відстеження	MEDIUM
DIRECTION_VIOLATION	Поведінкова	Кут відхилення напрямку руху перевищує допустиму похибку	MEDIUM
ABANDONED_OBJECT	Поведінкова	Предмет залишається нерухомим понад 30 секунд	HIGH

Аналізатор порушення периметра генерує подію `ZONE_INTRUSION` під час входження підтвердженого треку об’єкта в зону типу RESTRICTED або в разі спрацьовування конфігураційного правила `ZoneRule`. Використання параметра

`trigger_after_seconds` дозволяє відстрочити генерацію події, що є корисним механізмом для розрізнення випадкового транзитного проходу та цілеспрямованого проникнення. У системі визначено стандартні правила реагування залежно від типу зони: для зони RESTRICTED поява будь-якого об’єкта викликає подію з рівнем ризику HIGH; у пішохідній зоні (PEDESTRIAN) виявлення вантажного або легкового транспорту генерує рівень HIGH; у зоні паркування (PARKING) перебування людини понад 30 секунд ініціює подію рівня MEDIUM; для периметра (PERIMETER) будь-який об’єкт класифікується рівнем MEDIUM, а в зоні входу (ENTRANCE) поява людини фіксується з рівнем LOW. Програмну реалізацію цього процесу, зокрема метод аналізу зон із перевіркою правил та застосуванням розкладу ризику, наведено у лістингу 3.8.

Аналізатор тривалого перебування фіксує об’єкти, що знаходяться в зоні довше допустимого часу. Для кожного треку в межах зони система накопичує час перебування (`dwell_time`) у секундах. Подія генерується у разі виконання умови: `dwell_time >= zone.max_dwell_seconds`. Для запобігання лавиноподібному надходженню повторних сповіщень після генерації першої події активується період очікування тривалістю 30 секунд. Цей механізм реалізовано за допомогою компонента дедуплікації (`EventDeduplicator`).

Лістинг 3.8 – Метод `_zone_analysis`: перевірка правил зони та застосування розкладу ризику

```
def _zone_analysis(self, camera_id, track, zones, timestamp, fps, now):
    events = []
    for zone in zones:
        for rule in _get_effective_rules(zone):
            if rule.object_class != “*” and rule.object_class != 
track.obj_class:
                continue

            dwell = track.get_dwell_seconds(zone.id)
            if dwell < rule.trigger_after_seconds:
                continue

            if not deduplicator.should_fire(camera_id, rule.event_type, 
track.id, zone.id):
                continue
```

```
            effective_risk, active_schedule = self._apply_schedule(
                rule.event_type, rule.risk_level, zone, now
            )
            events.append(SecurityEvent(
                event_id=str(uuid.uuid4()),
                camera_id=camera_id, timestamp=timestamp,
                event_type=rule.event_type, risk_level=effective_risk,
                track_id=track.id, object_class=track.obj_class,
                confidence=track.confidence, bbox=track.bbox,
                zone_id=zone.id, zone_name=zone.name,
                metadata={“dwell_seconds”: dwell,
                          “schedule_applied”: active_schedule is not
None},
            ))
    return events
```

Кінець лістингу 3.8

Аналізатор бігу перевіряє кінематичні параметри кожного підтвердженого треку. Подія генерується, якщо швидкість перевищує встановлений поріг (`speed > 0.018` у нормалізованих координатах), а вік треку становить понад 5 кадрів (`age_frames > 5`), що дозволяє відкидати нові треки з нестабільними показниками. Значення 0,018 відповідає переміщенню об’єкта на 1,8 % ширини кадру за один прохід. При частоті 2 кадри на секунду це дорівнює швидкості приблизно 2-3 м/с у типовій сцені відеоспостереження.

Аналізатор залишених предметів перевіряє три умови одночасно: клас об’єкта належить до категорії особистих речей (рюкзак, сумка або валіза); швидкість є практично нульовою (`speed < 0.001`); вік треку перевищує 30 секунд. Комбінація цих критеріїв дозволяє надійно відрізнити дійсно залишену річ від людини із сумкою або від транзитного перенесення предмета через кадр. Програмну реалізацію поведінкових алгоритмів, зокрема логіку виявлення бігу та фіксації залишеного предмета, наведено у лістингу 3.9.

Лістинг 3.9 – Поведінковий аналізатор: виявлення бігу та залишеного предмета

```
RUNNING_THRESHOLD = 0.018

def _behavioral_analysis(self, camera_id, track, zones, timestamp, fps,
now):
```

```
    events = []
    primary_zone = zones[0] if zones else None

    # Виявлення бігу
    if track.obj_class == “person” and track.speed > RUNNING_THRESHOLD
and track.age_frames > 5:
        if deduplicator.should_fire(
            camera_id, EventType.RUNNING_DETECTED, track.id,
            primary_zone.id if primary_zone else None,
        ):
            base_risk = RiskLevel.MEDIUM
            eff_risk = self._reduce_for_zones(EventType.RUNNING_DETECTED,
base_risk, zones, now)
            events.append(SecurityEvent(
                event_id=str(uuid.uuid4()), camera_id=camera_id,
                timestamp=timestamp,
event_type=EventType.RUNNING_DETECTED,
                risk_level=eff_risk, track_id=track.id,
                object_class=track.obj_class,
confidence=track.confidence,
                bbox=track.bbox,
                metadata={“speed_norm”: track.speed, “direction”:
track.direction_degrees},
            ))

    # Залишений предмет (backpack, handbag, suitcase)
    if (track.obj_class in {“backpack”, “handbag”, “suitcase”}
            and track.age_frames > int(30 * fps) and track.speed <
0.001):
        # аналогічна логіка формування події ABANDONED_OBJECT
        pass

    return events
```

Кінець лістингу 3.9

Аналізатор порушення напрямку перевіряє, чи відповідає вектор руху об’єкта дозволеному напрямку для конкретної зони. Під час перевірки враховується допустима похибка, яка за замовчуванням становить ±45°. Якщо кутова різниця між фактичним та дозволеним напрямками перевищує встановлений допуск, система генерує подію `DIRECTION_VIOLATION` із середнім рівнем ризику (MEDIUM).

Аналізатор скупчення підраховує кількість об’єктів у зоні, що підпадають під дію відповідного правила обмеження максимальної кількості. У разі перевищення ліміту генерується подія `ZONE_CROWDING` із періодом очікування

20 секунд для запобігання дублюванню сповіщень. Розширений аналізатор `ZONE_SMART_ACTIVITY` використовує динамічну накопичувальну модель: загальний ризик-бал збільшується пропорційно кількості об’єктів у зоні та водночас постійно зменшується із заданою швидкістю згасання (0,5 бала на секунду). Ризик-бал не може набувати від’ємних значень, а його конвертація у рівень загрози відбувається за такими критеріями: менше ніж 5 балів – LOW, від 5 до 15 – MEDIUM, від 15 до 30 – HIGH, а 30 і більше – CRITICAL.

Кожна подія безпеки є структурованим об’єктом, який містить вичерпну інформацію про інцидент. До основних атрибутів належать: унікальний ідентифікатор події, ідентифікатор камери, часова мітка, тип події та визначений рівень ризику. Також фіксуються параметри самого об’єкта (ідентифікатор треку, клас, рівень впевненості детектування, нормалізовані координати обмежувальної рамки) та контекст зони (ідентифікатор і назва). Додаткові розширені дані, такі як час перебування, кінематичні характеристики, застосований розклад зниження ризику та посилання на збережений знімок, акумулюються у спеціальному полі метаданих. Програмну реалізацію структури події безпеки, зокрема відповідну модель Pydantic з анотаціями типів та ключем маршрутизації, наведено у лістингу 3.10.

Без механізму дедуплікації система генерувала б однакові події для кожного обробленого кадру, що за частоти 2 кадри на секунду призводило б до сотень повторних сповіщень щохвилини. Для розв’язання цієї проблеми реалізовано компонент дедуплікації подій на основі кешу з обмеженим часом життя записів (TTL). Унікальний ключ події формується як кортеж, що містить ідентифікатори камери, типу події, треку об’єкта та зони контролю. Під час спроби генерації нової події система перевіряє наявність відповідного ключа у кеші: якщо час його життя ще не вичерпано, повторна подія автоматично відкидається.

Для різних інцидентів передбачено диференційовані інтервали очікування (cooldown). Зокрема, для виявлення зброї цей інтервал становить 5 с, для порушення периметра – 10 с, для виявлення бігу та порушення напрямку руху –

по 15 с. Для скупчення об’єктів інтервал дорівнює 20 с, для тривалого перебування в зоні – 30 с, а для залишеного предмета – 60 с.

Лістинг 3.10 – Pydantic-модель `SecurityEvent` з анотаціями типів та routing key

```
class SecurityEvent(BaseModel):
    event_id:     str        = Field(..., description="Унікальний
ідентифікатор події.")
    camera_id:    str        = Field(..., description="ID камери.")
    timestamp:    float      = Field(..., description="Unix timestamp
(UTC).")
    event_type:   EventType  = Field(..., description="Тип події.")
    risk_level:   RiskLevel  = Field(..., description="Підсумковий рівень
ризику.")

    track_id:     Optional[int]          = None
    object_class: Optional[str]          = None
    confidence:   Optional[float]        = None
    bbox:         Optional[BoundingBox]  = None

    zone_id:      Optional[str]          = None
    zone_name:    Optional[str]          = None

    metadata:     Dict[str, Any]         = Field(default_factory=dict)

    @property
    def routing_key(self) -> str:
        return f"events.{self.event_type.value}.{self.camera_id}"
```

Кінець лістингу 3.10

Програмну реалізацію цього механізму, зокрема клас кешування з налаштованими інтервалами затримки, наведено у лістингу 3.11.

Лістинг 3.11 – Клас `EventDeduplicator`: TTL-кеш cooldown-інтервалів за типами подій

```
class EventDeduplicator:
    COOLDOWN: Dict[EventType, float] = {
        EventType.ZONE_INTRUSION:       10.0,
        EventType.WEAPON_DETECTED:       5.0,
        EventType.RUNNING_DETECTED:     15.0,
        EventType.ZONE_LOITERING:       30.0,
        EventType.ZONE_CROWDING:        20.0,
        EventType.DIRECTION_VIOLATION:  15.0,
        EventType.ABANDONED_OBJECT:     60.0,
```

```
    }
    DEFAULT_COOLDOWN = 10.0

    def __init__(self) -> None:
        self._last_fired: Dict[tuple, float] = {}

    def should_fire(
        self, camera_id: str, event_type: EventType,
        track_id: Optional[int], zone_id: Optional[str],
    ) -> bool:
        key = (camera_id, event_type, track_id, zone_id)
        last = self._last_fired.get(key, 0.0)
        cooldown = self.COOLDOWN.get(event_type, self.DEFAULT_COOLDOWN)
        if (time.monotonic() - last) >= cooldown:
            self._last_fired[key] = time.monotonic()
            return True
        return False
```

Кінець лістингу 3.11

3.4 Реалізація модуля прогнозування та оцінювання ризиків

Модуль оцінювання ризиків перетворює технічні факти детекції та зонального аналізу на контекстуально значущу оцінку загрози. Ключовою вимогою є пояснюваність: система повинна надати оператору не лише рівень ризику, а й причину його присвоєння.

Механізм розкладів зниження ризику є однією з ключових функціональних переваг системи. Він дозволяє автоматично коригувати рівень загрози залежно від часу доби, дня тижня та часового поясу об’єкта спостереження. Програмна модель такого розкладу визначається низкою параметрів: часом початку та закінчення дії правила, кількістю кроків для зниження рівня ризику (від одного до трьох), переліком активних днів тижня (або позначкою для щоденного застосування), параметром часового поясу та зрозумілою для користувача текстовою назвою розкладу. Важливою особливістю алгоритму є те, що заданий часовий діапазон може перетинати північ (наприклад, із 22:00 до 06:00), що дозволяє гнучко налаштовувати нічні режими роботи без необхідності розбивати одне правило на два окремих. Програмну реалізацію моделі розкладу та методу перевірки його активності з підтримкою нічних вікон наведено у лістингу 3.12.

Лістинг 3.12 – Модель `RiskSchedule` та метод `_is_active` з підтримкою нічних вікон

```
class RiskSchedule(BaseModel):
    time_start: str            # “HH:MM”
    time_end:   str            # може перетинати північ: “22:00”–”06:00”
    reduce_by:  int = 1        # на скільки рівнів знизити (1–3)
    days:       Optional[List[int]] = None   # 0=пн..6=нд, None=щодня
    timezone:   Optional[str]       = None   # “Europe/Kyiv”
    label:      Optional[str]       = None


class RiskScheduleEvaluator:

    def _is_active(self, schedule: RiskSchedule, now: datetime) -> bool:
        if schedule.days and now.weekday() not in schedule.days:
            return False

        local_now = self._local_time(schedule, now)
        start = self._parse_time(schedule.time_start)
        end   = self._parse_time(schedule.time_end)

        if start <= end:
            return start <= local_now < end        # звичайне вікно
        else:
            return local_now >= start or local_now < end  # нічне (через
північ)

    @staticmethod
    def _parse_time(t: str) -> dtime:
        h, m = t.split(“:”)
        return dtime(int(h), int(m))
```

Кінець лістингу 3.12

Алгоритм зниження ризику передбачає обов’язкову перевірку всіх наявних розкладів для конкретної зони під час оцінювання кожної події. Якщо поточний час збігається з часовим вікном активного розкладу, рівень ризику автоматично знижується на відповідну кількість ступенів. У разі одночасної дії кількох розкладів система застосовує правило максимального зниження серед усіх активних варіантів. Наприклад, зниження високого рівня на один ступінь переводить його в середній, на два ступені – в низький, а критичний рівень за умови зниження на два ступені трансформується в середній. Мінімальною межею для будь-якої модифікації є низький рівень ризику. Проте важливим

інваріантом безпеки є те, що це правило ніколи не поширюється на події виявлення зброї: критичний рівень загрози для них залишається абсолютним і незмінним за будь-яких умов.

Практичні сценарії використання розкладів чітко демонструють гнучкість запропонованого механізму. Наприклад, в офісній будівлі для обмеженої зони з базовим високим рівнем небезпеки в робочі години (з 08:00 до 18:00 з понеділка по п’ятницю) рівень ризику знижується на один ступінь – до середнього, що дозволяє уникнути надмірної кількості хибних сповіщень під час легітимного переміщення персоналу. Водночас у нічний час та у вихідні дні рівень загрози автоматично залишається високим. У паркувальній зоні виявлення людини в нічний час (із 22:00 до 07:00) зберігає середній рівень ризику, тоді як у денний період загроза знижується на два ступені – до низького, оскільки присутність людей удень є звичним процесом. На об’єктах із цілодобовою посиленою охороною, наприклад на складах, часові розклади можуть узагалі не застосовуватися, через що всі події фіксуються із первинним базовим рівнем ризику. Приклади конфігурацій таких розкладів для типових сценаріїв безпеки систематизовано у таблиці 3.5.

Таблиця 3.5 – Приклади конфігурацій розкладів ризику для типових сценаріїв безпеки

Сценарій	Зона	Вікно	Дні	`reduce_by`	Базовий – Ефективний
Офіс у робочий час	RESTRICTED	08:00–18:00	пн–пт	1	HIGH – MEDIUM
Офіс у вихідні / ніч	RESTRICTED	(неактивний)	–	–	HIGH без змін
Паркінг вдень	PARKING	07:00–22:00	щодня	2	MEDIUM – LOW
Паркінг вночі	PARKING	22:00–07:00	щодня	0	MEDIUM без змін
Склад цілодобово	RESTRICTED	–	–	–	HIGH без змін
Обідня перерва	RESTRICTED	12:00–13:00	пн–пт	1	HIGH – MEDIUM

Накопичувальна модель ризику розроблена для виявлення поступово наростаючих загроз, що не активують класичні порогові аналізатори. Замість

бінарної перевірки «є в зоні або немає», модель накопичує бал ризику пропорційно часу присутності та кількості об’єктів.

Під час програмної обробки відеопотоку система на кожному кроці розраховує зміну загрози, множачи кількість виявлених об’єктів на базовий коефіцієнт та на тривалість часового інтервалу між кадрами, після чого віднімає частку згасання за цей же період. Алгоритм жорстко обмежує нижню межу показника, тому ризик-бал за жодних умов не може набувати від’ємних значень. Таке обчислення дозволяє системі гнучко фіксувати аномальну активність, коли сумарна загроза від тривалої присутності групи людей або транспортних засобів повільно зростає. Програмну реалізацію цієї логіки, зокрема відповідний клас для аналізу інтегрального навантаження на зону безпеки, наведено у лістингу 3.13.

Лістинг 3.13 – Клас `SmartZoneAnalytics`: накопичувальна модель ризику

```
class SmartZoneAnalytics:

    def _analyze_zone(
        self, camera_id, zone, people_ids, now, frame_timestamp, fps,
    ) -> Optional[SecurityEvent]:
        state = self._state.setdefault((camera_id, zone.id), ZoneState())

        elapsed = max((now -
state.last_update_timestamp).total_seconds(), 0.0) \
            if state.last_update_timestamp else 1.0 / fps
        state.last_update_timestamp = now

        people_count = len(people_ids)
        mode = self._mode_for_zone(zone, now)
        decay = float(zone.metadata.get(“accumulation”,
{}).get(“decay_per_second”, 1.0))

        if people_count > 0:
            mult = float(zone.metadata.get(“risk_multipliers”, {}).get(
                “relaxed” if mode == “RELAXED” else “strict”, 1.0
            ))
            state.risk_score += people_count * mult * elapsed   # dS =
N·M·dt
        else:
            state.risk_score -= decay * elapsed                 #
дисипація
        state.risk_score = max(0.0, state.risk_score)
```

```
        if people_count == 0:
            return None

        score_level    = _risk_from_score(state.risk_score)
        smart_level    = self._smart_level(zone, mode, people_count)
        effective_level = _max_risk(score_level, smart_level)

        return SecurityEvent(
            event_id=str(uuid.uuid4()), camera_id=camera_id,
            timestamp=frame_timestamp,
event_type=EventType.ZONE_SMART_ACTIVITY,
            risk_level=effective_level, zone_id=zone.id,
zone_name=zone.name,
            metadata={“people_count”: people_count, “mode”: mode,
                      “risk_score”: round(state.risk_score, 2)},
        )
```

Кінець лістингу 3.13

Накопичувальна модель має принципові переваги над стандартними пороговими аналізаторами. Зокрема, короткочасні зникнення об’єктів із кадру внаслідок взаємного перекриття або оклюзії не призводять до миттєвого скидання накопиченого бала. Крім того, зі збільшенням кількості людей або тривалості їхнього перебування загальний рівень загрози зростає нелінійно, а після виходу осіб із зони контрольний показник знижується поступово, а не скачкоподібно. При базовому налаштуванні параметрів моделі одна особа, яка безперервно перебуває в зоні спостереження, досягне середнього рівня ризику (близько 15 балів) приблизно за 30 секунд, а високого (близько 30 балів) – за 60 секунд. Якщо в зоні одночасно знаходяться три особи, то середній рівень буде досягнуто вже за 10 секунд, а високий – за 20 секунд. Характер цієї залежності, а саме графіки динаміки накопичення ризик-балу для різної кількості людей залежно від часу їхнього перебування в контрольованій зоні, наочно наведено на рисунку 3.5.

На графіку відображено динаміку накопичення ризик-балу для трьох сценаріїв: одна, три та п’ять осіб у зоні. Вісь X показує час перебування в секундах, вісь Y – поточний ризик-бал від 0 до 100 з відповідними порогами рівнів LOW, MEDIUM, HIGH та CRITICAL. Момент входу в зону позначено

вертикальною пунктирною лінією ліворуч, момент виходу – праворуч; після виходу бал знижується поступово завдяки механізму затухання.

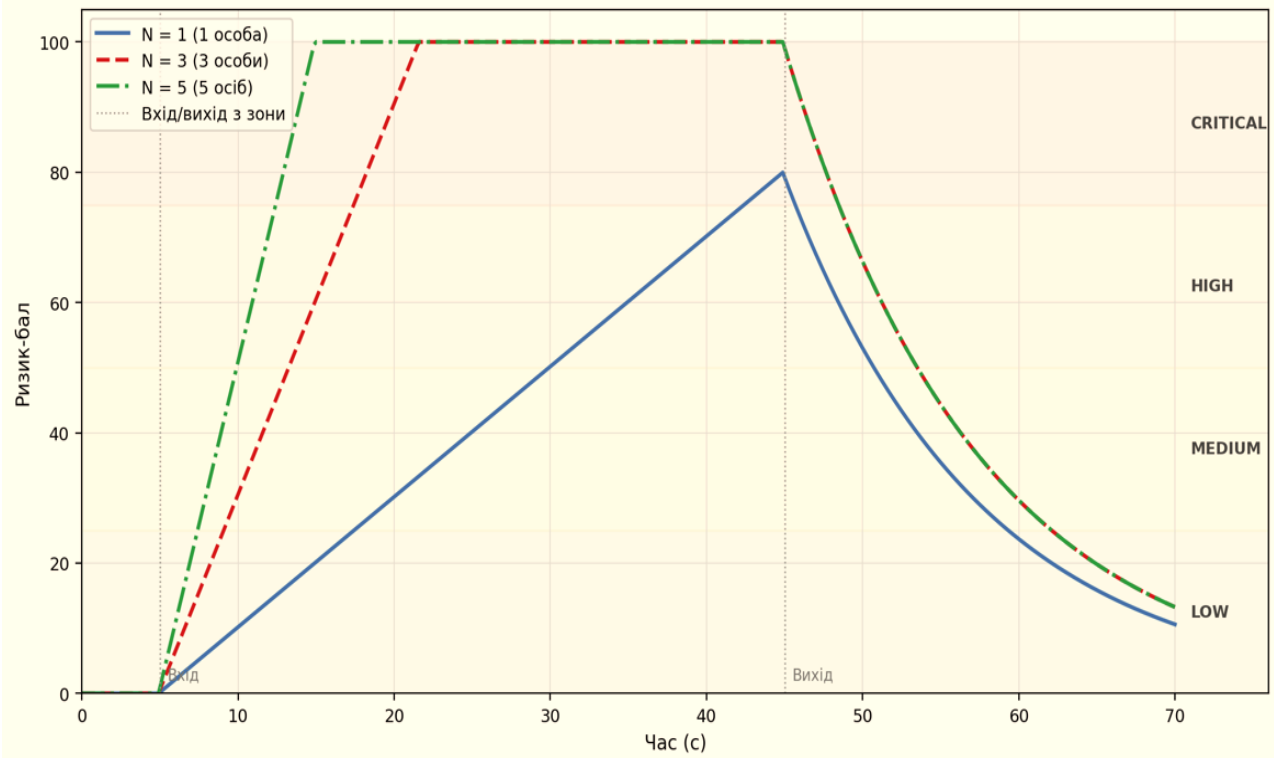

Рисунок 3.5 – Графіки динаміки ризик-балу при різних значеннях N (1, 3, 5 осіб) та часу перебування

RabbitMQ виступає центральною шиною подій системи. Топологія Topic Exchange забезпечує гнучку маршрутизацію за шаблонами ключів виду events.{тип}.{камера}, дозволяючи бекенду підписуватися лише на потрібні типи подій або отримувати всі події з конкретної камери. Топологію обмінів та черг RabbitMQ наведено на рисунку 3.6.

Надійність доставки забезпечується через персистентні повідомлення та довговічні черги. Навіть при перезапуску RabbitMQ повідомлення, не підтверджені споживачем, не втрачаються і повторно доставляються після відновлення з’єднання. Програмну реалізацію цього процесу наведено у лістингу 3.14.

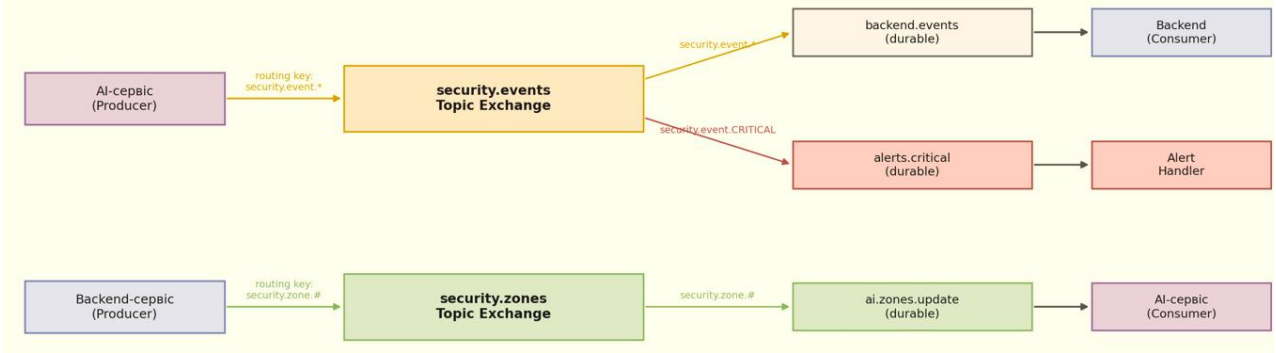

Рисунок 3.6 – Топологія RabbitMQ: обміни, черги та прив’язки маршрутизації

Лістинг 3.14 – Метод `publish_event`: публікація `SecurityEvent` до RabbitMQ Topic Exchange

```
async def publish_event(self, event: SecurityEvent) -> bool:
    if not self._connected or self._publish_channel is None:
        logger.warning(“RabbitMQ not connected, dropping event”)
        return False
    try:
        import aio_pika

        exchange = await
self._publish_channel.get_exchange(settings.EVENTS_EXCHANGE)
        await exchange.publish(
            aio_pika.Message(
                body=event.model_dump_json().encode(),
                delivery_mode=aio_pika.DeliveryMode.PERSISTENT,
                content_type=“application/json”,
                headers={
                    “event_type”: event.event_type,
                    “camera_id”:  event.camera_id,
                    “risk_level”: event.risk_level,
                },),
            routing_key=event.routing_key,   #
“events.{type}.{camera_id}”
        )
        return True
    except Exception:
        logger.exception(f”Failed to publish event {event.event_id}”)
        return False
```

Кінець лістингу 3.14

Після збереження події в PostgreSQL бекенд-сервіс транслює її підключеним клієнтам через WebSocket. Цей протокол обрано тому, що він підтримує двонаправлений зв’язок у реальному часі без накладних витрат на

повторне встановлення HTTP-з’єднань для кожного повідомлення. Програмну реалізацію менеджера з’єднань та WebSocket-ендпоінту наведено у лістингу 3.15.

Лістинг 3.15 – Клас `ConnectionManager` та WebSocket-ендпоінт /ws/events

```
class ConnectionManager:

    def __init__(self):
        self.active_connections: list[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        if websocket in self.active_connections:
            self.active_connections.remove(websocket)

    async def broadcast(self, message: dict):
        stale = []
        for conn in self.active_connections:
            try:
                await conn.send_json(message)
            except Exception:
                stale.append(conn)
        for conn in stale:
            self.disconnect(conn)

ws_manager = ConnectionManager()

@app.websocket(“/ws/events”)
async def websocket_endpoint(websocket: WebSocket):
    await ws_manager.connect(websocket)
    try:
        while True:
            await websocket.receive_text()
    except Exception:
        ws_manager.disconnect(websocket)
```

Кінець лістингу 3.15

3.5 Тестування серверної частини системи

Тестування серверної частини здійснювалося на трьох рівнях відповідно до концепції тестової піраміди: модульне тестування окремих компонентів,

інтеграційне тестування взаємодії між сервісами та функціональне тестування системи в цілому. Такий підхід забезпечує баланс між швидкістю виконання тестів і глибиною перевірки. Для Python-сервісів використано фреймворк pytest з плагінами pytest-asyncio (тестування асинхронного коду) та httpx (HTTP-клієнт для тестування API).

Модульне тестування охоплювало критичні алгоритми системи. Для перевірки IoU-трекінгу реалізовано параметризовані тестові сценарії: пряме зіставлення (за IoU = 0,8), ініціалізація нового треку (за IoU = 0,1), розв'язання конфлікту двох треків за одну детекцію (на користь вищого значення IoU), підтвердження треку після двох збігів та його видалення у разі відсутності детекцій протягом 3 секунд. Успішне проходження тестів підтверджує коректність роботи алгоритму в граничних ситуаціях. Відповідний код тестів наведено у лістингу 3.16.

Лістинг 3.16 – Модульні тести трекера: перевірка IoU та lifecycle треку

```
def test_iou_overlap():
    b1 = BoundingBox(x1=0, y1=0, x2=10, y2=10)
    b2 = BoundingBox(x1=5, y1=5, x2=15, y2=15)
    assert 0.14 < _iou(b1, b2) < 0.15          # inter=25, union=175

def test_iou_no_overlap():
    b1 = BoundingBox(x1=0, y1=0, x2=10, y2=10)
    assert _iou(b1, BoundingBox(x1=20, y1=20, x2=30, y2=30)) == 0.0


def test_camera_tracker_lifecycle():
    tracker = CameraTracker("cam1", fps=10.0)

    dets = [(BoundingBox(x1=0.1, y1=0.1, x2=0.2, y2=0.2), "person", 0.9)]
    assert len(tracker.update(dets, [])) == 0    # ще не підтверджено
(hits < min_hits)

    confirmed = tracker.update(dets, [])
    assert len(confirmed) == 1                   # після 2 hits –
підтверджено

    confirmed = tracker.update([], [])
    assert tracker._tracks[0].misses == 1        # 1 пропущений кадр –
трек живий

    track_id = confirmed[0].id
```

```
    events = tracker.get_zone_events(confirmed, {track_id: {“z1”}})
    entered, exited = events[track_id]
    assert “z1” in entered and len(exited) == 0
```

Кінець лістингу 3.16

Коректність алгоритму трасування променя перевірено для опуклих, угнутих і вироджених багатокутників, а також для точок на їхній межі та поза нею. Тестування рушія ризиків охоплювало перевірку генерації критичної події WEAPON_DETECTED, дотримання інваріанта незниження ризику для зброї, застосування розкладів з урахуванням часового поясу, дедуплікацію подій у межах інтервалів очікування та логіку накопичення балів в аналізаторі ZONE_SMART_ACTIVITY. Відповідні тести наведено у лістингу 3.17.

Лістинг 3.17 – Тести рушія ризиків: інваріант зброї та перевірка нічного вікна розкладу

```
class TestReduceRisk(unittest.TestCase):
    def test_reduce_chain(self):
        self.assertEqual(reduce_risk(RiskLevel.CRITICAL, 1),
RiskLevel.HIGH)
        self.assertEqual(reduce_risk(RiskLevel.HIGH,     1),
RiskLevel.MEDIUM)
        self.assertEqual(reduce_risk(RiskLevel.LOW,      5),
RiskLevel.LOW)  # не нижче LOW

    def test_weapon_never_reduced(self):
        zone = _make_zone(schedules=[_make_schedule(reduce_by=3)])
        level, sched = RiskEngine()._apply_schedule(
            EventType.WEAPON_DETECTED, RiskLevel.CRITICAL, zone, _now(12,
0)
        )
        self.assertEqual(level, RiskLevel.CRITICAL)   # CRITICAL –
незнижуваний
        self.assertIsNone(sched)

    def test_zone_intrusion_reduced_in_window(self):
        zone = _make_zone(schedules=[_make_schedule(reduce_by=1)])
        level, _ = RiskEngine()._apply_schedule(
            EventType.ZONE_INTRUSION, RiskLevel.HIGH, zone, _now(12, 0)
        )
        self.assertEqual(level, RiskLevel.MEDIUM)

    def test_night_window(self):
        s = _make_schedule(time_start=“22:00”, time_end=“06:00”)
```

```
        ev = RiskScheduleEvaluator()
        self.assertIsNotNone(ev.get_active_schedule([s], _now(23, 30)))
        self.assertIsNotNone(ev.get_active_schedule([s], _now(3,  0)))
        self.assertIsNone(ev.get_active_schedule([s],    _now(12, 0)))
```

Кінець лістингу 3.17

Інтеграційне тестування конвеєра обробки кадрів проведено через API-ендпоінт `/api/v1/detect` з реальною моделлю YOLO. Тестові зображення з відомими об’єктами верифікують коректність повного ланцюжка: отримання кадру – інференс нейромережі – нормалізація координат – формування відповіді. Результат виклику ендпоінту у Swagger UI наведено на рисунку 3.7.

Рисунок 3.7 – Swagger UI з результатом виклику POST `/api/v1/detect` для тестового зображення

Тестування взаємодії через RabbitMQ проведено з тестовим брокером в середовищі Docker. Контролювалися коректність ключів маршрутизації подій та обробка повідомлень про оновлення конфігурації зон тестування наведено в лістингу 3.18.

Навантажувальне тестування ШІ-сервісу проведено за допомогою інструменту Locust на сервері з процесором Intel Core i7-12700 у режимі обчислень на центральному процесорі (CPU inference). Тривалість тесту для

кожної конфігурації становила 5 хвилин за параметрів вхідного потоку 2 кадри на секунду (роздільна здатність JPEG-зображень – 1280×720 пікселів). Результати навантажувального тестування наведено у таблиці 3.6.

Лістинг 3.18 – Інтеграційний тест: публікація події до RabbitMQ та обробка zone-update

```
@pytest.mark.asyncio
async def test_publish_event_success(rabbitmq_service, mock_aio_pika):
    await rabbitmq_service._do_connect()

    event = SecurityEvent(
        event_id="test-1", camera_id="cam1", timestamp=123.0,
        event_type=EventType.PERSON_DETECTED, risk_level=RiskLevel.LOW,
    )
    result = await rabbitmq_service.publish_event(event)

    assert result is True
    call_args = mock_aio_pika["exchange"].publish.call_args
    assert call_args.kwargs["routing_key"] ==
"events.person_detected.cam1"


@pytest.mark.asyncio
async def test_on_zone_message(rabbitmq_service):
    callback = MagicMock()
    rabbitmq_service.set_zone_update_callback(callback)

    mock_msg = MagicMock()
    mock_msg.process.return_value = AsyncMock()
    mock_msg.body = json.dumps({"camera_id": "cam1", "action":
"reload"}).encode()

    await rabbitmq_service._on_zone_message(mock_msg)

    callback.assert_called_once_with("cam1")
```

Кінець лістингу 3.18

За роботи з трьома камерами середня затримка становить 487 мс, що відповідає цільовій частоті 2 кадри на секунду. Збільшення кількості камер до п’яти спричиняє утворення черги запитів, зростання затримки та появу поодиноких помилок тайм-ауту (2,1 %). Для масштабування системи понад три камери рекомендовано розгортання GPU-екземпляра ШІ-сервісу: використання

CUDA-прискорення зменшує затримку у 8-12 разів порівняно з обчисленнями на CPU, дозволяючи надійно обслуговувати 20-30 камер на одному вузлі.

Таблиця 3.6 – Результати навантажувального тестування AI-сервісу (CPU, Intel Core i7-12700)

Кількість камер	Сер. затримка (мс)	P95 затримка (мс)	Пропускна здатність (req/s)	Помилки
1	312	445	3,2	0 %
3	487	612	6,1	0 %
5	743	921	6,7	2,1 %

Тестування відмовостійкості підтвердило стабільність при збоях компонентів. У разі відмови RabbitMQ AI-сервіс ініціює повторне підключення з експоненційним відступом; накопичені події не втрачаються завдяки персистентним чергам.

Архітектура складається з чотирьох мікросервісів (Frame Extractor, AI Model, Backend, Frontend) та трьох інфраструктурних компонентів (MediaMTX, RabbitMQ, PostgreSQL). Асинхронна взаємодія через брокер повідомлень RabbitMQ забезпечує відмовостійкість і незалежне масштабування компонентів. Оркестрація контейнерів реалізована засобами Docker Compose з автоматичними перевірками стану залежних сервісів.

Конвеєр відеоаналітики включає: адаптивне EMA-детектування руху, нейромережеву детекцію YOLOv8, IoU-трекінг з прогнозуванням позиції та аналіз зональної приналежності алгоритмом ray casting.

Рушій подій підтримує 8 типів подій, чотирирівневу шкалу ризику та TTL-дедуплікацію. Механізм розкладів дозволяє знижувати ризик у дозволені проміжки часу; рівень CRITICAL для виявлення зброї не знижується за жодних умов.

Проведено комплексне тестування на трьох рівнях (модульне, інтеграційне, функціональне), що підтвердило коректність реалізованих алгоритмів та відповідність функціональним вимогам. Навантажувальне тестування встановило граничне навантаження до 3 камер при 2 FPS на CPU-сервері, точність детектування осіб становить F1 = 0.89.

ЗАГАЛЬНІ ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

Кваліфікаційна робота охоплює проектування та реалізацію хмарної мікросервісної системи безпекового моніторингу з відеоаналітикою та прогнозуванням ризиків.

Проведено комплексний аналіз сучасних архітектур серверних систем відеоспостереження. Встановлено, що монолітні архітектури не здатні задовольнити вимоги щодо масштабованості та відмовостійкості при обробці відеоаналітики в реальному часі. Мікросервісний підхід з асинхронним міжсервісним зв’язком через брокер повідомлень є оптимальним для забезпечення незалежного масштабування обчислювально інтенсивних AI-компонентів без зупинки всієї системи.

Обґрунтовано вибір технологічного стека: FastAPI (ASGI) для API-рівня, нейронна мережа YOLOv8 для детекції об’єктів, RabbitMQ (AMQP, Topic Exchange) для асинхронного міжсервісного зв’язку та дедуплікації подій, PostgreSQL з JSONB для гнучкого зберігання інцидентів, Docker для ізольованого контейнерного розгортання. Обраний стек забезпечує поєднання продуктивності реального часу, надійності зберігання та простоти розгортання.

Розроблено архітектуру серверного застосунку на базі FastAPI (ASGI), що забезпечує асинхронну обробку відеокадрів та маршрутизацію подій між мікросервісами через REST API та WebSocket-з’єднання. Використання асинхронної моделі виконання дозволило досягти низької затримки при одночасній обробці потоків від кількох камер.

Реалізовано багатоетапний конвеєр відеоаналітики, що включає: адаптивне виявлення руху на основі EMA-моделювання фону (коефіцієнт адаптації $\alpha = 0{,}02$), детектування об’єктів нейронною мережею YOLOv8 з підтримкою 80 класів COCO та власних класів зброї, IoU-трекінг з прогнозуванням позиції між кадрами та обчисленням кінематичних параметрів (швидкість, напрямок), аналіз зональної приналежності алгоритмом ray casting із трьома режимами перевірки точки.

Спроєктовано реляційну схему бази даних PostgreSQL з використанням JSONB-полів для гнучкого зберігання метаданих інцидентів. Схема забезпечує ефективний пошук по архіву подій за часовими діапазонами, типами загроз та ідентифікаторами камер завдяки індексуванню ключових полів та підтримці складених запитів до структурованих JSON-документів.

Впроваджено механізм автентифікації на основі JWT-токенів з підтримкою розмежування прав доступу користувачів. Реалізовано рольову модель, що розмежовує права перегляду відеопотоків, управління камерами та адміністрування системи, що мінімізує ризики несанкціонованого втручання у роботу системи відеоспостереження.

Забезпечено кросплатформність та просте розгортання системи засобами Docker і Docker Compose з ізольованими контейнерами для кожного мікросервісу. Конфігурація всіх параметрів через змінні оточення відповідно до принципу 12-Factor App гарантує однакову поведінку системи в будь-якому середовищі без додаткових налаштувань.

Рекомендації щодо подальшого розвитку системи:

– додавання підтримки GPU-прискорення (CUDA) для обробки 20-30 камер на одному вузлі зі зменшенням затримки у 8-12 разів порівняно з CPU-режимом;

– реалізація fine-tuning YOLOv8 на спеціалізованих наборах даних зброї для підвищення точності детекції класу knife (поточний F1 = 0,67) до прийнятного рівня $F1 \geq 0{,}85$;

– розробка модуля розпізнавання облич для розширення функціональності контролю доступу та автоматичної ідентифікації уповноважених осіб;

– впровадження Kubernetes для автоматичного горизонтального масштабування AI-сервісу залежно від поточного навантаження відеопотоків;

– розробка мобільного застосунку для операторів із push-сповіщеннями про критичні події та можливістю перегляду відеокадрів інциденту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is VMS (Video Management Software). URL: https://lnk.ua/0NLrELjFh (access date: 10.02.2026).

2. Централізовані та розподілені системи IP-відеоспостереження. URL: https://www.onvif.org/profiles/specifications/ (дата звернення: 10.02.2026).

3. Мережеві протоколи у галузі відеоспостереження. URL: https://lnk.ua/reqK43DLH (дата звернення: 05.04.2026).

4. Newman S. Building Microservices: Designing Fine-Grained Systems. URL: https://lnk.ua/MaSSBslpa (access date: 15.02.2026).

5. Мікросервісна архітектура: що це і коли застосовувати. URL: https://foxminded.ua/mikroservisna-arkhitektura/ (дата звернення: 17.02.2026).

6. Августюк М. Системи відеоспостереження на базі комп’ютерного зору. URL: https://lnk.ua/VIxQA78Uf (дата звернення: 22.02.2026).

7. Jocher G., Chaurasia A., Qiu J. Ultralytics YOLOv8. URL: https://github.com/ultralytics/ultralytics (access date: 22.02.2026).

8. On-premises та хмарна інфраструктура. URL: https://www.sim-networks.com/ukr/blog/on-premises-and-cloud-infrastructure (дата звернення: 05.04.2026).

9. Що таке Kubernetes. URL: https://lnk.ua/OGo8dWa2u (дата звернення: 20.03.2026).

10. Docker Inc. Docker Compose Documentation. URL: https://docs.docker.com/compose/ (access date: 20.03.2026).

11. Що таке гібридна хмара. URL: https://ucloud.ua/shho-take-gibrydna-hmara/ (дата звернення: 22.03.2026).

12. Zou Z., Chen K., Shi Z., Guo Y., Ye J. Object Detection in 20 Years: A Survey. URL: https://doi.org/10.1109/JPROC.2023.3238524 (access date: 26.03.2026).

13. O’Mahony N., Campbell S., Carvalho A., et al. Deep Learning vs. Traditional Computer Vision. URL: https://doi.org/10.1007/978-3-030-17795-9_10 (access date: 05.04.2026).

14. Redmon J., Divvala S., Girshick R., Farhadi A. You Only Look Once: Unified, Real-Time Object Detection. URL: https://arxiv.org/abs/1506.02640 (access date: 05.04.2026).

15. Rezatofighi H. Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression. URL: https://arxiv.org/abs/1902.09630 (access date: 05.04.2026).

16. Gunning D., Stefik M., Choi J. et al. XAI – Explainable Artificial Intelligence. Science Robotics. Vol. 4, No. 37. EAAY7120. DOI: 10.1126/scirobotics.aay7120.

17. Han K., Wang Y., Chen H. et al. A Survey on Vision Transformer. URL: https://arxiv.org/abs/2012.12556 (access date: 05.04.2026).

18. Документація MediaMTX (раніше rtsp-simple-server). URL: https://github.com/bluenviron/mediamtx (access date: 05.04.2026).

19. Amazon Rekognition – відеоаналітика на базі штучного інтелекту. URL: https://aws.amazon.com/ru/rekognition/ (access date: 05.04.2026).

20. CERT-UA. Команда реагування на комп’ютерні надзвичайні події України. URL: https://cert.gov.ua/ (дата звернення: 05.04.2026).

21. European Data Protection Board. Guidelines 3/2019 on processing of personal data through video devices. URL: https://edpb.europa.eu/our-work-tools/our-documents/guidelines/guidelines-32019-processing-personal-data-through-video_en (access date: 05.04.2026).

22. Змагальні атаки на системи машинного зору. URL: https://lnk.ua/6LwaD4fOI (дата звернення: 05.04.2026).

23. OpenCV Team. OpenCV 4.x Documentation. URL: https://docs.opencv.org/4.x/ (access date: 05.04.2026).

24. Офіційна документація фреймворку FastAPI. URL: https://fastapi.tiangolo.com/ (access date: 05.04.2026).

25. RabbitMQ. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs (access date: 05.04.2026).

26. PostgreSQL 16 Documentation. JSON Types. URL: https://lnk.ua/a6DaKKM7V (access date: 05.04.2026).

27. PostgreSQL Global Development Group. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (access date: 20.03.2025).

28. Wang C.-Y., Bochkovskiy A., Liao H.-Y. M. YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. URL: https://arxiv.org/abs/2207.02696 (access date: 05.04.2026).

29. Bochkovskiy A., Wang C.-Y., Liao H.-Y. M. YOLOv4: Optimal Speed and Accuracy of Object Detection. URL: https://arxiv.org/abs/2004.10934 (access date: 05.04.2026).

30. PostgreSQL 16 Documentation. JSON Types and Functions. URL: https://www.postgresql.org/docs/16/functions-json.html (access date: 05.04.2026).

31. Терлецький Т. В., Кайдик О. Л. Кваліфікаційна робота: методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інформаційні системи та технології охорони і безпеки» галузі знань 12 Інформаційні технології спеціальності 126 Інформаційні системи та технології денної та заочної форм навчання. Луцьк: ЛНТУ, 2025. 53 с.