Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних систем та мереж
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: *Комп'ютерна система оцінювання завантаженості вулиць транспортними засобами та пішоходами.*

Виконав: студент 4 курсу, групи СІ-42
спеціальності 123 «Комп'ютерна інженерія»
(шифр і назва спеціальності)

		Котовський Д.М.
	(підпис)	(прізвище та ініціали)
Керівник		*Тиш Є.В.*
	(підпис)	(прізвище та ініціали)
Нормоконтроль		*Луцик Н.С.*
	(підпис)	(прізвище та ініціали)
Завідувач кафедри		Осухівська Г.М.
	(підпис)	(прізвище та ініціали)
Рецензент		*Дмитроца Л.П.*
	(підпис)	(прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет *комп'ютерно-інформаційних систем і програмної інженерії*
(повна назва факультету)

Кафедра *комп'ютерних систем та мереж*
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Осухівська Г.М.
(підпис) (прізвище та ініціали)
«25» квітня 2026 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *бакалавр*
(назва освітнього ступеня)

за спеціальністю *123 «Комп'ютерна інженерія»*
(шифр і назва спеціальності)

студента *Котовського Дениса Мирославовича*
(прізвище, ім'я, по батькові)

1. Тема роботи *Комп'ютерна система оцінювання завантаженості вулиць транспортними засобами та пішоходами.*

Керівник роботи
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від *« 24 » квітня 2026 року № 4/9-188.*

2. Термін подання студентом завершеної роботи *17.06*

3. Вихідні дані до роботи *Мікрокомп'ютер Raspberry Pi 3 Model B; модуль камери (CSI);* сервер обробки на базі Apple Silicon (Neural Engine); Python 3.12; нейромережа YOLOv8m (CoreML); FastAPI; SQLite; OpenCV.

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз предметної області та обґрунтування переходу до розподіленої архітектури
2. Проєктування розподіленої системи оцінювання завантаженості вулиць
3. Практична реалізація системи та оцінка її ефективності
4. Безпека життєдіяльності, основи охорони праці.
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Схема електрична структурна
2. Блок-схема алгоритму
3. Схема електрична принципова
4. Схема електрична з'єднань

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці			

7. Дата видачі завдання *25.04.2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Розробка технічного завдання*	*26.01 – 02.02*	
2.	*Робота над першим розділом «Аналіз предметної області та обґрунтування переходу до розподіленої архітектури»*	*03.02 – 15.02*	
3.	*Робота над другим розділом «Проєктування розподіленої системи оцінювання завантаженості вулиць»*	*20.04 – 28.04*	
4.	*Робота над третім розділом «Практична реалізація системи та оцінка її ефективності»*	*29.04 – 08.05*	
5.	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*10.05 – 25.05*	
6.	*Оформлення пояснювальної записки і графічного матеріалу*	*26.05 – 7.06*	
7.	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*	
8.	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*	
9.	*Захист кваліфікаційної роботи бакалавра*	*24.06*	

Студент _____ *Котовський Д.М.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Тиш Є.В.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Котовський Д.М. Комп’ютерна система оцінювання завантаженості вулиць транспортними засобами та пішоходами: робота на здобуття кваліфікаційного ступеня бакалавра: спец. 123 — комп’ютерна інженерія. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2026.

Ключові слова: комп’ютерний зір, відеоаналітика, YOLOv8, розподілена архітектура, Raspberry Pi, моніторинг трафіку, FastAPI.

У кваліфікаційній роботі розроблено розподілену апаратно-програмну систему для оцінювання завантаженості вулиць транспортними засобами та пішоходами в реальному часі. Проаналізовано недоліки монолітних Edge-систем та обґрунтовано перехід до клієнт-серверної архітектури. Апаратний вузол на базі мікрокомп'ютера Raspberry Pi 3 виконує захоплення та трансляцію відеопотоку, що дозволило усунути проблему перегріву процесора. Обробка даних делегована високопродуктивному серверу, де реалізовано ізольований конвеєр комп'ютерного зору з використанням нейромережі YOLOv8m та алгоритму центроїдного трекінгу. Розроблено серверну частину на базі фреймворку FastAPI та інтерактивний вебдашборд, який отримує статистику через WebSockets за патерном Backend-Driven UI. Система забезпечує стабільну обробку відео з частотою 20 кадрів за секунду та точність підрахунку понад 97%.

ANNOTATION

Kotovskyi D.M. Computer System for Assessing Street Congestion by Vehicles and Pedestrians: Bachelor's Graduation Thesis: speciality 123 — computer engineering. Ternopil: Ternopil Ivan Puluj National Technical University, 2026.

Keywords: computer vision, video analytics, YOLOv8, distributed architecture, Raspberry Pi, traffic monitoring, FastAPI.

In the qualification work, a distributed hardware-software system for real-time assessment of street congestion by vehicles and pedestrians was developed. The disadvantages of monolithic Edge systems were analyzed, and the transition to a client-server architecture was justified. The hardware node based on the Raspberry Pi 3 microcomputer captures and streams the video, which eliminated the processor overheating problem. Data processing is delegated to a high-performance server, where an isolated computer vision pipeline is implemented using the YOLOv8m neural network and a centroid tracking algorithm. A server backend based on the FastAPI framework and an interactive web dashboard receiving statistics via WebSockets using the Backend-Driven UI pattern were developed. The system ensures stable video processing at 20 frames per second and a counting accuracy of over 97%.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

API – Application Programming Interface (Інтерфейс прикладного програмування)

CNN – Convolutional Neural Network (Згорткова нейронна мережа)

CPU – Central Processing Unit (Центральний процесор)

CSI – Camera Serial Interface (Послідовний інтерфейс камери)

CV – Computer Vision (Комп'ютерний зір)

DOM – Document Object Model (Об'єктна модель документа)

FPS – Frames Per Second (Кадри за секунду)

GPU – Graphics Processing Unit (Графічний процесор)

HTTP – HyperText Transfer Protocol (Протокол передачі гіпертексту)

IoT – Internet of Things (Інтернет речей)

JSON – JavaScript Object Notation (Текстовий формат обміну даними)

mAP – Mean Average Precision (Усереднена середня точність детекції)

NPU – Neural Processing Unit (Нейронний процесор / Neural Engine)

ORM – Object-Relational Mapping (Об'єктно-реляційне відображення)

REST – Representational State Transfer (Архітектурний стиль взаємодії компонентів)

ROI – Region of Interest (Зона інтересу)

SQL – Structured Query Language (Мова структурованих запитів)

TCP – Transmission Control Protocol (Протокол управління передачею)

UI – User Interface (Інтерфейс користувача)

YOLO – You Only Look Once (Нейромережа для детекції об'єктів)

ВСТУП

В умовах стрімкого розвитку концепції «Розумне місто» (Smart City) та постійного зростання темпів урбанізації, ефективне управління транспортною та пішохідною інфраструктурою набуває особливої актуальності. Зі збільшенням кількості транспортних засобів міста зіштовхуються з проблемою заторів, що вимагає точних даних про інтенсивність руху для оптимізації роботи світлофорів, розвантаження дорожніх артерій та підвищення загального рівня безпеки [6, 8].

Традиційні методи збору даних, такі як індукційні петлі або радари, часто є занадто дорогими у впровадженні та не здатні забезпечити гнучку класифікацію об'єктів (наприклад, відрізнити пішохода від легкового авто) [4]. Натомість системи відеоаналітики на базі комп'ютерного зору (Computer Vision) демонструють значний потенціал у контексті створення інтелектуальних транспортних систем. Моделі глибинного навчання, зокрема нейромережі типу YOLO, дозволяють виконувати детекцію об’єктів у реальному часі з високою точністю [16].

Проте, зростаючий інтерес до таких систем вимагає вирішення проблеми апаратних обмежень. Як показали попередні етапи досліджень, використання виключно локальної архітектури на малопотужних мікрокомп'ютерах (наприклад, Raspberry Pi) стикається з низькою частотою обробки кадрів та перегрівом процесора. Це стає передумовою для розробки гнучких, розподілених систем, де ресурсоємна аналітика делегується високопродуктивному серверу, а мікрокомп'ютер виконує роль вузла відеотрансляції. Такий підхід дає змогу розширити функціональність систем відеоспостереження та створювати ефективні, масштабовані рішення без залучення дороговартісного обладнання.

У межах кваліфікаційної роботи розглядається питання створення розподіленої комп’ютерної системи, що забезпечує розпізнавання, трекінг та підрахунок об’єктів дорожнього руху в режимі реального часу з виведенням

статистики на вебдашборд. Мета даної роботи полягає у проєктуванні, реалізації та тестуванні автономної системи оцінювання завантаженості вулиць транспортними засобами та пішоходами. Отже, тема цієї кваліфікаційної роботи бакалавра є актуальною, враховуючи високий попит на інтелектуальні, доступні за ціною та точні системи відеоаналітики, здатні працювати в розподіленому середовищі.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ПЕРЕХОДУ ДО РОЗПОДІЛЕНОЇ АРХІТЕКТУРИ

1.1 Огляд існуючих методів та систем моніторингу дорожнього трафіку

Розвиток сучасних міст та концепції інтелектуальних транспортних систем неможливий без впровадження надійних підсистем збору даних про параметри дорожнього руху. Отримання точної інформації про кількість транспортних засобів, їх класифікацію та інтенсивність пішохідних потоків є базовою вимогою для прийняття рішень щодо керування світлофорними об'єктами та планування міської інфраструктури. Існуючі методи моніторингу трафіку класифікують на дві основні категорії: інтрузивні (контактні) та неінтрузивні (безконтактні).

Інтрузивні методи передбачають фізичне втручання в дорожнє покриття під час їх встановлення. Найбільш поширеним представником цієї групи є індукційні петлі, які монтуються під верхній шар асфальту і реагують на зміну електромагнітного поля при проходженні металевого об'єкта. До переваг цього методу належить висока точність підрахунку транспортних засобів та незалежність від погодних умов чи рівня освітлення. Проте індукційні петлі мають ряд суттєвих недоліків: висока вартість монтажу та обслуговування, необхідність перекриття руху під час встановлення, а також вразливість до деформацій дорожнього полотна. Крім того, такі системи мають вкрай обмежені можливості щодо класифікації об'єктів і абсолютно не здатні фіксувати пішоходів чи велосипедистів [3].

До неінтрузивних методів належать мікрохвильові радари, інфрачервоні датчики, лідари та системи відеоаналітики. Радарні комплекси використовують ефект Доплера для вимірювання швидкості та підрахунку автомобілів. Вони демонструють високу надійність у складних погодних умовах, однак їхня здатність до точної класифікації об'єктів у щільному міському потоці залишається низькою. Лідарні системи забезпечують високу точність побудови тривимірної картини простору, проте їхня висока вартість робить масове впровадження економічно недоцільним для більшості муніципалітетів.

Окрему і найбільш перспективну нішу займають системи відеоаналітики на базі комп'ютерного зору. Використання цифрових камер у поєднанні з алгоритмами обробки зображень дозволяє отримувати найбільш повний спектр даних: не лише рахувати об'єкти, але й визначати їхній тип (легковий автомобіль, вантажівка, автобус, мотоцикл), колір, траєкторію руху та швидкість. Головною перевагою відеоаналітики є її універсальність – одна камера здатна одночасно моніторити як проїжджу частину, так і тротуари, забезпечуючи точний підрахунок пішохідних потоків, що є критично важливим для комплексного оцінювання завантаженості вулиць [5].

Історично головним недоліком систем відеоаналітики була їхня чутливість до умов освітлення (тіні, відблиски, нічний час) та погодних явищ. Однак із розвитком методів глибинного машинного навчання, зокрема згорткових нейронних мереж (CNN), ці проблеми вдалося значною мірою нівелювати. Сучасні архітектури нейромереж, такі як сімейство YOLO (You Only Look Once), здатні виконувати детекцію об'єктів у реальному часі з високою точністю навіть за складних візуальних умов [16].

Впровадження систем відеоаналітики може здійснюватися за двома основними архітектурними підходами: централізованим (хмарним) та локальним (Edge Computing). Хмарний підхід передбачає передачу "сирого" відеопотоку з камер на віддалені сервери для обробки, що створює значне

навантаження на канали зв'язку. Локальний підхід передбачає розміщення обчислювальних потужностей безпосередньо біля камери (наприклад, використання мікрокомп'ютерів). Проте, використання складних нейромережевих моделей вимагає значних апаратних ресурсів, що створює нові інженерні виклики при проєктуванні вбудованих систем [2].

1.2 Аналіз результатів попередніх досліджень та апаратних обмежень Edge-архітектури

Концепція граничних обчислень (Edge Computing) передбачає перенесення процесів обробки даних максимально близько до джерела їх виникнення, тобто безпосередньо на кінцеві IoT-пристрої. У контексті відеоаналітики це означає, що камера не просто транслює відеопотік, а самостійно виконує детекцію та класифікацію об'єктів за допомогою вбудованого мікрокомп'ютера. Такий підхід дозволяє суттєво розвантажити мережеву інфраструктуру, оскільки замість важкого відеопотоку на центральний сервер передаються лише легкі текстові метадані (наприклад, JSON-файли з кількістю розпізнаних автомобілів).

На попередніх етапах дослідження, у межах виконання курсового проєкту, було розроблено та протестовано прототип такої автономної системи на базі одноплатного мікрокомп'ютера Raspberry Pi 3 Model B. Для забезпечення роботи системи в умовах жорстко обмежених апаратних ресурсів було обрано легковажну згорткову нейромережу MobileNet SSD, яка оптимізована для мобільних пристроїв. Алгоритм відстеження об'єктів базувався на методі центроїдного трекінгу (Centroid Tracking).

Проведені натурні випробування прототипу виявили ряд критичних апаратних та алгоритмічних обмежень, які унеможливлюють використання такої архітектури для високонавантажених міських магістралей. Головною проблемою стала вкрай низька продуктивність системи: швидкість обробки

відеопотоку не перевищувала 4-5 кадрів за секунду (FPS). Це пояснюється тим, що процесор ARM Cortex-A53, встановлений на Raspberry Pi 3, не має спеціалізованих апаратних блоків (NPU або TPU) для прискорення тензорних обчислень, тому весь тягар інференсу нейромережі лягає на ядра центрального процесора, завантаження яких стабільно трималося на рівні 100%.

Низька частота кадрів має руйнівний вплив на точність підрахунку транспортних засобів. Автомобіль, що рухається з типовою міською швидкістю 50 км/год, долає близько 13,8 метрів за секунду. При частоті обробки 4 FPS часовий інтервал між сусідніми кадрами становить 0,25 секунди, за які об'єкт зміщується майже на 3,5 метри. Такі різкі просторові стрибки об'єктів між кадрами призводять до збоїв у роботі алгоритму трекінгу: система втрачає ідентифікатор (ID) автомобіля і привласнює йому новий, що спричиняє хибне подвоєння результатів підрахунку.

Другим суттєвим обмеженням став температурний режим роботи мікрокомп'ютера. Безперервне стовідсоткове завантаження процесора призводило до його швидкого нагрівання до 75-80 градусів за Цельсієм. При досягненні критичних температур операційна система автоматично активувала механізм тротлінгу (зниження тактової частоти для запобігання фізичному пошкодженню кристала), що призводило до подальшого падіння FPS до 1-2 кадрів за секунду та повної втрати працездатності системи в реальному часі.

Крім того, використання полегшеної моделі MobileNet SSD, хоч і було вимушеним кроком через слабкість апаратної платформи, негативно позначилося на загальній точності детекції (mAP). Модель демонструвала низьку ефективність при розпізнаванні дрібних об'єктів (пішоходів на дальній відстані) та в умовах часткового перекриття об'єктів один одним у щільному потоці.

Аналіз отриманих результатів дозволив зробити висновок, що побудова системи відеоаналітики виключно на базі бюджетних одноплатних

комп'ютерів (Edge-архітектура) є неефективною для задач, що вимагають використання сучасних, високоточних нейромереж. Для забезпечення стабільної частоти обробки на рівні 20 FPS та використання передових моделей детекції виникає об'єктивна необхідність у перегляді архітектурних підходів та винесенні обчислювального навантаження за межі кінцевого пристрою.

1.3 Обґрунтування доцільності переходу до розподіленої клієнт-серверної архітектури

Виявлені апаратні обмеження мікрокомп'ютерів при виконанні ресурсоємних завдань комп'ютерного зору зумовлюють необхідність зміни архітектурної парадигми. Оптимальним рішенням для забезпечення високої точності та швидкодії є перехід від концепції граничних обчислень (Edge Computing) до розподіленої клієнт-серверної архітектури. Суть цього підходу полягає у чіткому розділенні функцій захоплення даних та їх інтелектуальної обробки між різними апаратними вузлами.

У запропонованій розподіленій моделі мікрокомп'ютер Raspberry Pi виконує виключно роль вузла відеотрансляції (Thin Client). Його завдання зводиться до апаратного захоплення відеопотоку з камери, його стиснення (наприклад, у формат H.264) та передачі по локальній мережі за допомогою протоколу TCP. Оскільки трансляція відеопотоку з використанням апаратного кодувальника вимагає мінімальних обчислювальних ресурсів, загальне завантаження центрального процесора Raspberry Pi знижується в середньому до 20-25% (показник Load Average становить близько 1.00 при 4 доступних апаратних ядрах). Завдяки цьому повністю вирішується проблема критичного перегріву: як показали натурні тестування, навіть у спекотні літні дні робоча температура процесора стабільно тримається в межах 58-65 °C. Цей показник є значно нижчим за поріг активації апаратного тротлінгу (80-85 °C), що

дозволяє пристрою безперебійно працювати в автономному режимі 24/7 без необхідності встановлення активних систем охолодження.

У свою чергу, завдання з детекції, трекінгу та класифікації об'єктів переносяться на виділений високопродуктивний сервер обробки (Thick Server). Використання сучасних серверних або десктопних процесорів (зокрема, архітектури Apple Silicon із вбудованими співпроцесорами Neural Engine для апаратного прискорення машинного навчання) дозволяє кардинально змінити програмний стек системи.

По-перше, зняття жорстких апаратних обмежень дає змогу відмовитися від компромісних легковажних нейромереж (MobileNet) на користь передових архітектур, таких як YOLOv8 (You Only Look Once, версія 8). Моделі сімейства YOLO забезпечують найвищий на сьогодні рівень точності (mAP) при розпізнаванні об'єктів у реальному часі. Вони здатні ефективно виявляти дрібні об'єкти, працювати в умовах часткового перекриття транспортних засобів та адаптуватися до складних умов освітлення.

По-друге, потужний сервер здатний обробляти відеопотік із частотою 20 і більше кадрів за секунду. Як було зазначено раніше, високий FPS є критично важливою умовою для коректної роботи алгоритмів трекінгу (наприклад, Centroid Tracker або BoT-SORT). При частоті 20 FPS зміщення автомобіля між кадрами становить менше півметра, що гарантує надійне утримання ідентифікатора (ID) об'єкта та виключає можливість хибного дублювання при підрахунку перетинів контрольної лінії.

Крім того, перехід до клієнт-серверної архітектури відкриває широкі можливості для масштабування та інтеграції. Централізований сервер може одночасно приймати та обробляти відеопотоки з кількох IoT-камер, встановлених в різних місцях. Також на боці сервера стає можливою реалізація повноцінного вебзастосунку з використанням сучасних фреймворків (наприклад, FastAPI). Це дозволяє не лише зберігати статистику в реляційних

базах даних, але й організувати багатокористувацький доступ до аналітики через інтерактивні вебдашборди в режимі реального часу.

Таким чином, розподілена клієнт-серверна архітектура забезпечує ідеальний баланс між вартістю інфраструктури та її продуктивністю. Дешеві мікрокомп'ютери використовуються як масові точки збору даних, а дорогі обчислювальні потужності централізуються, що робить розроблену систему економічно доцільною для впровадження в інфраструктуру «Розумного міста».

1.4 Формування технічних вимог до апаратно-програмного комплексу

На основі проведеного аналізу предметної області, виявлених апаратних обмежень вбудованих систем та обґрунтування переходу до розподіленої архітектури, було сформовано комплекс технічних вимог до розроблюваної системи оцінювання завантаженості вулиць. Вимоги поділяються на апаратні, функціональні та нефункціональні (вимоги до продуктивності та надійності).

Вимоги до апаратного забезпечення:

– вузол відеофіксації (Thin Client) повинен базуватися на мікрокомп'ютері рівня Raspberry Pi 3 Model B (або подібному) з підключеним модулем камери через інтерфейс CSI для мінімізації затримок;

– сервер обробки (Thick Server) повинен володіти достатньою обчислювальною потужністю для запуску сучасних нейромереж. Цільовою платформою визначено комп'ютери на базі архітектури Apple Silicon (процесори серії M) з підтримкою апаратного прискорення через Neural Engine.

Функціональні вимоги до програмного забезпечення:

– підсистема трансляції повинна забезпечувати безперебійне захоплення відеопотоку та його передачу по локальній мережі за протоколом TCP;

– модуль комп'ютерного зору повинен виконувати детекцію об'єктів у реальному часі з використанням нейромережі YOLOv8, розпізнаючи ключові класи: пішоходи, легкові автомобілі, вантажівки, автобуси та мотоцикли;

– система повинна включати алгоритм трекінгу (відстеження переміщення центроїдів об'єктів між кадрами) для запобігання хибному дублюванню підрахунку;

– підрахунок об'єктів має здійснюватися за фактом перетину віртуальної контрольної лінії із записом події (тип об'єкта, мітка часу, стан трафіку) до реляційної бази даних;

– вебсервер повинен надавати REST API для отримання історичних даних та підтримувати протокол WebSocket для двосторонньої взаємодії з клієнтами;

– графічний інтерфейс (вебдашборд) повинен відображати відеотрансляцію з накладеною графікою (Bounding Boxes, лічильники) та інтерактивні графіки статистики.

Вимоги до продуктивності та оптимізації ресурсів:

– швидкість обробки відеопотоку на стороні сервера повинна становити не менше 20 кадрів за секунду (FPS) для забезпечення високої точності трекінгу;

– температурний режим роботи мікрокомп'ютера Raspberry Pi під час трансляції не повинен перевищувати 65-70 °C, а завантаження процесора має залишатися в межах 25%;

– система повинна підтримувати механізм розумного рендерингу (Smart Rendering): кодування кадрів у формат JPEG для вебінтерфейсу має виконуватися виключно за наявності активних підключень (глядачів) на дашборді, що дозволить суттєво економити ресурси центрального процесора сервера;

– архітектура вебдашборду повинна базуватися на патерні Backend-Driven UI, де сервер виступає єдиним джерелом правди (Single Source of Truth) для генерації часових міток, що виключає проблему розсинхронізації часових поясів на клієнтських пристроях.

Вимоги до надійності та відмовостійкості:

– програмне забезпечення сервера повинно автоматично виявляти втрату зв'язку з камерою (відсутність кадрів) та ініціювати процес перепідключення по SSH;

– архітектура бази даних повинна підтримувати систему міграцій для безпечного оновлення структури таблиць;

– запис логів до бази даних має виконуватися в ізольованому фоновому потоці (через чергу повідомлень) з використанням пакетного вставлення (Batch Inserts), щоб уникнути блокування конвеєра комп'ютерного зору операціями вводу-виводу.

Сформульовані вимоги є основою для подальшого проєктування архітектури системи, вибору конкретних програмних фреймворків та практичної реалізації апаратно-програмного комплексу, що буде детально розглянуто в наступних розділах роботи.

РОЗДІЛ 2 ПРОЄКТУВАННЯ РОЗПОДІЛЕНОЇ СИСТЕМИ ОЦІНЮВАННЯ ЗАВАНТАЖЕНОСТІ ВУЛИЦЬ

2.1 Розробка узагальненої структури апаратно-програмного комплексу

Для задоволення сформованих технічних вимог та подолання апаратних обмежень вбудованих систем, розроблюваний апаратно-програмний комплекс спроєктовано за принципами розподіленої клієнт-серверної архітектури. Загальна структура системи базується на концепції Domain-Driven Design (предметно-орієнтоване проєктування), що передбачає чітке логічне та фізичне розділення інфраструктурного рівня (захоплення та обробка відео) від рівня бізнес-логіки (збереження даних та надання доступу користувачам).

Узагальнену структурну схему розробленого апаратно-програмного комплексу наведено на рис. 2.1. Архітектурно систему можна розділити на три основні логічні блоки: вузол відеофіксації (Thin Client), сервер обробки та аналітики (Thick Server) та клієнтський вебзастосунок (Frontend Dashboard).

Вузол відеофіксації є апаратним компонентом системи, що розміщується безпосередньо в зоні моніторингу. Його базовим елементом є мікрокомп'ютер, до якого через спеціалізований шлейф підключено модуль цифрової камери. Головна і єдина функція цього вузла полягає у безперервному захопленні відеокадрів, їх апаратному стисненні та трансляції у локальну мережу через TCP-сокет. Відсутність будь-якої аналітичної логіки на цьому вузлі гарантує його високу енергоефективність та стійкість до перегріву.

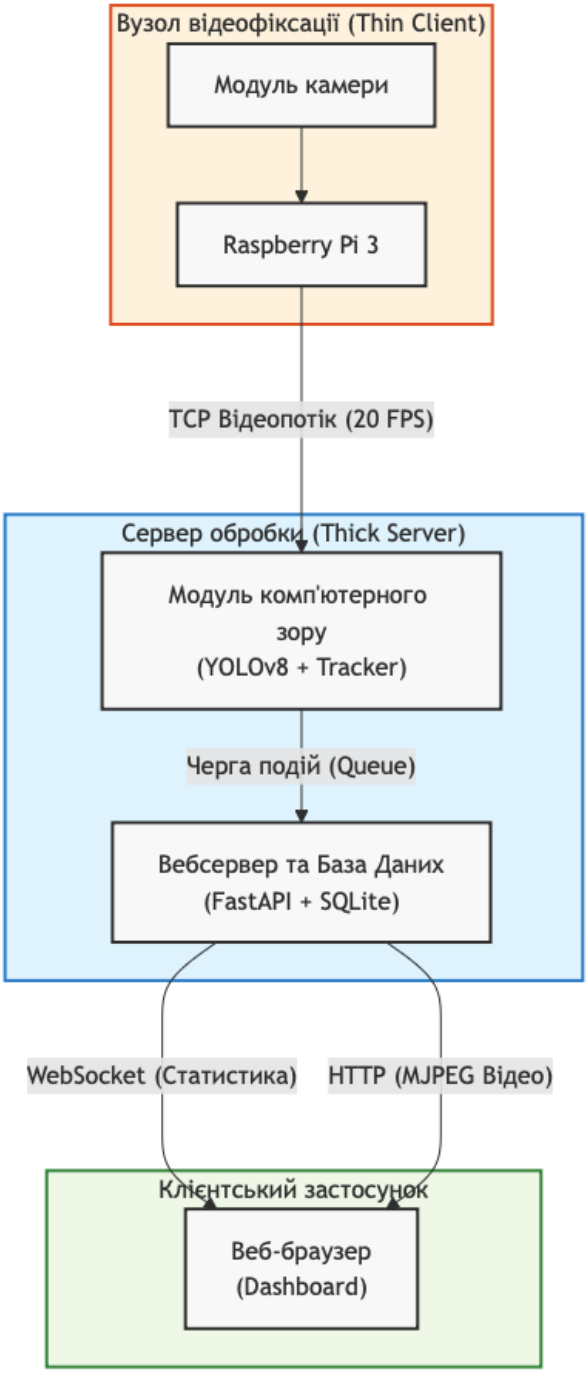

Рисунок 2.1 – Структурна схема апаратно-програмного комплексу

Сервер обробки та аналітики є центральним ядром системи, яке розгортається на високопродуктивній обчислювальній платформі. Програмна архітектура сервера фізично розділена на два незалежні домени, які взаємодіють між собою через потокобезпечну чергу повідомлень (Message Queue).

Перший домен сервера – це конвеєр комп'ютерного зору. Він відповідає за підключення до TCP-потоку камери, декодування кадрів та їх передачу до згорткової нейронної мережі для детекції об'єктів. Після отримання координат об'єктів (Bounding Boxes), у роботу вступає алгоритм трекінгу, який відстежує траєкторію руху кожного транспортного засобу чи пішохода. При перетині віртуальної контрольної лінії конвеєр генерує подію (Event), яка містить тип об'єкта та поточний стан завантаженості дороги, після чого ця подія передається до другого домена.

Другий домен сервера – це вебсервер та рівень бази даних. Він працює в ізольованому асинхронному потоці, що гарантує відсутність блокувань (I/O Blocking) для конвеєра комп'ютерного зору. Цей домен приймає події з черги та виконує їх пакетне збереження (Batch Inserts) до реляційної бази даних. Одночасно з цим, вебсервер агрегує історичні дані та керує пулом активних WebSocket-з'єднань, розсилаючи оновлену статистику всім підключеним клієнтам у режимі реального часу.

Клієнтський вебзастосунок є кінцевою точкою взаємодії користувача із системою. Він реалізований за принципом «тонкого клієнта», де вся логіка (розрахунок часових інтервалів, визначення станів трафіку) виконується на бекенді. Браузер користувача лише встановлює WebSocket-з'єднання для отримання готових метрик та відображає їх у вигляді інтерактивних графіків, а також приймає MJPEG-потік для візуалізації процесу розпізнавання.

Запропонована структура забезпечує високу модульність системи. Вихід з ладу або оновлення одного з компонентів (наприклад, заміна моделі нейромережі або оновлення вебфреймворку) не вимагає втручання в роботу інших вузлів, що є критично важливою вимогою для сучасних систем моніторингу.

2.2 Обґрунтування вибору апаратного забезпечення для вузла трансляції та сервера обробки

Реалізація розподіленої архітектури вимагає ретельного підбору апаратних компонентів, які б відповідали специфічним вимогам кожного з вузлів системи. Апаратна база поділяється на дві складові: пристрій для захоплення та трансляції відеопотоку (Thin Client) та високопродуктивний сервер для інференсу нейромереж і маршрутизації даних (Thick Server).

В якості апаратної платформи для вузла відеофіксації було обрано одноплатний мікрокомп'ютер Raspberry Pi 3 Model B. Незважаючи на те, що обчислювальної потужності його процесора (ARM Cortex-A53) недостатньо для запуску сучасних моделей комп'ютерного зору, цей пристрій є ідеальним рішенням для ролі "тонкого клієнта". Ключовою перевагою Raspberry Pi є наявність спеціалізованого апаратного інтерфейсу CSI-2 (Camera Serial Interface), до якого підключається модуль камери (Raspberry Pi Camera Module). На відміну від стандартних USB-камер, інтерфейс CSI забезпечує прямий доступ до графічного співпроцесора (VideoCore IV), минаючи центральний процесор [19].

Це дозволяє використовувати вбудований апаратний кодувальник (Hardware Encoder) для стиснення "сирого" відеопотоку у формат H.264 "на льоту". Завдяки цьому, процес трансляції відео з частотою 20 кадрів за секунду створює мінімальне навантаження на систему. Як було встановлено під час тестування, загальне завантаження процесора становить лише 20-25%, а робоча температура не перевищує 65 °C. Такі показники гарантують високу надійність вузла при його розміщенні у вуличних умовах (наприклад, у захисних боксах на стовпах) без ризику перегріву та без необхідності використання вентиляторів.

В якості сервера обробки та аналітики було обрано комп'ютер на базі архітектури Apple Silicon (процесори серії M). Вибір цієї платформи

зумовлений її унікальними архітектурними особливостями, які ідеально підходять для завдань машинного навчання та розгортання бекенд-сервісів.

Головним аргументом на користь Apple Silicon є наявність інтегрованого нейропроцесора – 16-ядерного співпроцесора Neural Engine (NPU), який апаратно прискорює виконання матричних та тензорних операцій. Традиційні сервери вимагають встановлення дискретних відеокарт (GPU) для досягнення прийнятної швидкості роботи нейромереж типу YOLO, що суттєво збільшує їхню вартість та енергоспоживання. Натомість використання Neural Engine дозволяє виконувати інференс важких згорткових мереж з мінімальними енергозатратами [23].

Для максимальної утилізації апаратних можливостей Apple Silicon, оригінальна модель YOLOv8 (у форматі PyTorch) конвертується у спеціалізований формат CoreML. Це дозволяє операційній системі macOS перенести майже весь тягар тензорних обчислень на Neural Engine, практично повністю розвантаживши центральний процесор (CPU). Як показали результати тестування на базі чипа Apple M1, сервер здатний обробляти вхідний відеопотік із частотою понад 20 FPS при мінімальному енергоспоживанні. Температура процесора під час безперервного трекінгу стабільно тримається на рівні 50-60 °C, що дозволяє системі працювати в абсолютно безшумному режимі (кулери залишаються вимкненими). Таке радикальне зниження навантаження на CPU гарантує наявність величезного запасу вільних ресурсів для паралельної та безперебійної роботи асинхронного вебсервера FastAPI, системи управління базою даних SQLite та пулу WebSocket-з'єднань, навіть якщо серверна машина паралельно використовується для інших завдань.

Таким чином, обрана апаратна конфігурація забезпечує синергетичний ефект: дешевий та енергоефективний Raspberry Pi виконує роль надійного джерела даних, а потужний сервер на базі Apple Silicon гарантує безкомпромісну швидкість та точність їх аналізу.

2.3 Обґрунтування вибору програмного стеку та фреймворків

Реалізація розподіленої архітектури та забезпечення взаємодії між модулями комп'ютерного зору, базою даних і клієнтським вебзастосунком вимагає використання сучасного, високопродуктивного та надійного програмного стеку. Базовою мовою програмування для розробки серверної частини та конвеєра обробки відео було обрано Python. Цей вибір зумовлений наявністю найпотужнішої екосистеми бібліотек для машинного навчання та комп'ютерного зору, а також підтримкою сучасних механізмів асинхронного програмування.

Для вирішення завдань комп'ютерного зору базовим інструментом обрано бібліотеку OpenCV, яка забезпечує ефективну роботу з матрицями зображень, зміну розміру кадрів, накладання графічних елементів (Bounding Boxes) та кодування обробленого відеопотоку у формат MJPEG для трансляції на клієнт [18]. У якості ядра інтелектуальної аналітики використано фреймворк Ultralytics та модель YOLOv8. На відміну від попередніх поколінь, восьма версія YOLO пропонує найкращий компроміс між точністю (mAP) та швидкістю інференсу, а також має вбудовану підтримку експорту моделей у формат CoreML для апаратного прискорення на процесорах Apple Silicon [16, 23].

Основою для побудови серверної частини та маршрутизації обрано асинхронний вебфреймворк FastAPI. На відміну від традиційних синхронних фреймворків (наприклад, Django або Flask), FastAPI побудований поверх ASGI-сервера Starlette, що забезпечує надзвичайно високу пропускну здатність, порівнянну з рішеннями на базі Node.js або Go. Нативна підтримка асинхронності є критично важливою для даного проєкту, оскільки дозволяє серверу одночасно обробляти вхідний відеопотік, виконувати запити до бази даних та підтримувати постійні WebSocket-з'єднання з клієнтами без блокування основного потоку виконання (I/O Blocking) [17, 24].

Для управління конфігураціями та секретними даними (наприклад, SSH-доступами до камери) застосовано бібліотеку Pydantic Settings. Вона забезпечує сувору типізацію та автоматичну валідацію змінних оточення з файлу .env. Такий підхід реалізує принцип «швидкого збою» (Fail-Fast): якщо конфігураційний файл відсутній або містить некоректні типи даних, система миттєво зупиняє запуск із чітким повідомленням про помилку, що унеможливлює виникнення прихованих збоїв під час роботи [21].

Збереження історичних даних про трафік реалізовано на базі легкої реляційної системи управління базами даних SQLite. Оскільки система генерує прості текстові логи (мітки часу, типи об'єктів та стани трафіку), використання важких серверних СУБД (наприклад, PostgreSQL) на даному етапі є надлишковим. Взаємодія з базою даних здійснюється через бібліотеку об'єктно-реляційного відображення (ORM) SQLAlchemy. Це дозволяє абстрагуватися від сирих SQL-запитів, підвищує безпеку (захист від SQL-ін'єкцій) та дає змогу в майбутньому легко мігрувати на іншу СУБД без зміни бізнес-логіки. Для керування версіями схеми бази даних інтегровано інструмент Alembic, який забезпечує безпечне застосування міграцій при оновленні структури таблиць [25, 27].

Клієнтська частина (вебдашборд) побудована з використанням класичного стеку HTML5, CSS3 (фреймворк Bootstrap 5 для адаптивної верстки) та чистого JavaScript (Vanilla JS). Відмова від використання важких фронтенд-фреймворків (React, Vue) обґрунтована застосуванням архітектурного патерну Backend-Driven UI. Оскільки вся складна бізнес-логіка, включаючи розрахунок часових інтервалів та агрегацію статистики, виконується на бекенді, завдання фронтенду зводиться виключно до встановлення WebSocket-з'єднання та оновлення DOM-дерева. Для візуалізації історичних даних інтегровано бібліотеку Chart.js, яка забезпечує плавне та оптимізоване відмальовування графіків з підтримкою анімацій [22].

2.4 Проєктування алгоритму детекції та трекінгу транспортних засобів і пішоходів

Для забезпечення високої точності підрахунку об'єктів у реальному часі розроблено багатоетапний алгоритмічний конвеєр (Pipeline), який складається з чотирьох послідовних стадій: детекції, просторової фільтрації, трекінгу та логічного підрахунку.

Першим етапом є детекція об'єктів на вхідному кадрі. Після попередньої обробки (зміни розміру та нормалізації) кадр передається на вхід нейромережі YOLOv8. Оскільки модель здатна розпізнавати десятки різних класів, алгоритм виконує жорстку фільтрацію результатів. З усього масиву знайдених об'єктів система залишає лише ті, що належать до цільових класів: пішоходи, легкові автомобілі, мотоцикли, автобуси та вантажівки. Додатково застосовується поріг впевненості (Confidence Threshold), який відсікає хибні спрацювання (False Positives), якщо нейромережа не впевнена у класифікації об'єкта на достатньому рівні (наприклад, менше 25%).

Другим етапом є просторова фільтрація за допомогою зони інтересу (Region of Interest, ROI). Для уникнення підрахунку об'єктів, що знаходяться поза межами проїжджої частини або тротуару (наприклад, автомобілів на парковці на задньому фоні), програмно задається віртуальний полігон дороги. Алгоритм перевіряє координати кожного знайденого об'єкта: якщо його центр знаходиться поза межами заданого полігону, об'єкт ігнорується і не передається на наступні етапи обробки.

Третім, критично важливим етапом є трекінг (відстеження) об'єктів. Оскільки нейромережа аналізує кожен кадр незалежно, вона не зберігає інформацію про те, що автомобіль на поточному кадрі і на попередньому – це один і той самий фізичний об'єкт. Для вирішення цієї проблеми спроєктовано алгоритм центроїдного трекінгу (Centroid Tracking).

Алгоритм трекінгу працює за наступним принципом:

1. Для кожної обмежувальної рамки (Bounding Box), отриманої від нейромережі, обчислюється її геометричний центр (центроїд) за координатами X та Y.

2. Система порівнює центроїди поточного кадру з центроїдами об'єктів, зафіксованих на попередньому кадрі, обчислюючи Евклідову відстань між ними.

3. Якщо відстань між точками є мінімальною і не перевищує заданого порогу, алгоритм робить висновок, що це той самий об'єкт, який просто змістився у просторі, і зберігає за ним його унікальний ідентифікатор (ID).

4. Якщо для існуючого ID не знайдено відповідного центроїда на новому кадрі протягом певної кількості ітерацій (наприклад, 40 кадрів), об'єкт вважається таким, що покинув зону видимості, і його ID видаляється з оперативної пам'яті (Deregistration).

Четвертим етапом є логічний підрахунок. На зображенні програмно задається віртуальна контрольна лінія (по осі X або Y). Алгоритм аналізує вектор руху кожного ідентифікованого об'єкта, порівнюючи його попередню та поточну позиції відносно цієї лінії. Якщо на попередньому кадрі центроїд об'єкта знаходився до лінії, а на поточному – після неї (або навпаки), фіксується факт перетину.

Для запобігання дублюванню підрахунку (наприклад, якщо об'єкт зупинився прямо на лінії або його координати "тремтять" через похибки детекції), система використовує кеш зареєстрованих ідентифікаторів. Після першого перетину ID об'єкта заноситься до спеціальної хеш-таблиці, і всі його подальші переміщення ігноруються лічильником. Для запобігання витокам пам'яті (Memory Leaks) розмір цього кешу жорстко обмежений, і старі записи автоматично витісняються новими за принципом FIFO (First-In, First-Out).

Запропонований алгоритмічний конвеєр забезпечує високу точність підрахунку, стійкість до перекриття об'єктів (Occlusion) та оптимально

використовує обчислювальні ресурси сервера.

2.5 Проєктування бази даних та моделі збереження історичних логів

Для накопичення статистики, побудови історичних графіків та подальшого аналізу інтенсивності дорожнього руху спроєктовано підсистему збереження даних. Враховуючи характер даних (прості текстові логи без складних зв'язків), базовою системою управління базами даних (СУБД) обрано SQLite. Взаємодія з базою даних реалізована на рівні об'єктно-реляційного відображення (ORM) за допомогою бібліотеки SQLAlchemy, що забезпечує високий рівень абстракції та захист від SQL-ін'єкцій.

Основним елементом моделі даних є таблиця traffic_logs, яка фіксує кожен факт перетину контрольної лінії розпізнаним об'єктом. Детальна структура та типи даних полів цієї таблиці наведені у табл. 2.1.

Таблиця 2.1 – Структура таблиці бази даних traffic_logs

Назва поля	Тип даних	Опис
id	Integer	Унікальний ідентифікатор запису (Primary Key) з автоінкрементом
timestamp	DateTime	Точна мітка часу фіксації події (зберігається у форматі UTC)
object_class	Enum	Тип розпізнаного об'єкта (PERSON, CAR, MOTORCYCLE, BUS, TRUCK)
traffic_state	Enum	Стан завантаженості дороги на момент фіксації (FREE, NORMAL, HEAVY)

Важливим архітектурним рішенням є збереження міток часу (timestamp) виключно у всесвітньому координованому часі (UTC). Такий підхід гарантує відсутність проблем при переході на літній/зимовий час та унеможливлює

розсинхронізацію даних при доступі до системи клієнтів з різних часових поясів. Конвертація часу в локальний часовий пояс користувача відбувається виключно на етапі форматування відповідей через REST API.

Для забезпечення цілісності даних на рівні бази, поля object_class та traffic_state реалізовані з використанням перерахувань (Enum). Це унеможливлює запис до бази некоректних або непередбачених типів даних. Керування схемою бази даних (створення таблиць, зміна типів колонок) здійснюється за допомогою системи міграцій Alembic, що дозволяє безпечно оновлювати структуру БД у процесі розвитку проєкту без втрати накопиченої історії.

Критичним викликом при проєктуванні систем реального часу є проблема блокування вводу-виводу (I/O Blocking). Оскільки СУБД SQLite блокує файл бази даних на час виконання транзакції запису, пряме звернення до бази з головного циклу комп'ютерного зору призвело б до мікрозатримок та падіння загальної частоти кадрів (FPS).

Для вирішення цієї проблеми спроєктовано механізм асинхронного фонового запису. Головний конвеєр обробки відео не взаємодіє з базою даних напряму, а лише поміщає згенеровані події до потокобезпечної черги повідомлень (queue.Queue). Паралельно з цим працює ізольований фоновий потік (Worker), який зчитує події з черги та виконує їх збереження. З метою мінімізації кількості транзакцій та зниження навантаження на дискову підсистему, воркер використовує механізм пакетного вставлення (Batch Inserts): він накопичує події протягом короткого таймауту і записує їх у базу даних єдиним блоком. Така архітектура гарантує, що операції збереження даних жодним чином не впливають на продуктивність нейромережі та алгоритмів трекінгу.

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ ТА ОЦІНКА ЇЇ ЕФЕКТИВНОСТІ

3.1 Реалізація підсистеми захоплення та трансляції відеопотоку на базі Raspberry Pi

Практична реалізація вузла відеофіксації (Thin Client) базується на використанні вбудованих апаратних можливостей мікрокомп'ютера Raspberry Pi та програмного стеку libcamera. Для забезпечення максимальної автономності системи було прийнято рішення відмовитися від ручного запуску трансляції на мікрокомп'ютері. Натомість реалізовано механізм віддаленого програмного керування через протокол SSH, що дозволяє центральному серверу повністю оркеструвати життєвий цикл відеопотоку.

Для реалізації цього механізму мовою Python розроблено клас-контролер SSHCameraController, який використовує бібліотеку Paramiko для встановлення захищеного з'єднання. З метою забезпечення безпечного управління ресурсами та уникнення витоків мережевих з'єднань, клас реалізує патерн «Контекстний менеджер» (Context Manager), перевизначаючи магічні методи ініціалізації та завершення контексту. Це гарантує, що при будь-якому аварійному завершенні роботи головної програми, SSH-сесія буде коректно закрита, а процес трансляції на Raspberry Pi – примусово зупинений [29].

Процес ініціалізації трансляції складається з кількох послідовних кроків. Спочатку контролер зчитує конфігураційні дані (IP-адресу, логін та пароль) з об'єкта налаштувань, згенерованого бібліотекою Pydantic. Після успішної авторизації на Raspberry Pi, сервер надсилає команду для примусового завершення всіх попередніх сесій камери, щоб звільнити апаратний інтерфейс CSI.

Наступним кроком є запуск самої трансляції за допомогою утиліти rpicam-vid. Фрагмент коду, що відповідає за формування команди та ініціалізацію відеопотоку через SSH, наведено на рис. 3.1. Команда формується таким чином, щоб активувати апаратне кодування у формат H.264, встановити частоту кадрів на рівні 20 FPS та запустити процес у фоновому режимі (через утиліту nohup). Відеопотік спрямовується на TCP-сокет, який очікує підключення від сервера.

```
logger.info("Запуск трансляції...")
cmd = (
    f"nohup rpicam-vid -t 0 --framerate 20 "
    f"--saturation 0 --inline --listen -o "
    f"tcp://0.0.0.0:{StreamConfig.STREAM_PORT} "
    f">/dev/null 2>&1 &"
)
self.ssh.exec_command(cmd)

logger.info("Очікування підняття порту...")
wait_cmd = (
    f"timeout {StreamConfig.STREAM_TIMEOUT} bash -c "
    f"'while ! ss -ltn | grep -q :{StreamConfig.STREAM_PORT}; "
    "do sleep 0.1; done; echo READY'"
)
_, stdout, _ = self.ssh.exec_command(wait_cmd)
```

Рисунок 3.1 – Лістинг ініціалізації відеопотоку через SSH.

Оскільки ініціалізація камери та підняття мережевого порту на Raspberry Pi потребує певного часу (зазвичай 1-2 секунди), у контролері реалізовано механізм активного очікування (Polling). Сервер циклічно опитує стан мережевих інтерфейсів мікрокомп'ютера, очікуючи появи відкритого порту трансляції. Лише після отримання підтвердження про готовність порту, контролер повертає управління головному конвеєру комп'ютерного зору для початку зчитування кадрів.

Для забезпечення високої відмовостійкості системи, у головному циклі обробки відео (клас VideoPipeline) реалізовано механізм автоматичного відновлення з'єднання. Якщо сервер фіксує відсутність нових кадрів у відеопотоці протягом заданого таймауту (наприклад, через тимчасову втрату

Wi-Fi сигналу), він автоматично розриває поточне з'єднання, витримує паузу та ініціює процес підключення до камери з самого початку. Це дозволяє системі працювати в режимі 24/7 без втручання оператора.

3.2 Програмна реалізація ізольованого конвеєра комп'ютерного зору

Практична реалізація розробленого апаратно-програмного комплексу базується на принципах модульності та предметно-орієнтованого проєктування (Domain-Driven Design). Для забезпечення зручності підтримки та ізоляції бізнес-логіки від інфраструктурних компонентів, кодову базу проєкту було чітко розділено на відповідні директорії (рис. 3.2).

```
TrafficCounter/
├── camera/                    # Домен комп'ютерного зору (CV)
│   ├── camera_controller.py # SSH-менеджер для підключення до Raspberry Pi
│   ├── line_counter.py      # Математика перетину об'єктами контрольних ліній
│   ├── overlay.py           # Відмальовка UI елементів поверх відео
│   ├── state.py             # Thread-safe керування станом (Shared Memory)
│   ├── traffic_state.py     # Логіка визначення інтенсивності (Free, Normal, Heavy)
│   └── video_processor.py   # Головний конвеєр: захоплення кадру -> YOLO -> трекінг
├── web/                       # Домен веб-сервера та бізнес-логіки
│   ├── alembic/             # Файли міграцій БД
│   ├── static/              # CSS стилі та JS (дашборд з Chart.js)
│   ├── templates/           # HTML шаблони
│   ├── app.py               # Ініціалізація FastAPI додатку
│   ├── db.py                # Підключення до SQLite (SQLAlchemy engine)
│   ├── history_service.py   # Логіка вибірки даних та форматування часових міток
│   ├── models.py            # ORM моделі таблиць (TrafficLog)
│   ├── routes.py            # API ендпоінти та WebSocket Broadcast Manager
│   ├── schemas.py           # Pydantic схеми для валідації API запитів/відповідей
│   └── workers.py           # Фонові задачі (запис статистики в БД)
├── yolov8l.mlpackage/       # Скомпільована модель під Apple Neural Engine
├── .env                       # Локальні секрети (IP, логіни)
├── .gitignore                 # Правила ігнорування файлів для Git
├── alembic.ini                # Конфігурація системи міграцій
├── config.py                  # Єдине джерело правди для налаштувань (Pydantic + Classes)
├── custom_tracker.yaml        # Налаштування алгоритму трекінгу об'єктів (ByteTrack/BoT-SORT)
├── logging_config.py          # Централізоване налаштування логування системи
├── main.py                    # Точка входу, що об'єднує CV-потік та Веб-сервер
├── pyproject.toml             # Файл конфігурації проєкту та список залежностей
├── traffic.db                 # Локальна SQLite база даних (генерується автоматично)
├── traffic.log                # Системні логи
└── uv.lock                    # Жорстко зафіксовані версії залежностей (lock-файл)
```

Рисунок 3.2 – Структура директорій та файлів програмного комплексу

Основою програмної реалізації інтелектуальної обробки відеоданих є клас VideoPipeline. Його головна задача – об'єднати процеси отримання кадрів, детекції об'єктів, їх трекінгу та логічного підрахунку в єдиний безперервний конвеєр. Для забезпечення максимальної продуктивності цей конвеєр запускається в окремому фоновому потоці (Thread), що повністю ізолює його від операцій вебсервера та бази даних.

Ініціалізація конвеєра починається із завантаження моделі комп'ютерного зору. У проєкті використано бібліотеку ultralytics, яка надає зручний інтерфейс для роботи з нейромережею YOLOv8. Модель завантажується у скомпільованому форматі CoreML (.mlpackage), що автоматично активує апаратне прискорення на співпроцесорі Neural Engine. Одночасно з цим ініціалізуються допоміжні класи: LineCounter для математичного підрахунку перетинів та TrafficStateTracker для аналізу інтенсивності руху методом ковзного вікна.

Процес покадрової обробки відеопотоку реалізовано у внутрішньому методі _process_stream. Алгоритм зчитує кадри з об'єкта VideoStream, виконує їх масштабування до заданої роздільної здатності та передає на вхід методу model.track(). Фрагмент коду, що відповідає за детекцію та реєстрацію перетинів, наведено на рис. 3.3.

```
results = self.model.track( #
    frame, conf=MLConfig.CONFIDENCE_THRESHOLD,
    iou=MLConfig.IOU_THRESHOLD, persist=True,
    tracker=MLConfig.TRACKER_CONFIG,
    classes=MLConfig.TARGET_CLASSES,
    imgsz=MLConfig.INFERENCE_SIZE, verbose=False
)

if results and len(results):
    events, new_crossings = self.counter.process_tracks(
        results[0].boxes, self.model.names
    )
    if new_crossings:
        for obj_class in new_crossings:
            self.traffic_state.register_crossing()
            if obj_class == "PERSON":
                self.total_persons += 1
            else:
                self.total_transport += 1

        intensity, state, state_color = (
            self.traffic_state.get_metrics()
        )
        for obj_class in new_crossings:
            global_state.log_queue.put((obj_class, state))
    else:
        intensity, state, state_color = (
            self.traffic_state.get_metrics()
        )
else:
    events = []
    intensity, state, state_color = (
        self.traffic_state.get_metrics()
    )
```

Рисунок 3.3 – Лістинг алгоритму детекції та реєстрації перетинів

Як видно з лістингу на рис. 3.3, метод track інкапсулює в собі як саму детекцію об'єктів, так і алгоритм трекінгу, повертаючи масив обмежувальних рамок (Bounding Boxes) із прив'язаними до них унікальними ідентифікаторами. Після успішної детекції результати передаються до класу LineCounter. Якщо алгоритм фіксує факт перетину віртуальної лінії новим об'єктом, оновлюються локальні лічильники, а таймстемп події додається до TrafficStateTracker для перерахунку поточного стану завантаженості дороги.

Критично важливим архітектурним рішенням на цьому етапі є спосіб передачі даних для збереження. Замість прямого виклику функцій бази даних, конвеєр використовує потокобезпечну чергу повідомлень. Метод global_state.log_queue.put() миттєво додає кортеж із типом об'єкта та станом трафіку до оперативної пам'яті, після чого конвеєр негайно переходить до наступного етапу – оновлення стану та візуалізації (рис. 3.4).

```
global_state.update_stats(
    self.counter.total_objects, intensity, state,
    self.total_persons, self.total_transport
)

if global_state.has_viewers():
    if not results or not len(results):
        ret, buffer = cv2.imencode('.jpg', frame)
    else:
        frame_rendered = Visualizer.render(
            frame, events, self.counter.total_objects,
intensity,
            state, state_color, self.counter.line_color
        )
        ret, buffer = cv2.imencode('.jpg', frame_rendered)

    if ret:
        global_state.update_frame(buffer.tobytes())
else:
    global_state.clear_frame()
```

Рисунок 3.4 – Лістинг оновлення глобального стану та умовного рендерингу

3.3 Розробка вебсервера, REST API та WebSockets

Для забезпечення взаємодії між ядром комп'ютерного зору, базою даних та клієнтськими пристроями розроблено серверну частину на базі асинхронного фреймворку FastAPI. Архітектура вебсервера поділяється на два основні комунікаційні рівні: традиційний REST API для отримання історичних даних та систему реального часу на базі протоколу WebSocket для миттєвої передачі телеметрії.

Рівень REST API реалізовано через ендпоінт /api/history, який приймає параметри часового інтервалу та типу об'єктів. Обробка запиту делегується

класу HistoryService, який виконує агрегацію даних безпосередньо на рівні SQL-запитів до бази даних SQLite. Це дозволяє уникнути вивантаження великих масивів даних в оперативну пам'ять сервера, що є критично важливим для забезпечення стабільності системи при тривалій експлуатації [20, 25].

Особливої уваги заслуговує реалізація системи реального часу. Оскільки інформація про інтенсивність трафіку оновлюється щосекунди, використання класичних HTTP-запитів (Polling) з боку клієнта призвело б до надмірного навантаження на мережу та сервер. Тому для передачі поточних метрик використано двостороннє з'єднання WebSocket.

Для ефективного управління пулом активних клієнтів спроєктовано клас ConnectionManager (рис. 3.5), який реалізує архітектурний патерн «Видавець-Підписник» (Broadcast Pattern).

Як видно з лістингу на рис. 3.5, замість того, щоб кожен клієнтський потік незалежно звертався до глобального стану системи (що створювало б проблему конкурентного доступу та блокувань), сервер використовує єдину асинхронну фонову задачу stats_broadcaster. Ця задача один раз на секунду зчитує актуальні метрики з оперативної пам'яті та розсилає їх усім підключеним клієнтам через метод broadcast. Такий підхід забезпечує складність операції читання стану O(1) незалежно від кількості активних користувачів на дашборді.

Крім того, у броадкастері реалізовано концепцію Backend-Driven UI (інтерфейс, керований сервером). Метод HistoryService.get_current_buckets(now) генерує готові часові мітки для всіх можливих фільтрів фронтенду (хвилина, година, день тощо) на основі єдиного серверного часу. Передача готових міток у JSON-пакеті повністю звільняє клієнтський застосунок від необхідності виконувати математичні операції з датами та унеможливлює розсинхронізацію графіків у випадку, якщо клієнтський пристрій знаходиться в іншому часовому поясі [17].

```
class ConnectionManager:
    """Менеджер для керування активними WebSocket
з'єднаннями."""

    def __init__(self) -> None:
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket) -> None:
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket) -> None:
        self.active_connections.remove(websocket)

    async def broadcast(self, data: dict) -> None:
        for connection in self.active_connections:
            await connection.send_json(data)

manager = ConnectionManager()

async def stats_broadcaster() -> None:
    while global_state.is_running:
        if manager.active_connections:
            stats = global_state.get_stats()
            now = datetime.now(ZoneInfo(AppConfig.TIMEZONE))
            await manager.broadcast({
                "time": now.strftime("%H:%M:%S"),
                "buckets":
HistoryService.get_current_buckets(now),
                "intensity": stats["intensity"],
                "total_objects": stats["total_objects"],
                "traffic_state": stats["traffic_state"],
                "total_persons": stats.get("total_persons", 0),
                "total_transport": stats.get("total_transport",
0)
            })
        await asyncio.sleep(1)
```

Рисунок 3.5 – Лістинг реалізації менеджера WebSocket-з'єднань та броадкастера

Додатково вебсервер реалізує ендпоінт /video_feed, який забезпечує потокову передачу оброблених кадрів у форматі MJPEG (Motion JPEG) за допомогою класу StreamingResponse. Цей ендпоінт тісно інтегрований з механізмом умовного рендерингу: при підключенні клієнта викликається метод increment_viewers(), а при розриві з'єднання – decrement_viewers(), що дозволяє конвеєру комп'ютерного зору динамічно вмикати або вимикати

ресурсоємні операції кодування зображень.

3.4 Створення графічного інтерфейсу користувача

Для візуалізації результатів роботи системи та надання користувачеві зручного інструменту моніторингу розроблено інтерактивний вебдашборд. Інтерфейс побудовано за принципом адаптивної верстки (Responsive Design) з використанням CSS-фреймворку Bootstrap 5, що гарантує коректне відображення як на десктопних моніторах, так і на мобільних пристроях. Для зниження навантаження на зір оператора та підвищення контрастності відеопотоку застосовано темну тему оформлення [26].

Головний екран дашборду логічно поділено на дві основні зони: блок відеотрансляції та аналітичний блок (рис. 3.6).

Відеотрансляція реалізована максимально легковажним способом – через стандартний HTML-тег <img>, джерелом якого вказано ендпоінт /video_feed. Сучасні браузери нативно підтримують обробку потоку multipart/x-mixed-replace, що дозволяє безперервно оновлювати зображення (MJPEG) без використання сторонніх відеоплеєрів чи складних JavaScript-бібліотек. На відео в реальному часі накладається графічний інтерфейс (HUD): полігон зони інтересу, контрольна лінія, обмежувальні рамки об'єктів (Bounding Boxes) та їхні унікальні ідентифікатори.

Аналітичний блок містить панель загальної статистики, інтерактивний графік та панель фільтрів. Графік побудовано за допомогою бібліотеки Chart.js, яка забезпечує високу продуктивність рендерингу на елементі HTML5 Canvas та підтримує плавні анімації. Користувач має змогу фільтрувати дані за типом об'єктів (всі, транспорт, люди) та за часовими інтервалами (від 1 хвилини до року). При зміні типу об'єкта JavaScript-код динамічно змінює кольорову палітру графіка для кращого візуального сприйняття (наприклад, зелений для загального потоку, синій для транспорту, оранжевий для людей).

Рисунок 3.6 – Головний екран вебдашборду системи моніторингу

Ключовою особливістю клієнтського застосунку є його реалізація за патерном «Тонкий клієнт» (Thin Client). Увесь JavaScript-код (файл dashboard.js) відповідає виключно за оновлення DOM-дерева та рендеринг графіка. При отриманні JSON-пакета через WebSocket, скрипт виконує наступні дії:

1. Оновлює бейдж загальної кількості об'єктів.
2. Змінює текст та колір індикатора стану трафіку (зелений для стану FREE, жовтий для NORMAL, червоний для HEAVY).
3. Зчитує готову часову мітку (Bucket), згенеровану сервером для поточного режиму відображення.

4. Обчислює дельту (приріст) об'єктів відносно попереднього стану та додає її до відповідної точки на графіку.

Для оптимізації використання ресурсів клієнтського пристрою впроваджено механізм умовного рендерингу графіка. Метод chart.update() викликається лише тоді, коли фактично відбулася зміна даних (зафіксовано новий об'єкт) або почався новий часовий інтервал. Це запобігає зайвому перемальовуванню Canvas-елемента щосекунди і суттєво знижує навантаження на графічний процесор (GPU) пристрою користувача. Крім того, для запобігання стану гонки (Race Condition) під час перемикання фільтрів, реалізовано прапорець isChartInitialized, який блокує обробку WebSocket-повідомлень до моменту повного завантаження історичних даних через REST API.

3.5 Тестування системи, аналіз продуктивності та порівняння результатів

Для перевірки ефективності запропонованих архітектурних рішень було проведено комплексне тестування розробленої розподіленої системи в реальних умовах експлуатації. Головною метою тестування було підтвердження гіпотези про те, що винесення обчислювального навантаження за межі мікрокомп'ютера (Edge-вузла) дозволяє вирішити проблеми низької частоти кадрів, перегріву та недостатньої точності трекінгу, які були виявлені на попередніх етапах дослідження.

Тестовий стенд складався з двох вузлів: мікрокомп'ютера Raspberry Pi 3 Model B (вузол трансляції), розміщеного поблизу проїжджої частини, та сервера обробки на базі процесора Apple M1, підключених до єдиної локальної бездротової мережі (Wi-Fi). Зовнішній вигляд зібраного експериментального стенду (вузла трансляції), змонтованого для проведення натурних випробувань, наведено на рис. 3.7.

Рисунок 3.7 – Зовнішній вигляд експериментального стенду на базі Raspberry Pi 3

Першим етапом було оцінено апаратне навантаження на вузол трансляції. Завдяки використанню апаратного кодувальника H.264 та відмові від локального інференсу нейромережі, загальне завантаження центрального процесора Raspberry Pi (CPU Load) знизилося зі 100% до стабільних 20-25% (показник Load Average становив близько 1.00). Температурний режим процесора під час безперервної трансляції зафіксувався на позначці 58-65 °C

максимально. Це повністю усунуло явище апаратного тротлінгу, яке раніше виникало при досягненні 80 °C, та забезпечило стабільну передачу відеопотоку з частотою 20 FPS без пропуску кадрів.

Другим етапом було проаналізовано продуктивність сервера обробки. Завдяки конвертації моделі YOLOv8m (Medium) у формат CoreML, операційна система macOS успішно делегувала тензорні обчислення на співпроцесор Neural Engine. Це дозволило серверу обробляти вхідний потік (20 FPS) у реальному часі без жодних затримок, маючи при цьому значний запас обчислювальної потужності (потенційна пропускна здатність перевищує 30 FPS). Температура процесора Apple M1 Pro під час роботи конвеєра комп'ютерного зору залишалася в межах 50-60 °C, що дозволило системі охолодження працювати в пасивному (безшумному) режимі. Процес безперервного логування розпізнаних об'єктів на сервері та одночасний моніторинг температурного режиму мікрокомп'ютера під час натурних випробувань наведено на рис. 3.8.

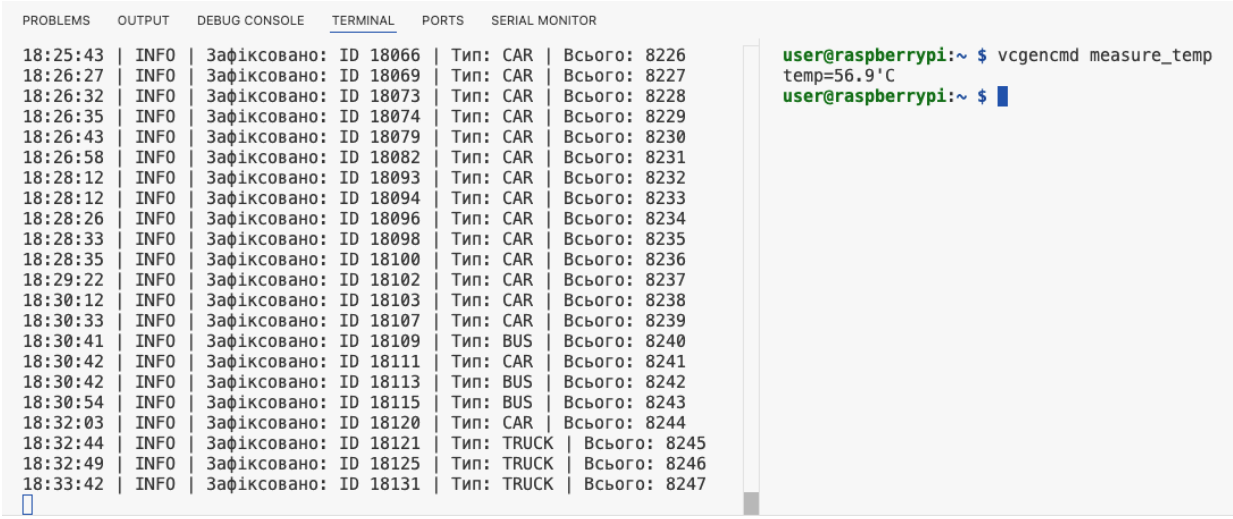

Рисунок 3.8 – Логування подій перетину та моніторинг температури вузла трансляції

Третім етапом стала оцінка точності детекції та трекінгу. Перехід від легковажної моделі MobileNet SSD до сучасної архітектури YOLOv8m (Medium) кардинально покращив якість розпізнавання. Вибір саме версії

Medium зумовлений пошуком оптимального балансу («золотої середини») між високою точністю та енергоефективністю: на відміну від важкої версії Large, вона забезпечує аналогічну якість трекінгу, але суттєво знижує теплове навантаження на сервер обробки. Система продемонструвала здатність впевнено детектувати транспортні засоби та пішоходів на значно більшій відстані та в умовах часткового перекриття об'єктів. Крім того, стабільна частота надходження кадрів (20 FPS) забезпечила плавне переміщення центроїдів об'єктів між ітераціями. Це повністю вирішило проблему втрати ідентифікаторів (ID) алгоритмом Centroid Tracker, яка раніше призводила до хибного дублювання підрахунку через різкі просторові стрибки об'єктів при 4-5 FPS. Загальна точність підрахунку (Accuracy) зросла з 89% (у монолітній архітектурі) до понад 97% у розподіленій.

Для наочної демонстрації ефективності проведених змін, порівняльні характеристики монолітної та розподіленої архітектур зведено у табл. 3.1.

Таблиця 3.1 – Порівняння характеристик монолітної та розподіленої архітектур

Показник	Монолітна архітектура	Розподілена архітектура
Модель нейромережі	MobileNet SSD (Caffe)	YOLOv8m (CoreML)
Частота обробки (FPS)	4 - 5 FPS	20 FPS (стабільно)
Завантаження CPU (RPi)	100%	20 - 25%
Максимальна температура (RPi)	80 - 85 °C (Тротлінг)	58 - 65 °C (Норма)
Точність підрахунку	~ 89% (в ідеальних умовах на прикладі тестового відеопотоку)	> 97% (в реальному часі в реальних умовах експлуатації)

РОЗДІЛ 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Ергономіка робочого місця оператора ПК

Організація робочого місця розробника програмного забезпечення виконана з урахуванням вимог НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями», ДСанПіН 3.3.2.007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами», а також стандартів ДСТУ ISO 9241-5:2004 та ДСТУ 8604:2015 щодо ергономіки робочого місця [9, 10, 13, 14].

Просторове компонування робочого місця спроєктовано таким чином, щоб забезпечити зручну позу розробника, мінімізувати статичне напруження м'язів шийно-плечового поясу та спини. Робочий стіл має висоту 720 мм, що відповідає антропометричним даним користувача. Простір під столом має висоту 600 мм, ширину 500 мм та глибину 650 мм, що забезпечує вільне розміщення ніг та можливість зміни пози під час робочої зміни. Поверхня столу має матове покриття з коефіцієнтом відбиття 0,3-0,5, що запобігає появі відблисків від джерел освітлення.

Для забезпечення фізіологічно правильної робочої пози використовується робоче крісло, яке відповідає вимогам ДСТУ 7951:2015 [15]. Крісло оснащене підйомним механізмом для регулювання висоти сидіння (в межах 400-500 мм) та кута нахилу спинки. Спинка крісла має анатомічну форму з підтримкою поперекового відділу хребта, що знімає навантаження з міжхребцевих дисків під час тривалого написання коду. Клавіатура розташована на поверхні столу на відстані 100-300 мм від краю, що дозволяє зручно спирати передпліччя на стіл, знімаючи статичне навантаження з плечових суглобів.

Особливу увагу приділено візуальній ергономіці, оскільки робота з інтегрованими середовищами розробки (IDE) та терміналами вимагає високої концентрації зору. Екран ноутбука (Apple MacBook) розташований на оптимальній відстані 600-700 мм від очей користувача. Кут зору на центр екрана становить 15-20 градусів нижче горизонтальної лінії погляду, що запобігає перенапруженню очних м'язів та розвитку синдрому «сухого ока». Для уникнення відблисків на екрані, робочий стіл розміщено збоку від вікна таким чином, щоб природне світло падало зліва.

Освітлення робочого місця організовано відповідно до ДБН В.2.5-28:2018 «Природне і штучне освітлення» [11]. Використовується комбінована система освітлення: природне світло доповнюється загальним штучним освітленням (світлодіодні лампи з колірною температурою 4000К, що максимально наближена до природного спектра). Рівень освітленості на поверхні робочого столу становить 400 лк. Для усунення прямої та відбитої блискості на вікнах встановлено жалюзі, а екран пристрою має антиблікове покриття.

Мікроклімат у приміщенні підтримується згідно з ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» [12]. Оскільки робота програміста належить до категорії легкої фізичної праці (категорія Іа), у приміщенні забезпечується температура повітря на рівні 22-24 °С, відносна вологість 40-60% та швидкість руху повітря не більше 0,1 м/с. Для підтримання цих параметрів та забезпечення припливу свіжого повітря передбачено регулярне провітрювання приміщення та використання системи кондиціонування.

Для запобігання розвитку перевтоми та монотонії під час розробки та тестування алгоритмів комп'ютерного зору, впроваджено раціональний режим праці та відпочинку. Відповідно до санітарних норм, після кожної години безперервної роботи за екраном передбачено регламентовану перерву тривалістю 10 хвилин. Під час перерв виконується комплекс гімнастичних вправ для зняття зорової втоми та розминки м'язів спини і рук.

4.2 Вплив електромагнітних полів на людину та заходи захисту

Специфіка розробленого апаратно-програмного комплексу передбачає використання електронного обладнання, яке є джерелом електромагнітних полів (ЕМП) різних частотних діапазонів. До основних джерел ЕМП на робочому місці належать: персональний комп'ютер (ноутбук), мікрокомп'ютер Raspberry Pi 3, Wi-Fi роутер, який забезпечує передачу відеопотоку по локальній мережі, та периферійні пристрої.

Електромагнітні поля характеризуються напруженістю електричного поля (В/м) та магнітною індукцією (мкТл). Вплив ЕМП на організм людини має неіонізуючий характер і поділяється на тепловий та біологічний (інформаційний) ефекти. Тепловий ефект виникає при поглинанні енергії ЕМП тканинами організму, що призводить до їх нагрівання. Проте при роботі з комп'ютерною технікою інтенсивність випромінювання є недостатньою для вираженого теплового ефекту. На перший план виходить біологічний вплив слабких електромагнітних полів при їх тривалій експозиції.

Найбільш чутливими до впливу електромагнітного випромінювання є центральна нервова, серцево-судинна, імунна та ендокринна системи. Біологічний ефект від тривалого впливу ЕМП радіочастотного діапазону (зокрема, від модулів Wi-Fi, що працюють на частотах 2.4 ГГц та 5 ГГц) проявляється у вигляді астенічного синдрому. У користувача можуть спостерігатися швидка втомлюваність, головний біль, порушення сну, зниження концентрації уваги та погіршення пам'яті, що є критично неприпустимим при виконанні складних інженерних завдань з написання програмного коду. Крім того, низькочастотні поля (50 Гц), що генеруються блоками живлення та елементами материнської плати, можуть викликати зміни у серцевому ритмі, коливання артеріального тиску та пригнічення імунної реактивності організму.

Для мінімізації негативного впливу електромагнітних полів та забезпечення безпечних умов праці, у межах даної роботи реалізовано комплекс організаційних та інженерно-технічних заходів захисту.

Основним методом захисту від ЕМП, застосованим при організації робочого простору, є захист відстанню. Інтенсивність електромагнітного випромінювання зменшується обернено пропорційно квадрату відстані від джерела. З огляду на це, Wi-Fi роутер, який забезпечує бездротову передачу відеопотоку від Raspberry Pi до сервера обробки, розміщено на відстані понад 2 метри від постійного робочого місця розробника. Сам мікрокомп'ютер Raspberry Pi під час етапу тестування також розташовується на безпечній відстані (не менше 1 метра).

Для захисту від низькочастотних полів, що генеруються блоками живлення, застосовано метод захисту часом. Тривалість безперервного перебування в зоні дії ЕМП суворо регламентована режимом праці та відпочинку. Усі периферійні пристрої, які не використовуються в поточний момент часу, відключаються від мережі живлення.

Важливим інженерним заходом є використання обладнання, що відповідає міжнародним стандартам електромагнітної сумісності (EMC) та безпеки (наприклад, стандарти TCO та MPR II для дисплеїв). Корпус ноутбука (алюмінієвий сплав) виконує роль природного електромагнітного екрана, що суттєво знижує рівень випромінювання від внутрішніх компонентів (процесора, оперативної пам'яті).

Крім того, для усунення статичної електрики, яка може накопичуватися на екранах та корпусах пристроїв, забезпечено належне захисне заземлення всіх розеток у приміщенні. Для підключення обладнання використовуються виключно сертифіковані мережеві фільтри із заземлюючим контактом, що не лише захищає апаратуру від перепадів напруги, але й відводить статичний заряд, знижуючи напруженість електростатичного поля на робочому місці. Регулярне вологе прибирання приміщення та підтримання відносної вологості повітря на рівні 50% також сприяє швидкому стіканню статичних зарядів.

Реалізований комплекс заходів гарантує, що рівні електромагнітного випромінювання на робочому місці не перевищують гранично допустимих рівнів, встановлених санітарними нормами, що забезпечує збереження здоров'я та високу працездатність під час розробки та тестування апаратно-програмного комплексу.

ВИСНОВКИ

У кваліфікаційній роботі успішно реалізовано проєкт зі створення ефективної, точної та масштабованої системи моніторингу дорожнього руху. В результаті виконання роботи було досягнуто поставленої мети та отримано наступні основні результати:

1. На основі аналізу предметної області та результатів попередніх досліджень доведено неефективність використання монолітної Edge-архітектури на базі мікрокомп'ютерів для задач відеоаналітики в реальному часі. Обґрунтовано та спроєктовано розподілену клієнт-серверну архітектуру, яка розділяє процеси захоплення відео та його інтелектуальної обробки.

2. Розроблено та налаштовано апаратний вузол відеофіксації (Thin Client) на базі мікрокомп'ютера Raspberry Pi 3 Model B. Завдяки використанню апаратного кодувальника H.264 та передачі відеопотоку через TCP-сокет, вдалося знизити завантаження центрального процесора до 20-25% та стабілізувати робочу температуру на рівні 58-61 °C. Це повністю усунуло проблему апаратного тротлінгу та забезпечило стабільну трансляцію з частотою 20 FPS.

3. Спроєктовано та реалізовано високопродуктивний ізольований конвеєр комп'ютерного зору на базі сервера з архітектурою Apple Silicon (Thick Server). Використання сучасної нейромережі YOLOv8m, конвертованої у формат CoreML, дозволило делегувати тензорні обчислення на співпроцесор Neural Engine. У поєднанні з розробленим алгоритмом центроїдного трекінгу (Centroid Tracking) це забезпечило безперебійну обробку відеопотоку та підвищило загальну точність детекції і підрахунку об'єктів з 89% до понад 97%.

4. Розроблено серверну частину системи на базі асинхронного вебфреймворку FastAPI та реляційної бази даних SQLite (з використанням ORM SQLAlchemy та системи міграцій Alembic). Для усунення проблеми блокування вводу-виводу (I/O Blocking) впроваджено механізм фонового

збереження логів через потокобезпечну чергу повідомлень із застосуванням пакетного вставлення (Batch Inserts).

5. Створено систему передачі телеметрії в реальному часі на базі протоколу WebSocket. Розроблено менеджер з'єднань, який реалізує патерн Broadcast, забезпечуючи розсилку статистики всім підключеним клієнтам зі складністю O(1).

6. Розроблено інтерактивний графічний вебдашборд за архітектурним патерном Backend-Driven UI. Генерація часових міток на стороні сервера повністю усунула ризик розсинхронізації даних через різницю часових поясів на клієнтських пристроях. Впровадження механізму умовного рендерингу (Smart Rendering) дозволило суттєво оптимізувати використання обчислювальних ресурсів сервера та клієнта.

Загалом, розроблений апаратно-програмний комплекс є економічно ефективним, надійним та готовим до інтеграції в інфраструктуру «Розумного міста» для автоматизованого оцінювання завантаженості вулиць транспортними засобами та пішоходами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жаровський Р.О., Луцик Н.С., Осухівська Г.М., Паламар А.М., Тиш Є.В. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальністю 123 «Комп'ютерна інженерія» усіх форм навчання. Тернопіль: ТНТУ, 2024. 39 с.

2. Луцків А., Лупенко С., Пасічник В. Паралельні та розподільнені обчислення. Навчальний посібник. Львів: Видавництво «Магнолія 2006», 2024. 566 с.

3. Паламар М.І., Стрембіцький М.О., Паламар А.М. Проектування комп'ютеризованих вимірювальних систем і комплексів. Навчальний посібник. Тернопіль: ТНТУ. 2019. 150 с.

4. Кондратюк Р., Тиш Є. Машинний зір: сутність технології, принципи роботи та сфери застосування. XIII науково-технічна конференція «Інформаційні моделі, системи та технології» Тернопіль: ТНТУ. 2025. С. 120.

5. Кондратюк Р., Тиш Є. Opencv як основа сучасних систем комп'ютерного зору. XIII науково-технічна конференція «Інформаційні моделі, системи та технології» Тернопіль : ТНТУ. 2025. С. 121.

6. Антонюк В.І., Луцик Н.С., Паламар А.М. Комп'ютеризована IoT-система для аналізу споживання електроенергії у житлових приміщеннях. Актуальні задачі сучасних технологій : збірник тез доповідей XIV міжнародної науково-практичної конференції молодих учених та студентів (Тернопіль, 11-12 грудня 2025 року), Тернопіль: ФОП Паляниця В. А., 2025. С. 225.

7. Дерягін В., Дрогобицький М., Луцик Н. Методи моніторингу та оптимізації взаємодії мікросервісів в istio service mesh. Матеріали XIII науково-технічної конференції Тернопільського національного технічного університету імені Івана Пулюя «Інформаційні моделі, системи та технології» (17-18 грудня 2025 року). Тернопіль: ТНТУ. 2025. с. 111.

8. Романов Д.В., Осухівська Г.М., Паламар А.М. Система управління зовнішнім освітленням на основі Інтернету речей. Актуальні задачі сучасних технологій : збірник тез доповідей X міжнародної науково-практичної конференції молодих учених та студентів (Тернопіль, 24-25 листопада 2021 року), Тернопіль: ТНТУ, 2021. С. 120.

9. НПАОП 0.00-7.15-18. Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями. Затверджено наказом Мінсоцполітики України від 14.02.2018 № 207.

10. ДСанПіН 3.3.2.007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин. Київ: МОЗ України, 1998. 18 с.

11. ДБН В.2.5-28:2018. Природне і штучне освітлення. Київ: Мінрегіон України, 2018. 133 с.

12. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень. Київ: МОЗ України, 1999. 11 с.

13. ДСТУ ISO 9241-5:2004. Ергономічні вимоги до роботи з відеотерміналами в офісі. Частина 5. Вимоги до компонування робочого місця та до робочої пози. Київ: Держспоживстандарт України, 2005. 26 с.

14. ДСТУ 8604:2015. Дизайн і ергономіка. Робоче місце для виконання робіт у положенні сидячи. Загальні ергономічні вимоги. Київ: ДП «УкрНДНЦ», 2016. 15 с.

15. ДСТУ 7951:2015. Дизайн і ергономіка. Крісло оператора. Загальні ергономічні вимоги. Київ: ДП «УкрНДНЦ», 2016. 14 с.

16. Ultralytics YOLOv8 Documentation. URL: https://docs.ultralytics.com/ (дата звернення: 22.04.2026).

17. FastAPI Framework Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 22.04.2026).

18. OpenCV: Open Source Computer Vision Library. URL: https://opencv.org/ (дата звернення: 26.04.2026).

19. Raspberry Pi Documentation. URL: https://www.raspberrypi.com/documentation/ (дата звернення: 20.04.2026).

20. SQLAlchemy 2.0 Documentation. URL: https://docs.sqlalchemy.org/ (дата звернення: 25.04.2026).

21. Pydantic Settings Documentation. URL: https://docs.pydantic.dev/latest/concepts/pydantic_settings/ (дата звернення: 01.05.2026).

22. Chart.js Documentation. URL: https://www.chartjs.org/docs/latest/ (дата звернення: 03.05.2026).

23. Core ML Overview – Apple Developer. URL: https://developer.apple.com/machine-learning/core-ml/ (дата звернення: 21.04.2026).

24. Python 3.12 Documentation. URL: https://docs.python.org/3/ (дата звернення: 27.04.2026).

25. SQLite Official Documentation. URL: https://www.sqlite.org/docs.html (дата звернення: 25.04.2026).

26. Bootstrap 5.3 Documentation. URL: https://getbootstrap.com/docs/5.3/getting-started/introduction/ (дата звернення: 02.05.2026).

27. Alembic Documentation. URL: https://alembic.sqlalchemy.org/en/latest/ (дата звернення: 01.05.2026).

28. Uvicorn: The lightning-fast ASGI server. URL: https://www.uvicorn.org/ (дата звернення: 04.05.2026).

29. Paramiko Documentation (SSHv2 protocol library for Python). URL: https://www.paramiko.org/ (дата звернення: 28.04.2026).

Додаток А

Технічне завдання

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії

Кафедра комп'ютерних систем та мереж

"Затверджую"

Завідувач кафедри КС

_____ Осухівська Г.М.

" 2 " лютого 2026 р.

Комп'ютерна система оцінювання завантаженості
вулиць транспортними засобами та пішоходами.

ТЕХНІЧНЕ ЗАВДАННЯ

на _*8*_ листках

Вид робіт: Кваліфікаційна робота

На здобуття освітнього ступеня «Бакалавр»

Спеціальність 123 «Комп'ютерна інженерія»

«УЗГОДЖЕНО»	«ВИКОНАВЕЦЬ»
Керівник кваліфікаційної роботи	Студент групи СІ-42
_____ к.т.н., доц. Тиш Є.В.	_____ Котовський Д.М.
" 2 " лютого 2026 р.	" 2 " лютого 2026 р.

Тернопіль 2026

1 Загальні відомості

1.1 Повна назва та її умовне позначення

Повна назва теми кваліфікаційної роботи: «Комп’ютерна система оцінювання завантаженості вулиць транспортними засобами та пішоходами».

Умовне позначення кваліфікаційної роботи: КС КРБ 123.180.00.00

1.2 Виконавець

Студент групи СІ-42, факультету комп’ютерно-інформаційних систем і програмної інженерії, кафедри комп’ютерних систем та мереж, Тернопільського національного технічного університету імені Івана Пулюя, Котовський Д.М.

1.3 Підстава для виконання роботи

Підставою для виконання кваліфікаційної роботи є наказ по університету (№4/9-188 від 24.04.2026 р.)

1.4 Планові терміни початку та завершення роботи

Плановий термін початку виконання кваліфікаційної роботи – 26.01.2026 р.

Плановий термін завершення виконання кваліфікаційної роботи – 21.06.2026 р.

1.5 Порядок оформлення та пред’явлення результатів роботи

Порядок оформлення пояснювальної записки та графічного матеріалу здійснюється у відповідності до чинних норм та правил ISO, ЕСКД, ЕСПД та ДСТУ.

Пред’явлення проміжних результатів роботи з виконання кваліфікаційної роботи здійснюється у відповідності до графіку, затвердженого керівником роботи. Попередній захист кваліфікаційної роботи відбувається при готовності роботи – наявності пояснювальної записки та графічного матеріалу.

Пред’явлення результатів кваліфікаційної роботи відбувається шляхом захисту на відповідному засіданні ЕК, ілюстрацією основних досягнень за допомогою графічного матеріалу.

2 Призначення і цілі створення системи

2.1 Призначення системи

Система, що розробляється, призначена для:

– безперервного відеомоніторингу дорожнього руху в реальному часі;

– автоматичної детекції, класифікації та трекінгу транспортних засобів і пішоходів;

– підрахунку кількості об'єктів, що перетинають контрольну лінію;

– визначення поточного стану завантаженості вулиці (вільний рух, нормальний, затор);

– надання користувачам доступу до відеотрансляції та аналітики через вебдашборд.

2.2 Мета створення системи

Метою кваліфікаційної роботи є розробка розподіленої апаратно-програмної системи для високоточного оцінювання завантаженості вулиць з використанням мікрокомп'ютера Raspberry Pi, нейромережі YOLOv8 та сучасних вебтехнологій.

2.3 Характеристика об'єкту

Розроблювана система призначена для стаціонарного використання поблизу проїжджих частин, перехресть та пішохідних зон в умовах міської інфраструктури.

3 Вимоги до системи

3.1 Вимоги до системи в цілому

3.1.1 Вимоги до структури та функціонування системи

Система має складатися з трьох основних блоків:

– апаратного вузла відеофіксації (мікрокомп'ютер Raspberry Pi з модулем камери);

– сервера обробки та аналітики (високопродуктивний комп'ютер з підтримкою апаратного прискорення нейромереж);

– клієнтського вебзастосунку (інтерактивний дашборд).

3.1.2 Вимоги до способів та засобів зв’язку між компонентами системи

Основна вимога, яка ставиться до способів та засобів інформаційного обміну – це їх узгодженість.

Елементи системи мусять взаємодіяти за допомогою наступних каналів:

– вузол відеофіксації передає стиснений відеопотік (H.264) на сервер обробки через локальну мережу з використанням протоколу TCP;

– сервер обробки передає статистичні дані на клієнтський вебзастосунок через двостороннє з'єднання WebSocket (патерн Broadcast);

– сервер обробки транслює оброблений відеопотік на клієнтський застосунок у форматі MJPEG через протокол HTTP.

3.1.3 Вимоги до режимів функціонування системи

Для системи визначено два режими роботи:

– Нормальний режим роботи (безперервне захоплення відео, детекція об'єктів, запис логів у базу даних та трансляція на дашборд);

– Аварійний режим роботи (втрата з'єднання з камерою). У цьому режимі сервер повинен автоматично ініціювати процес перепідключення по SSH до відновлення стабільного відеопотоку.

3.1.4 Вимоги по діагностуванню системи

Для діагностування системи використовуються:

– інструменти системного логування на сервері (вивід інформації про ініціалізацію нейромережі, підключення камери та помилки виконання);

– візуальна діагностика через вебдашборд (відображення поточного стану підключення та системних повідомлень).

3.1.5 Перспективи розвитку, проектування системи

Цю систему можна вдосконалити через підключення декількох вузлів відеофіксації до одного центрального сервера, а також шляхом використання більш бюджетних апаратних платформ замість Raspberry Pi 3 для трансляції відеопотоку.

3.2 Показники призначення

Система має передбачати можливість масштабування. Архітектура бази даних та вебсервера повинна дозволяти обробку даних з кількох джерел одночасно.

3.2.1 Вимоги до надійності

Система повинна забезпечувати працездатність та відновлення своїх функцій при виникненні наступних ситуацій:

– при тимчасовій втраті мережевого з'єднання між вузлом фіксації та сервером (автоматичне перепідключення);

– при збереженні даних у БД (використання ізольованого фонового потоку та черги повідомлень для уникнення блокування конвеєра комп'ютерного зору).

3.3 Вимоги до безпеки

Вузол відеофіксації (Raspberry Pi) живиться від безпечної напруги 5В. Серверне обладнання повинно підключатися до електромережі з використанням захисного заземлення та мережевих фільтрів.

3.3.1 Вимоги до експлуатації, технічного обслуговування, ремонту і зберігання компонентів системи

Мікроклімат в приміщеннях повинен відповідати нормам виробничого мікроклімату по ДСН 3.3.6.042-99:

– температуру повітря в межах від +10°С до +35°С;

– відносну вологість повітря при 25°С в межах від 30% до 80%;

– атмосферний тиск 760±25 мм рт. ст.

Періодичне технічне обслуговування має проводитися відповідно до вимог технічної документації, але не рідше ніж один раз на рік і повинно включати перевірку і тестування всіх використовуваних апаратних засобів (мікрокомп'ютера, модуля камери та блоку живлення).

На підставі результатів тестування технічних засобів повинні проводитися аналіз причин виникнення виявлених дефектів і прийматися заходи по їх ліквідації.

Завдяки використанню апаратного кодувальника відео, завантаження процесора Raspberry Pi не повинно перевищувати 20-25%, що гарантує підтримання робочої температури в межах 58-65 °С та дозволяє експлуатувати пристрій без активного охолодження.

3.4 Вимоги до захисту інформації від несанкціонованого доступу

Управління вузлом відеофіксації та передача команд ініціалізації повинні здійснюватися виключно через захищений протокол SSH з використанням автентифікації.

3.4.1 Вимоги по збереженню інформації при аваріях

Історичні дані про трафік повинні зберігатися у реляційній базі даних SQLite на постійному носії сервера.

3.4.2 Вимоги по стандартизації і уніфікації

Система мусить відповідати вимогам ергономіки. Вебдашборд повинен мати адаптивний дизайн та підтримувати темну тему оформлення для зниження навантаження на зір оператора.

3.4.3 Вимоги до функцій (завдань), що виконуються системою:

– апаратне захоплення та трансляція відеопотоку з частотою 20 FPS;
– детекція та класифікація об'єктів (YOLOv8m);
– трекінг об'єктів (Centroid Tracker);
– підрахунок перетинів контрольної лінії;
– фоновий запис подій у базу даних;
– умовний рендеринг відео (Smart Rendering) лише за наявності активних глядачів на дашборді.

4 Вимоги до документації

Документація повинна відповідати вимогам ЄСКД та ДСТУ

Комплект документації повинен складатись з:

– пояснювальної записки;
– графічного матеріалу:

а) Схема електрична структурна.

б) Блок-схема алгоритму функціонування програми.

в) Схема електрична принципова.

г) Схема електрична з’єднань.

*Примітка: У комплект документації можуть вноситися зміни та доповнення в процесі розробки.

5 Стадії та етапи проектування

Таблиця 1 – Стадії та етапи виконання кваліфікаційної роботи бакалавра

№ етапу	Назва етапу виконання кваліфікаційної роботи бакалавра	Термін виконання
1	*Розробка технічного завдання*	*26.01 – 02.02*
2	*Робота над першим розділом «Аналіз предметної області та обґрунтування переходу до розподіленої архітектури»*	*03.02 – 15.02*
3	*Робота над другим розділом «Проєктування розподіленої системи оцінювання завантаженості вулиць»*	*20.04 – 28.04*
4	*Робота над третім розділом «Практична реалізація системи та оцінка її ефективності»*	*29.04 – 08.05*
5	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*10.05 – 25.05*
6	*Оформлення пояснювальної записки і графічного матеріалу*	*26.05 – 7.06*
7	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*
8	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*
9	*Захист кваліфікаційної роботи бакалавра*	*24.06*

6 Додаткові умови виконання кваліфікаційної роботи

Під час виконання кваліфікаційної роботи у дане технічне завдання можуть вноситися зміни та доповнення.

Додаток Б

Перелік елементів

Позн.	Найменування	К-сть	Примітка
	Мікрокомп'ютери та модулі		
A1	Мікрокомп'ютер Raspberry Pi 3 Model B	1	
B1	Модуль камери Raspberry Pi Camera Module	1	
	Роз'єми та інтерфейси (на платі A1)		
X1	Роз'єм USB 2.0 Type-A	1	Не задіяно
X2	Роз'єм USB 2.0 Type-A	1	Не задіяно
X3	Роз'єм LAN (RJ-45 Ethernet)	1	Не задіяно
X4	Конектор GPIO 40-pin	1	Не задіяно
X5	Роз'єм Audio/Video (3.5mm Jack)	1	Не задіяно
X6	Роз'єм інтерфейсу камери (CSI-2 15-pin)	1	
X7	Роз'єм відеовиходу HDMI (Type A)	1	Не задіяно
X8	Слот для карт пам'яті MicroSD	1	ОС Raspberry Pi
X9	Роз'єм живлення (Micro-USB 5V)	1	
X10	Роз'єм інтерфейсу дисплея (DSI 15-pin)	1	Не задіяно
	Кабелі та шлейфи		
W1	Шлейф гнучкий FFC 15-pin (крок 1 мм)	1	

					КС КРБ 123.180.00.00 ПЕ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Котовський Д.М.			Комп'ютерна система оцінювання завантаженості вулиць транспортними засобами та пішоходами. Перелік елементів	Літ.	Арк. 1 / Акрушів 1
Перевір.		Тиш Є.В.					
Реценз.		Дмитроца Л.П.				ТНТУ, каф. КС, гр. СІ-42	
Н. Контр.		Луцик Н.С.					
Затверд.		Осухівська Г.М.					

Додаток В

Лістинг програмного коду

Повний вихідний код проєкту доступний у репозиторії GitHub: https://github.com/Den-k0/traffic_counter/tree/develop

camera/camera_controller.py:

```
import time
import paramiko
import logging
from typing import Optional, Type, Any
from config import StreamConfig

logger = logging.getLogger("TrafficAnalyzer.Camera")


class SSHCameraController:
    """Контролер SSH-з'єднання для віддаленого запуску камери на
Raspberry Pi.

    Керує життєвим циклом стріму (rpicam-vid), використовуючи
протокол TCP.
    Реалізує інтерфейс Context Manager для безпечного закриття.

    Args:
        ip (str): IP адреса Raspberry Pi у локальній мережі.
        user (str): SSH користувач.
        password (str): SSH пароль.
        max_retries (int, optional): Кількість спроб
підключення.
                                     За замовчуванням 3.
    """

    def __init__(
        self, ip: str, user: str, password: str, max_retries:
int = 3
    ) -> None:
        self.ip: str = ip
        self.user: str = user
        self.password: str = password
        self.max_retries: int = max_retries
        self.ssh: Optional[paramiko.SSHClient] = None
        self.stream_url: str =
f"tcp://{self.ip}:{StreamConfig.STREAM_PORT}"

    def __enter__(self) -> "SSHCameraController":
        """Встановлює з'єднання при вході в контекст.
```

```
        Returns:
            SSHCameraController: Екземпляр контролера.

        Raises:
            ConnectionError: Якщо підключитися не вдалося після
всіх спроб.
        """
        if not self.start_stream():
            raise ConnectionError(
                "Не вдалося запустити трансляцію з Raspberry
Pi."
            )
        return self

    def __exit__(
        self,
        exc_type: Optional[Type[BaseException]],
        exc_val: Optional[BaseException],
        exc_tb: Optional[Any],
    ) -> bool:
        """Гарантовано вимикає камеру та
        закриває сесію при виході з контексту.
        """
        self.stop_stream()
        if exc_type:
            logger.error(f"Аварійне завершення роботи камери:
{exc_val}")
        return False

    def start_stream(self) -> bool:
        """Спроба підключитися по SSH і запустити трансляцію
`rpicam-vid`.

        Вбиває старі завислі процеси, запускає новий та чекає
відкриття порту.

        Returns:
            bool: True при успішному запуску, False при невдачі.
        """
        if not all([self.ip, self.user, self.password]):
            logger.error(
                "Відсутні SSH-доступи (IP, USER або PASS) у .env
файлі!"
            )
            return False

        self.ssh = paramiko.SSHClient()

self.ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        for attempt in range(1, self.max_retries + 1):
            try:
```

```
                logger.info(
                    f"Підключення до {self.ip} "
                    f"(спроба {attempt}/{self.max_retries})..."
                )
                self.ssh.connect(
                    self.ip,
                    username=self.user,
                    password=self.password,
                    timeout=5,
                    look_for_keys=False,
                    allow_agent=False
                )

                self.ssh.exec_command("pkill rpicam-vid")
                time.sleep(1)

                logger.info("Запуск трансляції...")
                cmd = (
                    f"nohup rpicam-vid -t 0 --framerate 20 "
                    f"--saturation 0 --inline --listen -o "
                    f"tcp://0.0.0.0:{StreamConfig.STREAM_PORT} "
                    f">/dev/null 2>&1 &"
                )
                self.ssh.exec_command(cmd)

                logger.info("Очікування підняття порту...")
                wait_cmd = (
                    f"timeout {StreamConfig.STREAM_TIMEOUT} bash
-c "
                    f"'while ! ss -ltn | grep -q
:{StreamConfig.STREAM_PORT}; "
                    "do sleep 0.1; done; echo READY'"
                )
                _, stdout, _ = self.ssh.exec_command(wait_cmd)

                if stdout.read().decode().strip() == "READY":
                    logger.info("Порт відкритий. Буферизація
H.264...")
                    return True
                else:
                    raise Exception("Таймаут очікування порту")

            except Exception as e:
                logger.error(f"Помилка ініціалізації: {e}")
                if attempt == self.max_retries:
                    logger.critical(
                        "Не вдалося запустити камеру після всіх
спроб."
                    )
                    return False
                time.sleep(2)

        return False
```

```
    def stop_stream(self) -> None:
        """Зупиняє процес трансляції на
        віддаленому хості та закриває SSH-сесію.
        """
        if self.ssh:
            logger.info("Вимкнення камери та закриття SSH...")
            try:
                self.ssh.exec_command("pkill rpicam-vid")
                self.ssh.close()
            except Exception as e:
                logger.debug(f"Помилка при закритті SSH: {e}")
```

camera/line_counter.py:

```
import cv2
import logging
import numpy as np
from collections import OrderedDict
from typing import Any

logger = logging.getLogger("TrafficAnalyzer.Counter")


class LineCounter:
    """Алгоритм підрахунку перетинів контрольної лінії
об'єктами.

    Аналізує треки від YOLO, перевіряє їх на входження у зону
інтересу (ROI)
    та реєструє факт перетину заданої X-координати.

    Args:
        polygon (np.ndarray): Полігон ROI (дороги), що ігнорує
зайвий фон.
        line_position (int): X-координата вертикальної лінії
підрахунку.
        max_cache_size (int, optional): Розмір кеша унікальних
ID об'єктів
                                        для запобігання
дублюванню.
                                        За замовчуванням 1000.
    """

    def __init__(
        self,
        polygon: np.ndarray,
        line_position: int,
        max_cache_size: int = 1000,
    ) -> None:
        self.polygon: np.ndarray = polygon
        self.line_position: int = line_position
```

```
        self.total_objects: int = 0
        self.counted_ids: OrderedDict[int, bool] = (
            OrderedDict()
        )  # черга для безпечного очищення
        self.max_cache_size: int = max_cache_size

        self.previous_centroids: dict[int, tuple[int, int]] = {}
        self.line_color: tuple[int, int, int] = (0, 255, 255)

    def process_tracks(
        self, boxes: Any, class_names: dict[int, str]
    ) -> tuple[list[dict[str, Any]], list[str]]:
        """Обробляє результати трекера YOLO для поточного кадру.

        Args:
            boxes (Any): Об'єкт Ultralytics із результатами
                         детекції та трекінгу.
            class_names (dict[int, str]): Словник відповідності
                                          ID класів та їх назв.

        Returns:
            tuple:
                - list[dict]: Список подій трекінгу
                              (координати, ID, label) для
рендеру.
                - list[str]: Список назв класів об'єктів,
                             які щойно перетнули лінію.
        """
        current_centroids: dict[int, tuple[int, int]] = {}
        self.line_color = (0, 255, 255)
        events: list[dict[str, Any]] = []
        new_crossings: list[str] = []

        for box in boxes:
            if box.id is None:
                continue

            track_id = int(box.id.item())
            cls_id = int(box.cls[0])
            label = class_names[cls_id].upper()

            x1, y1, x2, y2 = map(int, box.xyxy[0])
            cx, cy = (x1 + x2) // 2, (y1 + y2) // 2

            # Перевірка полігону
            if cv2.pointPolygonTest(self.polygon, (cx, cy),
False) >= 0:
                current_centroids[track_id] = (cx, cy)
                events.append(
                    {
                        "id": track_id,
                        "label": label,
```

```
                        "bbox": (x1, y1, x2, y2),
                        "centroid": (cx, cy),
                    }
                )

                if track_id in self.previous_centroids:
                    prev_x =
self.previous_centroids[track_id][0]

                    crossed_lr = (
                        prev_x < self.line_position
                        and cx >= self.line_position
                    )
                    crossed_rl = (
                        prev_x >= self.line_position
                        and cx < self.line_position
                    )

                    if (
                        crossed_lr or crossed_rl
                    ) and track_id not in self.counted_ids:
                        self.total_objects += 1
                        self.counted_ids[track_id] = True
                        self.line_color = (0, 0, 255)

                        new_crossings.append(label)

                        logger.info(
                            f"Зафіксовано: ID {track_id} | Тип:
{label} | "
                            f"Всього: {self.total_objects}"
                        )

        self.previous_centroids = current_centroids.copy()

        while len(self.counted_ids) > self.max_cache_size:
            self.counted_ids.popitem(last=False)

        return events, new_crossings
```

camera/overlay.py:

```
import cv2
import numpy as np
from typing import Any
from config import GeometryConfig


class Visualizer:
    """Утиліта для накладання графіки (HUD) на кадри
відеопотоку.
```

```
    Малює Bounding Boxes, ID об'єктів, полігон дороги та
контрольну лінію.
    """

    @staticmethod
    def draw_text_with_outline(
        img: np.ndarray,
        text: str,
        pos: tuple[int, int],
        font_scale: float,
        text_color: tuple[int, int, int],
        thickness: int = 2,
    ) -> None:
        """Малює текст з чорним контуром для кращої
читабельності."""
        cv2.putText(
            img,
            text,
            pos,
            cv2.FONT_HERSHEY_SIMPLEX,
            font_scale,
            (0, 0, 0),
            thickness + 3,
        )
        cv2.putText(
            img,
            text,
            pos,
            cv2.FONT_HERSHEY_SIMPLEX,
            font_scale,
            text_color,
            thickness,
        )

    @staticmethod
    def render(
        frame: np.ndarray,
        events: list[dict[str, Any]],
        total_objects: int,
        intensity: int,
        state: str,
        state_color: tuple[int, int, int],
        line_color: tuple[int, int, int],
    ) -> np.ndarray:
        """Накладає ROI, лінію підрахунку, bounding box'и та HUD
на кадр.

        Args:
            frame (np.ndarray): OpenCV image.
            events (list[dict[str, Any]]): список подій трекінгу

({id,label,bbox,centroid}).
            total_objects (int): загальна кількість за сесію.
```

```
            intensity (int): інтенсивність (об'єктів за
хвилину).
            state (str): поточний стан трафіку.
            state_color (tuple[int, int, int]): колір тексту
стану.
            line_color (tuple[int, int, int]): колір лінії
                                                (змінюється при
перетині).
        Returns:
            np.ndarray: frame з накладеним UI.
        """
        h, _ = frame.shape[:2]

        # Статичні зони
        cv2.polylines(
            frame,
            [GeometryConfig.ROAD_POLYGON],
            isClosed=True,
            color=(255, 0, 0),
            thickness=2,
        )
        cv2.line(
            frame,
            (GeometryConfig.LINE_POSITION, 0),
            (GeometryConfig.LINE_POSITION, h),
            line_color,
            3 if line_color == (0, 0, 255) else 2,
        )

        # Динамічні об'єкти (з tracking events)
        for ev in events:
            x1, y1, x2, y2 = ev["bbox"]
            cx, cy = ev["centroid"]
            label = f"{ev['label']} {ev['id']}"

            cv2.circle(frame, (cx, cy), 4, (0, 255, 0), -1)
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255,
0), 1)
            Visualizer.draw_text_with_outline(
                frame, label, (x1, y1 - 10), 0.5, (0, 255, 0), 1
            )

        return frame
```

camera/state.py:

```
import threading
import queue
import logging

logger = logging.getLogger("TrafficAnalyzer.State")
```

```
class VideoStreamState:
    """Клас для потокобезпечного керування глобальним станом
застосунку.

    Відповідає за зберігання останнього кадру, бізнес-метрик
трафіку
    та керування чергою логів для запису в БД.
    """
    def __init__(self) -> None:
        """Ініціалізує початковий стан та примітиви
синхронізації."""
        self._latest_frame: bytes = b""
        self._total_objects: int = 0
        self._intensity: int = 0
        self._traffic_state: str = "FREE"
        self._total_persons: int = 0
        self._total_transport: int = 0
        self._active_viewers: int = 0

        self.is_running: bool = False

        self._lock = threading.Lock()
        self.log_queue: queue.Queue = queue.Queue()

    def update_frame(self, frame_bytes: bytes) -> None:
        """Потокобезпечно оновлює останній згенерований кадр.

        Args:
            frame_bytes (bytes): Закодоване зображення у форматі
JPEG.
        """
        with self._lock:
            self._latest_frame = frame_bytes

    def get_frame(self) -> bytes:
        """Потокобезпечно повертає останній кадр.

        Returns:
            bytes: Закодоване зображення.
        """
        with self._lock:
            return self._latest_frame

    def clear_frame(self) -> None:
        """Очищає буфер кадру для економії пам'яті, коли
глядачів немає."""
        with self._lock:
            self._latest_frame = b""

    def increment_viewers(self) -> None:
        """Реєструє нового глядача відеопотоку."""
        with self._lock:
```

```
            self._active_viewers += 1
            logger.info(
                f"👥 Нове підключення до відеопотоку. "
                f"Активних глядачів: {self._active_viewers}"
            )

    def decrement_viewers(self) -> None:
        """Відміняє реєстрацію глядача відеопотоку."""
        with self._lock:
            if self._active_viewers > 0:
                self._active_viewers -= 1
            logger.info(
                f"👥 Відключення від відеопотоку. "
                f"Активних глядачів: {self._active_viewers}"
            )

    def has_viewers(self) -> bool:
        """Перевіряє, чи є активні глядачі.

        Returns:
            bool: True, якщо відеопотік переглядають, інакше
False.
        """
        with self._lock:
            return self._active_viewers > 0

    def update_stats(
        self,
        total: int,
        intensity: int,
        state: str,
        total_persons: int,
        total_transport: int
    ) -> None:
        """Потокобезпечно оновлює метрики трафіку.

        Args:
            total (int): Загальна кількість зафіксованих
об'єктів.
            intensity (int): Кількість об'єктів за останнє
часове вікно.
            state (str): Поточний стан трафіку (FREE, NORMAL,
HEAVY).
            total_persons (int): Загальна кількість людей.
            total_transport (int): Загальна кількість
транспорту.
        """
        with self._lock:
            self._total_objects = total
            self._intensity = intensity
            self._traffic_state = state
            self._total_persons = total_persons
            self._total_transport = total_transport
```

```
    def get_stats(self) -> dict:
        """Потокобезпечно повертає поточну статистику.

        Returns:
            dict: Словник з ключами total_objects, intensity,
                  traffic_state, total_persons, total_transport.
        """
        with self._lock:
            return {
                "total_objects": self._total_objects,
                "intensity": self._intensity,
                "traffic_state": self._traffic_state,
                "total_persons": self._total_persons,
                "total_transport": self._total_transport
            }

global_state = VideoStreamState()
```

camera/traffic_state.py:

```
import time
from collections import deque
from web.models import TrafficStatus


class TrafficStateTracker:
    """Аналізатор інтенсивності трафіку за
    допомогою методу ковзного вікна (Sliding Window).

    Зберігає таймстемпи перетинів та видаляє
    старі дані для підрахунку "машин за хвилину".

    Args:
        free_limit (int): Верхня межа кількості авто
                          для стану FREE (Вільна дорога).
        jam_limit (int): Верхня межа кількості авто
                         для стану NORMAL. Вище - HEAVY (Затор).
        window_seconds (int): Розмір ковзного вікна в секундах
(наприклад, 60).
    """

    def __init__(
        self,
        free_limit: int = 10,
        jam_limit: int = 25,
        window_seconds: int = 60,
    ) -> None:
        self.crossing_timestamps: deque[float] = deque()
        self.free_limit: int = free_limit
        self.jam_limit: int = jam_limit
        self.window_seconds: int = window_seconds
```

```
    def register_crossing(self) -> None:
        """Додає поточний таймстемп при виявленому перетині
лінії."""
        self.crossing_timestamps.append(time.time())

    def get_metrics(self) -> tuple[int, str, tuple[int, int,
int]]:
        """Повертає (intensity, state, color) для поточної
множини таймстемпів.

        Видаляє старіші записи за межами `window_seconds`.
        """
        current_time = time.time()

        # Швидке очищення O(1): видаляємо старі записи зліва
(найстаріші)
        while (
            self.crossing_timestamps
            and current_time - self.crossing_timestamps[0]
            > self.window_seconds
        ):
            self.crossing_timestamps.popleft()

        intensity = len(self.crossing_timestamps)

        if intensity <= self.free_limit:
            state = TrafficStatus.FREE
            color = (0, 255, 0)
        elif intensity <= self.jam_limit:
            state = TrafficStatus.NORMAL
            color = (0, 255, 255)
        else:
            state = TrafficStatus.HEAVY
            color = (0, 0, 255)

        return intensity, state, color
```

camera/video_processor.py:

```
import cv2
import time
import logging

from ultralytics import YOLO
from imutils.video import VideoStream

from config import MLConfig, GeometryConfig, TrafficConfig,
SSHConfig
from camera.camera_controller import SSHCameraController
from camera.line_counter import LineCounter
from camera.traffic_state import TrafficStateTracker
```

```
from camera.overlay import Visualizer
from camera.state import global_state

logger = logging.getLogger("TrafficAnalyzer.Pipeline")


class VideoPipeline:
    """Ізольований клас для обробки відеопотоку та детекції
об'єктів.

    Відповідає за ініціалізацію нейромережі, отримання кадрів з
камери,
    трекінг об'єктів та генерацію подій перетину контрольної
лінії.
    """

    def __init__(self, initial_totals: dict, initial_timestamps:
list) -> None:
        """Ініціалізує пайплайн та відновлює стан з бази даних.

        Args:
            initial_totals (dict): Словник з початковими
                                   лічильниками об'єктів.
            initial_timestamps (list): Список таймстемпів
перетинів
                                       за останню хвилину.

        Raises:
            Exception: Якщо не вдалося завантажити YOLO модель.
        """
        logger.info("Ініціалізація VideoPipeline...")
        try:
            self.model = YOLO(MLConfig.YOLO_MODEL,
task="detect")
        except Exception as e:
            logger.error(f"Помилка завантаження моделі: {e}")
            raise e

        self.counter = LineCounter(
            polygon=GeometryConfig.ROAD_POLYGON,
            line_position=GeometryConfig.LINE_POSITION
        )
        self.counter.total_objects = initial_totals.get("total",
0)
        self.total_persons = initial_totals.get("persons", 0)
        self.total_transport = initial_totals.get("transport",
0)

        self.traffic_state = TrafficStateTracker(
            free_limit=TrafficConfig.FREE_FLOW_LIMIT,
            jam_limit=TrafficConfig.JAM_LIMIT,
            window_seconds=TrafficConfig.INTENSITY_WINDOW_SEC
        )
```

```
self.traffic_state.crossing_timestamps.extend(initial_timestamps
)

        init_intensity, init_state, _ =
self.traffic_state.get_metrics()
        global_state.update_stats(
            self.counter.total_objects, init_intensity,
init_state,
            self.total_persons, self.total_transport
        )

    def run(self) -> None:
        """Головний цикл пайплайну з авто-перепідключенням до
камери.

        Спрощує підключення по SSH, запускає трансляцію та
делегує
        обробку кадрів внутрішньому методу. Відновлює з'єднання
при збоях.
        """
        logger.info("Фоновий потік обробки відео запущено.")
        while global_state.is_running:
            try:
                logger.info(
                    "Спроба встановити SSH з'єднання з Raspberry
Pi..."
                )
                with SSHCameraController(
                    SSHConfig.RPI_IP,
                    SSHConfig.RPI_USER,
                    SSHConfig.RPI_PASS
                ) as camera:
                    cap = VideoStream(camera.stream_url).start()
                    logger.info("Трансляцію з камери успішно
запущено.")
                    self._process_stream(cap)
            except Exception as e:
                logger.error(f"Помилка в циклі обробки
відео/SSH-камери: {e}")

            if global_state.is_running:
                logger.info(
                    "Пауза 5 секунд перед наступною "
                    "спробою підключення..."
                )
                time.sleep(5)

        logger.info("Потік обробки відео зупинено.")

    def _process_stream(self, cap: VideoStream) -> None:
        """Внутрішній цикл покадрової обробки відеопотоку.
```

```
        Виконує детекцію через YOLO, рахує перетини, оновлює
глобальний стан
        і рендерить UI, якщо на сайті є активні глядачі.

        Args:
            cap (VideoStream): Об'єкт відеопотоку (imutils).
        """
        consecutive_none_frames = 0

        while global_state.is_running:
            frame = cap.read()

            if frame is None:
                consecutive_none_frames += 1
                if consecutive_none_frames > 300:
                    logger.warning(
                        "⚠️ Втрачено зв'язок із стрімом. "
                        "Ініціюємо перепідключення..."
                    )
                    break
                time.sleep(0.03)
                continue

            consecutive_none_frames = 0
            frame = cv2.resize(
                frame,
                (
                    GeometryConfig.FRAME_WIDTH,
                    GeometryConfig.FRAME_HEIGHT
                )
            )

            results = self.model.track(
                frame, conf=MLConfig.CONFIDENCE_THRESHOLD,
                iou=MLConfig.IOU_THRESHOLD, persist=True,
                tracker=MLConfig.TRACKER_CONFIG,
                classes=MLConfig.TARGET_CLASSES,
                imgsz=MLConfig.INFERENCE_SIZE, verbose=False
            )

            if results and len(results):
                events, new_crossings =
self.counter.process_tracks(
                    results[0].boxes, self.model.names
                )
                if new_crossings:
                    for obj_class in new_crossings:
                        self.traffic_state.register_crossing()
                        if obj_class == "PERSON":
                            self.total_persons += 1
                        else:
                            self.total_transport += 1
```

```
                    intensity, state, state_color = (
                        self.traffic_state.get_metrics()
                    )
                    for obj_class in new_crossings:
                        global_state.log_queue.put((obj_class,
state))
                else:
                    intensity, state, state_color = (
                        self.traffic_state.get_metrics()
                    )
            else:
                events = []
                intensity, state, state_color = (
                    self.traffic_state.get_metrics()
                )

            global_state.update_stats(
                self.counter.total_objects, intensity, state,
                self.total_persons, self.total_transport
            )

            if global_state.has_viewers():
                if not results or not len(results):
                    ret, buffer = cv2.imencode('.jpg', frame)
                else:
                    frame_rendered = Visualizer.render(
                        frame, events,
self.counter.total_objects, intensity,
                        state, state_color,
self.counter.line_color
                    )
                    ret, buffer = cv2.imencode('.jpg',
frame_rendered)

                if ret:
                    global_state.update_frame(buffer.tobytes())
            else:
                global_state.clear_frame()

            time.sleep(0.03)

        cap.stop()
```

web/static/style.css:

```
body { background-color: #121212; color: #ffffff; padding-
bottom: 50px; }
.video-container { text-align: center; margin-bottom: 30px; }
.chart-container { background-color: #1e1e1e; padding: 25px;
border-radius: 12px; border: 1px solid #333; }
```

```
.state-badge { font-size: 1.1rem; padding: 6px 16px; border-
radius: 6px; font-weight: bold; border: 1px solid currentColor;
display: inline-block;}
#time-filters .btn-outline-secondary { color: #aaa; border-
color: #444; }
#time-filters .btn-outline-secondary:hover { background-color:
#333; border-color: #555; color: #fff;}
```

web/static/dashboard.js:

```
// Глобальний стан фільтрів та кеш лічильників
let currentMode = '1min';
let currentType = 'all';
window.latestTotals = { all: 0, person: 0, transport: 0 };

// Запобіжник стану гонки (Race Condition) при завантаженні
історії
window.isChartInitialized = false;

const ctx =
document.getElementById('trafficChart').getContext('2d');
const chart = new Chart(ctx, {
    type: 'line',
    data: {
        labels: [],
        datasets: [{
            label: 'Зафіксовано об\'єктів',
            data: [],
            borderColor: '#00ff00',
            backgroundColor: 'rgba(0, 255, 0, 0.1)',
            borderWidth: 2,
            pointRadius: 3,
            pointBackgroundColor: '#00ff00',
            fill: true,
            tension: 0.3
        }]
    },
    options: {
        responsive: true,
        animation: { duration: 300 },
        scales: {
            x: { ticks: { color: '#aaa', maxRotation: 45,
maxTicksLimit: 15 }, grid: { display: false } },
            y: { ticks: { color: '#aaa', stepSize: 1 }, grid: {
color: '#333' }, beginAtZero: true }
        },
        plugins: { legend: { display: false } }
    }
});

async function fetchChartData() {
    try {
```

```
        const response = await
fetch(`/api/history?interval=${currentMode}&obj_type=${currentTy
pe}`);
        const data = await response.json();

        chart.data.labels = data.map(item =>
item.timestamp_group);
        chart.data.datasets[0].data = data.map(item =>
item.total_objects);
        chart.update();

        updateTotalBadge();
        window.isChartInitialized = true; // Дозволяємо
WebSocket малювати нові точки
    } catch (error) {
        console.error("Помилка завантаження історії:", error);
    }
}

function setMode(mode, event) {
    currentMode = mode;
    const buttons = document.querySelectorAll('#time-filters
.btn');
    buttons.forEach(btn => {
        btn.classList.remove('btn-outline-success', 'active');
        btn.classList.add('btn-outline-secondary');
    });

    const targetBtn = event ? event.currentTarget :
document.querySelector(`[onclick*="setMode('${mode}'"]`);
    if (targetBtn) {
        targetBtn.classList.remove('btn-outline-secondary');
        targetBtn.classList.add('btn-outline-success',
'active');
    }

    window.isChartInitialized = false;
    fetchChartData();
}

function setType(type, event) {
    currentType = type;
    const buttons = document.querySelectorAll('#type-filters
.btn');
    buttons.forEach(btn => {
        if(btn.classList.contains('active'))
btn.classList.remove('active');
    });

    const targetBtn = event ? event.currentTarget :
document.querySelector(`[onclick*="setType('${type}'"]`);
    if (targetBtn) targetBtn.classList.add('active');
```

```
    if (type === 'all') {
        chart.data.datasets[0].borderColor = '#00ff00';
        chart.data.datasets[0].backgroundColor = 'rgba(0, 255,
0, 0.1)';
        chart.data.datasets[0].pointBackgroundColor = '#00ff00';
    } else if (type === 'transport') {
        chart.data.datasets[0].borderColor = '#0dcaf0';
        chart.data.datasets[0].backgroundColor = 'rgba(13, 202,
240, 0.1)';
        chart.data.datasets[0].pointBackgroundColor = '#0dcaf0';
    } else if (type === 'person') {
        chart.data.datasets[0].borderColor = '#fd7e14';
        chart.data.datasets[0].backgroundColor = 'rgba(253, 126,
20, 0.1)';
        chart.data.datasets[0].pointBackgroundColor = '#fd7e14';
    }

    updateTotalBadge();
    window.isChartInitialized = false;
    fetchChartData();
}

function updateTotalBadge() {
    if (chart.data.datasets[0].data.length > 0) {
        const dataArray = chart.data.datasets[0].data;
        document.getElementById('total').innerText =
dataArray[dataArray.length - 1];
    }
}

fetchChartData();

/*
================================================================
=====
 * WEBSOCKET (Backend-Driven UI)
 * Використовуються мітки (buckets) від сервера.
 *
================================================================
===== */
const ws = new
WebSocket(`ws://${window.location.host}/ws/stats`);

ws.onmessage = function(event) {
    const data = JSON.parse(event.data);

    if (window.lastTotalObjects === undefined) {
        window.lastTotalObjects = data.total_objects;
        window.lastTotalPersons = data.total_persons;
        window.lastTotalTransport = data.total_transport;
    }

    window.latestTotals.all = data.total_objects;
```

```
    window.latestTotals.person = data.total_persons;
    window.latestTotals.transport = data.total_transport;

    updateTotalBadge();

    // Оновлення бейджа стану трафіку
    const stateEl = document.getElementById('state');
    stateEl.innerText = data.traffic_state;
    stateEl.className = `state-badge align-middle ${
        data.traffic_state === "FREE" ? "text-success border-
success" :
        data.traffic_state === "NORMAL" ? "text-warning border-
warning" :
        "text-danger border-danger"
    }`;

    if (!window.isChartInitialized) return;

    // Готова мітка від сервера для поточного режиму
    const expectedLabel = data.buckets[currentMode];

    if (expectedLabel) {
        let delta = 0;

        if (currentType === 'all') delta = data.total_objects -
window.lastTotalObjects;
        else if (currentType === 'person') delta =
data.total_persons - window.lastTotalPersons;
        else if (currentType === 'transport') delta =
data.total_transport - window.lastTotalTransport;

        const labels = chart.data.labels;
        const chartData = chart.data.datasets[0].data;

        if (labels.length > 0) {
            let shouldUpdateChart = false; // рендер тільки при
змінах

            if (labels[labels.length - 1] === expectedLabel) {
                if (delta > 0) {
                    chartData[chartData.length - 1] += delta;
                    shouldUpdateChart = true;
                }
            } else {
                labels.push(expectedLabel);
                chartData.push(delta);

                if (labels.length > 20) {
                    labels.shift();
                    chartData.shift();
                }
                shouldUpdateChart = true;
            }
```

```
            if (shouldUpdateChart) {
                chart.update();
            }
        }
    }

    window.lastTotalObjects = data.total_objects;
    window.lastTotalPersons = data.total_persons;
    window.lastTotalTransport = data.total_transport;
};
```

web/templates/index.html:

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Traffic Counter</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/boot
strap.min.css" rel="stylesheet">
    <link href="/static/css/style.css" rel="stylesheet">
    <script
src="https://cdn.jsdelivr.net/npm/chart.js"></script>
</head>
<body class="mt-4">
    <div class="container">
        <h2 class="mb-4 text-center fw-bold text-white">🚦
Traffic Counter</h2>

        <div class="row justify-content-center">
            <div class="col-lg-8 video-container">
                <img src="/video_feed" class="img-fluid border
border-secondary rounded shadow" alt="Live Video" style="width:
100%; max-height: 50vh; object-fit: cover;">
            </div>
        </div>

        <div class="row justify-content-center">
            <div class="col-lg-10 chart-container shadow mb-5">

                <div class="d-flex justify-content-between
align-items-center mb-4">
                    <h4 class="mb-0 text-white">Статистика
руху</h4>
                    <div class="text-end">
                        <span class="me-3 text-light fs-5 align-
middle">Всього: <strong id="total" class="text-white fs-3 ms-
1">0</strong></span>
                        <span id="state" class="state-badge
text-success align-middle">FREE</span>
```

```
                    </div>
                </div>

                <canvas id="trafficChart" height="80"></canvas>

                <div class="d-flex flex-column flex-md-row
justify-content-between align-items-center mt-4 pt-3 border-top
border-dark">
                    <div class="btn-group mb-3 mb-md-0 shadow-
sm" role="group" id="type-filters">
                        <button type="button" class="btn btn-
outline-success active" onclick="setType('all',
event)">Всі</button>
                        <button type="button" class="btn btn-
outline-success" onclick="setType('transport',
event)">Транспорт</button>
                        <button type="button" class="btn btn-
outline-success" onclick="setType('person',
event)">Люди</button>
                    </div>

                    <div class="btn-group shadow-sm flex-wrap"
role="group" id="time-filters">
                        <button type="button" class="btn btn-
outline-success active" onclick="setMode('1min', event)">1
хв</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('15min', event)">15
хв</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('hour',
event)">Година</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('day', event)">День</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('week',
event)">Тиждень</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('month',
event)">Місяць</button>
                        <button type="button" class="btn btn-
outline-secondary" onclick="setMode('year', event)">Рік</button>
                    </div>
                </div>
            </div>
        </div>
    </div>

    <script src="/static/js/dashboard.js"></script>
</body>
</html>
```

web/app.py:

```
import threading
from contextlib import asynccontextmanager
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
import logging

from camera.state import global_state
from camera.video_processor import VideoPipeline
from web.workers import db_writer_worker
from web.routes import router
from web.db import SessionLocal
from web.models import TrafficLog, ObjectType
from config import TrafficConfig

logger = logging.getLogger("TrafficAnalyzer.App")


def fetch_initial_stats_from_db():
    """Вивантажує початкову статистику з бази даних при старті
сервера.

    Необхідно для відновлення роботи системи після перезапуску
(щоб не скидати
    лічильники в 0). Читає загальну кількість та перетини за
останню хвилину.

    Returns:
        tuple:
            - Словник із ключами total, persons, transport.
            - Список float таймстемпів перетинів за останню
хвилину.
    """
    try:
        with SessionLocal() as db:
            initial_total = db.query(TrafficLog).count()
            initial_persons = db.query(TrafficLog).filter(
                TrafficLog.object_class == ObjectType.PERSON
            ).count()
            initial_transport = initial_total - initial_persons

            one_min_ago = datetime.now(ZoneInfo("UTC")) -
timedelta(
                seconds=TrafficConfig.INTENSITY_WINDOW_SEC
            )
            recent_logs = db.query(TrafficLog).filter(
                TrafficLog.timestamp >= one_min_ago
            ).all()
            crossing_timestamps = [
                log.timestamp.timestamp() for log in recent_logs
```

```
            ]

            return {
                "total": initial_total,
                "persons": initial_persons,
                "transport": initial_transport
            }, crossing_timestamps
    except Exception as e:
        logger.error(f"Помилка ініціалізації з БД: {e}")
        return {"total": 0, "persons": 0, "transport": 0}, []

bg_threads = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Керує життєвим циклом (Startup/Shutdown) FastAPI
застосунку.

    Оркеструє запуск та коректне завершення фонових потоків
(відео та БД).

    Args:
        app (FastAPI): Інстанс FastAPI.
    """
    global_state.is_running = True

    initial_totals, initial_timestamps =
fetch_initial_stats_from_db()

    try:
        pipeline = VideoPipeline(initial_totals,
initial_timestamps)
    except Exception as e:
        logger.critical(f"Не вдалося запустити VideoPipeline:
{e}")
        yield
        return

    bg_threads['video'] = threading.Thread(target=pipeline.run,
daemon=True)
    bg_threads['db'] = threading.Thread(target=db_writer_worker,
daemon=True)

    bg_threads['video'].start()
    bg_threads['db'].start()

    yield

    global_state.is_running = False

    if 'video' in bg_threads:
        bg_threads['video'].join(timeout=5.0)
    if 'db' in bg_threads:
```

```
        bg_threads['db'].join(timeout=5.0)

app = FastAPI(title="Traffic Counter API", lifespan=lifespan)
app.mount("/static", StaticFiles(directory="web/static"),
name="static")
app.include_router(router)
```

web/db.py:

```
"""Модуль налаштування підключення до бази даних (SQLAlchemy).

Створює підключення до SQLite та визначає
базовий декларативний клас для моделей.
"""
from pathlib import Path
from sqlalchemy import create_engine
from sqlalchemy.orm import declarative_base, sessionmaker,
Session

# База даних знаходиться у корені проєкту
PROJECT_DIR = Path(__file__).parent.parent
DB_PATH = PROJECT_DIR / "traffic.db"
SQLALCHEMY_DATABASE_URL = f"sqlite:///{DB_PATH}"

engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread":
False}
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False,
bind=engine)

Base = declarative_base()


def get_db() -> Session:
    """Генератор сесій бази даних.

    Автоматично закриває з'єднання після завершення обробки
HTTP-запиту.

    Yields:
        Session: Активна сесія SQLAlchemy.
    """
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

web/history_service.py:

```
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
from sqlalchemy.orm import Session
from sqlalchemy import func
from fastapi import HTTPException
from typing import Callable, Tuple

from web.models import TrafficLog, TrafficStatus, ObjectType
from web.schemas import HistoricalTrafficData
from config import AppConfig


class HistoryService:
    """Сервіс для агрегації історичних даних трафіку з бази
даних."""

    @staticmethod
    def _get_interval_config(
        interval: str, now_local: datetime
    ) -> Tuple[datetime, timedelta, Callable[[datetime], str],
str]:
        """Визначає параметри групування даних
        для SQL запиту залежно від обраного інтервалу.

        Args:
            interval (str): Назва інтервалу (1min, hour, day,
week, etc.).
            now_local (datetime): Поточний локальний час.

        Returns:
            Tuple[datetime, timedelta, Callable, str]:
                - Час початку вибірки.
                - Крок часу для генерації порожніх бакетів.
                - Функція для форматування дати у мітку (Label).
                - Формат дати для SQL функції strftime.

        Raises:
            HTTPException: Якщо передано невалідний інтервал.
        """
        if interval == "1min":
            return (
                now_local - timedelta(minutes=15),
                timedelta(minutes=1),
                lambda dt: dt.strftime("%H:%M"), '%Y-%m-%d
%H:%M'
            )
        elif interval == "15min":
            return (
                now_local - timedelta(hours=4),
                timedelta(minutes=15),
```

```
                lambda dt: f"{dt.hour:02d}:{(dt.minute // 15) *
15:02d}",
                '%Y-%m-%d %H:%M'
            )
        elif interval == "hour":
            return (
                now_local - timedelta(hours=23),
                timedelta(hours=1),
                lambda dt: dt.strftime("%H:00"),
                '%Y-%m-%d %H:00'
            )
        elif interval == "day":
            return (
                now_local - timedelta(days=7),
                timedelta(days=1),
                lambda dt: dt.strftime("%d.%m"),
                '%Y-%m-%d'
            )
        elif interval == "week":
            def get_week_label(dt: datetime) -> str:
                monday = dt - timedelta(days=dt.weekday())
                sunday = monday + timedelta(days=6)
                return f"{monday.strftime('%d.%m')}-
{sunday.strftime('%d.%m')}"
            return (
                now_local - timedelta(weeks=4),
                timedelta(days=7),
                get_week_label,
                '%Y-%m-%d'
            )
        elif interval == "month":
            return (
                now_local - timedelta(days=365),
                timedelta(days=20),
                lambda dt: dt.strftime("%m.%Y"),
                '%Y-%m'
            )
        elif interval == "year":
            return (
                now_local - timedelta(days=365 * 5),
                timedelta(days=100),
                lambda dt: dt.strftime("%Y"),
                '%Y'
            )
        else:
            raise HTTPException(
                status_code=400,
                detail="Invalid interval specified"
            )

    @staticmethod
    def get_historical_data(
        interval: str, obj_type: str, db: Session
```

```
    ) -> list[HistoricalTrafficData]:
        """Витягує та агрегує історичні дані трафіку для
графіка.

        Використовує SQL GROUP BY для оптимізації пам'яті
        (замість вивантаження всіх записів).

        Args:
            interval (str): Часовий інтервал угруповання.
            obj_type (str): Тип об'єкта для фільтрації
                            ('all', 'person', 'transport').
            db (Session): Сесія SQLAlchemy.

        Returns:
            list[HistoricalTrafficData]: Список агрегованих
даних для
                                         побудови графіка на
фронтенді.
        """
        tz_local = ZoneInfo(AppConfig.TIMEZONE)
        now_local = datetime.now(tz_local)

        start_time_local, step, get_bucket_key, sql_fmt = (
            HistoryService._get_interval_config(interval,
now_local)
        )
        start_time_utc =
start_time_local.astimezone(ZoneInfo("UTC"))

        labels = []
        curr = start_time_local
        while curr <= now_local:
            labels.append(get_bucket_key(curr))
            curr += step
        labels.append(get_bucket_key(now_local))
        labels = list(dict.fromkeys(labels))

        grouped_data = {label: {"total": 0, "states": {}} for
label in labels}

        query = db.query(
            func.strftime(sql_fmt,
TrafficLog.timestamp).label('bucket'),
            TrafficLog.traffic_state,
            func.count(TrafficLog.id).label('count')
        ).filter(TrafficLog.timestamp >= start_time_utc)

        if obj_type == "person":
            query = query.filter(TrafficLog.object_class ==
ObjectType.PERSON)
        elif obj_type == "transport":
            query = query.filter(TrafficLog.object_class !=
ObjectType.PERSON)
```

```
        query = query.group_by('bucket',
TrafficLog.traffic_state)
        db_results = query.all()

        parse_fmt_map = {
            '%Y-%m-%d %H:%M': '%Y-%m-%d %H:%M',
            '%Y-%m-%d %H:00': '%Y-%m-%d %H:00',
            '%Y-%m-%d': '%Y-%m-%d',
            '%Y-%m': '%Y-%m',
            '%Y': '%Y'
        }
        parse_fmt = parse_fmt_map[sql_fmt]

        for bucket_str, state, count in db_results:
            if not bucket_str:
                continue

            dt_utc = datetime.strptime(bucket_str,
parse_fmt).replace(
                tzinfo=ZoneInfo("UTC")
            )
            dt_local = dt_utc.astimezone(tz_local)
            time_key = get_bucket_key(dt_local)

            if time_key in grouped_data:
                grouped_data[time_key]["total"] += count
                grouped_data[time_key]["states"][state] = (
                    grouped_data[time_key]["states"].get(state,
0) + count
                )

        results = []
        for time_key in labels:
            data = grouped_data[time_key]
            states = data["states"]

            dominant = max(
                states, key=states.get
            ) if states else TrafficStatus.FREE
            state_str = dominant.value if isinstance(
                dominant, TrafficStatus
            ) else dominant

            results.append(HistoricalTrafficData(
                timestamp_group=time_key,
                total_objects=data["total"],
                average_intensity=float(data["total"]),
                dominant_state=state_str
            ))

        return results
```

```
    @staticmethod
    def get_current_buckets(now: datetime) -> dict:
        """Генерує готові мітки часу для всіх можливих фільтрів
фронтенда.

        Використовується WebSocket-броадкастером
        для реалізації патерну Backend-Driven UI.

        Args:
            now (datetime): Поточний локальний час сервера.

        Returns:
            dict: Словник з готовими відформатованими мітками
для графіку.
        """
        monday = now - timedelta(days=now.weekday())
        sunday = monday + timedelta(days=6)
        return {
            "1min": now.strftime("%H:%M"),
            "15min": f"{now.hour:02d}:{(now.minute // 15) *
15:02d}",
            "hour": now.strftime("%H:00"),
            "day": now.strftime("%d.%m"),
            "week": f"{monday.strftime('%d.%m')}-
{sunday.strftime('%d.%m')}",
            "month": now.strftime("%m.%Y"),
            "year": now.strftime("%Y")
        }
```

web/models.py:

```
"""Моделі бази даних (SQLAlchemy) та перерахування (Enums)
системи."""
import enum
from datetime import datetime
from zoneinfo import ZoneInfo
from sqlalchemy.orm import Mapped, mapped_column
from web.db import Base


class TrafficStatus(str, enum.Enum):
    """Стани завантаженості дороги."""
    FREE = "FREE"
    NORMAL = "NORMAL"
    HEAVY = "HEAVY"


class ObjectType(str, enum.Enum):
    """Типи об'єктів, які розпізнає нейромережа."""
    PERSON = "PERSON"         # клас 0
    CAR = "CAR"               # клас 2
    MOTORCYCLE = "MOTORCYCLE" # клас 3
```

```
    BUS = "BUS"                # клас 5
    TRUCK = "TRUCK"            # клас 7


class TrafficLog(Base):
    """Модель ORM для таблиці логів трафіку.

    Кожен запис — це один зафіксований перетин контрольної
лінії.
    """
    __tablename__ = "traffic_logs"

    id: Mapped[int] = mapped_column(primary_key=True,
index=True)

    timestamp: Mapped[datetime] = mapped_column(
        default=lambda: datetime.now(ZoneInfo("UTC"))
    )

    object_class: Mapped[ObjectType] = mapped_column(index=True)
    traffic_state: Mapped[TrafficStatus] =
mapped_column(index=True)
```

web/routes.py:

```
import asyncio
from datetime import datetime
from zoneinfo import ZoneInfo
from fastapi import APIRouter, Depends, WebSocket,
WebSocketDisconnect, Request
from fastapi.responses import StreamingResponse, FileResponse
from sqlalchemy.orm import Session
from typing import List

from web.db import get_db
from web.schemas import HistoricalTrafficData
from web.history_service import HistoryService
from camera.state import global_state
from config import AppConfig

router = APIRouter()


class ConnectionManager:
    """Менеджер для керування активними WebSocket
з'єднаннями."""

    def __init__(self) -> None:
        """Ініціалізує список активних клієнтів."""
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket) -> None:
```

```
        """Приймає підключення та додає до списку розсилки."""
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket) -> None:
        """Видаляє підключення зі списку при відключенні
клієнта."""
        self.active_connections.remove(websocket)

    async def broadcast(self, data: dict) -> None:
        """Розсилає JSON повідомлення всім
        підключеним клієнтам (Broadcast Pattern).
        """
        for connection in self.active_connections:
            await connection.send_json(data)

manager = ConnectionManager()

async def stats_broadcaster() -> None:
    """Асинхронне фонове завдання для розсилки телеметрії.

    Раз на секунду зчитує глобальний стан, генерує часові мітки
для графіка
    і розсилає ці дані всім WebSocket клієнтам.
    """
    while global_state.is_running:
        if manager.active_connections:
            stats = global_state.get_stats()
            now = datetime.now(ZoneInfo(AppConfig.TIMEZONE))
            await manager.broadcast({
                "time": now.strftime("%H:%M:%S"),
                "buckets":
HistoryService.get_current_buckets(now),
                "intensity": stats["intensity"],
                "total_objects": stats["total_objects"],
                "traffic_state": stats["traffic_state"],
                "total_persons": stats.get("total_persons", 0),
                "total_transport": stats.get("total_transport",
0)
            })
        await asyncio.sleep(1)

@router.on_event("startup")
async def startup_event() -> None:
    """Хук, що спрацьовує при старті FastAPI для запуску
броадкастера."""
    asyncio.create_task(stats_broadcaster())

@router.get("/")
async def index() -> FileResponse:
    """Віддає головну HTML сторінку дашборду."""
    return FileResponse("web/templates/index.html")
```

```
@router.websocket("/ws/stats")
async def websocket_stats(websocket: WebSocket) -> None:
    """Ендпоінт для встановлення WebSocket з'єднання з
клієнтом."""
    await manager.connect(websocket)
    try:
        while True:
            await websocket.receive_text()
    except (WebSocketDisconnect, asyncio.CancelledError):
        manager.disconnect(websocket)

@router.get("/video_feed")
async def video_feed(request: Request) -> StreamingResponse:
    """Ендпоінт для стрімінгу обробленого відео (MJPEG).

    Реєструє підключення для оптимізації рендеру відео на
бекенді.
    Якщо клієнтів немає — рендер та кодування кадрів
вимикається.
    """
    async def frame_generator():
        global_state.increment_viewers()
        try:
            last_sent_frame = None
            while global_state.is_running:
                if await request.is_disconnected():
                    break
                current_frame = global_state.get_frame()
                if current_frame and current_frame !=
last_sent_frame:
                    last_sent_frame = current_frame
                    yield (
                        b"--frame\r\nContent-Type:
image/jpeg\r\n\r\n"
                        + last_sent_frame
                        + b"\r\n"
                    )
                await asyncio.sleep(0.03)
        finally:
            global_state.decrement_viewers()

    return StreamingResponse(
        frame_generator(),
        media_type="multipart/x-mixed-replace; boundary=frame"
    )

@router.get("/api/history",
response_model=list[HistoricalTrafficData])
async def get_historical_data(
    interval: str,
    obj_type: str = "all",
    db: Session = Depends(get_db)
):
```

```
    """REST API ендпоінт для отримання історичних даних для
графіку.

    Args:
        interval (str): Назва часового фільтра (1min, 15min,
hour, etc.).
        obj_type (str): Фільтр за типом об'єкта (all, transport,
person).
        db (Session): Сесія підключення до БД (впорскується
автоматично).

    Returns:
        list[HistoricalTrafficData]: Список агрегованих точок
даних.
    """
    return HistoryService.get_historical_data(
        interval=interval, obj_type=obj_type, db=db
    )
```

web/schemas.py:

```
"""Схеми Pydantic для валідації та серіалізації даних API."""
from datetime import datetime
from pydantic import BaseModel, ConfigDict
from web.models import TrafficStatus, ObjectType


class TrafficLogBase(BaseModel):
    """Базова схема логування трафіку."""
    object_class: ObjectType
    traffic_state: TrafficStatus


class TrafficLogCreate(TrafficLogBase):
    """Схема для створення нового запису."""
    pass


class TrafficLogResponse(TrafficLogBase):
    """Схема відповіді API з деталями перетину."""
    id: int
    timestamp: datetime

    model_config = ConfigDict(from_attributes=True)


class HistoricalTrafficData(BaseModel):
    """Схема агрегованих даних для побудови графіка на
фронтенді.

    Attributes:
        timestamp_group (str): Згрупована мітка часу
```

```
                                (наприклад, '14:00' або '15.06').
        total_objects (int): Сумарна кількість об'єктів за цей
бакет часу.
        average_intensity (float): Середня інтенсивність
(швидкість потоку).
        dominant_state (str): Найчастіший стан трафіку у цьому
бакеті
                                (FREE/NORMAL/HEAVY).
    """
    timestamp_group: str
    total_objects: int
    average_intensity: float
    dominant_state: str
```

web/workers.py:

```
import logging
import queue
from camera.state import global_state
from web.db import SessionLocal
from web.models import TrafficLog

logger = logging.getLogger("TrafficAnalyzer.Worker")


def db_writer_worker() -> None:
    """Ізольований фоновий потік для пакетного запису в базу
даних.

    Читає події з черги `global_state.log_queue` та виконує
масовий (batch)
    insert у базу даних SQLite. Гарантує, що операції I/O з
базою не блокують
    комп'ютерний зір.
    """
    logger.info("Фоновий потік БД запущено.")

    while global_state.is_running or not
global_state.log_queue.empty():
        try:
            item = global_state.log_queue.get(timeout=0.5)
            batch = [item]

            while not global_state.log_queue.empty():
                try:

batch.append(global_state.log_queue.get_nowait())
                except queue.Empty:
                    break

            with SessionLocal() as db:
                try:
```

```
                    for obj_class, traffic_state in batch:
                        db.add(TrafficLog(
                            object_class=obj_class,
                            traffic_state=traffic_state)
                        )
                    db.commit()
                except Exception as e:
                    logger.error(f"Помилка фонового запису в БД:
{e}")
                    db.rollback()

        except queue.Empty:
            continue

    logger.info("Фоновий потік БД успішно зупинено.")
```

config.py:

```
"""Конфігураційний модуль застосунку.

Використовує pydantic-settings для безпечного завантаження
змінних оточення
та звичайні класи для організації конфігураційного простору імен
(namespaces).
"""
import numpy as np
from pydantic_settings import BaseSettings, SettingsConfigDict


class EnvSettings(BaseSettings):
    """Схема валідації змінних оточення з файлу .env.

    Гарантує (Fail-Fast), що застосунок не запуститься
    без обов'язкових облікових даних до Raspberry Pi.
    """
    model_config = SettingsConfigDict(
        env_file=".env",
        env_file_encoding="utf-8",
        extra="ignore"
    )

    rpi_ip: str
    rpi_user: str
    rpi_pass: str

env = EnvSettings()


class SSHConfig:
    """SSH доступи до Raspberry Pi."""
    RPI_IP: str = env.rpi_ip
    RPI_USER: str = env.rpi_user
```

```
    RPI_PASS: str = env.rpi_pass


class StreamConfig:
    """Мережеві налаштування відеотрансляції."""
    STREAM_PORT: int = 8888
    STREAM_TIMEOUT: int = 15


class MLConfig:
    """Налаштування нейромережі (YOLO) та параметри трекера."""
    YOLO_MODEL: str = "yolov8m.mlpackage"
    CONFIDENCE_THRESHOLD: float = 0.25
    IOU_THRESHOLD: float = 0.4
    INFERENCE_SIZE: int = 640
    TRACKER_CONFIG: str = "custom_tracker.yaml"
    # 0 - людина, 2 - авто, 3 - мото, 5 - автобус, 7 -
вантажівка
    TARGET_CLASSES: list[int] = [0, 2, 3, 5, 7]


class TrafficConfig:
    """Логіка підрахунку та визначення станів трафіку."""
    FREE_FLOW_LIMIT: int = 10
    JAM_LIMIT: int = 25
    INTENSITY_WINDOW_SEC: int = 60


class GeometryConfig:
    """Геометрія кадру (ROI) та параметри обробки зображення."""
    FRAME_WIDTH: int = 640
    FRAME_HEIGHT: int = 360
    LINE_POSITION: int = 350
    ROAD_POLYGON: np.ndarray = np.array([
        [0, 160],   # Верхній лівий кут
        [640, 160], # Верхній правий кут
        [640, 330], # Нижній правий кут
        [0, 330]    # Нижній лівий кут
    ], np.int32)


class AppConfig:
    """Загальні налаштування бекенд-застосунку."""
    TIMEZONE: str = "Europe/Kyiv"
```

logging_config.py:

```
LOGGING_CONFIG = {
    "version": 1,
    "disable_existing_loggers": False,
    "formatters": {
        "standard": {
```

```
            "format": "%(asctime)s | %(levelname)s |
%(message)s",
            "datefmt": "%H:%M:%S",
        },
    },
    "handlers": {
        "console": {
            "level": "INFO",
            "class": "logging.StreamHandler",
            "formatter": "standard",
        },
        "file": {
            "level": "INFO",
            "class": "logging.FileHandler",
            "filename": "traffic.log",
            "encoding": "utf-8",
            "formatter": "standard",
        },
    },
    "loggers": {
        "": {
            "handlers": ["console", "file"],
            "level": "INFO",
        },
        "uvicorn": {
            "handlers": ["console", "file"],
            "level": "INFO",
            "propagate": False,
        },
        "uvicorn.error": {
            "level": "INFO",
        },
        "uvicorn.access": {
            "handlers": ["console", "file"],
            "level": "INFO",
            "propagate": False,
        },
    },
}
```

main.py:

```
"""Головний вхідний файл застосунку.

Відповідає за конфігурацію логера та запуск ASGI-сервера
Uvicorn.
"""
import logging
import logging.config
from dotenv import load_dotenv
import uvicorn
```

```
load_dotenv()

from logging_config import LOGGING_CONFIG

if __name__ == "__main__":
    logging.config.dictConfig(LOGGING_CONFIG)
    logger = logging.getLogger("TrafficAnalyzer")
    logger.info("🚀 Запуск сервера Traffic Counter...")

    uvicorn.run(
        "web.app:app",
        host="0.0.0.0",
        port=8000,
        reload=False,
        log_config=None
    )
```