Міністерство освіти і науки України
Західноукраїнський національний університет
Факультет комп’ютерних інформаційних технологій
Кафедра спеціалізованих комп’ютерних систем

Музика Сергій Русланович

КОНТЕЙНЕРИЗОВАНА IOT-ПЛАТФОРМА ДЛЯ АВТОМАТИЗОВАНОГО МОНІТОРИНГУ ВИРОБНИЧОЇ ЛІНІЇ З ВИКОРИСТАННЯМ КОМП'ЮТЕРНОГО ЗОРУ / CONTAINERIZED IOT PLATFORM FOR AUTOMATED PRODUCTION LINE MONITORING USING COMPUTER VISION

спеціальність: 151 – Автоматизація та комп’ютерно-інтегровані технології
освітньо-професійна програма – Автоматизація та комп’ютерно-інтегровані технології

Кваліфікаційна робота

Виконав студент групи АКІТ-41
Музика С.Р.
_____

Науковий керівник:
д.т.н., Николайчук Я.М.
_____

Дипломну роботу допущено до захисту:
"____"_____20___ р.

Завідувач кафедри
_____ А.І. Сегін

Тернопіль 2026

Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра спеціалізованих комп'ютерних систем
Освітній ступінь "бакалавр"
Спеціальність: 151 - Автоматизація та комп'ютерно-інтегровані технології
Освітньо-професійна програма – Автоматизація та комп'ютерно-інтегровані технології

"ЗАТВЕРДЖУЮ"

Т.в.о. Завідувача кафедри СКС
_____ А.І.Сегін
"____"_____20__ р.

З А В Д А Н Н Я

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Музики Сергія Руслановича
(прізвище, ім'я по-батькові)

1. Тема випускної кваліфікаційної роботи
Контейнеризована IOT-Платформа для моніторингу виробничої лінії з використанням комп'ютерного зору / Containerized IOT platform for automated production line monitoring using computer vision
керівник роботи д.т.н., Николайчук Я.М.

затверджені наказом по університету від 1.12.2025р №894

2. Строк подання студентом закінченої випускної кваліфікаційної роботи:
17 травня 2021р.

3. Вихідні дані до випускної кваліфікаційної роботи:
1. Виробнича лінія підприємства з виробництва пластикових виробів
2. IP-камери з підтримкою протоколу RTSP та промислові I/O-пристрої з інтерфейсом RS485
3. Перелік контрольованих параметрів виробничого процесу

4. Основні питання, які потрібно розробити:
1. Аналіз існуючих рішень для промислового IoT-моніторингу.
2. Розроблення архітектури контейнеризованої IoT-платформи
3. Розроблення програмних модулів системи.

5. Перелік графічного матеріалу у роботі:
1. Структурна схема архітектури системи
2. Блок-схема алгоритму аналізу зображення

6. Консультанти розділів випускної кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 1 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назви етапів випускної кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз існуючих аналогів	01.12.2025р. – 15.01.2026р.	
2	Розроблення структурної схеми системи	16.01.2026р. – 15.03.2026р.	
3	Розроблення програмних модулів та тестування системи	16.03.2026р. – 30.04.2026р.	
4	Остаточне оформлення та подача кваліфікаційної роботи на перевірку щодо плагіату	1.05.2026р. – 25.05.2026	

Студент _____ Музика С.Р.
(підпис)

Керівник роботи _____ д.т.н., Николайчук Я.М.
(підпис)

РЕФЕРАТ

Робота виконана на 81 сторінках та містить 12 рисунків, 1 додаток, 2 таблиці, 20 джерел з переліком посилань.

Мета роботи. Метою роботи є розробка контейнеризованої IoT-платформи для автоматизованого моніторингу виробничої лінії на підприємстві з виробництва пластикових виробів із застосуванням методів комп'ютерного зору та збором даних із промислових датчиків.

Методи дослідження. Методи та засоби розроблення мікросервісних систем на основі контейнеризації, методи комп'ютерного зору для обробки відеопотоків у реальному часі, промислові протоколи передачі даних та технології збору й візуалізації метрик.

Результати роботи. Результатом роботи є розроблена та впроваджена на діючому підприємстві IoT-платформа, яка забезпечує одночасний моніторинг чотирьох виробничих ліній з п'ятьма камерами та дев'ятьма I/O-пристроями (72 дискретних входи) з часом реагування до 0,3 секунди.

Рекомендації по використанню результатів роботи полягають у застосуванні розробленої платформи для автоматизованого контролю якості продукції на малих і середніх виробничих підприємствах із мінімальними витратами на впровадження завдяки використанню виключно відкритого програмного забезпечення.

Можливі напрямки розвитку. При використанні отриманих результатів можливе вдосконалення системи шляхом застосування методів машинного навчання для підвищення точності класифікації дефектів та розширення переліку контрольованих параметрів продукції.

Ключові слова: ІОТ-ПЛАТФОРМА, ПРОМИСЛОВИЙ МОНІТОРИНГ, КОМП'ЮТЕРНИЙ ЗІР, DOCKER, MODBUS TCP, OPENCV, FLASK, PROMETHEUS, КОНТРОЛЬ ЯКОСТІ, INDUSTRY 4.0.

ABSTRACT

Work is executed on 81 pages and including 12 illustrations, 1 appendix, 2 tables, 20 sources after the list of references.

The goal of the work. The aim of the work is to develop a containerized IoT platform for automated monitoring of a production line at a plastic products manufacturing enterprise using computer vision methods and industrial sensor data acquisition.

Research methods. Methods and tools for developing microservice systems based on containerization, computer vision methods for real-time video stream processing, industrial data transfer protocols, and metrics collection and visualization technologies.

Results of work. The result of work is a developed and deployed IoT platform at an active manufacturing facility, ensuring simultaneous monitoring of four production lines with five cameras and nine I/O devices (72 discrete inputs) with a response time of up to 0.3 seconds.

Recommendations for the use of the results of the work are to apply the developed platform for automated product quality control at small and medium-sized manufacturing enterprises with minimal implementation costs due to the exclusive use of open-source software.

Possible directions of development. The system can be improved by applying machine learning methods to increase the accuracy of defect classification and expanding the list of monitored product parameters.

Keywords: IOT PLATFORM, INDUSTRIAL MONITORING, COMPUTER VISION, DOCKER, MODBUS TCP, OPENCV, FLASK, PROMETHEUS, QUALITY CONTROL, INDUSTRY 4.0.

ЗМІСТ

ВСТУП

Сучасне промислове виробництво все частіше стикається з проблемою, яку важко вирішити традиційними методами: контроль якості продукції на конвеєрі вимагає постійної уваги, але людські ресурси на це обмежені. Саме тут з'являється потреба в автоматизованих системах моніторингу — і саме в цьому контексті розвивається напрям Industry 4.0, що об'єднує кіберфізичні системи, IoT та аналіз даних у реальному часі.

Виробництво пластикових виробів не є винятком: це технологічно складний процес, де відхилення в геометрії продукту можна виявити лише якщо стежити за лінією постійно. Ручний контроль тут має очевидні обмеження — оператор втомлюється, не може бути одночасно на кількох лініях, а брак виявляється із запізненням. Автоматизація через камери та датчики вирішує цю проблему без збільшення штату.

Docker-контейнеризація стала зручним способом розгортати такі системи на виробництві: не потрібно думати про залежності, оновлення окремих компонентів не ламає решту, а у разі збою контейнер перезапускається сам. У поєднанні з комп'ютерним зором та промисловими протоколами це дозволяє будувати досить компактні, але надійні рішення для моніторингу.

Актуальність теми дипломної роботи обумовлена необхідністю розробки комплексного програмно-апаратного рішення для автоматизованого моніторингу виробничої лінії, яке поєднує збір даних із промислових датчиків, аналіз відеопотоків із камер за допомогою алгоритмів комп'ютерного зору та надання операторам зручного веб-інтерфейсу для управління системою.

Метою дипломної роботи є проектування та розробка контейнеризованої IoT-платформи для автоматизації та моніторингу виробничої лінії з використанням методів комп'ютерного зору.

Для досягнення поставленої мети необхідно вирішити такі задачі:

1) проаналізувати предметну область та існуючі рішення для промислового IoT-моніторингу;

2) розробити модуль збору та обробки даних із промислових датчиків через протокол TCP та інтерфейс RS485;
3) реалізувати алгоритм комп'ютерного зору для контролю розмірів продукції на основі аналізу зображень із камер;
4) розробити веб-інтерфейс для операторів із функціями налаштування, живого відеопотоку та системою сповіщень;
5) контейнеризувати всі компоненти системи за допомогою Docker та забезпечити їх взаємодію;
6) налаштувати систему логування метрик за допомогою Prometheus для подальшого аналізу.

Об'єктом дослідження є процес автоматизованого моніторингу виробничої лінії на підприємстві з виробництва пластикових виробів.

Предметом дослідження є методи та засоби розробки контейнеризованих IoT-платформ із застосуванням комп'ютерного зору для промислового моніторингу.

Практичне значення роботи полягає у розробці та впровадженні реальної системи моніторингу виробничих ліній, що забезпечує автоматичне виявлення відхилень у розмірах продукції та несправностей датчиків, і яка пройшла дослідну експлуатацію на діючому підприємстві.

Структура дипломної роботи. Робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Характеристика виробничого процесу та задачі автоматизації

Виробництво пластикових виробів є складним багатоетапним технологічним процесом, що включає підготовку сировини, лиття під тиском або екструзію, охолодження, обрізку та контроль якості готової продукції. Кожен із цих етапів вимагає дотримання чітких технологічних параметрів: температури розплаву, тиску у прес-формі, швидкості подачі матеріалу та геометричних характеристик кінцевого виробу.

В умовах сучасного виробництва підприємства прагнуть автоматизувати контроль якості — і для цього є вагомі причини. Людський фактор при візуальному контролі неминучий: монотонна робота знижує уважність, і дефекти пропускаються. До того ж ручна перевірка фізично не встигає за темпом швидкісних ліній — перевірити кожен виріб просто нереально. А затримка між виникненням дефекту та його виявленням означає, що до того моменту вже накопичилась партія браку.

Концепція Industry 4.0 передбачає інтеграцію виробничих процесів з інформаційними технологіями через так звані кіберфізичні системи (Cyber-Physical Systems, CPS) — сукупності фізичного обладнання та програмних компонентів, що взаємодіють у реальному часі. Ключовою складовою таких систем є Інтернет речей (IoT) — мережа фізичних пристроїв, оснащених датчиками, програмним забезпеченням та засобами підключення для збору та обміну даними. У виробничому контексті промисловий IoT (IIoT) дозволяє отримувати оперативну інформацію про стан обладнання та параметри продукції без участі людини, що є основою для побудови систем предиктивного технічного обслуговування та адаптивного управління якістю.

Традиційним підходом до автоматизації промислових підприємств є використання SCADA-систем (Supervisory Control and Data Acquisition) — програмних комплексів для диспетчерського контролю та збору даних. SCADA-

системи добре зарекомендували себе для управління технологічними процесами, однак вони орієнтовані переважно на роботу зі стандартизованими промисловими протоколами (Modbus, OPC-UA, PROFIBUS) та не мають вбудованих засобів комп'ютерного зору. Крім того, комерційні SCADA-системи є дорогими та вимагають спеціалізованих знань для налаштування та підтримки. Сучасні IoT-підходи на базі відкритого програмного забезпечення пропонують більш гнучку альтернативу, особливо для підприємств, де поєднуються класичні дискретні датчики з сучасними IP-камерами.

Об'єктом автоматизації в даній роботі є підприємство з виробництва пластикових виробів, що має шість виробничих ліній. На кожній активній лінії встановлено промислові камери для відеоконтролю та дискретні датчики на базі пристроїв вводу/виводу з інтерфейсом RS485. Камери розміщені у контрольних точках конвеєра і фіксують геометричні параметри продукції у процесі виробництва. Датчики відстежують ключові технологічні параметри: наявність продукту на конвеєрі, спрацювання кінцевих вимикачів, стан виконавчих механізмів тощо.

Кожен I/O-пристрій має вісім дискретних входів, кожен із яких відповідає за окремий параметр і повертає значення типу «істина/хибність». Такий підхід є типовим для промислової автоматики: дискретні сигнали надійні, прості в інтерпретації та стійкі до перешкод. Обробка цих сигналів полягає у порівнянні зчитаного значення з еталонним (очікуваним для нормального стану лінії) та присвоєнні відповідного статусу — OK або ERROR. Сукупність статусів усіх датчиків і камер формує загальну картину стану виробничої лінії в реальному часі.

Основними задачами автоматизації для даного підприємства є: безперервний збір даних із датчиків та камер у режимі реального часу; автоматичний аналіз геометричних параметрів продукції та виявлення відхилень від норми; своєчасне сповіщення оператора про виникнення аварійних ситуацій; зберігання та аналіз статистики роботи обладнання для прогнозування технічного обслуговування.

Специфіка промислового середовища висуває додаткові вимоги до програмного забезпечення: висока надійність та відмовостійкість, оскільки зупинка системи моніторингу може призвести до пропуску браку; здатність працювати в умовах нестабільного мережевого з'єднання з окремими пристроями; простота використання для технічного персоналу без спеціальної підготовки в IT-сфері; можливість налаштування під специфіку конкретної виробничої лінії.

Окремим аспектом, що враховується при проектуванні системи, є воєнний стан в Україні та пов'язані з ним ризики переривання електропостачання. Для забезпечення оперативного інформування персоналу про події на виробничій лінії незалежно від доступності інтернету система оснащена автономною апаратною сигналізацією — світловою мигалкою та звуковою сиреною, що активуються безпосередньо через I/O-пристрій при виникненні помилки. Додатково на сторінці веб-інтерфейсу інтегровано карту тривог України для інформування операторів про повітряні тривоги в регіоні.

1.2 Огляд існуючих IoT-платформ для промислового моніторингу

На ринку промислового IoT представлено широкий спектр платформ для моніторингу виробничих процесів. Розглянемо найбільш поширені рішення та проаналізуємо їх відповідність вимогам даного проєкту.

Siemens MindSphere є хмарною IoT-платформою промислового рівня, що забезпечує збір та аналіз даних із промислового обладнання. Платформа підтримує широкий спектр протоколів передачі даних, має потужні інструменти аналітики та машинного навчання. Однак її використання вимагає значних фінансових витрат на ліцензування, а хмарна архітектура передбачає обов'язкову передачу виробничих даних на сервери постачальника, що може бути неприйнятним з огляду на конфіденційність технологічних процесів підприємства.

Платформа PTC ThingWorx є однією з провідних промислових IoT-платформ, що надає інструменти для побудови застосунків для моніторингу та управління обладнанням. Вона підтримує широкий набір протоколів, включаючи OPC-UA, MQTT та REST API. Суттєвим недоліком є висока вартість ліцензії та значна складність налаштування і підтримки, що робить її недоцільною для малих і середніх підприємств.

Node-RED є відкритим інструментом для потокового програмування, що широко використовується в IoT-рішеннях. Він надає візуальне середовище для побудови обробних ланцюгів даних та підтримує численні протоколи. Проте Node-RED не має вбудованих засобів комп'ютерного зору, а реалізація складних алгоритмів аналізу зображень у потоковій парадигмі є нетривіальним завданням.

Відкрита платформа OpenHAB орієнтована переважно на розумний будинок, але може застосовуватись і у промислових середовищах. Вона підтримує сотні різних пристроїв та протоколів, має активну спільноту розробників. Однак платформа не має засобів комп'ютерного зору і не розрахована на обробку відеопотоків у реальному часі.

Grafana у поєднанні з Prometheus є популярним стеком для моніторингу та візуалізації метрик. Prometheus забезпечує збір та зберігання часових рядів даних, а Grafana — їх візуалізацію у вигляді дашбордів. Цей стек широко використовується для моніторингу IT-інфраструктури, але потребує додаткових компонентів для інтеграції з промисловим обладнанням та не має вбудованих засобів комп'ютерного зору.

Окрему категорію складають SCADA-системи промислового рівня, такі як Wonderware AVEVA, Ignition від Inductive Automation та ICONICS Genesis64. Ці платформи є потужними інструментами для диспетчерського керування та збору даних на великих підприємствах. Вони підтримують широкий спектр промислових протоколів, мають розвинені засоби побудови операторських панелей та засоби тривожної сигналізації. Проте їх вартість, складність розгортання та відсутність вбудованої підтримки комп'ютерного зору роблять їх непридатними для використання в умовах даного проєкту.

Жодна з розглянутих платформ не закриває всі потреби проєкту в готовому вигляді. Комерційні рішення коштують більше ніж виправдано для підприємства такого масштабу, а відкриті потребують суттєвого доопрацювання — особливо в частині промислових протоколів і обробки відео. Тому було прийнято рішення будувати власне рішення на базі відкритих бібліотек, підібраних під конкретні вимоги задачі.

1.3 Аналіз методів комп'ютерного зору для контролю якості продукції

Комп'ютерний зір (Computer Vision) є розділом штучного інтелекту, що займається отриманням, обробкою та аналізом зображень з метою отримання значущої інформації. У виробничих застосуваннях методи комп'ютерного зору використовуються для автоматичного контролю якості, вимірювання геометричних параметрів виробів, виявлення дефектів поверхні та ідентифікації об'єктів.

Традиційні методи машинного зору базуються на класичних алгоритмах обробки зображень, серед яких ключове місце займає виявлення країв (edge detection). Ця техніка дозволяє знаходити різкі зміни яскравості у зображенні, що відповідають межам об'єктів. Найпоширенішими алгоритмами виявлення країв є оператори Canny, Sobel та Prewitt.

Алгоритм Canny, розроблений Джоном Кенні у 1986 році, є одним із найбільш ефективних методів виявлення країв і широко використовується у задачах промислового контролю. Він складається з кількох етапів: згладжування зображення гаусівським фільтром для зменшення шуму, обчислення градієнту яскравості, придушення немаксимумів та гістерезисне порогування. Результатом є чіткі одно-піксельні лінії, що відповідають краям об'єктів.

Для вимірювання геометричних параметрів виробів у системах технічного зору застосовується підхід, що базується на визначенні положення країв об'єкта у зображенні та обчисленні відстаней між ними у пікселях із подальшим

перерахунком у фізичні одиниці на основі відомого масштабного коефіцієнта. Точність таких вимірювань залежить від якості оптики камери, рівномірності освітлення та стабільності положення продукту відносно камери.

Бінаризація зображення є важливим попереднім кроком у більшості алгоритмів машинного зору. Вона полягає у перетворенні зображення у відтінках сірого на двоколірне зображення шляхом порівняння яскравості кожного пікселя з пороговим значенням. Пікселі яскравіші за поріг позначаються як білі (об'єкт), темніші — як чорні (фон), або навпаки, залежно від умов освітлення. Адаптивна бінаризація, що використовує локальне порогове значення для різних ділянок зображення, забезпечує кращі результати при нерівномірному освітленні.

У даній роботі застосовується метод контролю, що базується на аналізі лінійного профілю зображення уздовж заданого вектора. Оператор визначає дві точки у просторі зображення, між якими проводиться лінія контролю. Після отримання кадру з камери він перетворюється у відтінки сірого, потім бінаризується. Вздовж лінії контролю визначається профіль яскравості, за яким встановлюється, чи виходить продукт за межі лінії, чи наближається до неї на критичну відстань, чи відсутній взагалі. Такий підхід є простим у реалізації, обчислювально ефективним і добре підходить для контролю продуктів з передбачуваною геометрією на конвеєрі.

Важливим аспектом систем машинного зору є спосіб отримання зображень із камер. У промислових застосуваннях широко використовуються IP-камери, що передають відеопотік за протоколом RTSP (Real Time Streaming Protocol). RTSP є прикладним протоколом для керування потоковими медіаданими, що працює поверх транспортних протоколів TCP або UDP. Він дозволяє клієнту підключитися до камери, вибрати потрібний потік та отримувати кадри у стиснутому форматі (зазвичай H.264). Бібліотека OpenCV надає зручний інтерфейс для роботи з RTSP-потоками через клас VideoCapture, що приймає RTSP-URL як аргумент. Для систем контролю якості, де аналізується не відеопотік цілком, а окремі кадри через певний інтервал, такий підхід є оптимальним з точки зору навантаження на процесор та мережу.

Конвеєр обробки зображення працює наступним чином. Спочатку на основі двох точок, заданих оператором, будується бінарна маска розміром з вихідний кадр, на якій відмічено лише ті пікселі, що лежать на відрізку між координатами (x1, y1) та (x2, y2). З вихідного кадру витягується одновимірний масив пікселів, що відповідають цій лінії — таким чином тривимірне зображення зводиться до послідовності кольорових значень уздовж контрольного вектора. Ця послідовність переводиться у відтінки сірого, після чого до неї застосовується інверсна бінаризація з пороговим значенням T: пікселі, темніші за поріг, позначаються як «1» (продукт), яскравіші — як «0» (фон). Таке інверсне порогування є доцільним, оскільки продукт на конвеєрі, як правило, темніший за освітлений фон.

Після бінаризації визначаються позиції пікселів продукту вздовж лінії та знаходяться крайні з них — min_pos та max_pos. Логіка присвоєння статусу базується на тому, де саме вздовж лінії розташовані ці крайні точки. Лінія умовно ділиться на три зони: дві крайові зони попередження завдовжки W% від загальної довжини лінії з кожного боку, та центральна зона між ними. Якщо продукт не виявлено взагалі — присвоюється статус ERROR. Якщо обидва кінці продукту (min_pos і max_pos) знаходяться в центральній зоні — статус OK. Якщо хоча б один кінець потрапляє у крайову зону попередження — статус WARNING. В усіх інших випадках, тобто коли продукт виходить за межі контрольної лінії — статус ERROR. (Схема алгоритму аналізу зображення з камери подана у Додатку А)

Альтернативою класичним методам є підходи на основі глибокого навчання (Deep Learning), зокрема згорткові нейронні мережі (CNN). Вони демонструють вищу точність при виявленні складних дефектів поверхні та класифікації об'єктів, однак вимагають значних обчислювальних ресурсів та великих обсягів навчальних даних. Для задачі лінійного вимірювання на конвеєрі класичні методи є більш доцільними завдяки їх простоті, швидкості та детермінованості результатів.

1.4 Огляд технологій контейнеризації та оркестрації

Контейнеризація є технологією ізоляції програмного забезпечення, при якій застосунок разом із усіма його залежностями упаковується у стандартизований контейнер. На відміну від традиційних віртуальних машин, контейнери використовують ядро операційної системи хоста і не вимагають окремої гостьової ОС, що забезпечує значно нижні накладні витрати та вищу щільність розгортання.

Docker є найбільш поширеною платформою контейнеризації. Він надає інструменти для збірки образів контейнерів (Dockerfile), їх зберігання у реєстрах образів та запуску у вигляді ізольованих контейнерів. Docker compose дозволяє декларативно описати багатоконтейнерні застосунки, визначивши конфігурацію кожного сервісу, мережі та томи у файлі docker-compose.yml. Це значно спрощує розгортання та управління складними системами.

Ключовими перевагами контейнеризації для промислових систем є: відтворюваність середовища виконання — контейнер поводиться однаково на будь-якому хості з Docker; ізоляція компонентів — збій в одному контейнері не впливає на роботу інших; спрощене оновлення — нову версію компонента можна розгорнути без зупинки всієї системи; легкість масштабування — за необхідності кількість екземплярів контейнера можна збільшити.

Для складніших сценаріїв із великою кількістю мікросервісів існують системи оркестрації контейнерів, серед яких найбільш відомим є Kubernetes. Він автоматично розподіляє контейнери по вузлах кластера, стежить за їх станом та перезапускає у разі відмови. Проте для систем невеликого масштабу, що розгортаються на одному хості, Docker compose є достатнім і значно простішим у використанні рішенням.

Міжконтейнерна комунікація у Docker може здійснюватись кількома способами: через внутрішню мережу Docker (за допомогою імен сервісів як DNS-назв), через відображені порти хоста, а також через спільні томи для обміну

файлами. У даному проєкті два контейнери — основний застосунок та сервіс опитування датчиків — обмінюються даними через внутрішню мережу Docker, що забезпечує ізоляцію від зовнішніх мережевих загроз.

1.5 Обґрунтування вибору технологій та інструментів

На основі проведеного аналізу предметної області та існуючих рішень сформовано набір технологій та бібліотек для розробки IoT-платформи. Вибір кожної технології обґрунтований нижче.

Python обрано основною мовою розробки з кількох практичних міркувань. Головне — екосистема: OpenCV для роботи з зображеннями, socket для мережевого рівня, Flask для веб-частини, prometheus-client для метрик — все це є і добре документовано. Динамічна типізація та інтерпретованість прискорюють розробку і полегшують налагодження, що було важливо в умовах обмеженого часу. Нарешті, Python широко відомий у технічних колах, тобто підтримувати такий код зможе не лише той хто його написав.

OpenCV — бібліотека комп'ютерного зору з відкритим кодом, яка охоплює більшість потреб при роботі із зображеннями: від базової обробки до складніших алгоритмів. Для цього проєкту важливою була підтримка RTSP — стандартного протоколу IP-камер — через клас VideoCapture. Бінаризація, обробка пікселів, робота з масивами зображень — все це є в OpenCV «з коробки» і не потребує написання з нуля.

Flask обраний тому, що для цієї задачі важлива гнучкість, а не масштаб. Django чи FastAPI принесли б зайву складність: тут не потрібна ORM, складна авторизація чи розгалужена маршрутизація. Flask дозволяє зробити MJPEG-стрим, REST-ендпоінти та сторінки налаштувань без зайвого «скелету» навколо — і саме це було потрібно.

Стандартна бібліотека socket Python використовується для реалізації TCP-клієнта для зв'язку з I/O-пристроями RS485. Власна реалізація TCP-клієнта на

низькому рівні надає повний контроль над форматом пакетів, таймаутами та обробкою помилок з'єднання, що є критично важливим для надійної роботи з промисловими протоколами.

Prometheus є відкритою системою моніторингу та сповіщень, розробленою у SoundCloud та зараз підтримуваною Cloud Native Computing Foundation. Вона використовує модель витягування (pull model), при якій сервер Prometheus сам звертається до застосунків для збору метрик через HTTP-ендпоінти. Бібліотека prometheus-client для Python дозволяє легко додавати метрики будь-яких типів (лічильники, gauge, гістограми) до застосунку. Prometheus ідеально підходить для збору часових рядів виробничих даних з подальшою візуалізацією в Grafana.

Docker та Docker Compose обрано для контейнеризації системи, що забезпечує ізоляцію компонентів та спрощує розгортання. Розбиття на два контейнери — основний застосунок (main_program) та сервіс опитування датчиків (tcps) — дозволяє незалежно масштабувати та оновлювати кожен компонент. Така архітектура також підвищує надійність: при проблемах зі зв'язком з датчиками основна програма продовжує працювати.

Важливою перевагою Docker є механізм перевірки стану контейнерів (health check), що дозволяє автоматично перезапускати контейнер у разі його відмови. Директива restart: always у конфігурації Docker Compose забезпечує автоматичний запуск контейнерів після перезавантаження хост-системи, що є критично важливим для виробничих середовищ, де система повинна відновлювати роботу без ручного втручання після планових або позапланових перезавантажень.

Мережева ізоляція Docker є ще однією перевагою для промислових систем. За замовчуванням контейнери в одній мережі Docker Compose можуть спілкуватися між собою за іменами сервісів, тоді як зовнішній доступ дозволено лише через явно відображені порти. Це формує природний мережевий периметр безпеки: сервіс опитування датчиків tcps взаємодіє з основним застосунком виключно через внутрішню мережу Docker, не будучи доступним ззовні. Веб-

інтерфейс Flask, навпаки, відображає свій порт назовні для доступу операторів з локальної мережі підприємства.

Багатопотоковість (Python threading) забезпечує паралельну обробку потоків від кожної камери та кожного датчика. Це є критично важливим для систем реального часу: якщо один з пристроїв перестає відповідати, відповідний потік блокується у стані очікування, тоді як всі інші продовжують нормальну роботу. Використання окремих потоків для кожного пристрою також дозволяє масштабувати систему шляхом простого додавання нових потоків при підключенні нових камер або датчиків.

Слід зазначити, що Python має обмеження у вигляді глобального блокування інтерпретатора (GIL — Global Interpreter Lock), що не дозволяє двом потокам Python виконувати байт-код одночасно в рамках одного процесу. Однак для задач введення/виведення (I/O-bound tasks), таких як очікування відповіді від мережевого пристрою або отримання кадру з камери, GIL знімається на час операції, що дозволяє потокам ефективно працювати паралельно. Оскільки основна частина роботи кожного потоку полягає саме в очікуванні мережевих відповідей, архітектура на основі потоків є обґрунтованим вибором для даної задачі. Альтернативою могло б бути використання asyncio, однак бібліотеки OpenCV та socket мають синхронний API, що ускладнило б їх інтеграцію з асинхронною парадигмою.

1.6 Висновки до розділу 1

Аналіз, проведений у першому розділі, показав: готового рішення яке б закривало всі вимоги проєкту — немає. Комерційні IoT-платформи орієнтовані на великі підприємства з відповідними бюджетами, а відкриті інструменти або не мають підтримки промислових протоколів, або не вміють працювати з відеопотоком.

Проаналізовано методи комп'ютерного зору для контролю якості продукції. Показано, що для задачі лінійного вимірювання геометричних параметрів продукції на конвеєрі класичні методи на основі бінаризації та аналізу профілю яскравості є більш доцільними порівняно з підходами глибокого навчання, оскільки вони є обчислювально ефективними, детермінованими та не вимагають великих обсягів навчальних даних.

Розглянуто технологію контейнеризації на базі Docker та обґрунтовано її застосування для ізоляції компонентів системи. Розбиття на два незалежних контейнери — для основного застосунку та для сервісу опитування датчиків — підвищує надійність системи та спрощує її розгортання та обслуговування.

На основі проведеного аналізу обґрунтовано вибір технологічного стеку: Python як основна мова програмування, OpenCV для обробки зображень, Flask для веб-інтерфейсу, бібліотека socket для TCP-комунікації з датчиками, Prometheus для збору метрик, Docker та Docker Compose для контейнеризації. Кожен з обраних інструментів вирішує конкретну задачу і добре поєднується з рештою — саме це і було критерієм вибору.

На основі цього аналізу у наступному розділі сформульовано архітектурні рішення системи. Окремо встановлено, що специфіка умов експлуатації — виробниче середовище, необхідність автономної сигналізації та інтеграції з картою тривог — обумовлює ряд додаткових проектних рішень, які не є характерними для типових IoT-платформ і реалізовані в рамках даної роботи як оригінальні компоненти системи.

2. ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Загальна архітектура системи

Розроблена IoT-платформа побудована за мікросервісною архітектурою та розгортається за допомогою Docker Compose. Файл docker-compose.yml описує чотири контейнери, що утворюють повний стек системи: main_program, tcps, prometheus та grafana. Такий підхід забезпечує ізоляцію кожного компонента, незалежне управління його життєвим циклом та можливість перезапуску окремого сервісу без зупинки решти.

Перший контейнер — main_program — є центральним компонентом системи. Він відповідає за захоплення та аналіз зображень із IP-камер, агрегацію статусів усіх пристроїв, веб-інтерфейс для операторів, ініціювання сигналізації та публікацію метрик для Prometheus. Другий контейнер — tcps — є спеціалізованим сервісом для роботи з промисловими I/O-пристроями через протокол Modbus TCP. Він надає HTTP API, через який main_program отримує статуси датчиків та відправляє команди на фізичну сигналізацію.

Третій і четвертий контейнери — prometheus та grafana — утворюють підсистему моніторингу. Контейнер Prometheus налаштований на збір метрик із main_program через ендпоінт /metrics з визначеним інтервалом опитування, а також на зберігання зібраних часових рядів протягом заданого проміжку часу. Контейнер Grafana підключений до Prometheus як джерела даних. Оскільки уся система розгортається в локальній мережі підприємства та доступ до Grafana мав виключно замовник, авторизацію в Grafana було вимкнено для спрощення доступу — питання мережевої безпеки в умовах закритої локальної мережі не є критичним. Налаштування дашбордів у Grafana виконував замовник відповідно до власних потреб аналізу виробничих даних.

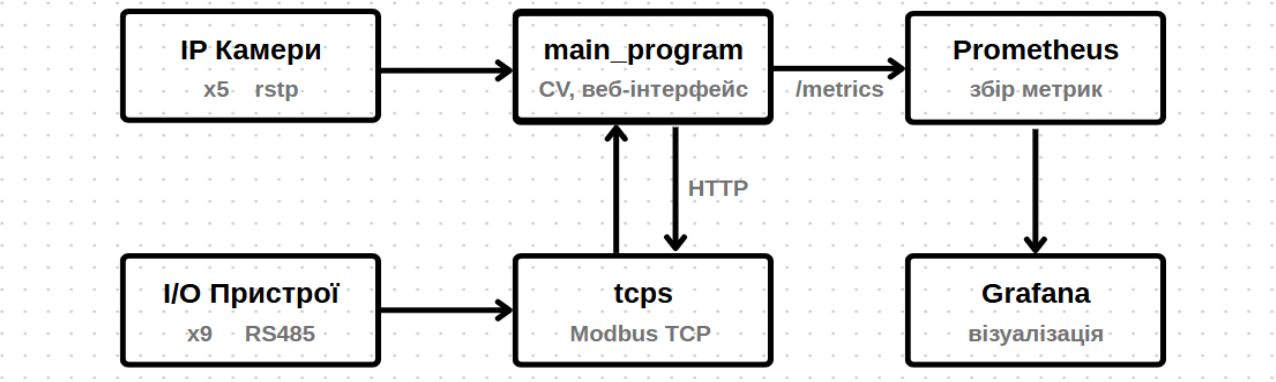


Рисунок 2.1 — Структурна схема архітектури системи

Потоки даних у системі організовані наступним чином. IP-камери передають відеопотік за протоколом RTSP безпосередньо до контейнера main_program, де кожна камера обробляється в окремому потоці. I/O-пристрої з датчиками підключені до мережі підприємства і спілкуються з контейнером tcps за протоколом Modbus TCP. Контейнер tcps циклічно опитує кожен пристрій у своєму потоці та накопичує поточні статуси у пам'яті. Контейнер main_program звертається до tcps через HTTP для отримання актуальних даних з датчиків, а також для надсилання команд активації сигналізації при виникненні помилок. Prometheus зі своєї сторони звертається до main_program для збору метрик, а Grafana зчитує дані з Prometheus для відображення на дашбордах.

Ключовим архітектурним рішенням є ізоляція модуля роботи з датчиками в окремий контейнер tcps. Це рішення обумовлено декількома міркуваннями. По-перше, Modbus TCP потребує постійного підтримання з'єднань з кожним I/O-пристроєм, і будь-який збій у цій логіці не повинен впливати на роботу основного застосунку та веб-інтерфейсу. По-друге, сервіс tcps може бути перезапущений незалежно при оновленні або відновленні після помилки. По-третє, така архітектура дозволяє в майбутньому масштабувати сервіс опитування датчиків, запускаючи кілька екземплярів tcps для різних груп пристроїв.

Автоматичний перезапуск усіх контейнерів при збоях та після перезавантаження хост-системи забезпечується директивою restart: always у конфігурації Docker Compose. Це гарантує, що система відновлює роботу без

ручного втручання після планових або позапланових перезавантажень промислового ПК. Детальний опис структури конфігураційних файлів системи наведено у підрозділі 2.5.

2.2 Проектування модуля збору даних з датчиків

Модуль збору даних із датчиків реалізований у вигляді окремого контейнера tcps і складається з двох логічних частин: підсистеми опитування I/O-пристроїв через Modbus TCP та HTTP API для взаємодії з основним застосунком.

На фізичному рівні I/O-пристрої підключені до мережі підприємства через інтерфейс RS485, перетворений у Ethernet за допомогою відповідних конверторів. Кожен пристрій має вісім дискретних входів, кожен із яких підключений до певного датчика або кінцевого вимикача на виробничій лінії. Дискретний вхід повертає лише два стани: «активний» (True) або «неактивний» (False), що є типовим для промислової автоматики і забезпечує стійкість сигналу до електромагнітних завад.

Для комунікації з I/O-пристроями використовується протокол Modbus RTU поверх TCP-з'єднання — підхід, при якому RTU-фрейм передається безпосередньо через TCP-сокет без додаткового обгортання в MBAP-заголовок. Це можливо завдяки тому, що TCP-шлюз (конвертер RS485/Ethernet) налаштований у прозорому режимі передачі: він пересилає RTU-фрейм до фізичного пристрою як є і повертає відповідь у тому самому форматі. Для читання стану дискретних входів використовується функціональний код 0x02 (Read Discrete Inputs), що дозволяє зчитати стан восьми входів одним запитом.

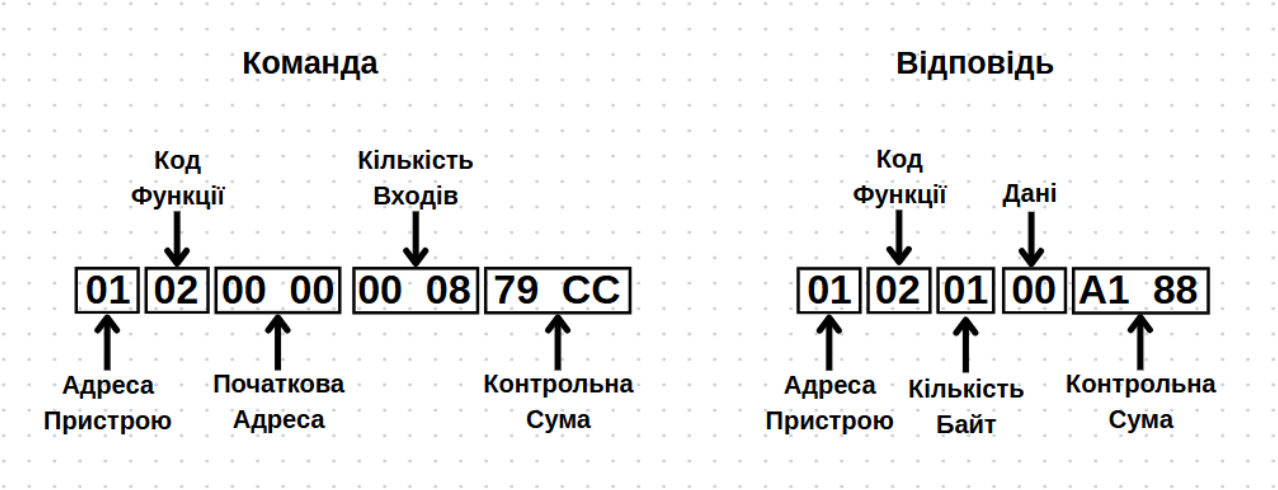


Рисунок 2.2 — Структура Modbus RTU пакету

Архітектура сервісу tcps побудована на моделі окремого потоку для кожного I/O-пристрою. При старті сервіс зчитує конфігурацію з JSON-файлу та записує її в спільну змінну в пам'яті. Для кожного налаштованого пристрою створюється окремий потік виконання, який звертається до цієї змінної на кожній ітерації циклу опитування. Таке рішення дозволяє динамічно підхоплювати зміни в конфігурації: якщо оператор змінює параметри через веб-інтерфейс і файл конфігурації оновлюється, спільна змінна також оновлюється в той самий момент, і вже при наступній ітерації потоки використовують нові параметри без жодного перезапуску.

Кожен потік підтримує TCP-з'єднання зі своїм I/O-пристроєм і циклічно, з заданим інтервалом, відправляє Modbus-запит на читання всіх восьми дискретних входів. Отримана відповідь розбирається: значення кожного входу порівнюється з полем Normal_state із конфігурації. Якщо зчитане значення збігається з еталонним — вхід отримує статус OK. Якщо ні — перевіряється поле monitored: якщо воно False, відхилення ігнорується (дані збираються, але помилка не генерується); якщо True, перевіряється поле critical, яке визначає тип помилки — ERROR або WARNING.

Обробка нештатних ситуацій є критично важливою для промислового застосування і реалізована на двох рівнях. Якщо втрачається зв'язок з конкретним I/O-пристроєм, усі його вісім входів переводяться у стан False, а

пристрою присвоюється статус CONNECTION_ERROR — оператор бачить не конкретну помилку датчика, а сигнал про проблему зв'язку з пристроєм. Якщо ж втрачається з'єднання на рівні TCP-шлюзу, усі I/O-пристрої, що підключені через нього, отримують статус CONNECTION_ERROR, а сам TCP-шлюз також позначається як недоступний. При цьому потоки інших незалежних пристроїв продовжують нормальну роботу — відмова одного сегменту мережі не паралізує всю систему моніторингу.

HTTP API сервісу tcps реалізований на базі Flask і надає два типи ендпоінтів. Перший тип — ендпоінти для читання поточних статусів датчиків: main_program звертається до них із заданою в конфігурації Times periodicity для отримання актуальних даних, які далі відображаються на дашборді та передаються в Prometheus. Другий тип — ендпоінти для керування фізичними виходами I/O-пристрою, через які main_program надсилає команди активації мигалки або сирени. Такий поділ відповідальності дозволяє main_program не знати деталей Modbus-протоколу — вся низькорівнева логіка інкапсульована в tcps, а взаємодія між контейнерами залишається простою та зрозумілою.

2.3 Проектування модуля комп'ютерного зору

Модуль комп'ютерного зору є частиною контейнера main_program і відповідає за підключення до IP-камер, захоплення та аналіз зображень, визначення статусу кожної камери та збереження результатів для веб-інтерфейсу і системи моніторингу. Як і модуль датчиків, він побудований на моделі окремого потоку для кожного пристрою з динамічним оновленням конфігурації.

При старті контейнера main_program конфігурація зчитується з JSON-файлу і записується в спільну змінну в пам'яті. Для кожної налаштованої камери створюється окремий потік, що на кожній ітерації звертається до цієї змінної за актуальними параметрами. Завдяки цьому зміни, внесені оператором через веб-інтерфейс, набирають чинності при наступній ітерації потоку без його

перезапуску. Потік підключається до камери за RTSP URL за допомогою класу VideoCapture бібліотеки OpenCV, захоплює черговий кадр та виконує його попередню обробку.

Першим кроком обробки є обрізання кадру до заданої оператором прямокутної області інтересу (ROI — Region of Interest). Це особливо важливо для камер із роздільною здатністю 4K: аналіз і передача повного кадру були б обчислювально надлишковими, а потрібна зона огляду займає лише частину зображення. Обрізаний кадр зберігається в спільній структурі даних у пам'яті — словнику camera_frames, де ключем є ім'я камери. Саме цей збережений кадр є вхідними даними як для алгоритму аналізу, так і для формування зображень для веб-інтерфейсу.

Після збереження кадру, якщо камера увімкнена (параметр ON у конфігурації), запускається функція аналізу analyze_frame, що повертає поточний статус. Алгоритм аналізу описаний у підрозділі 1.3 та у Додатку А. Результат — один із статусів OK, WARNING або ERROR — зберігається в пам'яті разом із останнім кадром. При втраті з'єднання з камерою потік намагається відновити підключення, а камері присвоюється статус OFF до моменту відновлення зв'язку.

Окремою функцією є get_frame, яка викликається виключно для потреб веб-інтерфейсу і формує зображення з нанесеною на нього візуальною розміткою контрольної зони. Функція приймає параметр threshold, що визначає режим відображення. Незалежно від режиму, функція попередньо обчислює позиції чотирьох ключових точок на контрольній лінії: дві крайні точки (p1 та p2), задані оператором, та дві внутрішні точки (p5 та p95), що позначають межі зони попередження й розраховуються як зміщення від країв на відстань Yellow% від загальної довжини лінії.

При threshold=True функція перетворює збережений кадр у відтінки сірого, застосовує бінаризацію з порогом T та повертає бінарне зображення з нанесеними жовтими точками в позиціях p5 та p95. Цей режим призначений для налаштування оператором: він дозволяє наочно бачити результат бінаризації з

поточним порогом і зрозуміти, як саме алгоритм «бачить» продукт, до того як зберегти параметри. При threshold=False до кадру спочатку застосовується регулювання контрасту (параметр Contrast), після чого на зображення наносяться лінії: жовті відрізки від p1 до p5 і від p95 до p2 позначають зони попередження, а зелений відрізок між p5 та p95 — центральну зону OK. Цей режим є стандартним для відображення на дашборді та сторінці живого відеопотоку.

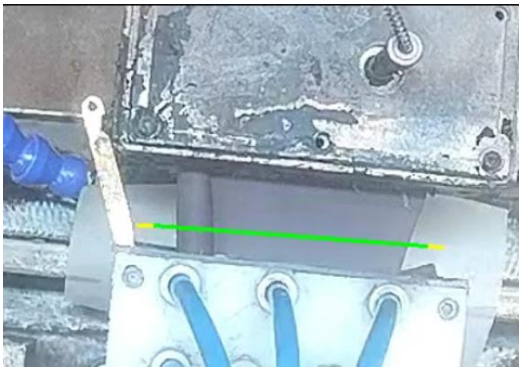


Рисунок 2.3 — Приклад відображення контрольної розмітки на кадрі.

Таке розділення двох режимів відображення має важливе практичне значення. Оператор, налаштовуючи камеру на новій лінії або при зміні умов освітлення, може перемкнутися в режим threshold=True і безпосередньо спостерігати, як змінюється бінаризоване зображення при коригуванні порогу T, не вдаючись до технічних знань про роботу алгоритму.

2.4 Проектування веб-інтерфейсу та системи сповіщень

Веб-інтерфейс системи реалізований на базі мікрофреймворку Flask і є частиною контейнера main_program. Він орієнтований на використання нетехнічним персоналом — операторами виробничих ліній — і складається з трьох основних сторінок: дашборду зі статусами, сторінки живого відеопотоку та сторінки налаштувань.

Головна сторінка — дашборд — відображає поточний стан усіх виробничих ліній у зведеному вигляді. Кадри з камер на дашборді оновлюються

п'ять разів на секунду. Важливо зазначити, що передаються саме окремі кадри, а не відеопотік: браузер із заданим інтервалом запитує новий кадр, який сервер формує з функції get_frame у режимі threshold=False і повертає як base64-зображення. Такий підхід є менш ресурсомістким порівняно з повноцінним відеопотоком і добре підходить для відображення одразу кількох камер на одній сторінці. Інтервал оновлення кадрів налаштовується в секції Times конфігураційного файлу.

На дашборді відображається чотири слоти для камер, визначених у секції Main_cameras конфігурації. Четвертий слот зарезервований під карту повітряних тривог України — важливий елемент в умовах воєнного стану, що дозволяє операторам стежити за ситуацією в регіоні безпосередньо з робочого місця. Для кожної виробничої лінії відображаються агреговані статуси камер і датчиків із відповідним кольоровим маркуванням.

Додатковим елементом керування на дашборді є кнопка Blue Button. Вона дозволяє оператору тимчасово зупинити активну сигналізацію на час усунення виявленої помилки. Після натискання звукова та світлова сигналізація вимикається, проте статус помилки на екрані залишається видимим до фактичного усунення проблеми. Це рішення дозволяє уникнути ситуації, коли тривала сирена заважає роботі персоналу, що вже знає про помилку і активно її усуває.

Сторінка живого відеопотоку дозволяє оператору обрати будь-яку доступну камеру зі списку та переглянути її поточне зображення. Відеопотік передається за технологією MJPEG (Motion JPEG) — послідовністю JPEG-кадрів, що надсилаються у відповідь на один HTTP-запит із Content-Type: multipart/x-mixed-replace. Це широко підтримуваний браузерами підхід для потокового відео без додаткових плагінів або WebSocket-з'єднань. На відміну від дашборду, де кадри передаються через окремі запити, MJPEG-потік є безперервним з'єднанням, що забезпечує плавніше відображення. На зображення в режимі реального часу накладається розмітка контрольної зони у режимі

threshold=False, що дозволяє оператору оцінювати коректність налаштувань лінії прямо в процесі роботи.

Сторінка налаштувань складається з двох окремих секцій: налаштувань камер та налаштувань датчиків. Вони надають оператору повний контроль над усіма параметрами системи без необхідності редагувати конфігураційні файли вручну.

Секція налаштувань датчиків відображає список усіх I/O-пристроїв та їхніх дискретних входів у вигляді зручної таблиці. Для кожного з восьми входів кожного пристрою оператор може індивідуально керувати трьома параметрами. По-перше, перемикач активності входу — вмикає або вимикає конкретний вхід, що дозволяє тимчасово ігнорувати несправний або не підключений датчик без зміни логіки роботи решти системи. По-друге, перемикач monitored визначає, чи слід генерувати помилку при відхиленні значення цього входу від норми: вимкнений monitored означає, що дані з датчика збираються та логуються в Prometheus, але оператор не отримує сповіщення — це корисно для некритичних параметрів, щоб не перевантажувати операторів зайвими сигналами. По-третє, перемикач critical визначає тип помилки при відхиленні: увімкнений critical генерує статус ERROR із відповідною реакцією сигналізації, вимкнений — лише WARNING зі світловою мигалкою.

Секція налаштувань камер організована у вигляді форм для кожної лінії та кожної камери. Для кожної камери доступні такі параметри: координати двох точок контрольної лінії (x1, y1, x2, y2), що визначають вектор контролю на зображенні; порогове значення яскравості для бінаризації T; розмір зони попередження Yellow%; параметри обрізання кадру (координати ROI); рівень контрасту для відображення на дашборді; перемикачі реакції на статуси WARNING та ERROR окремо — дозволяють для конкретних ліній вимикати певний тип сповіщень, якщо він є штатним для цієї лінії; перемикач активності камери в цілому. Також на сторінці доступний вибір камери для відображення на сторінці живого відеопотоку.

Усі зміни з обох секцій після підтвердження оператором записуються у відповідний JSON-файл конфігурації та набирають чинності при наступній ітерації відповідного потоку без перезапуску системи.

Рисунок 2.4 — Сторінка налаштувань камери

Система фізичної сигналізації є важливим компонентом безпеки виробничого процесу. При виникненні помилки на будь-якому з датчиків або камер main_program аналізує тип помилки та формує відповідну команду для tcps. При статусі WARNING активується лише світлова мигалка, що привертає увагу оператора без переривання виробничого процесу. При статусі ERROR активуються одночасно мигалка і звукова сирена на три секунди, після чого сирена вимикається автоматично, тоді як мигалка продовжує роботу до усунення помилки або натискання Blue Button. Такий градуйований підхід дозволяє операторові відрізняти попереджувальні стани від аварійних без зайвого звукового навантаження на виробничому майданчику.

Команда на активацію сигналізації відправляється від main_program до tcps у вигляді HTTP POST-запиту на відповідний ендпоінт. Сервіс tcps, отримавши команду, формує Modbus-запит на запис значення до дискретного виходу I/O-пристрою, до якого підключена мигалка або сирена. Таким чином, main_program залишається повністю абстрагованим від деталей Modbus-протоколу: для нього активація сигналізації є простим HTTP-запитом, а вся низькорівнева взаємодія з фізичним обладнанням інкапсульована в tcps.

2.5 Проектування підсистеми моніторингу та конфігурації

Підсистема конфігурації є фундаментом усієї системи, оскільки від неї залежить поведінка кожного компонента. Вона базується на двох окремих JSON-файлах відповідно до архітектурного принципу розділення відповідальності: кожен контейнер має власний файл конфігурації, що містить лише ті параметри, які необхідні саме йому. JSON-формат обраний через його простоту, читабельність для людини та нативну підтримку в Python через стандартний модуль json. Обидва файли монтуються в контейнери через механізм Docker volumes, що забезпечує персистентність конфігурації між перезапусками: навіть після збою контейнер завантажить актуальні параметри з диску хост-системи.

При старті кожного контейнера відповідний конфігураційний файл зчитується і завантажується у спільну змінну в пам'яті, що є єдиним джерелом параметрів для всіх потоків усередині контейнера. Якщо оператор вносить зміни через веб-інтерфейс, JSON-файл записується на диск, а спільна змінна оновлюється одночасно — нові параметри підхоплюються всіма потоками при їхній наступній ітерації без перезапуску. Це забезпечує «живе» оновлення налаштувань у виробничих умовах, де зупинка моніторингу навіть на короткий час є небажаною.

Конфігураційний файл контейнера main_program має таку ієрархічну структуру. На верхньому рівні знаходиться секція Main_settings, що містить

глобальний перемикач ON — він переводить всю систему між режимом активного аналізу та режимом пасивного спостереження (коли камери транслюють зображення, але аналіз не виконується). Підсекція Times містить три часових параметри: інтервал між оновленнями кадрів на головному дашборді, інтервал зчитування нових кадрів із камер та інтервал опитування датчиків. Ці значення дозволяють балансувати між актуальністю даних та навантаженням на мережу і процесор залежно від специфіки виробництва.

Секція Main_cameras описує чотири слоти для камер, що виводяться на головному дашборді. Четвертий слот зарезервований під карту повітряних тривог України — рішення, прийняте в умовах воєнного стану для забезпечення своєчасного інформування операторів без необхідності відволікатися від робочого місця. Секція Lines є масивом об'єктів, кожен із яких описує одну виробничу лінію. Кожна лінія містить підсекцію Cameras з детальними параметрами кожної камери (координати контрольної лінії, поріг бінаризації T, Yellow%, параметри ROI, контраст, прапорці ON та реакцій на WARNING і ERROR) та підсекцію Sensors з описом датчиків лінії у форматі, зручному для управління через веб-інтерфейс. Дані з підсекції Sensors передаються до контейнера tcps для зіставлення з низькорівневими Modbus-параметрами I/O-пристроїв.

Таблиця 2.1

Структура конфігураційного файлу main_program

Поле	Тип	Опис
Settings		
ON	bool	Активний аналіз або режим пасивного спостереження
Air_Alert	bool	Відображення карти тривог на дашборді
Update_times.*	float	Інтервали оновлення: дашборд, кадри камер, сторінка налаштувань, опитування датчиків (сек)
Main_cameras.{1–4}.Name	string	Камера для відповідного слоту дашборду; слот 4 — карта тривог
Cameras[]		
Name, Line_name	string	Назва камери та лінія до якої вона належить

Поле	Тип	Опис
Lines[]		
Name, ON	string / bool	Назва виробничої лінії та її активність
Lines[] → Cameras[] → Parameters		
Link	string	RTSP URL камери
X_1,Y_1,X_2,Y_2	int	Координати двох точок контрольної лінії
Top/Bottom/Left/Right_crop	int	Межі обрізання кадру (ROI)
Yellow%	float	Розмір зони попередження (% від довжини лінії)
Threshold	int	Поріг бінаризації T
Contrast	float	Коефіцієнт контрасту для відображення
Warning_reaction, Error_reaction	bool	Активація сигналізації при відповідному статусі
Lines[] → Sensors[]		
Name	string	Назва датчика (збігається з полем Name у конфігурації tcps)
Controller, Slave_ID, Address	int	Ідентифікатор контролера, Slave ID та адреса входу для зіставлення з tcps
ON	bool	Чи активовувати відповідну помилку про помилці

Конфігураційний файл контейнера tcps описує підключення до фізичних пристроїв на рівні протоколу Modbus і не містить жодних параметрів, специфічних для логіки застосунку — виключно мережеві адреси та ідентифікатори. Секція TCP містить IP-адресу та порт TCP-шлюзу RS485. Підсекція Slaves описує перелік I/O-пристроїв, підключених через цей шлюз, із їхніми Modbus Slave ID та ідентифікаторами контролерів (CONTROLLER_ID), що використовуються для зіставлення з конфігурацією main_program. Для кожного пристрою секція Parameters описує всі вісім дискретних входів. Кожен вхід має такі поля: Name — людська назва параметра для відображення в інтерфейсі; CONTROLLER_ID та SLAVE_ID — ідентифікатори для зіставлення з відповідним записом у конфігурації main_program; адреса I/O входу на пристрої за специфікацією Modbus; Normal_state — очікуваний стан входу в нормі (True або False), з яким порівнюється зчитане значення; monitored — прапорець

генерації помилок (якщо False, відхилення фіксується в метриках, але оператор не сповіщується); critical — тип помилки при відхиленні (True відповідає ERROR, False — WARNING).

Таблиця 2.2.

Структура конфігураційного файлу tcps

Поле	Тип	Опис
Контролер (кореневий масив)		
IDX, Name	int / string	Ідентифікатор та назва контролера; IDX збігається з полем Controller у main_program
IP, Port	string / int	IP-адреса та порт TCP-шлюзу RS485
ON	bool	Активність контролера
Slaves[]		
SlaveID, Name	int / string	Modbus Slave ID та назва I/O-пристрою на шині RS485
ON	bool	Активність пристрою
Slaves[] → Parameters[] (8 записів на кожен пристрій)		
Name	string	Назва параметра; збігається з полем Name у Sensors конфігурації main_program
Controller_IDX, Slave_ID, Address	int	Ідентифікатори для зіставлення з main_program та адреса входу (1–8)
Normal_state	bool	Очікуваний стан входу в нормі; зчитане значення порівнюється з цим полем
Monitored	bool	true → генерується помилка при відхиленні; false → дані збираються без сповіщення
Critical	bool	true → ERROR; false → WARNING при відхиленні від Normal_state

Підсистема моніторингу побудована на основі Prometheus — відкритої системи збору та зберігання часових рядів метрик. У контейнері main_program за допомогою бібліотеки prometheus-client оголошуються метрики типу Gauge для кожного пристрою: числовий статус кожної камери (0 — OFF, 1 — ON, 2 — OK, 3 — WARNING, 4 — ERROR) та бінарний статус кожного дискретного входу датчика (0 — OK, 1 — ERROR або CONNECTION_ERROR). Flask надає

ендпоінт /metrics, до якого контейнер Prometheus звертається з налаштованим інтервалом scraping.

Контейнер Prometheus налаштований з двома змінами відносно стандартної конфігурації: збільшений термін зберігання зібраних даних відповідно до потреб замовника (керується прапорцем --storage.tsdb.retention.time; за замовчуванням Prometheus зберігає дані 15 днів) та визначений інтервал опитування ендпоінту /metrics. Контейнер Grafana підключений до Prometheus як джерело даних через стандартний HTTP-інтерфейс. Авторизацію в Grafana вимкнено, оскільки система розгорнута в закритій локальній мережі підприємства і доступ до аналітики мав виключно замовник, який самостійно налаштував дашборди відповідно до власних потреб аналізу виробничих даних.

2.6 Висновки до розділу 2

У другому розділі виконано проектування всіх компонентів IoT-платформи для моніторингу виробничих ліній. Визначено загальну чотириконтейнерну архітектуру системи на базі Docker Compose: main_program, tcps, prometheus та grafana. Встановлено чіткий розподіл відповідальності між компонентами — контейнер main_program відповідає за обробку зображень, веб-інтерфейс та агрегацію даних, контейнер tcps інкапсулює всю логіку взаємодії з промисловим обладнанням через Modbus TCP, а стек Prometheus і Grafana забезпечує накопичення та візуалізацію метрик.

Детально спроектовано підсистему конфігурації на базі двох JSON-файлів із повним описом їхньої ієрархічної структури: файл main_program (Main_settings з Times, Main_cameras, Lines із підсекціями Cameras та Sensors) та файл tcps (TCP, Slaves, Parameters із полями Normal_state, monitored, critical). Визначено механізм динамічного оновлення конфігурації через спільну змінну в пам'яті без перезапуску потоків. Спроектовано підсистему моніторингу на базі Prometheus та Grafana із відображенням числових статусів камер і датчиків як метрик типу Gauge.

Спроектовано модуль збору даних із датчиків з підтримкою паралельного опитування через Modbus TCP, гранульованою логікою статусів (monitored, critical) та дворівневою обробкою мережевих відмов. Розроблено архітектуру модуля комп'ютерного зору із двома режимами відображення get_frame (threshold та нормальний) для зручного налаштування операторами.

Спроектовано веб-інтерфейс із дашбордом реального часу (оновлення кадрів 5 разів на секунду), MJPEG-потоком для перегляду окремих камер, сторінкою налаштувань камер та окремою сторінкою налаштувань датчиків із індивідуальним керуванням кожним із восьми входів кожного I/O-пристрою (активність, monitored, critical), а також Blue Button для керування сигналізацією. Визначено градуйовану систему фізичної сигналізації: WARNING активує мигалку, ERROR активує мигалку та сирену на 3 секунди.

Загалом архітектура вийшла достатньо гнучкою: більшість параметрів змінюється без перезапуску, а відмова одного компонента не тягне за собою решту. Третій розділ описує безпосередню реалізацію.

3. РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Структура проекту та організація коду

Проект організований у вигляді двох незалежних Python-застосунків — main_program та tcps — кожен із яких має власний Dockerfile і розміщений у відповідній директорії репозиторію. Обидва застосунки описані як сервіси в єдиному файлі docker-compose.yml разом із сервісами prometheus та grafana, що забезпечує спільне розгортання та управління всім стеком однією командою.

Усередині кожного застосунку дотримується принцип єдиної відповідальності: файл main.py є виключно точкою входу й не містить бізнес-логіки. В застосунку main_program ініціалізація відбувається через імпорт функції initialize_camera_threads з модуля processing. Цей імпорт ініціалізує модуль processing, який у свою чергу ініціалізує модуль веб-застосунку (app) та всі залежні модулі, що й запускає весь процес. Такий підхід дозволяє чітко розмежувати точку входу та логіку ініціалізації, залишаючи main.py мінімальним.

Застосунок main_program складається з кількох модулів. Модуль camera інкапсулює всю логіку, пов'язану з відеокамерами: підключення до RTSP-потоків, захоплення та попередню обробку кадрів (crop, контраст), функцію analyze_frame для аналізу зображення та функцію get_frame для формування розміченого зображення з нанесеною контрольною зоною. Модуль processing є центральним координуючим модулем: він відповідає за комунікацію з веб-інтерфейсом, агрегацію та аналіз помилок з усіх джерел, інтеграцію карти тривог та взаємодію з модулем camera.

Модуль app реалізує всі Flask-маршрути: дашборд, MJPEG-стрим, сторінки налаштувань камер і датчиків та ендпоінт /metrics. Окремий модуль відповідає за HTTP-взаємодію з сервісом tcps: отримання статусів датчиків та відправку команд сигналізації. Модуль конфігурації реалізує читання та запис

JSON-файлу, а також управління спільною змінною з поточними налаштуваннями.

Застосунок tcps також розбитий на модулі. Ключовим є модуль із описом класу контролера: для кожного активного TCP-контролера, описаного в конфігурації, при старті ініціалізується окрема копія цього класу. Окремий потік виконання створюється не для кожного I/O-пристрою, а для кожного контролера — оскільки один контролер може обслуговувати кілька пристроїв у межах одного TCP-з'єднання. HTTP API реалізований в окремому Flask-модулі, який при обробці запитів звертається до кожного екземпляра класу контролера, збирає останні накопичені дані та агрегує їх у відповідь для main_program.

Dockerfile кожного застосунку базується на офіційному образі python:3.11-slim. Залежності встановлюються з файлу requirements.txt командою pip install, після чого визначається робоча директорія та команда запуску через ENTRYPOINT. Використання slim-образу замість повного зменшує розмір фінального Docker-образу, що прискорює розгортання та зменшує поверхню потенційних вразливостей.

Файл docker-compose.yml визначає чотири сервіси, внутрішню мережу Docker для їх взаємодії та volume для монтування JSON-конфігів із хост-системи в обидва контейнери. Критично важливим параметром є політика перезапуску restart: unless-stopped для всіх чотирьох контейнерів. Ця політика відрізняється від restart: always тим, що контейнер автоматично перезапускається після будь-якої нештатної зупинки або перезавантаження хост-системи, але не запускається, якщо він був зупинений вручну — наприклад, розробником під час налагодження. Це усуває ситуацію, коли зупинений для діагностики контейнер автоматично стартує знову і заважає процесу налагодження.

Для забезпечення автоматичного запуску Docker при старті операційної системи сервіс Docker налаштований через systemd: команда systemctl enable docker додає його до списку сервісів, що запускаються при завантаженні. Таким чином, при будь-якому перезавантаженні промислового сервера — плановому або спричиненому збоєм живлення — операційна система запускає Docker,

Docker запускає всі контейнери згідно з політикою unless-stopped, і система моніторингу відновлює роботу повністю автономно, без участі оператора або адміністратора.

Лістинг 3.1 — Файлова структура проекту

```
project/
├── main_program/
│   ├── data/
│   │   ├── lines.json          # конфігураційний файл main_program
│   │   └── log.txt             # лог помилок коду для розробника
│   ├── static/
│   │   ├── camera_lost_connection.png  # зображення-заглушка при
втраті зв'язку
│   │   ├── styles.css
│   │   └── test.png
│   ├── templates/              # HTML-шаблони Flask
│   │   ├── base.html
│   │   ├── camera.html         # сторінка налаштувань камери
│   │   ├── camera_view.html    # сторінка живого відеопотоку
│   │   ├── index.html          # головний дашборд
│   │   ├── sensor.html         # сторінка налаштувань датчиків
│   │   └── settings.html       # сторінка основних налаштувань
│   ├── app.py                  # Flask маршрути та веб-інтерфейс
│   ├── initializer.py          # ініціалізація потоків та модулів
│   ├── main.py                 # точка входу застосунку
│   ├── processing.py           # обробка зображень, аналіз,
агрегація помилок
│   ├── Dockerfile
│   └── requirements.txt
├── tcps/
│   ├── data.json               # конфігураційний файл tcps
│   ├── main.py                 # точка входу, HTTP API
│   ├── controller.py           # клас опитування датчиків
│   ├── Dockerfile
│   └── requirements.txt
├── docker-compose.yml          # опис усіх чотирьох контейнерів
└── prometheus.yml              # конфігурація Prometheus (scrape,
retention)
```

3.2 Реалізація сервісу опитування датчиків (tcps)

Сервіс tcps реалізований на основі класу контролера, що інкапсулює всю логіку роботи з одним TCP-з'єднанням. При старті застосунку конфігураційний файл зчитується повністю, після чого для кожного активного контролера,

описаного в секції TCP конфігурації, створюється окремий екземпляр класу. Конструктор кожного екземпляра ініціалізує структури даних для зберігання поточних статусів і метрик Prometheus, словники для відстеження попередніх значень спеціальних параметрів (prev_special_values), а також словники для зберігання стану затримок (chiller_hold, vacuum_hold) та таймерів Blue Button (blue_button_timers). Для кожного екземпляра запускається окремий daemon-потік, що виконує нескінченний цикл опитування пристроїв цього контролера. HTTP API модуль при обробці запитів від main_program ітерує по всіх екземплярах класу, збирає їхні поточні дані та агрегує загальну відповідь.

Взаємодія з I/O-пристроями реалізована через власний клас на базі стандартної бібліотеки socket. Клас формує Modbus RTU фрейм вручну: Slave ID пристрою, функціональний код 0x02, початкова адреса входів (0x0000) та їх кількість (0x0008). До шести байт команди обчислюється контрольна сума CRC16, яка додається у кінці фрейму молодшим байтом вперед. Отримана відповідь розбирається побайтово: байт Byte Count вказує скільки байт даних слідує далі, після чого кожен біт байта даних відповідає стану одного дискретного входу в порядку зростання адрес. Перед використанням відповіді перевіряється її CRC для виявлення помилок передачі. Клас також реалізує запис до дискретних виходів (функціональний код 0x05, Force Single Coil) для керування мигалкою та сиреною.

Лістинг 3.1 — Формування Modbus RTU фрейму та розбір відповіді

```
def _read_slave_inputs(self, slave):
    slave_id = slave["SlaveID"]
    # Формуємо RTU-фрейм: slave_id + fc 0x02 + адреса 0x0000 +
кількість 0x0008
    command = [slave_id, 0x02, 0x00, 0x00, 0x00, 0x08, 0, 0]
    crc = self._get_crc(command[:6])
    command[6] = crc & 0xFF   # молодший байт CRC
    command[7] = crc >> 8     # старший байт CRC

    with self.sock_lock:
        try:
            self.sock.sendall(bytes(command))
            time.sleep(0.3)
```

```
            # Спочатку зчитуємо 3-байтовий заголовок відповіді
            header = self._recv_exact(3)  # slave_id, fc,
byte_count
            byte_count = header[2]        # кількість байт даних

            # Зчитуємо байти даних + 2 байти CRC
            data = self._recv_exact(byte_count + 2)
            response = header + data

            if not self._validate_crc(response):
                raise SlaveResponseError("CRC mismatch")

            # Розпаковуємо біти: кожен біт = стан одного
дискретного входу
            bits = []
            for b in response[3:3 + byte_count]:
                bits.extend([(b >> i) & 1 for i in range(8)])
            return bits[:8]

        except SlaveResponseError:
            raise
        except ControllerConnectionError:
            raise
```

На кожній ітерації циклу опитування клас контролера зчитує актуальну конфігурацію зі спільної змінної, що дозволяє підхоплювати зміни без перезапуску потоку. Для кожного Slave, описаного в конфігурації, відправляється Modbus-запит на читання всіх восьми дискретних входів. Отримана відповідь обробляється послідовно: для кожного параметра спочатку перевіряється наявність спеціального обробника за іменем або адресою, і якщо він знайдений — виконується відповідна логіка з оператором continue в кінці. Якщо жодного спеціального обробника не знайдено, виконується стандартна логіка: зчитане значення порівнюється з полем Normal_state. При збігу метрика встановлюється в 0 (норма). При відхиленні метрика встановлюється в 1, і якщо поле Monitored має значення True, запис про помилку додається до структури nested_errors з типом ERROR або WARNING залежно від поля Critical. Структура nested_errors накопичує всі помилки поточної ітерації та після завершення обходу всіх параметрів передається в агрегатор статусів лінії.

Лістинг 3.2 — Стандартна логіка обробки дискретного входу датчика

```
if value != normal_state:
    # Вхід відхилився від норми — оновлюємо метрику Prometheus
    self.metrics["Slaves"][sname]["Parameters"][pname].set(1)

    if par.get("Monitored", False):
        # Генеруємо помилку лише якщо датчик позначено як
моніторований
        nested_errors.setdefault("SLAVES", {})
        nested_errors["SLAVES"].setdefault(
            sname, {"Status": "OK", "PARAMETERS": {}}
        )
        # Тип помилки визначається полем Critical
        err_type = "ERROR" if par.get("Critical", False) else
"WARNING"
        nested_errors["SLAVES"][sname]["PARAMETERS"][pname] =
err_type
else:
    # Вхід у нормі — обнуляємо метрику
    self.metrics["Slaves"][sname]["Parameters"][pname].set(0)
```

Окрім стандартної логіки, реалізовано чотири спеціальні обробники для датчиків із нестандартною поведінкою. Кожен обробник ідентифікується за іменем або адресою параметра і виконується замість стандартної логіки через оператор continue в кінці блоку, що повністю виключає параметр зі стандартного шляху обробки.

Перший спеціальний обробник призначений для датчика з SlaveID=1 та адресою входу 4, що є апаратним перемикачем режиму роботи системи. При виявленні висхідного фронту сигналу (попереднє значення в prev_special_values відрізняється від поточного) обробник викликає функцію request_turn_off_analysis(), що через HTTP надсилає команду до main_program для глобального вмикання або вимикання аналізу камер. Використання детекції висхідного фронту замість перевірки поточного значення гарантує, що команда відправляється рівно один раз при зміні стану, а не повторюється на кожній ітерації. Метрика оновлюється при кожній ітерації незалежно від наявності зміни.

Другий спеціальний обробник реалізує буферну фільтрацію для датчиків типу Vacuum_Error. Вакуумні датчики схильні до короткочасних збоїв сигналу

через електромагнітні завади у виробничому середовищі, тому одиничне спрацювання не повинно активувати сигналізацію. Алгоритм при першій появі помилки фіксує поточний час у словнику vacuum_hold під ключем номера лінії. При кожній наступній ітерації, поки помилка присутня, обчислюється різниця між поточним часом і зафіксованим моментом першої появи. Лише якщо помилка стабільно присутня щонайменше три секунди, вона передається в nested_errors для активації сигналізації. При зникненні помилки запис видаляється зі словника, скидаючи лічильник. Додатково для кожного такого датчика в Prometheus публікується метрика з суфіксом _unfiltered, що відображає миттєве нефільтроване значення.

Лістинг 3.3 — Обробник Vacuum_Error: буферна фільтрація ≥3 сек

```
if "Vacuum_Error" in par["Name"]:
    # Нефільтроване значення — для аналізу коротких збоїв у
Grafana
    self.metrics["Slaves"][sname]["Parameters"]
        [f"{pname}_unfiltered"].set(value != normal_state)

    line_number = int(par["Name"][-1])
    now = time.time()
    hold = self.vacuum_hold.get(line_number)

    if value != normal_state:
        if not hold:
            # Перша поява помилки — фіксуємо час
            self.vacuum_hold[line_number] = {"start": now}

        hold = self.vacuum_hold[line_number]
        if now - hold["start"] >= 3:  # помилка підтверджена ≥3
сек

self.metrics["Slaves"][sname]["Parameters"][pname].set(1)
            if par.get("Monitored", False):
                nested_errors.setdefault("SLAVES", {})
                nested_errors["SLAVES"].setdefault(
                    sname, {"Status": "OK", "PARAMETERS": {}}
                )
                err_type = "ERROR" if par.get("Critical", False)
else "WARNING"

nested_errors["SLAVES"][sname]["PARAMETERS"][pname] = err_type
    else:
        if hold:
```

```
            self.vacuum_hold.pop(line_number, None)  # скидаємо
лічильник
        self.metrics["Slaves"][sname]["Parameters"][pname].set(0)
    continue
```

Третій спеціальний обробник реалізує затримку помилки для датчиків типу Chiller_Error. Чилери (агрегати охолодження) при виникненні помилки автоматично перезапускаються, під час чого I/O-пристрій тимчасово не повертає жодних даних і система могла б хибно вважати, що помилка усунена. При появі відхилення від Normal_state в словнику chiller_hold фіксується час закінчення утримання — поточний момент плюс 30 секунд. Поки поточний час не перевищив цей рубіж, запис про ERROR передається в nested_errors незалежно від фактичного поточного значення датчика. Після закінчення затримки запис видаляється зі словника і обробка повертається до нормального стану. Таким чином навіть після успішного перезапуску чилера та повернення датчика в норму оператор бачить активний ERROR протягом 30 секунд — достатньо для усвідомлення події і прийняття рішення.

Лістинг 3.4 — Обробник Chiller_Error: утримання помилки на 30 секунд

```
if "Chiller_Error" in pname:
    self.metrics["Slaves"][sname]["Parameters"][pname].set(value)
    line_number = int(par["Name"][-1])
    now = time.time()

    if value != par["Normal_state"]:
        # Помилка з'явилась — фіксуємо час до якого ERROR
утримуватиметься
        self.chiller_hold[line_number] = {"until": now + 30}

    hold = self.chiller_hold.get(line_number)
    if hold and now < hold["until"]:
        # Передаємо ERROR навіть якщо датчик вже повернувся в
норму
        nested_errors.setdefault("SLAVES", {})
        nested_errors["SLAVES"].setdefault(
            sname, {"Status": "OK", "PARAMETERS": {}}
        )
        nested_errors["SLAVES"][sname]["PARAMETERS"][pname] =
"ERROR"
    else:
        if hold:
```

```
            self.chiller_hold.pop(line_number, None)  # затримка
завершилась
    continue
```

Четвертий спеціальний обробник реалізує логіку фізичної кнопки Blue Button. При виявленні висхідного фронту (prev_special_values показує 0, поточне значення — 1) обробник виконує дві паралельні дії: надсилає HTTP-запит до main_program для вимкнення відповідної виробничої лінії через функцію set_line_state та запускає таймер threading.Timer(180, self._reactivate_line, args=[line_number]). Таймер спрацьовує через 180 секунд і автоматично викликає метод _reactivate_line, що реактивує лінію, відновлюючи нормальний режим моніторингу. Якщо кнопка натиснута повторно до закінчення таймера, попередній таймер скасовується методом cancel() і запускається новий — оператор може необмежено продовжувати паузу повторними натисканнями. Номер лінії витягується з імені параметра методом rsplit з розбором останнього сегменту після символу підкреслення, що стійко до варіацій іменування.

Лістинг 3.5 — Обробник Blue Button: threading.Timer(180) та _reactivate_line

```
if "Blue_Button" in pname:
    self.metrics["Slaves"][sname]["Parameters"][pname].set(value)
    parts = par["Name"].rsplit("_", 1)
    line_number = int(parts[-1])  # "Blue_Button_4" -> 4

    prev = self.prev_special_values.get((sname, pname), 0)
    is_pressed = bool(value)

    if is_pressed and not prev:  # висхідний фронт — кнопка щойно
натиснута
        set_line_state(f"Line{line_number}", False)  # вимикаємо
лінію

    if is_pressed:
        # Скасовуємо попередній таймер і запускаємо новий на 3
хвилини
        old_timer = self.blue_button_timers.get(line_number)
        if old_timer:
            old_timer.cancel()
        t = threading.Timer(180, self._reactivate_line,
args=[line_number])
        self.blue_button_timers[line_number] = t
        t.start()
```

```
    # Зберігаємо поточне значення для детекції фронту в наступній
ітерації
    self.prev_special_values[(sname, pname)] = 1 if is_pressed
else 0
    continue

def _reactivate_line(self, line_number: int):
    # Викликається threading.Timer через 180 сек після натискання
    line_name = f"Line{line_number}"
    set_line_state(line_name, True)   # реактивуємо лінію
    self.blue_button_timers.pop(line_number, None)
```

HTTP API сервісу tcps реалізований на Flask в окремому модулі. При надходженні GET-запиту від main_program обробник ітерує по всіх екземплярах класу контролера, збирає з кожного поточні агреговані статуси та формує єдиний JSON-об'єкт із повним зрізом стану всіх датчиків. POST-ендпоінти приймають команди керування фізичними виходами: активація мигалки, активація сирени та деактивація обох. Отримавши команду, відповідний екземпляр класу формує Modbus write-запит (функціональний код 0x05) на дискретний вихід I/O-пристрою, до якого підключений відповідний виконавчий пристрій. Розділення читання статусів і запису команд на різні типи ендпоінтів (GET і POST) відповідає семантиці HTTP і спрощує налагодження — GET-запити можна перевіряти напряму через браузер.

3.3 Реалізація модуля комп'ютерного зору

Модуль комп'ютерного зору реалізований у вигляді набору функцій, що запускаються в окремих потоках — по одному на кожну камеру. При старті main_program зчитує конфігурацію та для кожної камери в секції Lines створює потік, передаючи йому об'єкт камери з конфігурації та посилання на спільні структури даних: словник camera_frames для збережених кадрів та словник statuses для поточних статусів.

На початку кожної ітерації потік зчитує актуальні параметри камери зі спільної змінної конфігурації — це забезпечує динамічне підхоплення змін без перезапуску потоку. Далі з об'єкта VideoCapture зчитується черговий кадр методом cap.read(). До кадру застосовується обрізання відповідно до параметрів ROI: якщо оператор задав координати crop, кадр обрізається до вказаної прямокутної області. Оброблений кадр зберігається в словнику camera_frames під ключем імені камери з використанням threading.Lock для захисту від одночасного запису з кількох потоків. Якщо камера увімкнена (параметр ON=True), до збереженого кадру застосовується функція analyze_frame, що повертає поточний статус. Результат зберігається в словнику statuses.

Для відображення кадру на сторінці налаштувань камери не створюється окремий потік — замість цього клієнтська сторінка запитує кадри з підвищеною частотою, використовуючи той самий ендпоінт, що й головний дашборд. Такий підхід обраний з кількох причин. По-перше, невідомо, чи дозволяє конкретна IP-камера одночасно кілька незалежних RTSP-з'єднань — а основний потік камери вже утримує активне з'єднання для дашборду. По-друге, логіку нанесення розмітки get_frame не потрібно дублювати в окремому потоці. По-третє, підвищена частота запитів потрібна лише під час активного налаштування оператором — після закриття сторінки налаштувань навантаження повертається до норми. Це рішення також не накладає обмежень на кількість одночасних підключень: оператор на головній сторінці та технік на сторінці налаштувань з іншого пристрою отримують кадри з одного й того ж потоку без конфліктів.

Лістинг 3.7 — Основний цикл потоку камери: захоплення кадру, crop та аналіз

```
while not stop_flag.is_set():
    ret, frame = cap.read()

    if ret and frame is not None:
        # Обрізання кадру до заданої оператором зони ROI
        p = camera["Parameters"]
```

```
        if p["Top_crop"] and p["Bottom_crop"] and p["Left_crop"]
and p["Right_crop"]:
            frame = frame[p["Top_crop"]:p["Bottom_crop"],
                          p["Left_crop"]:p["Right_crop"]]

        if camera["ON"]:  # аналіз увімкнено
            status = analyze_frame(frame=frame, camera=camera)
            # Зберігаємо статус лише якщо він змінився (для
сигналізації)
            if
statuses[camera["Line_name"]]["Cameras"][camera["Name"]] !=
status:
                changed_data[camera["Name"]] = status

statuses[camera["Line_name"]]["Cameras"][camera["Name"]] = status

            # Оновлюємо набір з 4 бінарних метрик Prometheus
            if status == "OK":              metrics = [1, 0, 0, 1]
            elif status == "WARNING":       metrics = [1, 1, 0, 0]
            elif status == "ERROR":         metrics = [1, 0, 1, 0]
            elif status == "CONNECTION_ERROR": metrics = [0, 0, 0,
0]
        else:
            # Камера вимкнена — зберігаємо кадр але не аналізуємо

statuses[camera["Line_name"]]["Cameras"][camera["Name"]] =
Status.OFF.value
            metrics = [1, 0, 0, 0]

        # Зберігаємо оброблений кадр для веб-інтерфейсу та
get_frame
        camera_frames[camera["Name"]] = frame

    # Оновлюємо всі 4 Gauge-метрики після кожної ітерації
    metric1.set(metrics[0])   # online
    metric2.set(metrics[1])   # warning
    metric3.set(metrics[2])   # error
    metric4.set(metrics[3])   # ok

    time.sleep(interval)

cap.release()  # звільняємо ресурси при зупинці потоку
```

Функція get_frame формує зображення з візуальною розміткою для відображення в веб-інтерфейсі. Вона зчитує збережений кадр із camera_frames та обчислює позиції чотирьох ключових точок: p1 і p2 — крайні точки контрольної лінії, задані оператором; p5 і p95 — межі зони попередження, що обчислюються як зміщення від p1 та p2 на відстань Yellow% від загальної довжини вектора

p1→p2. Напрямний вектор нормується, що забезпечує коректний розрахунок незалежно від кута нахилу лінії.

Перед поверненням статусу функція get_frame також перевіряє параметри Warning_reaction та Error_reaction із конфігурації камери. Ці параметри дозволяють оператору вибірково вимикати реакцію на певний тип помилки для конкретної камери. Якщо Warning_reaction=False і алгоритм визначив статус WARNING — функція повертає OK замість WARNING, оскільки для цієї камери зона попередження не є значущою. Аналогічно при Error_reaction=False статус ERROR замінюється на OK. Такий механізм необхідний для виробничих ліній, де певні відхилення є штатними через особливості технологічного процесу: наприклад, камера контролює зону, де продукт легітимно може виходити за межу на певному етапі виробництва. Замість постійного відключення камери оператор просто вимикає відповідну реакцію, залишаючи пристрій у моніторинговому режимі.

У режимі threshold=True кадр перетворюється у відтінки сірого та бінаризується з порогом T. На бінарне зображення наносяться дві жовті точки в позиціях p5 та p95. Цей режим призначений виключно для налаштування порогу бінаризації оператором: він дозволяє наочно бачити, як алгоритм інтерпретує зображення при поточному значенні T, до збереження параметрів. У режимі threshold=False до кадру застосовується регулювання контрасту через cv2.convertScaleAbs з коефіцієнтом Contrast. На зображення наносяться три відрізки: жовті від p1 до p5 і від p95 до p2 позначають зони попередження, зелений між p5 та p95 — центральну зону OK. Цей режим використовується на дашборді та сторінці MJPEG-стриму.

Найбільш нетривіальною частиною реалізації модуля є обробка втрати з'єднання з камерою. OpenCV при роботі з RTSP-потоком не генерує виключення при розриві з'єднання — натомість метод cap.read() починає зависати нескінченно або повертає порожній кадр без будь-якого сигналу про помилку. Це унеможливлює використання стандартного блоку try/except для виявлення

проблеми, оскільки сам виклик не кидає виключення — він просто ніколи не повертає результат.

Рішення базується на перевірці стану об'єкта VideoCapture методом cap.isOpened() та валідації отриманого кадру. Якщо cap.isOpened() повертає False або кадр порожній, потік переходить у режим відновлення з'єднання. Спочатку викликається cap.release() для явного звільнення апаратного дескриптора і мережевих ресурсів — без цього наступна спроба підключення могла б отримати вже закриту або заблоковану сесію. Камері присвоюється статус CONNECTION_ERROR, а в словник camera_frames записується заздалегідь підготовлене зображення-заглушка. Це важливий UX-аспект: замість відображення останнього валідного кадру, що міг би ввести оператора в оману, веб-інтерфейс показує чітке повідомлення про відсутність зв'язку.

Далі починається цикл reconnect: потік з інтервалом 15 секунд створює новий об'єкт VideoCapture з RTSP URL камери. Критичною деталлю є перевірка першого кадру після відкриття з'єднання: навіть якщо cap.isOpened() повертає True, перший виклик cap.read() може повернути порожній кадр при нестабільному або щойно відновленому з'єднанні. Тому після успішного cap.isOpened() обов'язково виконується один cap.read() та перевірка валідності отриманого кадру. Лише після цього подвійного підтвердження цикл reconnect завершується, камері повертається статус OK і потік продовжує нормальну роботу.

Лістинг 3.6 — Механізм відновлення RTSP-з'єднання при втраті зв'язку

```
# --- При старті потоку: перевірка першого підключення ---
cap = cv2.VideoCapture(camera["Link"], cv2.CAP_FFMPEG)

if not cap.isOpened():
    status = Status.CONNECTION_ERROR.value
    statuses[camera["Line_name"]]["Cameras"][camera["Name"]] =
status
    # Записуємо заглушку — оператор бачить що камера недоступна
    camera_frames[camera["Name"]] = no_connection_image

    while not cap.isOpened():   # цикл reconnect з інтервалом 15
сек
```

```
        time.sleep(15)
        cap = cv2.VideoCapture(camera["Link"], cv2.CAP_DSHOW)

    status = Status.OK.value    # з'єднання відновлено
    statuses[camera["Line_name"]]["Cameras"][camera["Name"]] =
status

# --- В основному циклі: виявлення розриву під час роботи ---
ret, frame = cap.read()
if not ret or frame is None:
    # OpenCV не кидає виключення — перевіряємо ret і frame вручну
    status = Status.CONNECTION_ERROR.value
    statuses[camera["Line_name"]]["Cameras"][camera["Name"]] =
status
    camera_frames[camera["Name"]] = no_connection_image




     # Цикл повторних спроб — до відновлення або сигналу зупинки
    while not cap.isOpened():
        time.sleep(15)
        cap = cv2.VideoCapture(camera["Link"], cv2.CAP_DSHOW)

    # Відновлення підтверджено — знімаємо статус помилки
    status = Status.OK.value
    statuses[camera["Line_name"]]["Cameras"][camera["Name"]] =
status
```

Аналогічний механізм reconnect реалізований для TCP-з'єднань у сервісі tcps. При виникненні мережевої помилки або таймауту socket поточне з'єднання закривається, всім входам пристрою присвоюється статус CONNECTION_ERROR і запускається цикл повторних спроб із фіксованою затримкою між ітераціями. Дворівнева обробка відмов (рівень I/O-пристрою та рівень TCP-шлюзу) дозволяє точно локалізувати проблему: оператор бачить, чи недоступний конкретний пристрій, чи весь сегмент мережі.

3.4 Реалізація веб-інтерфейсу на Flask

Веб-інтерфейс реалізований як набір маршрутів Flask у модулі app застосунку main_program. Flask запускається в окремому daemon-потоці після

ініціалізації всіх потоків камер і датчиків, що гарантує наявність актуальних даних у спільних змінних до першого запиту від браузера. Використання threaded=True при запуску Flask дозволяє обробляти кілька HTTP-запитів одночасно, що є необхідним при паралельному оновленні кількох зображень на дашборді різними браузерними вкладками.

Головна сторінка дашборду повертає HTML-сторінку зі статусами всіх ліній, камер та датчиків. Зображення з камер не вбудовуються в HTML безпосередньо — натомість на сторінці розміщені теги img із src, що вказують на окремі ендпоінти отримання кадрів. JavaScript-код встановлює таймер, що з інтервалом, визначеним у конфігурації Times, оновлює атрибут src кожного img-тегу, додаючи поточний timestamp як параметр запиту для запобігання кешуванню браузером. Сервер у відповідь повертає поточний кадр із camera_frames, відформатований функцією get_frame у режимі threshold=False, як base64-рядок у JSON.

Якщо камера недоступна і знаходиться в режимі reconnect, словник camera_frames містить для неї заздалегідь підготовлене зображення-заглушку — статичний PNG-файл із позначенням відсутності зв'язку, що зберігається в директорії static/. Саме це зображення повертається на дашборд замість реального кадру. Такий підхід є важливим із погляду зручності використання: оператор одразу бачить, що певна камера недоступна, і не може переплутати статичний останній кадр із реальним відеопотоком. Четвертий слот на дашборді замість камери містить iframe з картою повітряних тривог України, що оновлюється в реальному часі.

Для кожної виробничої лінії на дашборді відображається агрегований статус: якщо хоча б одна камера або датчик лінії має статус ERROR — лінія позначається червоним, при наявності лише WARNING — жовтим, при всіх OK — зеленим. Це дозволяє оператору миттєво оцінити стан всього виробництва, не аналізуючи кожен пристрій окремо. Також на дашборді розміщена кнопка Blue Button для кожної лінії: при натисканні браузер відправляє POST-запит до Flask,

який передає команду до tcps, де активується таймер вимкнення лінії на 3 хвилини.

Сторінка живого відеопотоку реалізує MJPEG-стрим через Python-генератор. Flask-маршрут повертає об'єкт Response з генератором та Content-Type: multipart/x-mixed-replace; boundary=frame. Генератор у нескінченному циклі зчитує поточний кадр із camera_frames — або реальний, або заглушку при відсутності зв'язку — кодує його у JPEG формат засобами OpenCV через cv2.imencode та формує частину multipart-відповіді з заголовком Content-Type: image/jpeg та Content-Length. Браузер інтерпретує такий потік як послідовність зображень, що створює ефект відеотрансляції без WebSocket або спеціальних плагінів. На відміну від дашборду, де кадри передаються через окремі HTTP-запити, MJPEG-потік є безперервним з'єднанням, що забезпечує плавніше відображення та меншу затримку. На кожному кадрі відображається розмітка контрольної зони у режимі threshold=False.

Сторінка налаштувань камер обробляє POST-запити з даними форм. При отриманні нових параметрів сервер валідує вхідні дані — перевіряє типи, діапазони допустимих значень — оновлює відповідний запис у JSON-файлі конфігурації та одночасно оновлює спільну змінну конфігурації в пам'яті. Завдяки цьому зміни набирають чинності при наступній ітерації відповідного потоку камери без перезапуску. Сторінка налаштувань датчиків працює аналогічно: зміни параметрів monitored, critical та активності кожного з восьми входів зберігаються в JSON-файлі tcps і передаються сервісу tcps через HTTP POST для оновлення спільної змінної конфігурації в його пам'яті.

3.5 Реалізація багатопотоковості та синхронізації даних

Ініціалізація системи починається з імпорту функції initialize_camera_threads у main.py. Цей імпорт запускає ланцюгову ініціалізацію: модуль processing ініціалізує модуль app та всі залежні модулі, реєструє Flask-

маршрути та готує спільні структури даних. Після повернення з імпорту main.py викликає initialize_camera_threads, що запускає потоки для камер та контролерів у суворо визначеному порядку: спочатку ініціалізуються структури даних і спільні змінні, потім запускаються потоки камер і датчиків як daemon-потоки, і лише після цього стартує Flask-сервер. Такий порядок гарантує, що до моменту першого HTTP-запиту всі потоки вже запущені й почали заповнювати camera_frames та statuses актуальними даними.

Параметр daemon=True при створенні кожного потоку є важливим архітектурним рішенням. Daemon-потоки автоматично завершуються при зупинці головного процесу — Python не чекає їх завершення перед виходом. Це означає, що зупинка Flask-сервера (наприклад, через Docker stop) автоматично завершує всі потоки камер і датчиків без необхідності явного управління їх життєвим циклом.

Для координації коректного завершення потоків у внутрішніх циклах — зокрема в циклі reconnect — використовується об'єкт threading.Event під назвою stop_flag. Кожен потік перевіряє stop_flag.is_set() на кожній ітерації циклу reconnect та з певною частотою в основному циклі. При отриманні сигналу зупинки потік виходить з циклу і завершується, навіть якщо він перебував у стані очікування time.sleep(15). Без цього механізму потоки в циклі reconnect залишалися б заблокованими на sleep навіть після команди зупинки контейнера, що затримувало б весь процес завершення.

Спільні змінні, до яких одночасно звертаються кілька потоків як для читання, так і для запису, захищені об'єктами threading.Lock. Зокрема, словник camera_frames захищений окремим lock-об'єктом: кожен потік камери при записі нового кадру захоплює lock у блоці with, виконує запис і звільняє lock. Flask-обробники, що зчитують кадри для дашборду або MJPEG-стриму, також виконують читання під захистом того ж lock. Це гарантує, що браузер ніколи не отримає частково перезаписаний кадр. Аналогічно захищений словник statuses, до якого записують потоки датчиків і з якого читає логіка агрегації статусів для сигналізації та Prometheus.

Оновлення спільної змінної конфігурації при зміні налаштувань через веб-інтерфейс також виконується атомарно: спочатку новий конфіг записується в JSON-файл на диск, потім одним присвоєнням замінюється об'єкт спільної змінної. Оскільки в CPython присвоєння об'єкта є атомарною операцією завдяки GIL, потоки не можуть отримати частково оновлений стан конфігурації — вони або бачать повністю старий об'єкт, або повністю новий. Запис у JSON-файл відбувається під захистом окремого file_lock для запобігання одночасному запису з кількох HTTP-запитів.

3.6 Реалізація системи сигналізації та метрик Prometheus

Логіка сигналізації реалізована в main_program як окрема функція, що викликається після кожного оновлення статусів від tcps. Функція ітерує по всіх активних лініях і пристроях, агрегуючи найвищий рівень помилки: якщо хоча б один пристрій має статус ERROR — формується команда на активацію мигалки і сирени; якщо є лише WARNING і жодного ERROR — команда на активацію лише мигалки. Якщо всі пристрої мають статус OK або OFF — відправляється команда деактивації. Для уникнення надмірних HTTP-запитів до tcps стан сигналізації кешується: нова команда відправляється лише при зміні рівня сигналу порівняно з попередньою ітерацією.

Команда на активацію фізичної сигналізації відправляється від main_program до tcps у вигляді HTTP POST-запиту. Сервіс tcps, отримавши команду ERROR, активує обидва виходи I/O-пристрою (мигалка і сирена) через Modbus write-запити. Одночасно запускається threading.Timer(3, deactivate_siren): через три секунди таймер відправляє Modbus-команду на вимкнення виходу сирени, залишаючи мигалку активною. Якщо надходить нова команда ERROR до закінчення таймера, попередній таймер скасовується і запускається новий — сирена лунає ще три секунди від моменту повторного

ERROR. Деактивація всієї сигналізації відбувається при отриманні команди OK: обидва виходи вимикаються і жоден таймер не запускається.

Метрики Prometheus для камер реалізовані за принципом, що відрізняється від звичайного числового кодування. Для кожної камери оголошується не один Gauge з числовим значенням (0–4), а п'ять окремих Gauge-метрик — по одній на кожен можливий статус: camera_status_off, camera_status_on, camera_status_ok, camera_status_warning, camera_status_error. У кожен момент часу рівно одна з п'яти метрик має значення 1, решта — 0. Такий підхід суттєво спрощує побудову дашбордів у Grafana: замість написання умовних виразів для декодування числового коду, дашборд може напряму використовувати метрику camera_status_error як індикатор помилки або будувати стек-графік із п'яти метрик для відображення розподілу статусів у часі.

Для датчиків метрики залишаються бінарними: один Gauge на параметр зі значенням 0 (норма) або 1 (помилка або CONNECTION_ERROR). Для датчиків Vacuum_Error додатково оголошується метрика з суфіксом _unfiltered — вона отримує значення 1 одразу при першому спрацюванні, не чекаючи трисекундного підтвердження, що дозволяє в Grafana аналізувати частоту та тривалість коротких збоїв незалежно від логіки сповіщень.

Усі метрики оновлюються в реальному часі в потоках обробки: потік камери — після кожного аналізу кадру, потік датчика — після кожного Modbus-запиту. Flask-застосунок надає ендпоінт /metrics, що генерується автоматично бібліотекою prometheus-client у форматі OpenMetrics, сумісному з Prometheus. Контейнер Prometheus звертається до цього ендпоінту з налаштованим інтервалом scraping і зберігає отримані значення як часові ряди. Grafana, підключена до Prometheus як джерела даних, дозволяє будувати графіки зміни статусів у часі, виявляти закономірності та аналізувати ефективність кожної виробничої лінії з будь-якою глибиною ретроспективи в межах налаштованого терміну зберігання.

3.7 Висновки до розділу 3

У третьому розділі описано практичну реалізацію всіх компонентів IoT-платформи відповідно до архітектурних рішень, спроектованих у другому розділі. Реалізовано два незалежних Python-застосунки з модульною структурою коду: main_program з модулями processing, app та конфігурації, і tcps з класом контролера та Flask HTTP API. Обидва застосунки контейнеризовано в Docker із політикою перезапуску unless-stopped, що в поєднанні з автозапуском Docker через systemd забезпечує повністю автономне відновлення системи після будь-яких перезавантажень промислового сервера.

Реалізація сервісу tcps побудована на моделі окремого екземпляра класу для кожного TCP-контролера з власним потоком опитування. Власний клас на базі socket реалізує Modbus TCP без зовнішніх залежностей, що спрощує розгортання. Чотири спеціальні обробники вирішують реальні виробничі задачі: глобальне перемикання режиму аналізу (висхідний фронт, SlaveID=1/Address=4), буферна фільтрація вакуумних датчиків (≥3 сек із публікацією нефільтрованої метрики), утримання помилки чилерів (30 сек після нормалізації) та Blue Button (threading.Timer(180) з автоматичною реактивацією лінії). Параметри обробників визначені емпірично в процесі налаштування системи.

Найбільше часу при реалізації модуля камер пішло не на сам алгоритм аналізу, а на механізм reconnect — OpenCV при втраті зв'язку не кидає виключення, тому довелось будувати власну логіку виявлення і відновлення з трьома ітераціями. Параметри Warning_reaction та Error_reaction дозволяють гнучко налаштовувати реакцію кожної камери без зупинки моніторингу. Веб-інтерфейс реалізує два режими передачі зображень: base64-кадри через окремі HTTP-запити для дашборду та MJPEG-стрим для сторінки перегляду.

Багатопотокова архітектура забезпечена через threading.Thread з daemon=True, threading.Lock для camera_frames, statuses та запису у файл, threading.Event (stop_flag) для координації завершення потоків у циклах

reconnect. Метрики Prometheus для камер реалізовані як п'ять окремих бінарних Gauge-метрик (по одній на статус), що спрощує побудову дашбордів у Grafana порівняно з числовим кодуванням.

4. ТЕСТУВАННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Методологія та середовище тестування

Тестування системи проводилось у два послідовних етапи: локальне тестування на персональному комп'ютері розробника без підключення до реального промислового обладнання, та інтеграційне тестування безпосередньо на виробничому об'єкті із залученням замовника.

На етапі локального тестування перевірялась коректність роботи програмної логіки в ізольованому середовищі. Підхід до тестування кожного з двох основних модулів суттєво відрізнявся залежно від доступного обладнання.

Тестування модуля комп'ютерного зору проводилось із використанням вбудованої веб-камери ноутбука розробника як замінника промислових IP-камер. OpenCV підтримує той самий інтерфейс VideoCapture для роботи як з RTSP-потоками, так і з локальними USB-камерами — достатньо змінити аргумент з URL на числовий індекс пристрою. Це дозволило тестувати повний конвеєр обробки зображення без доступу до реальних камер. Для перевірки алгоритму аналізу та коректності присвоєння статусів використовувався фізичний об'єкт, що переміщувався в полі зору камери: знаходився в центральній зоні контрольної лінії для перевірки статусу OK, наближався до меж зони попередження для перевірки WARNING та виходив за межу для ERROR. Відсутність об'єкта в кадрі перевіряла коректне визначення помилки при відсутності продукту. Час від перетину контрольної лінії до зміни статусу на веб-сторінці не перевищував 0.3 секунди, що підтверджувало прийнятну затримку обробки для виробничого застосування. Такий підхід дозволив виявити та виправити більшість логічних помилок алгоритму ще до першого виїзду на фабрику, суттєво скоротивши час налагодження на реальному обладнанні.

Тестування модуля датчиків на локальному етапі мало якісно відмінний характер — використовувалось реальне промислове обладнання. Першу партію з семи I/O-пристроїв та двох TCP-контролерів було доставлено розробнику

додому для вивчення та розробки коду до монтажу обладнання на фабриці. Це дало змогу детально вивчити специфіку Modbus TCP на конкретних пристроях: формат пакетів, часові характеристики відповідей, поведінку при різних режимах роботи. Саме на цьому етапі був написаний і налагоджений власний клас для роботи з Modbus TCP на базі бібліотеки socket, реалізована логіка опитування та обробки дискретних входів. Кожен з семи пристроїв почергово підключався до розробницького ПК, а входи перемикались вручну для перевірки коректності зчитування значень та їх передачі в спільну змінну. Цей підхід дозволив не лише розробити стабільний код до виїзду на виробництво, але й особисто познайомитись із особливостями поведінки кожного конкретного пристрою — знання, що виявились критично важливими при подальшому налагодженні на фабриці.

На етапі інтеграційного тестування система розгорталась на виробничому сервері підприємства та підключалась до реального обладнання. Конфігурація тестового середовища включала чотири активних виробничих лінії з чотирма підключеними камерами, що працювали одночасно. П'ята камера була підключена на неактивній лінії виключно для перевірки роботи модуля при наявності недіючого обладнання. Мережева інфраструктура включала три TCP-контролери, на кожному з яких було встановлено по три I/O-пристрої — таким чином система одночасно опитувала дев'ять I/O-пристроїв і обробляла до 72 дискретних входів. Тестування проводилось спільно із замовником, який забезпечував доступ до обладнання, виконував монтаж датчиків та надавав фідбек щодо зручності використання інтерфейсу.

Апаратна частина тестування включала не лише перевірку програмного забезпечення, але й налаштування фізичної інфраструктури: перевірку правильності підключення датчиків, встановлення необхідних термінуючих резисторів на шині RS485, налаштування мережевих маршрутизаторів для забезпечення стабільного з'єднання з I/O-пристроями. Ці роботи виконувались паралельно з тестуванням програмного забезпечення, що дозволяло негайно перевіряти внесені зміни на реальному обладнанні. Паралельне проведення

апаратних і програмних робіт, з одного боку, прискорювало загальний процес введення системи в дію, а з іншого — ускладнювало діагностику: помилки апаратного підключення та програмні баги на початковому етапі проявлялись схожими симптомами, що вимагало систематичного підходу до ізоляції причин кожної проблеми.

4.2 Тестування модуля комп'ютерного зору

Тестування модуля комп'ютерного зору охоплювало перевірку алгоритму аналізу зображення, налаштування параметрів у реальних умовах освітлення, роботу допоміжних функцій відображення та стійкість до втрати з'єднання.

На локальному етапі коректність алгоритму перевірялась в реальному часі з використанням веб-камери ноутбука. Контрольна лінія налаштовувалась безпосередньо у веб-інтерфейсі, після чого розробник, перебуваючи перед камерою, фізично відтворював різні сценарії: рівномірне знаходження в центральній зоні для підтвердження статусу OK, поступове наближення до меж зони попередження для перевірки переходу у WARNING, вихід за межу лінії для ERROR та повний вихід з поля зору камери для перевірки статусу при відсутності об'єкта. Ключовою метрикою при цьому тестуванні була затримка реагування — час від фізичної зміни положення до оновлення статусу на дашборді. Виміряний час реагування не перевищував 0.3 секунди.

На виробничому етапі перевірялась коректність класифікації на реальних кадрах із промисловими камерами. Для кожної лінії контрольна лінія встановлювалась відповідно до реального положення конвеєра, після чого оператор або розробник вручну переміщував тестовий продукт через зону контролю для перевірки всіх трьох статусів. Алгоритм коректно реагував на присутність та відсутність продукту, його положення відносно контрольної лінії та наближення до меж зони попередження.

Основною складністю, виявленою при тестуванні на реальному виробничому об'єкті, стала залежність алгоритму бінаризації від умов

освітлення. Виробничий цех має природне освітлення через вікна, рівень якого суттєво змінюється протягом доби та залежить від хмарності. При налаштуванні порогу T в умовах яскравого сонячного освітлення той самий параметр давав хибні спрацювання у вечірні години або в похмуру погоду — і навпаки. Спроби знайти компромісне значення T, що задовільно працювало б при різних умовах освітлення, не давали стабільного результату: при низькому контрасті між продуктом та фоном бінаризація не могла чітко відокремити об'єкт незалежно від обраного порогу.

Проблему вирішено апаратним способом: під конвеєр у зону аналізу встановлено контрастні підкладки темного кольору. Це забезпечило стабільний темний фон під продуктом незалежно від зовнішнього освітлення, різко підвищивши контраст між об'єктом та фоном. Після встановлення підкладок налаштування порогу T стало стабільним протягом усього робочого дня: одне значення параметра забезпечувало коректну бінаризацію як вранці при яскравому сонці, так і ввечері при штучному освітленні цеху. Це рішення демонструє важливість урахування умов фізичного середовища при проектуванні систем технічного зору.

Для зручного налаштування порогу оператори активно використовували режим threshold=True у функції get_frame. Цей режим, розроблений спеціально для процесу налаштування, відображає результат бінаризації безпосередньо на сторінці веб-інтерфейсу. Оператор змінював значення T на сторінці налаштувань і після кожної зміни бачив актуальне бінаризоване зображення з позначеними межами зони попередження. Такий підхід дозволив операторам без технічних знань самостійно налаштовувати параметри камер, орієнтуючись виключно на візуальний результат. Ефективність процесу налаштування підтверджується тим, що після встановлення підкладок повне налаштування порогу для нової камери займало не більше п'яти хвилин.

Окремо тестувалась коректна робота параметрів Warning_reaction та Error_reaction, що дозволяють вибірково вимикати реакцію на певний тип помилки для конкретної камери. На одній з ліній через особливості

технологічного процесу продукт легітимно міг тимчасово наближатись до межі контрольної лінії на певному етапі виробничого циклу. Для цієї камери реакція на WARNING була вимкнена: камера продовжувала відображати зображення з розміткою, логувати метрики в Prometheus та за необхідності генерувати ERROR при повному виході за межу, однак попереджувальна сигналізація не активувалась. Це дозволило уникнути постійних хибних тривог, що знижували б довіру операторів до системи, без повного відключення камери від моніторингу.

Сценарій втрати з'єднання з камерою відтворювався шляхом фізичного відключення мережевого кабелю. При відключенні на дашборді замість кадру відображалось заздалегідь підготовлене зображення-заглушка, що чітко сигналізує про відсутність зв'язку.

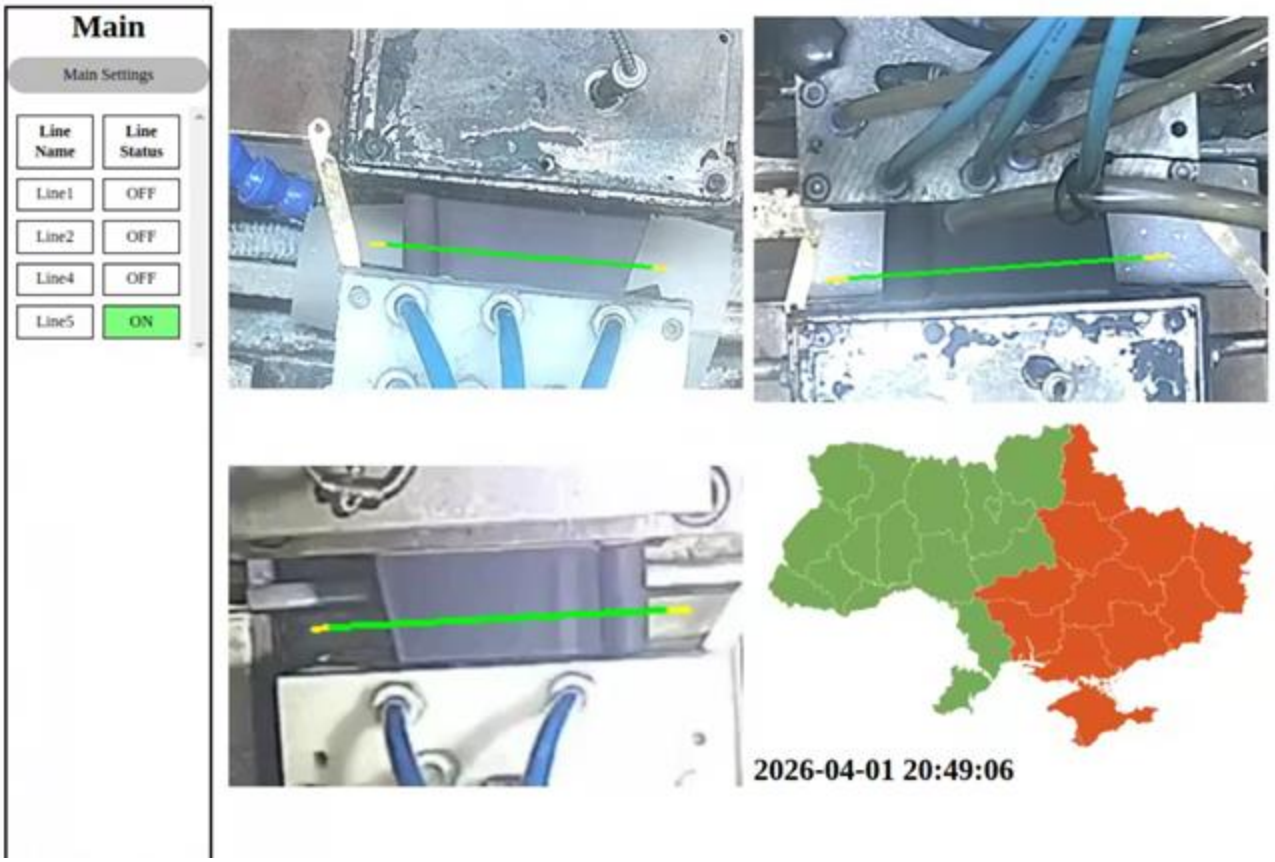


Рисунок 4.1 - Дашборд із активними камерами та статусами ліній

Статус камери змінювався на CONNECTION_ERROR і відображався у загальному статусі лінії відповідним кольором. Після відновлення кабельного

з'єднання система автоматично відновлювала підключення протягом наступних 15 секунд — максимального часу очікування в циклі reconnect. При цьому інші камери та датчики продовжували нормальну роботу без жодного впливу, що підтвердило ізоляцію відмов між потоками.

4.3 Тестування модуля збору даних з датчиків

Тестування модуля датчиків розпочалось ще до виїзду на виробництво: перша партія з семи I/O-пристроїв та двох TCP-контролерів була передана розробнику для вивчення в домашніх умовах. Цей підготовчий етап дозволив вивчити специфіку взаємодії з конкретними моделями обладнання: дослідити формат Modbus-пакетів на практиці, виявити особливості часових характеристик відповідей, налаштувати параметри таймаутів та розробити і налагодити власний клас для роботи з Modbus TCP. Кожен з семи пристроїв тестувався шляхом ручного перемикання входів і перевірки коректності відображення значень у відповідях на Modbus-запити. Такий підхід суттєво прискорив наступний етап: на фабриці вже не було необхідності одночасно розбиратись із протоколом і виявляти апаратні проблеми, оскільки поведінка обладнання була добре відома.

На фабриці тестування стандартної логіки обробки датчиків проводилось шляхом почергового переведення окремих дискретних входів I/O-пристроїв у стан, що відрізнявся від Normal_state. Для кожного входу перевірялась коректна зміна статусу на дашборді та активація або відсутність сигналізації відповідно до налаштованих параметрів Monitored та Critical. Перевірялись усі чотири можливі комбінації: Monitored=False (вхід не генерує сповіщень), Monitored=True і Critical=False (генерується WARNING), Monitored=True і Critical=True (генерується ERROR), вхід вимкнений у налаштуваннях (повністю ігнорується).

Тестування відмовостійкості проводилось шляхом фізичного відключення I/O-пристроїв та TCP-контролерів від мережі. При відключенні окремого I/O-пристрою всі його вісім входів негайно отримували статус

CONNECTION_ERROR, відображаючись відповідно на дашборді, тоді як інші пристрої на тому ж контролері продовжували нормальну роботу. При відключенні цілого TCP-контролера всі три підключені до нього I/O-пристрої одночасно переходили в стан CONNECTION_ERROR. В обох випадках відновлення з'єднання відбувалось автоматично — сервіс tcps самостійно відновлював Modbus-з'єднання без перезапуску і повертав коректні статуси при наступному успішному опитуванні.

Спеціальні обробники edge-cases тестувались відповідно до їхньої специфіки в умовах реального обладнання. Для Vacuum_Error короткочасні помилки тривалістю менше трьох секунд не призводили до активації сигналізації — лише нефільтрована метрика в Prometheus фіксувала коротке спрацювання. При тривалості понад три секунди сигналізація активувалась коректно. Grafana дозволяла порівнювати нефільтровану та фільтровану метрики і наочно бачити, скільки коротких збоїв було відфільтровано за зміну.

Для Chiller_Error після імітації помилки і її усунення перевірявся тридцятисекундний період утримання статусу ERROR — протягом цього часу сигналізація залишалась активною, хоча датчик вже повернув нормальне значення. Blue Button тестувався кількаразово: натискання зупиняло сигналізацію та вимикало лінію, повторне натискання до закінчення таймера скидало і перезапускало тримінутний відлік, а після закінчення таймера лінія автоматично повертала статус моніторингу.

Найбільш тривалим у діагностиці виявився збій, пов'язаний з одним із TCP-контролерів застарілої версії мікропрограмного забезпечення. Відповіді від цього контролера деколи надходили в нестандартному форматі: пакети могли бути обрізаними, накладатись один на одного або містити частини двох різних відповідей в одному буфері прийому TCP-сокета. Це призводило до систематичних помилок при парсингу Modbus-відповідей саме від цього пристрою, що зовні виглядало як нестабільне з'єднання або помилки читання конкретних входів.

Діагностику суттєво ускладнювало те, що на початковому етапі паралельно тривав монтаж датчиків та налаштування мережевої інфраструктури: помилки з'єднання від проблемного контролера і помилки від не повністю підключеного обладнання давали схожу симптоматику. Лише після стабілізації апаратної частини інших двох контролерів і зведення проблеми до єдиного проблемного пристрою почався детальний аналіз пакетного трафіку. Дослідження показало системний характер нестандартних відповідей саме від цього контролера при певних умовах навантаження, що вказувало на проблему на рівні прошивки. Контролер замінено на новіший екземпляр з актуальним програмним забезпеченням, що повністю усунуло проблему. Замінений пристрій було перенесено на інший, значно простіший проект без вимог до стабільності пакетного формату.

4.4 Тестування веб-інтерфейсу та виявлені проблеми

Тестування веб-інтерфейсу охоплювало перевірку всіх сторінок застосунку, роботу з налаштуваннями в реальному часі, поведінку системи при тривалій безперервній роботі та виявлення і усунення нетривіальних програмних помилок.

Дашборд тестувався при одночасному підключенні кількох браузерних клієнтів з різних пристроїв: оператор на основному моніторі виробничої зони та технік, що виконував налаштування з ноутбука. Обидва клієнти отримували актуальні кадри незалежно один від одного без затримок і без взаємного впливу, що підтвердило коректну роботу багатопотокового Flask при паралельних HTTP-запитах. Коректність відображення агрегованих статусів ліній перевірялась шляхом почергового введення датчиків та камер у стани OK, WARNING та ERROR і спостереження за відповідною зміною кольорового маркування ліній на дашборді.

Зміна налаштувань камери через веб-інтерфейс тестувалась з негайною перевіркою ефекту. Після збереження нових координат контрольної лінії або нового значення Yellow% наступний кадр на дашборді відображав оновлену розмітку — без будь-якого перезапуску. Зміна параметрів датчиків — перемикання Monitored та Critical для окремих входів — набирала чинності при наступному циклі опитування відповідного пристрою, тобто фактично миттєво з точки зору оператора. Це підтвердило коректну роботу механізму динамічного оновлення конфігурації через спільну змінну в пам'яті обох контейнерів.

Тестування сигналізації охоплювало перевірку повного ланцюга від виникнення помилки до фізичного сигналу: зміна стану датчика або камери → оновлення статусу в спільній змінній → HTTP-команда від main_program до tcps → Modbus write-запит на I/O-вихід → спрацювання фізичного пристрою. При статусі WARNING активувалась лише мигалка. При ERROR одночасно активувались мигалка і сирена; сирена автоматично вимикалась через три секунди за рахунок threading.Timer у tcps, тоді як мигалка залишалась активною. Кнопка Blue Button коректно зупиняла весь сигнал і запускала тримінутний таймер реактивації.

Найбільш трудомістким у виявленні виявився баг, пов'язаний з параметром use_reloader фреймворку Flask. За замовчуванням цей параметр має значення True, що призначено для зручності розробки: Flask автоматично перезапускається при виявленні змін у файлах коду без необхідності вручну перезапускати сервер. Реалізація цього механізму передбачає запуск Flask у двох процесах: основному (батьківському) та дочірньому, що є точною копією першого. У такій конфігурації всі глобальні змінні, оголошені на рівні модулів Python, ініціалізуються двічі — по одному екземпляру в кожному процесі. Відповідно, словники camera_frames та statuses, що слугують спільними структурами даних між потоками та Flask-обробниками, існували у двох незалежних копіях у двох різних процесах.

Потоки камер запускались в одному процесі і коректно записували дані у свою копію змінних; Flask-обробники HTTP-запитів виконувались в іншому

процесі і зчитували дані з власної, порожньої копії тих самих змінних — отримуючи у відповідь завжди порожні структури. Зовнішній прояв проблеми: дашборд повертав порожні відповіді і не відображав жодних кадрів, хоча в логах потоків камер були присутні записи про успішне захоплення і збереження кадрів. Ця невідповідність між логами та поведінкою тривалий час вводила в оману: здавалось, що проблема у Flask-маршрутах або в логіці читання з camera_frames.

Вирішення виявилось тривіальним: встановлення use_reloader=False при виклику app.run(). Ця проблема є прикладом того, як поширена практика розробки (увімкнений reloader) стає джерелом критичного дефекту при переході від розробки до production-архітектури з явними глобальними змінними між потоками.

Ще однією проблемою, виявленою в процесі тривалої роботи, було кешування кадрів браузером. При стандартному підході до оновлення зображення через зміну src атрибута img-тегу браузер на основі URL запиту міг повертати збережений у кеші попередній кадр замість запиту нового з сервера. Це призводило до «замороження» зображення на дашборді — все виглядало коректно при першому завантаженні, але поступово зображення переставало оновлюватись. Проблему вирішено стандартним підходом anti-cache: до URL кожного запиту за кадром додається унікальний параметр timestamp, що генерується JavaScript як поточний unix-час. URL виду /frame?camera=X&t=1706789123 є унікальним для кожного запиту, тому браузерний кеш ніколи не використовується, і кожне оновлення гарантовано отримує свіжий кадр із сервера.

Паралельно з проблемою кешування виявилась ще одна, глибша проблема: при тривалій безперервній роботі — порядку кількох годин — у деяких браузерах виникало поступове накопичення стану, що призводило до зависання JavaScript-таймера оновлення зображень. Динамічне оновлення зупинялось, хоча сторінка залишалась відкритою і сервер продовжував нормальну роботу. Для вирішення цієї проблеми в HTML-шаблон додано мета-тег автооновлення сторінки з директивою http-equiv та значенням content рівним 60 секундам. Він

викликає примусове повне перезавантаження сторінки браузером раз на хвилину, що скидає весь накопичений JavaScript-стан і відновлює стабільне оновлення зображень.

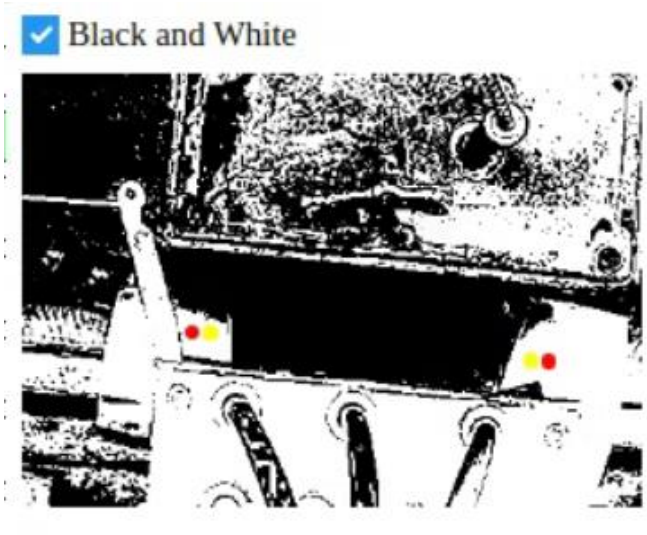


Рисунок 4.2 - Сторінка налаштувань камери з активним режимом threshold

4.5 Тестування надійності та розгортання

Тестування надійності системи включало перевірку поведінки при позаштатних ситуаціях на рівні інфраструктури: перезавантаження сервера, одночасна робота всіх активних ліній та тривала безперервна експлуатація. Окремо розглядались проблеми, виявлені в процесі розробки та розгортання, що стосуються стабільності механізмів відновлення з'єднань.

Сценарій перезавантаження сервера відтворювався шляхом примусового вимкнення живлення та подальшого увімкнення. Після завантаження операційної системи сервіс Docker автоматично запускався через systemd, а всі чотири контейнери (main_program, tcps, prometheus, grafana) відновлювали роботу відповідно до політики restart: unless-stopped. Час від увімкнення живлення до повної готовності системи — доступності веб-інтерфейсу та початку опитування датчиків — склав близько двох хвилин і визначався переважно часом завантаження операційної системи, а не самих контейнерів.

Перевірялась також відповідність політики unless-stopped своїй специфіці: контейнери, зупинені вручну командою docker stop перед перезавантаженням, після увімкнення сервера не запускались автоматично — на відміну від контейнерів, що зупинились через збій або перезавантаження системи. Це підтвердило коректне розмежування між навмисною зупинкою для налагодження та позаштатним завершенням роботи.

Одночасна робота чотирьох виробничих ліній із чотирма камерами та дев'ятьма I/O-пристроями протягом кількох годин не виявила проблем зі стабільністю. Системні ресурси сервера — використання процесора та оперативної пам'яті — залишались у прийнятних межах завдяки кільком архітектурним рішенням: обрізання кадрів до ROI перед аналізом суттєво знижувало обчислювальне навантаження від обробки 4K-зображень, а daemon-потоки з blocking I/O ефективно використовували час очікування відповідей від мережевих пристроїв.

Механізм відновлення RTSP-з'єднання пройшов через три послідовні ітерації вдосконалення протягом тестування на реальному обладнанні. Перша ітерація базувалась на стандартному підході: блок try/except навколо cap.read() і перевірка значення ret. Ця версія виявилась неефективною, оскільки OpenCV при втраті RTSP-з'єднання не генерує виключення — метод cap.read() або блокується нескінченно, або повертає порожній кадр без жодного сигналу про відмову. Потік камери фактично зависав при втраті з'єднання і не відновлювався самостійно.

На другій ітерації було додано явне звільнення ресурсів через cap.release() та цикл reconnect з інтервалом очікування між спробами. Це вирішило проблему зависання потоку і забезпечило базовий механізм відновлення. Однак при тестуванні виявилась нова проблема: після успішного відкриття нового VideoCapture перший виклик cap.read() при нестабільному з'єднанні міг повертати порожній кадр навіть за умови що cap.isOpened() повертав True. Це спричиняло передчасний вихід із циклу reconnect з подальшим негайним поверненням у нього.

Третя, фінальна, ітерація вирішила цю проблему через подвійну перевірку після відкриття з'єднання: після успішного cap.isOpened() обов'язково виконується один cap.read() і перевірка валідності отриманого кадру. Лише після проходження обох умов цикл reconnect завершується і потік повертається до нормальної роботи. Ця версія продемонструвала стабільне автоматичне відновлення при всіх типах розривів з'єднання

Варто окремо відзначити роль реальних умов виробництва у виявленні граничних випадків. Непередбачені відключення електроживлення, характерні для поточних умов в Україні, кілька разів відтворювали сценарій одночасної втрати з'єднання з усіма камерами та датчиками. У цих ситуаціях система коректно переводила всі пристрої в стан CONNECTION_ERROR, а після відновлення живлення і перезапуску контейнерів самостійно відновлювала всі з'єднання в порядку, визначеному конфігурацією, без жодного ручного втручання.

4.6 Аналіз результатів та практичне значення

За підсумками тестування система успішно виконує всі заявлені функції в умовах реального виробництва. Чотири виробничі лінії моніторяться одночасно в режимі реального часу: камери аналізують геометричні параметри продукції з затримкою реагування до 0.3 секунди, датчики відстежують стан обладнання з інтервалом опитування, визначеним у конфігурації, а веб-інтерфейс надає операторам централізовану точку контролю. Усі основні функціональні вимоги виконані: збір та аналіз даних з 72 дискретних входів і 5 камер в єдиній системі, автоматична сигналізація з диференційованою реакцією на різні рівні серйозності, гнучке налаштування всіх параметрів без перезапуску системи, а також автоматичне відновлення після будь-яких мережевих відмов.

Система продемонструвала стійкість до типових відмов у промисловому середовищі. Втрата мережевого з'єднання з камерами та I/O-пристроями

коректно обробляється з автоматичним відновленням і відображенням заглушки замість застиглого кадру. Перезавантаження сервера після відключень електроживлення не потребує ручного втручання — система відновлює роботу автономно протягом двох хвилин. Паралельна робота кількох десятків потоків забезпечується без взаємного блокування і з прийнятним навантаженням на ресурси сервера.

Модуль комп'ютерного зору виявив практичне обмеження, характерне для класичних методів порогової бінаризації: суттєву залежність від умов зовнішнього освітлення. В умовах виробничого цеху з природним освітленням рівень яскравості змінюється протягом доби, що унеможливлює використання одного фіксованого значення порогу T протягом усього часу роботи. Поточне рішення — контрастні підкладки під конвеєром — ефективне для наявної конфігурації, але потребує фізичного втручання при зміні виробничого процесу або розміщення обладнання. Потенційним вдосконаленням є застосування адаптивної бінаризації (adaptive thresholding), що автоматично підлаштовує поріг до локального рівня яскравості, або нормалізація гістограми зображення перед аналізом. Ці методи дозволили б знизити залежність від умов освітлення без апаратних змін.

Практичне значення впровадженої системи для підприємства полягає у централізації контролю виробничих ліній. До впровадження системи оператор мав фізично перебувати поруч із кожною лінією або регулярно обходити їх для перевірки стану обладнання. Після впровадження один оператор може одночасно контролювати всі активні лінії з єдиної робочої станції. Система сигналізації з градуйованою реакцією — WARNING активує лише мигалку, ERROR активує сирену — дозволяє оператору відразу оцінити терміновість реагування без погляду на екран. Логування всіх статусів у Prometheus з візуалізацією в Grafana надає замовнику інструмент для аналізу виробничих показників у часі: можна відстежити частоту відхилень по конкретних лініях, порівняти продуктивність різних змін або виявити закономірності в появі певних типів помилок.

Поточна конфігурація охоплює чотири з шести виробничих ліній підприємства — лінії 3 і 6 залишаються неактивними і не потребують моніторингу на поточному етапі. Завдяки конфігураційному підходу підключення нових ліній не потребує змін у коді: достатньо додати відповідний запис у JSON-конфігурацію і перезапустити контейнери. Серед напрямків подальшого розвитку системи найбільш перспективними є: дослідження адаптивних методів бінаризації для зниження залежності від освітлення; розширення набору метрик Prometheus для більш детального аналізу виробничих показників; архівування кадрів при виникненні помилок для подальшого розслідування причин відхилень; та підключення ліній 3 і 6 в міру їх введення в експлуатацію.

4.7 Висновки до розділу 4

У четвертому розділі описано методологію та результати тестування IoT-платформи на двох рівнях: попереднє локальне тестування на обладнанні розробника та інтеграційне тестування на реальному виробничому об'єкті. Попереднє вивчення обладнання вдома і тест алгоритму на веб-камері зекономили багато часу на фабриці — більшість очевидних проблем вже була виправлена. Це суттєво скоротило час інтеграційного тестування і дозволило зосередитись на специфічних проблемах реального виробничого середовища.

Система успішно пройшла перевірку всіх ключових сценаріїв на виробничому обладнанні. Одночасна робота чотирьох виробничих ліній із п'ятьма камерами та дев'ятьма I/O-пристроями (72 дискретних входи) підтвердила стабільність багатопотокової архітектури. Час реагування системи на виникнення помилки — від зміни стану пристрою до відображення на дашборді — не перевищує 0.3 секунди для камер, що відповідає вимогам виробничого моніторингу в режимі реального часу. Автоматичне відновлення

після втрати мережевого з'єднання та перезавантаження сервера підтверджено на практиці, включно зі сценаріями повного відключення електроживлення.

В процесі тестування виявлено та усунено ряд нетривіальних проблем. Критичним виявився баг з параметром use_reloader=True у Flask, що спричиняв дублювання процесу і розділення спільних змінних. Механізм відновлення RTSP-з'єднання пройшов через три ітерації вдосконалення. Проблема кешування кадрів у браузері вирішена через унікальний timestamp у URL запитів, а зависання JavaScript при тривалій роботі — через щохвилинне примусове перезавантаження сторінки. Несумісність TCP-контролера застарілої версії прошивки з коректним форматом Modbus-відповідей виявлена і усунена заміною обладнання. Ізольоване вивчення апаратної частини до виїзду на фабрику дало розуміння специфіки конкретних пристроїв, що виявилось вирішальним при діагностиці складних апаратно-програмних проблем.

Виявлено та задокументовано практичне обмеження модуля комп'ютерного зору: залежність класичної порогової бінаризації від умов зовнішнього освітлення в умовах виробничого цеху з природним освітленням. Проблему частково вирішено апаратним способом через встановлення контрастних підкладок. Після запуску на підприємстві система працює: чотири лінії під наглядом з одного монітора, сигналізація спрацьовує коректно, замовник аналізує статистику через Grafana.

ВИСНОВКИ

У дипломній роботі виконано проектування та розробку контейнеризованої IoT-платформи для автоматизації та моніторингу виробничої лінії з використанням комп'ютерного зору. Система розроблена для реального підприємства з виробництва пластикових виробів, впроваджена в дослідну експлуатацію та підтвердила свою практичну ефективність. В результаті виконаної роботи отримано такі результати:

1. Проведено аналіз предметної області та існуючих рішень для промислового IoT-моніторингу. Встановлено, що комерційні платформи (Siemens MindSphere, PTC ThingWorx, SCADA-системи) є надмірно складними та дорогими для підприємств малого і середнього масштабу, а відкриті рішення (Node-RED, OpenHAB) не мають вбудованої підтримки необхідних промислових протоколів і методів комп'ютерного зору. Це обумовило доцільність розробки власного рішення. Обґрунтовано вибір технологічного стеку: Python, OpenCV, Flask, socket, prometheus-client, Docker та Docker Compose.
2. Спроектовано загальну архітектуру системи на основі чотирьох Docker-контейнерів: main_program (обробка зображень, веб-інтерфейс, агрегація даних), tcps (Modbus TCP взаємодія з промисловим обладнанням), prometheus та grafana. Спроектовано дворівневу систему конфігурації на базі двох JSON-файлів з механізмом динамічного оновлення параметрів без перезапуску потоків. Розроблено алгоритм комп'ютерного зору для контролю геометричних параметрів продукції на основі аналізу одновимірного профілю вздовж контрольної лінії з класифікацією статусів OK, WARNING та ERROR.
3. Реалізовано два незалежних Python-застосунки з модульною структурою коду. У застосунку main_program реалізовано: захоплення відеопотоку через RTSP, попередню обробку кадрів (crop, контраст), алгоритм analyze_frame та функцію get_frame з двома режимами відображення

(threshold та нормальний), Flask веб-інтерфейс з дашбордом реального часу, MJPEG-потоком, сторінками налаштувань камер і датчиків. У застосунку tcps реалізовано власний клас Modbus TCP на базі socket, чотири спеціальні обробники нестандартних датчиків: буферна фільтрація (Vacuum_Error, ≥3 сек), затримка відновлення (Chiller_Error, 30 сек), апаратний перемикач режиму (SlaveID=1/Address=4) та Blue Button з threading.Timer(180). Контейнеризацію виконано з політикою restart: unless-stopped та автозапуском через systemd.

4. Реалізовано надійну багатопотокову архітектуру із застосуванням threading.Lock для захисту спільних змінних, threading.Event (stop_flag) для координації завершення потоків та daemon=True для автоматичного завершення при зупинці процесу. Розроблено механізм відновлення RTSP-з'єднання у три ітерації: від базового try/except до фінальної реалізації з явним звільненням ресурсів cap.release() та подвійною перевіркою відновленого з'єднання. Аналогічний механізм реалізовано для TCP-з'єднань у сервісі tcps. Налаштовано публікацію метрик у Prometheus: п'ять бінарних Gauge-метрик на статус кожної камери та бінарні метрики для кожного дискретного входу датчика, включно з нефільтрованими значеннями для аналітики.
5. Проведено тестування системи у два етапи. На локальному етапі модуль комп'ютерного зору тестувався з використанням веб-камери та фізичного об'єкта, що дозволило виміряти час реагування системи — не більше 0.3 секунди. Перші I/O-пристрої та TCP-контролери вивчались вдома, що забезпечило глибоке розуміння специфіки обладнання до виїзду на виробництво. На виробничому етапі система успішно пройшла тестування з чотирма активними лініями, п'ятьма камерами та дев'ятьма I/O-пристроями (72 дискретних входи). Виявлено та усунено ряд нетривіальних проблем: критичний баг з Flask use_reloader=True, що спричиняв дублювання процесу і розділення спільних змінних (виявлення зайняло ~16 годин); проблему кешування кадрів у браузері (вирішено

через timestamp у URL та щохвилинне перезавантаження сторінки); несумісність TCP-контролера з застарілою прошивкою (замінено обладнання). Виявлено практичне обмеження класичної порогової бінаризації — залежність від умов освітлення — частково вирішене встановленням контрастних підкладок.

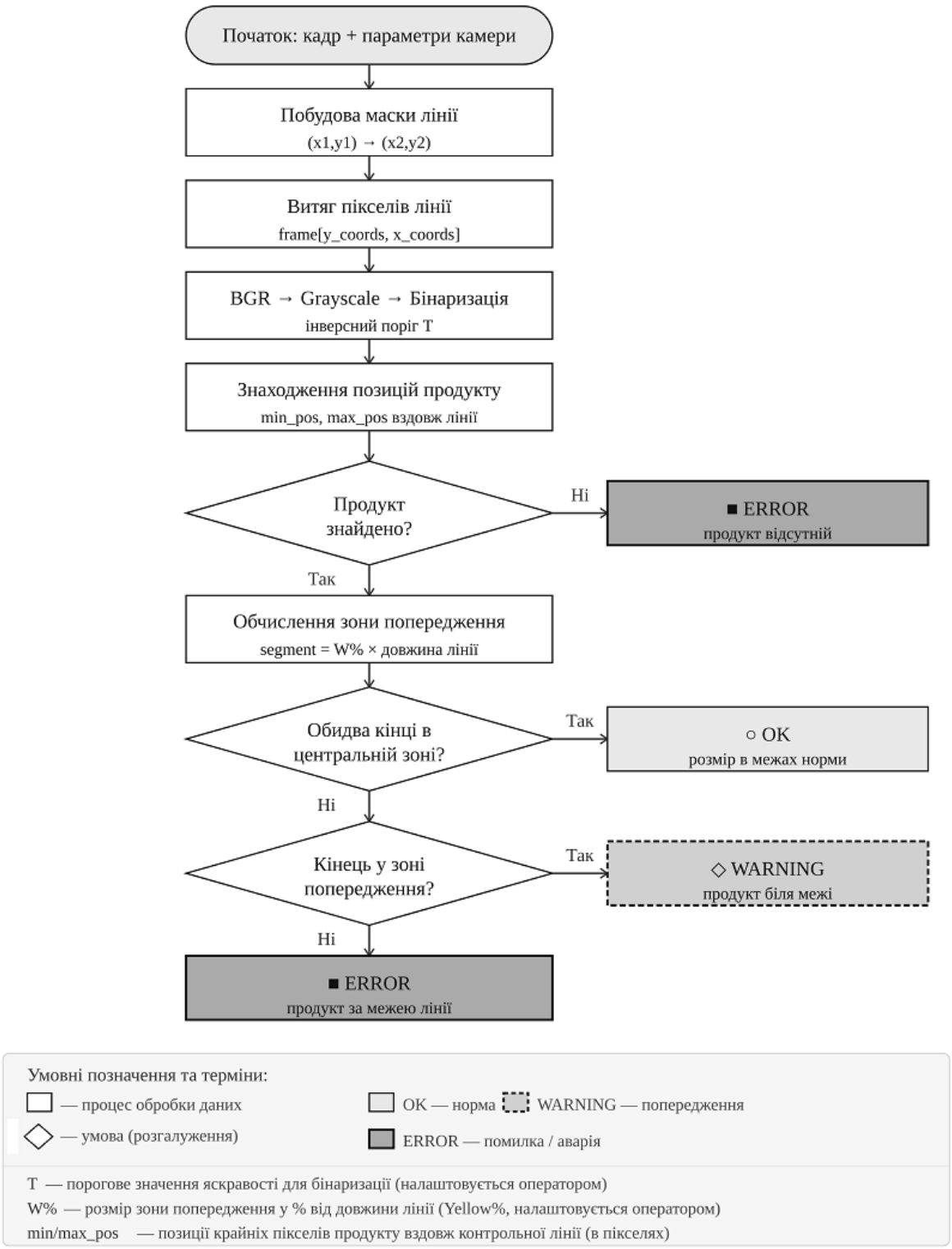
Поставлені задачі виконано. Система реально працює на виробництві: чотири лінії під постійним контролем, помилки виявляються автоматично, а накопичена статистика в Prometheus дає замовнику матеріал для аналізу. Найбільші труднощі були пов'язані не з алгоритмічною складністю, а з прихованими особливостями інструментів і реального виробничого середовища — баг Flask use_reloader=True, поведінка OpenCV при втраті RTSP-з'єднання, паралельний монтаж апаратури і програмне налагодження одночасно. Головний висновок простий: чим ретельніше перевірений кожен компонент окремо до інтеграції — тим менше часу витрачається на діагностику коли все зібрано разом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Schwab K. The Fourth Industrial Revolution. — Geneva : World Economic Forum, 2016. — 192 p.
2. Boyes H., Hallaq B., Cunningham J., Watson T. The industrial internet of things (IIoT): An analysis framework. // Computers in Industry. — 2018. — Vol. 101. — P. 1–12.
3. Lasi H., Fettke P., Kemper H.-G., Feld T., Hoffmann M. Industry 4.0 // Business & Information Systems Engineering. — 2014. — Vol. 6, No. 4. — P. 239–242.
4. Zhong R. Y., Xu X., Klotz E., Newman S. T. Intelligent Manufacturing in the Context of Industry 4.0: A Review // Engineering. — 2017. — Vol. 3, No. 5. — P. 616–630.
5. Bradski G., Kaehler A. Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library. — Sebastopol : O'Reilly Media, 2016. — 1024 p.
6. Szeliski R. Computer Vision: Algorithms and Applications. 2nd ed. — Cham : Springer, 2022. — 925 p.
7. Canny J. A computational approach to edge detection // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 1986. — Vol. 8, No. 6. — P. 679–698.
8. Kang G., Gao S., Yu L., Zhang D. Deep Learning-Based Plant Defect Detection // Sensors. — 2020. — Vol. 20, No. 23. — P. 6858.
9. Modbus Application Protocol Specification V1.1b3 [Електронний ресурс]. — Modbus Organization, 2012. — Режим доступу: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf (дата звернення: 01.03.2025).
10. Modbus Messaging on TCP/IP Implementation Guide V1.0b [Електронний ресурс]. — Modbus Organization, 2006. — Режим доступу: https://modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf (дата звернення: 01.03.2025).
11. Kane S., Matthias K. Docker: Up & Running. 3rd ed. — Sebastopol : O'Reilly Media, 2023. — 342 p.

12. Docker Documentation [Електронний ресурс]. — Docker Inc., 2025. — Режим доступу: https://docs.docker.com (дата звернення: 15.02.2025).
13. Python Software Foundation. Python 3 Documentation [Електронний ресурс]. — 2025. — Режим доступу: https://docs.python.org/3 (дата звернення: 10.02.2025).
14. Grinberg M. Flask Web Development: Developing Web Applications with Python. 2nd ed. — Sebastopol : O'Reilly Media, 2018. — 316 p.
15. OpenCV Documentation [Електронний ресурс]. — OpenCV team, 2025. — Режим доступу: https://docs.opencv.org (дата звернення: 10.02.2025).
16. Brazil B. Prometheus: Up & Running. 2nd ed. — Sebastopol : O'Reilly Media, 2023. — 448 p.
17. Prometheus Documentation [Електронний ресурс]. — The Prometheus Authors, 2025. — Режим доступу: https://prometheus.io/docs (дата звернення: 20.02.2025).
18. Grafana Documentation [Електронний ресурс]. — Grafana Labs, 2025. — Режим доступу: https://grafana.com/docs (дата звернення: 20.02.2025).
19. Tao F., Qi Q., Liu A., Kusiak A. Data-driven smart manufacturing // Journal of Manufacturing Systems. — 2018. — Vol. 48. — P. 157–169.
20. Wan J., Tang S., Li D., Wang S., Liu C., Abbas H., Vasilakos A. V. A Manufacturing Big Data Solution for Active Preventive Maintenance // IEEE Transactions on Industrial Informatics. — 2017. — Vol. 13, No. 4. — P. 2039–2047.

ДОДАТОК А. БЛОК-СХЕМА АЛГОРИТМУ АНАЛІЗУ ЗОБРАЖЕННЯ



					ДП.АКІТ.10658664.00.00.00.000 ПЗ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Музика С.Р.				Літ.	Арк.	Акрушів
Перевір.		Николайчук Я.М.					91	91
Консульт..						ЗУНУ,ФКІТ,АКІТ-41		
Н. Контр.		Заставний О. М						
Затверд.		Сегін А.І.						