НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	**Завідувач кафедри комп'ютерних наук**
_____ **Ігор БОЛБОТ**	_____ **Белла ГОЛУБ**
(підпис)	(підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему: «Програмне забезпечення системи моніторингу параметрів безпеки шахти»
Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент _____ **Ганна ВАЙГАНГ**
(підпис)

Керівник бакалаврської кваліфікаційної роботи **Віктор КИРИЧЕНКО**
_____
(підпис)
к.ф.-м.н., доцент

Виконав _____ **Назар ГАНЮК**
(підпис)

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук

к.т.н., доцент _____ Белла ГОЛУБ
(підпис)

«____» _____ 2025 р..

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Ганюку Назару Олександровичу
(прізвище, ім'я та по батькові)

Спеціальність «121 Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»
Тема бакалаврської кваліфікаційної роботи
«Програмне забезпечення системи моніторингу параметрів безпеки шахти»
затверджена наказом від «19» 12 2025 р. № 3196 «С»

Термін подання завершеної роботи на кафедру «___» _____ 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи: _____

Перелік питань, що підлягають дослідженню: 1.Аналіз галузі та ТЗ. 2.Проєктування архітектури. 3.Розробка БД та бекенду (Python/Django). 4.Реалізація алгоритмів роумінгу та тривог. 5. Розробка інтерфейсу з 2D-картою. 6.Тестування та розгортання системи на VPS.

Перелік графічних документів (за необхідності).

Дата видачі завдання «19» 12 2025р.

Керівник бакалаврської кваліфікаційної роботи _____ **Віктор КИРИЧЕНКО**
(підпис)

Завдання прийняв до виконання _____ **Назар ГАНЮК**
(підпис)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 5

ВСТУП 6

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 9

1.1 Опис предметної області 9

1.2 Аналіз вимог до програмної системи 10

1.2.1 Обмеження та припущення 12

1.3 Моделювання предметної області 13

1.4 Огляд інформаційних джерел та наявних рішень 15

1.5 Постановка завдання 19

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 21

2.1 Логічна модель даних 21

2.2 Проєктування архітектури та програмних компонентів 24

2.3 Проєктування алгоритмів роботи програмних модулів 26

2.4 Проєктування об’єктно-орієнтованої структури (UML – діаграма класів) 29

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 32

3.1 Система управління інформаційною базою 32

3.2 Розробка інформаційної бази 35

3.3 Вибір інструментарію для створення прикладного програмного забезпечення 37

3.4 Алгоритмізація та програмування програмних модулів 42

3.5 Конструктивне виконання та інтеграція апаратного модуля 50

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ 53
4.1 Тестування системи 53
4.2 Вимоги до апаратного та програмного забезпечення 58
4.3 Склад інсталяційного пакету 60
ВИСНОВКИ 65
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 67
ДОДАТОК А 69
ДОДАТОК Б 76
ДОДАТОК В 80
ДОДАТОК Д 81

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних.

ГДК – гранично допустима концентрація.

ПЗ – програмне забезпечення.

СУБД – система управління базами даних.

API – Application Programming Interface (інтерфейс прикладного програмування).

IoT – Internet of Things (інтернет речей).

LEL – Lower Explosive Limit (нижня межа вибухонебезпечності).

OTA – Over-The-Air (технологія оновлення програмного забезпечення повітрям).

RSSI – Received Signal Strength Indicator (показник рівня прийманого сигналу).

SPA – Single Page Application (односторінковий вебдодаток).

UI – інтерфейс користувача.

UX – досвід взаємодії користувача з системою.

Wi-Fi – технологія бездротової локальної мережі.

WSGI – стандарт взаємодії між Python-програмою та вебсервером.

ВСТУП

Актуальність теми. Гірничодобувна промисловість – це завжди ризик. Безпека шахтаря тримається на тому, чи працює моніторинг без перебоїв. У реальних шахтах ситуація гірша: газоаналізатори живуть своїм життям, вентиляція – своїм, зв'язок часто рветься. До того ж більшість систем закриті (proprietary). Якщо купили в одного постачальника – він і диктує умови на роки вперед.

Підземні виробки – це місце, де радіосигнал зникає не «іноді», а постійно. У критичний момент диспетчер може просто не дочекатися даних.

Саме тому ми взялися за відкриту систему на базі Глибина 4.0. Не черговий «зоопарк» із пропрієтарних модулів, а єдиний IoT-каркас, який працює під землею.

Створення відкритої архітектури, що поєднує IoT-модулі для збору персональної телеметрії, механізми безшовного роумінгу (Wi-Fi roaming) та відмовостійкі сервери для обробки даних у реальному часі, дозволить не лише фіксувати інциденти, але й ефективно запобігати їм завдяки алгоритмам інтелектуального аналізу тривог.

Мета розробки. Мета роботи – спроєктувати і розробити програмне забезпечення «Глибина 4.0». Система має робити такі речі одночасно:

- збирати телеметрію без втрат, навіть коли зв'язок рветься;
- показувати на карті, де зараз кожен гірник і чи все гаразд;
- автоматично бити тривогу при перевищенні метану, втраті зв'язку або SOS-сигналі.

Без ручного втручання. У реальному часі.

Методи та технології. Для досягнення поставленої мети та реалізації програмного комплексу використано сучасний стек технологій. Бекенд ми писали на Python з Django [5]. Чому? Бо швидко збирати прототип, і під реальне навантаження теж вивозить.

Дані телеметрії – це часові ряди. Їх дуже багато. Стандартні B-tree індекси роздувають базу до непристойних розмірів. Тому взяли PostgreSQL + BRIN-індекси [11]. Вони значно компактніші, хоча трохи повільніші на точних вибірках. Для моніторингу – добре.

Кеш – Redis [15]. Без нього при 50+ працівниках карта б мала затримку.

Клієнтська частина (Front-end) розроблена з використанням HTML5, CSS3 та JavaScript. Карту ми малюємо самі, через Canvas API. Чому не взяли готову бібліотеку типу Leaflet? Тому що треба було повний контроль над анімацією руху людей під землею. Готові рішення плавали при 200+ об'єктах. Наша flyTo на *requestAnimationFrame* дає плавний зум/пан без мікролагів. Для аналітичних звітів використано *Chart.js*.

Апаратний IoT-рівень системи побудований на базі мікроконтролерів ESP32 (мова C/C++) із реалізацією алгоритмів збереження даних у локальну чергу (offline queue) при втраті зв'язку та підтримкою бездротових оновлень (OTA). Захист вебінтерфейсу реалізовано за допомогою Cloudflare Turnstile та криптографічних перевірок паролів.

Апробація результатів роботи. Основні положення, результати досліджень та архітектурні рішення бакалаврської кваліфікаційної роботи демонструвались та обговорювалися на науково-практичній конференції. За матеріалами кваліфікаційної роботи підготовлено до публікації тези доповіді: Ганюк Н.О., Кириченко В.В. «Програмне забезпечення системи моніторингу параметрів безпеки шахти».

Структура записки. Пояснювальна записка складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі проведено системний аналіз предметної області, досліджено наявні рішення для моніторингу безпеки в шахтах та сформовано вимоги до програмної системи. У другому розділі виконано логічне та фізичне проєктування інформаційного і програмного забезпечення, розроблено ER-діаграму бази даних, діаграми класів, пакетів та компонентів. У третьому розділі описано процес розробки програмно-апаратного комплексу, обґрунтовано вибір

технологій (Django, PostgreSQL, ESP32), наведено алгоритми обробки телеметрії, Wi-Fi роумінгу та розумних тривог. У четвертому розділі наведено рекомендації щодо впровадження системи, описано процеси тестування (включаючи симуляцію навантаження) та вимоги до розгортання комплексу. Загальний обсяг роботи становить [81] сторінок машинописного тексту. Робота містить [22] рисунок, [8] таблиць та [16] джерел за списком літератури.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Гірничодобувна промисловість належить до галузей з найвищим рівнем виробничої небезпеки. Підземне середовище створює екстремальні умови праці, де життя та здоров'я гірників щомиті залежать від стану навколишнього середовища. Найбільшу загрозу становлять раптові викиди вибухонебезпечних газів – передусім метану (CH_4) та чадного газу (CO). Також небезпеку створюють обвали породи, пожежі та раптове затоплення виробок. У таких умовах швидкість реакції диспетчерської служби та наявність точних координат кожної людини під землею стають ключовими факторами для успішного проведення рятувальних операцій.

Аналіз сучасного стану українських вугледобувних підприємств показує значну технологічну нерівномірність. На окремих високорентабельних приватних шахтах східного регіону впроваджено комплексні автоматизовані системи моніторингу. Однак більшість підприємств, особливо в західному регіоні України, продовжують експлуатувати розрізнені та морально застарілі системи. Класичний підхід має фундаментальний недолік: газоаналізатори функціонують окремо, системи зв'язку – окремо. Крім того, наявні системи позиціювання базуються на зчитуванні RFID-міток лише на контрольних точках. Це дає диспетчеру інформацію про те, що працівник зайшов у певну зону (штрек), але не забезпечує щосекундного розуміння його точних координат. Традиційні засоби не здатні автоматично фіксувати стан тривалої бездіяльності працівника або його раптове падіння. Відсутність функції контролю рухової активності («Man Down») є критичним недоліком у випадку втрати свідомості гірником через отруєння чадним газом чи травмування. Ситуація додатково ускладнюється специфікою підземних виробок, де регулярно виникають обриви мережевих кабелів та локальні знеструмлення.

Іншою вагомою проблемою є монополізація ринку безпекових систем великими гравцями із закритими пропрієтарними протоколами передачі даних. Це створює так званий «vendor lock-in» – жорстку прив'язку підприємства до одного постачальника. Запровадивши таку систему, шахта втрачає можливість інтегрувати дешевші або більш сучасні датчики від сторонніх виробників. Як наслідок, модернізація інфраструктури стає економічно недоцільною та повільною.

Вирішенням описаного комплексу проблем є перехід до відкритих кіберфізичних систем у рамках парадигми Глибина 4.0. Сучасна система моніторингу має будуватися як єдина екосистема. Портативні IoT-пристрої, інтегровані в екіпірування гірника, збирають персональну телеметрію, відстежують рухову активність та використовують розгалужену мережу Wi-Fi репітерів для маршрутизації даних. При втраті зв'язку передбачається локальне збереження телеметрії з подальшою синхронізацією. Централізована серверна обробка даних дозволить не лише відображати стан шахти на інтерактивній карті, а й використовувати алгоритми розумного групування тривог для запобігання аваріям на ранніх стадіях.

1.2 Аналіз вимог до програмної системи

З огляду на специфіку предметної області та виявлені недоліки наявних систем, до програмного комплексу висувається ряд вимог. Їх доцільно розділити на функціональні та нефункціональні.

Функціональні вимоги визначають поведінку системи та перелік процесів, які вона має автоматизувати. Система логічно поділяється на три рівні, для кожного з яких сформульовано окремі вимоги.

Апаратний IoT-рівень

Модулі телеметрії на базі мікроконтролерів ESP32 забезпечують безперервний збір таких показників: рівень концентрації метану (CH_4), температура, вологість, стан заряду акумулятора, рівень прийманого сигналу

(RSSI). Пристрої оснащуються вбудованим акселерометром, що дає змогу відстежувати рухову активність та фіксувати раптове падіння працівника або його тривалу бездіяльність.

Важливою вимогою є наявність локальної енергонезалежної пам’яті (офлайн-черги). Це необхідно для буферизації зібраної телеметрії в разі обриву зв'язку з репітером – після відновлення з'єднання дані автоматично надсилаються на сервер. Крім того, передбачається підтримка оновлення програмного забезпечення «по повітрю» (OTA Updates) з механізмом часткового розгортання (Staged Rollout) для обраних тестових груп пристроїв.

Серверний рівень (бізнес-логіка):

Серверна частина має забезпечувати приймання та валідацію даних від IoT-модулів через REST API. Ключовим компонентом є модуль розумних тривог (Alert Engine), який автоматично виявляє аномалії та виконує дедуплікацію масових інцидентів. Наприклад, якщо одночасно втрачається зв'язок з кількома гірниками на одній ділянці, сервер генерує одну глобальну тривогу «Аварія інфраструктури (знеструмлено репітер)» замість десятків індивідуальних попереджень.

Окремою вимогою є суворий апаратний контроль життєвого циклу зміни. Статус завершення роботи встановлюється виключно тоді, коли пристрій працівника підключається до наземного базового репітера (AP-SURFACE) та встановлюється на зарядку в ламповій. Програмне забезпечення блокує здачу пристрою, якщо за гірником залишається активний критичний інцидент, який не закрито диспетчером.

Диспетчерський (клієнтський) рівень:

Диспетчерський інтерфейс має відображати стан шахти на інтерактивній 2D-карті в режимі реального часу. Реалізація виконується за принципами часткового SPA (Single Page Application) без перезавантаження сторінки. При надходженні критичних SOS-сигналів система автоматично генерує апаратну

звукову сирену (Web Audio API) та запускає візуальну пульсацію аварійних зон на карті. Також передбачено можливість перегляду історії інцидентів та автоматичну генерацію офіційних звітів у форматах PDF та CSV.

Нефункціональні вимоги визначають атрибути якості програмного продукту – надійність, швидкодію та безпеку.

Відмовостійкість та оптимізація високих навантажень. Серверна частина повинна витримувати інтенсивний потік даних від сотень датчиків. Для цього використовується in-memory кешування (Redis), що забезпечує швидку віддачу актуальних координат. Крім того, у СУБД PostgreSQL застосовуються спеціалізовані блокові індекси (BRIN) для компактного та швидкого збереження часових рядів телеметрії.

Інформаційна безпека. Доступ до API-ендпоінтів для ESP32 захищається статичними криптографічними ключами.

Захист вебінтерфейсу. Диспетчерська панель має багаторівневий захист: інтеграцію з алгоритмами розпізнавання ботів Cloudflare Turnstile, перевірку паролів на стійкість до зламів (інтеграція з базою джерел HIBP) та систему автоматичних email-сповіщень адміністратора при спробі входу в обліковий запис із невідомої IP-адреси або нового браузера.

Управління життєвим циклом даних (DLM). Телеметрія генерує значні обсяги даних, тому система підтримує фонові завдання (Cron) для автоматичної архівації та стиснення застарілих логів (старших за 30 днів). В оперативній базі залишаються лише дані щодо зафіксованих інцидентів.

1.2.1 Обмеження та припущення

Розроблена система «Глибина 4.0» орієнтована на роботу в умовах підземних виробок. При цьому її функціональність безпосередньо залежить від якості бездротового покриття. При значних згасаннях сигналу або повній відсутності Wi-Fi зв’язку телеметрія може надходити із затримкою. Для

мінімізації втрат даних на рівні IoT-модуля передбачено накопичення подій у локальній черзі з наступною синхронізацією після відновлення зв’язку.

Іншим обмеженням є точність позиціювання працівника за Wi-Fi параметрами (RSSI/BSSID). Такий підхід забезпечує достатню якість для оперативного диспетчерського контролю в межах зон покриття, але не дає сантиметрової точності. Крім того, він може вимагати калібрування карти або опорних точок для конкретної шахти.

Слід також враховувати, що датчики газу мають власну похибку та залежать від умов експлуатації (температура, вологість, старіння сенсора). Тому в системі передбачено періодичне калібрування та контроль порогів спрацювання.

Система містить механізми фільтрації та дедуплікації тривог. Однак повністю усунути ризик хибних спрацьовувань неможливо. Причинами можуть бути шум сенсора, короткочасні стрибки сигналу, механічні вібрації або нестандартна поведінка користувача. У таких випадках критичні події вимагають підтвердження диспетчером і додаткового аналізу історії телеметрії.

1.3 Моделювання предметної області

Для кращого розуміння функціональних вимог та логіки роботи системи виконано моделювання предметної області за допомогою діаграми прецедентів (Use-Case diagram) мовою UML (рис. 1.1). У розробленій кіберфізичній системі доцільно розділити ролі на людських користувачів та апаратних клієнтів. Виділено чотирьох основних акторів: «Диспетчер», «Адміністратор», «Гірник» та апаратний актор «Модуль ESP32».

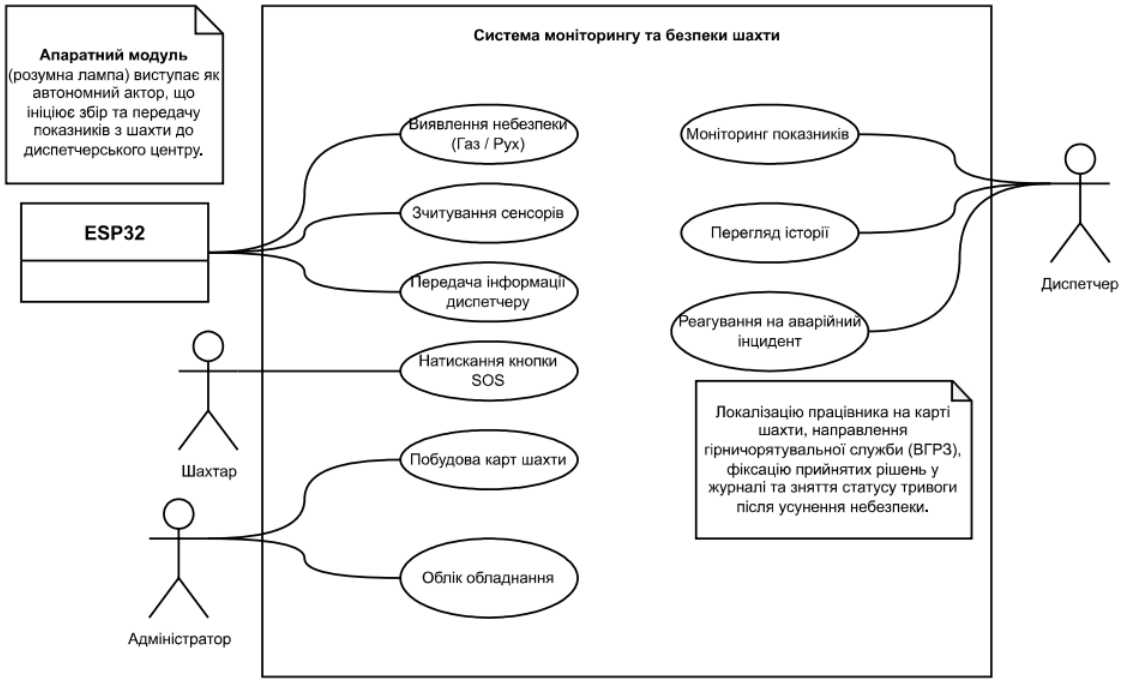

Рис. 1.1 Діаграма прецедентів кіберфізичної системи моніторингу

Актор **«Диспетчер»** є головним оперативним користувачем вебінтерфейсу системи. Його мета – безперервний моніторинг безпеки. До прецедентів диспетчера належать: безпечна авторизація в системі із захистом від автоматизованих атак (Turnstile); візуальний моніторинг переміщення персоналу та стану мікроклімату на інтерактивній 2D-карті; обробка інцидентів безпеки (перегляд деталей тривог, зміна статусів на «В процесі» або «Вирішено», внесення нотаток про вжиті заходи); а також генерація та вивантаження офіційних звітів у форматі PDF.

Актор **«Адміністратор»** відповідає за технічне обслуговування та конфігурацію системи на рівні бази даних і серверних налаштувань. Його прецеденти включають: управління парком обладнання (реєстрація нових модулів ESP32 та Wi-Fi репітерів, прив'язка MAC-адрес); конфігурація облікових записів персоналу та їхніх рівнів доступу; завантаження та оновлення топології шахти у вигляді JSON-карт; управління життєвим циклом прошивок (завантаження нових версій та запуск OTA-оновлень).

Актор **«Гірник»** – це кінцевий користувач, життя та безпека якого контролюються системою. Взаємодія цього актора з системою відбувається опосередковано через ношене обладнання. Прецеденти гірника: генерація рухової активності (відстежується системою як підтвердження життєдіяльності); ручна активація та деактивація сигналу SOS за допомогою апаратної кнопки на пристрої; завершення зміни – здача пристрою в зону дії базового репітера AP-SURFACE на поверхні.

Актор **«Модуль ESP32»** – це розумний IoT-пристрій, який виступає зовнішнім апаратним клієнтом для сервера (REST API). До його прецедентів належать: безперервний збір та надсилання телеметричних даних (рівень газу, температура, заряд батареї, показники акселерометра, рівень Wi-Fi сигналу); робота в офлайн-режимі (локальне накопичення логів у Flash-пам'яті при втраті з'єднання з наступною синхронізацією); виконання команд сервера (наприклад, примусове перезавантаження); отримання та встановлення оновлень програмного забезпечення «по повітрю» (OTA Updates).

Такий розподіл функцій наочно демонструє взаємодію між фізичним та програмним рівнями системи, забезпечуючи високий рівень інформаційної безпеки та керованості комплексу.

1.4 Огляд інформаційних джерел та наявних рішень

Для обґрунтування доцільності розробки програмно-апаратного комплексу «Глибина 4.0» виконано аналіз наявних підходів та систем моніторингу безпеки на гірничодобувних підприємствах. Розглянуто чотири класи рішень, кожен з яких оцінено за перевагами та недоліками.

1. Традиційні стаціонарні системи газового захисту (наприклад, УТАС). Цей клас систем базується на магістральних дротових лініях зв'язку та стаціонарних датчиках, жорстко закріплених на стінках виробок.

Переваги: висока надійність базової інфраструктури за нормальних умов експлуатації, багаторічний досвід використання, наявність державних сертифікацій.

Недоліки: повна відсутність індивідуального моніторингу персоналу. При аварії (наприклад, обвалі породи) та обриві магістрального кабелю ділянка за місцем обриву стає «сліпою зоною». Система не визначає точне місцеперебування заблокованих гірників.

2. Системи локального позиціювання на базі RFID (наприклад, рішення компанії Strata). Працівники оснащуються пасивними або напівактивними RFID-мітками, а зчитувачі встановлюються на ключових розгалуженнях штреків.

Переваги: базовий зональний контроль – фіксація факту спуску в шахту та переходу між великими ділянками.

Недоліки: дискретність даних – позиціювання відбувається лише в момент проходження повз зчитувач. Якщо працівник травмується або втратить свідомість між двома зчитувачами, диспетчер не отримає сповіщень. Крім того, RFID-мітки не є вимірювальними приладами і не відстежують персональний рівень вибухонебезпечних газів (метану, чадного газу) безпосередньо в зоні дихання працівника [1].

3. Сучасні пропрієтарні IoT-комплекси (наприклад, Becker Mining Systems або MST). Комплексні рішення закритого типу, які пропонують зв'язок та позиціювання на базі власних бездротових стандартів.

Переваги: висока функціональність, інтеграція різних типів датчиків в єдину мережу, високий рівень промислового захисту обладнання.

Недоліки: надзвичайно висока вартість впровадження та жорстка проблема «vendor lock-in» (прив'язка до одного виробника). Закриті пропрієтарні протоколи унеможливлюють інтеграцію дешевших датчиків або компонентів від інших розробників, що суттєво гальмує модернізацію інфраструктури українських шахт.

4. Запропоноване рішення – кіберфізична система «Глибина 4.0». Програмний комплекс, що розробляється в межах даної кваліфікаційної роботи, проєктується як альтернатива закритим комерційним системам.

Переваги: використання доступної апаратної бази (мікроконтролери ESP32) та відкритих вебпротоколів (REST API), що усуває проблему монопольної залежності від постачальника. Система забезпечує щосекундне позиціювання (через аналіз рівня сигналу Wi-Fi RSSI) та безперервний персональний газовий контроль. Інноваційною перевагою є механізм локальних офлайн-черг (буферизація телеметрії у Flash-пам'ять при втраті зв'язку) та апаратний контроль завершення зміни (пристрій переводиться у статус відпочинку виключно в зоні дії репітера поверхні AP-SURFACE).

Недоліки: на етапі лабораторного прототипу апаратна частина потребує процедури промислової сертифікації на іскробезпеку (стандарт ATEX) перед фізичним спуском у вибухонебезпечне середовище вугільної шахти. Програмна архітектура при цьому повністю готова до інтеграції з сертифікованими корпусами.

Для систематизації результатів аналізу та наочного порівняння технічних характеристик наявних рішень із запропонованим комплексом «Глибина 4.0» сформовано зведену порівняльну таблицю (табл. 1.1). Оцінювання виконано за критеріями, які безпосередньо впливають на безпеку гірників та економічну доцільність впровадження системи на підприємстві.

Таблиця 1.1 Порівняльний аналіз систем моніторингу безпеки на гірничодобувних підприємствах

Критерій оцінювання	**Традиційні системи (напр., УТАС)**	**Системи на базі RFID (напр., Strata)**	**Пропрієтарні IoT (Becker, MST)**	**Запропонована система «Глибина 4.0»**
Базова архітектура та зв'язок	Дротова магістральна мережа	Бездротові зчитувачі в контрольних точках	Закрита бездротова екосистема	Відкрита бездротова мережа (Wi-Fi 2.4 ГГц)
Персональний газовий контроль	Відсутній (лише стаціонарні датчики)	Відсутній (мітки не мають газових сенсорів)	Присутній (залежить від комплектації)	**Безперервний** (датчики MQ-серії на кожному працівнику)

Критерій оцінювання	Традиційні системи (напр., УТАС)	Системи на базі RFID (напр., Strata)	Пропрієтарні IoT (Becker, MST)	Запропонована система «Глибина 4.0»
Точність позиціювання	Відсутнє	Дискретне зональне (тільки в момент проходу повз зчитувач)	Точне, у режимі реального часу	**Точне, у режимі реального часу** (на базі Wi-Fi RSSI)
Контроль життєдіяльності (руху)	Не підтримується	Не підтримується	Підтримується	**Підтримується** (апаратна фіксація іммобілізації через датчик SW-420)
Відмовостійкість (робота в офлайн)	Висока (до моменту обриву магістрального кабелю)	Низька (дані не збираються поза зоною зчитувачів)	Висока	**Висока** (автоматична буферизація телеметрії у Flash-пам'ять модуля)
Вартість впровадження та масштабування	Середня	Середня	Дуже висока	**Низька** (використання доступних мікроконтролерів ESP32)
Залежність від виробника (Vendor lock-in)	Жорстка прив'язка до закритого протоколу	Жорстка прив'язка до стандарту міток	Жорстка прив'язка (неможливість підключення сторонніх датчиків)	**Відсутня** (використання відкритих протоколів HTTP/REST API та JSON)

Як видно з таблиці 1.1, розроблюваний програмно-апаратний комплекс поєднує переваги сучасних IoT-систем промислового рівня (персональний контроль мікроклімату, точне позиціювання, контроль рухової активності) з концепцією відкритої архітектури. Це робить систему стійкою до проблеми «vendor lock-in» і дозволяє підприємствам гнучко масштабувати мережу безпеки без надмірних капіталовкладень.

1.5 Постановка завдання

На основі проведеного системного аналізу предметної області та виявлених недоліків наявних рішень сформульовано головне завдання бакалаврської кваліфікаційної роботи: спроєктувати та розробити програмне забезпечення кіберфізичної системи моніторингу параметрів безпеки шахти (робоча назва продукту – «Глибина 4.0»).

Для досягнення поставленої мети необхідно вирішити такий комплекс завдань:

1. Проєктування архітектури та бази даних. Розробити логічну та фізичну моделі реляційної бази даних (ER-діаграму) для збереження телеметрії та відомостей про персонал, а також UML-діаграми класів, пакетів та компонентів програмної системи.

2. Розробка серверної частини (Back-end). Створити серверну інфраструктуру на базі фреймворку Django з підтримкою високих навантажень. Для цього використовуються СУБД PostgreSQL (із застосуванням спеціалізованих BRIN-індексів для часових рядів) та механізми in-memory кешування Redis.

3. Алгоритмізація апаратного IoT-рівня. Розробити логіку роботи клієнтських модулів на базі мікроконтролерів ESP32. Реалізувати алгоритми безшовного Wi-Fi роумінгу, локальної енергонезалежної офлайн-черги для збереження телеметрії при втраті зв'язку та механізм оновлення програмного забезпечення «по повітрю» (OTA Updates).

4. Розробка модулів бізнес-логіки та управління даними. Створити програмний модуль розумних тривог (Alert Engine) для інтелектуальної дедуплікації аварійних повідомлень. Впровадити апаратний контроль завершення зміни через ідентифікацію базового репітера на поверхні (AP SURFACE). Реалізувати механізм управління життєвим циклом даних (Data Lifecycle Management) для автоматизованої архівації застарілої телеметрії.

5. Створення диспетчерського інтерфейсу (Front-end). Розробити інтерактивну 2D-карту топології шахти на базі технології HTML5 Canvas API (2D context) для відображення переміщення персоналу в реальному часі. Також розробити підсистему генерації аналітичних статистичних звітів у форматі PDF.

6. Забезпечення безпеки та підготовка до розгортання. Реалізувати багаторівневий захист API-ендпоінтів та диспетчерського вебінтерфейсу (інтеграція алгоритмів розпізнавання ботів Cloudflare Turnstile, криптографічна валідація паролів). Підготувати вимоги до інсталяційного пакета програмного комплексу.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних

Для збереження, обробки та аналізу даних кіберфізичної системи моніторингу обрано об'єктно-реляційну СУБД PostgreSQL. Такий вибір зумовлений її високою надійністю, підтримкою складних типів даних (зокрема JSONB [12] для збереження картографічної інформації) та наявністю спеціалізованих механізмів індексування для часових рядів (BRIN-індекси).

Схема бази даних має комплексну структуру і логічно поділяється на системний рівень (інфраструктура фреймворку) та рівень предметної області. На рис. 2.1 зображено загальну архітектуру бази даних системи. Сірим кольором виділено таблиці розробленої бізнес-логіки, іншими кольорами – системні таблиці фреймворку Django (керування сесіями, аутентифікацією) та сторонніх модулів захисту (наприклад, axes для блокування брутфорс-атак).

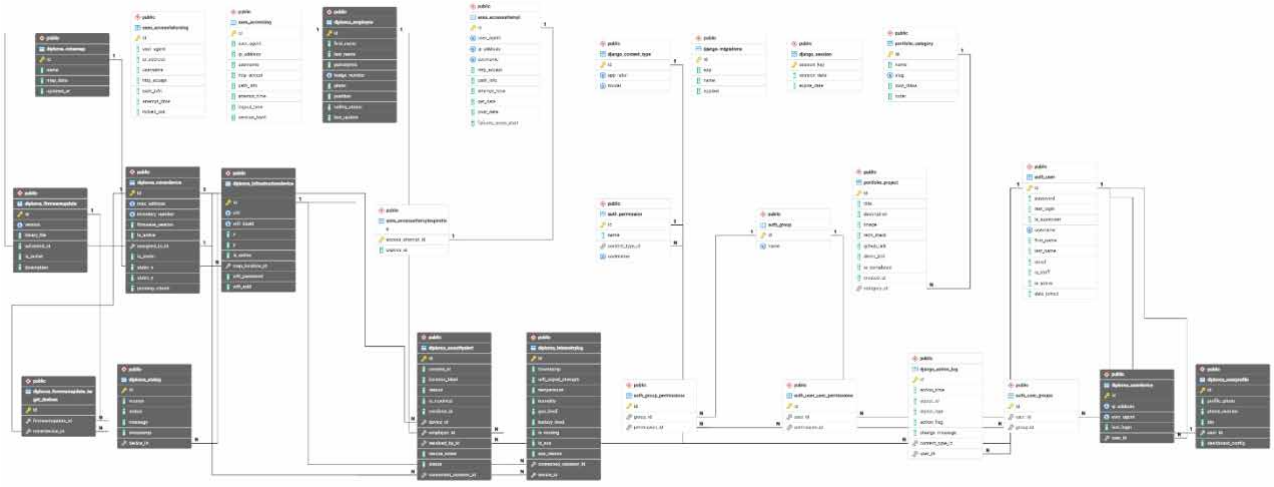

Рис. 2.1 ER-модель предметної області кіберфізичної системи

Для детального аналізу архітектури системи «Глибина 4.0» системні таблиці приховано. На рис. 2.2 наведено логічну модель даних, яка містить виключно розроблені сутності (з префіксом diploma_) та відображає зв'язки між ними.

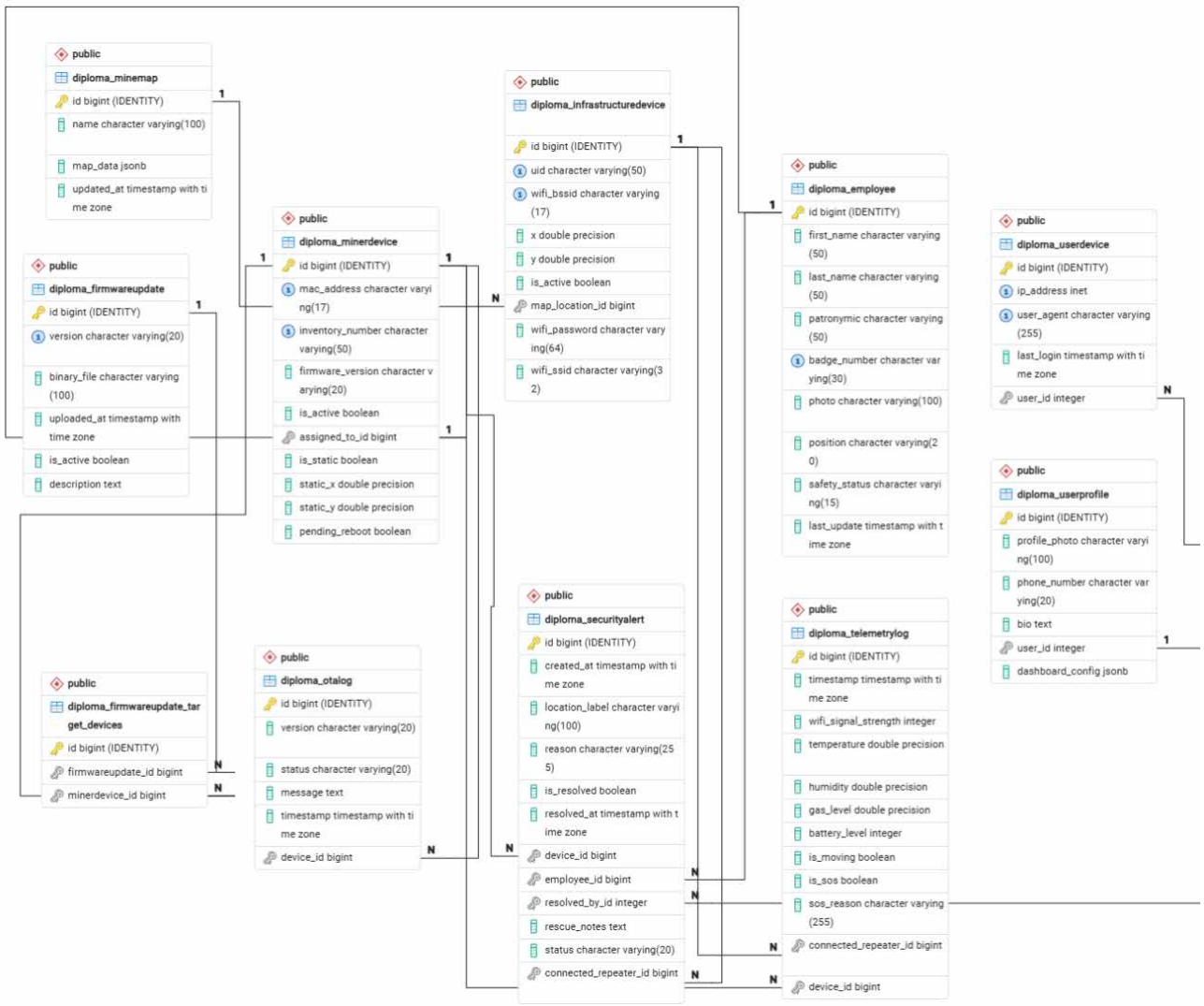

Рис. 2.2 ER-модель предметної області кіберфізичної системи

До головних сутностей бази даних належать такі таблиці:

1. **diploma_minemap** – зберігання картографічних даних. Містить назву горизонту та поле map_data (тип JSONB), де зберігається векторна топологія штреків для побудови 2D-карти у вебінтерфейсі.

2. **diploma_infrastructuredevice** – довідник стаціонарних вузлів зв'язку (Wi-Fi репітерів). Кожен запис містить координати розташування (x, y),

унікальний ідентифікатор та BSSID мережі. Таблиця пов'язана відношенням «один-до-багатьох» з картою шахти.

3. **diploma_employee** – відомості про персонал. Сутність зберігає ПІБ, згенерований табельний номер (жетон), посаду та поточний статус безпеки працівника (наприклад, OFF_SHIFT, OK, SOS).

4. **diploma_minerdevice** – реєстр апаратних IoT-модулів (касок гірників та стаціонарних газоаналізаторів). Пов'язана з таблицею diploma_employee відношенням «один-до-одного» (assigned_to_id), що гарантує закріплення одного мобільного пристрою за одним працівником на час зміни.

5. **diploma_telemetrylog** – наймасивніша таблиця системи. Вона зберігає часові ряди (Time-Series) телеметричних показників. Для оптимізації HighLoad-навантажень і швидкої вибірки історичних даних застосовано індексацію типу BRIN (Block Range Index) за полем timestamp, а також складений B-Tree індекс.

6. **diploma_securityalert** – журнал реєстрації аварійних подій. Зберігає інформацію про інциденти з прив'язкою до працівника, пристрою та найближчого репітера. Також відстежується життєвий цикл тривоги.

8. **diploma_firmwareupdate та diploma_otalog** – сутності для управління життєвим циклом вбудованого програмного забезпечення (OTA Updates). Вони зберігають бінарні файли прошивок та історію їх розгортання на ESP32.

9. **diploma_userprofile та diploma_userdevice** – сутності підсистеми безпеки. Перша розширює стандартну модель адміністратора (додаючи персональні налаштування). Друга забезпечує логування IP-адрес та User-Agent пристроїв, з яких виконується вхід, що дає змогу автоматично генерувати попередження про спроби несанкціонованого доступу.

Нормалізацію бази даних виконано до третьої нормальної форми (3NF). Це дозволило усунути дублювання даних та забезпечити цілісність посилань через механізми зовнішніх ключів (Foreign Keys).

2.2 Проєктування архітектури та програмних компонентів

Програмно-апаратний комплекс «Глибина 4.0» побудовано на принципах багаторівневої (n-tier) розподіленої архітектури. Для візуалізації структурних зв'язків між основними програмними та апаратними модулями розроблено UML-діаграму компонентів (рис. 2.3).

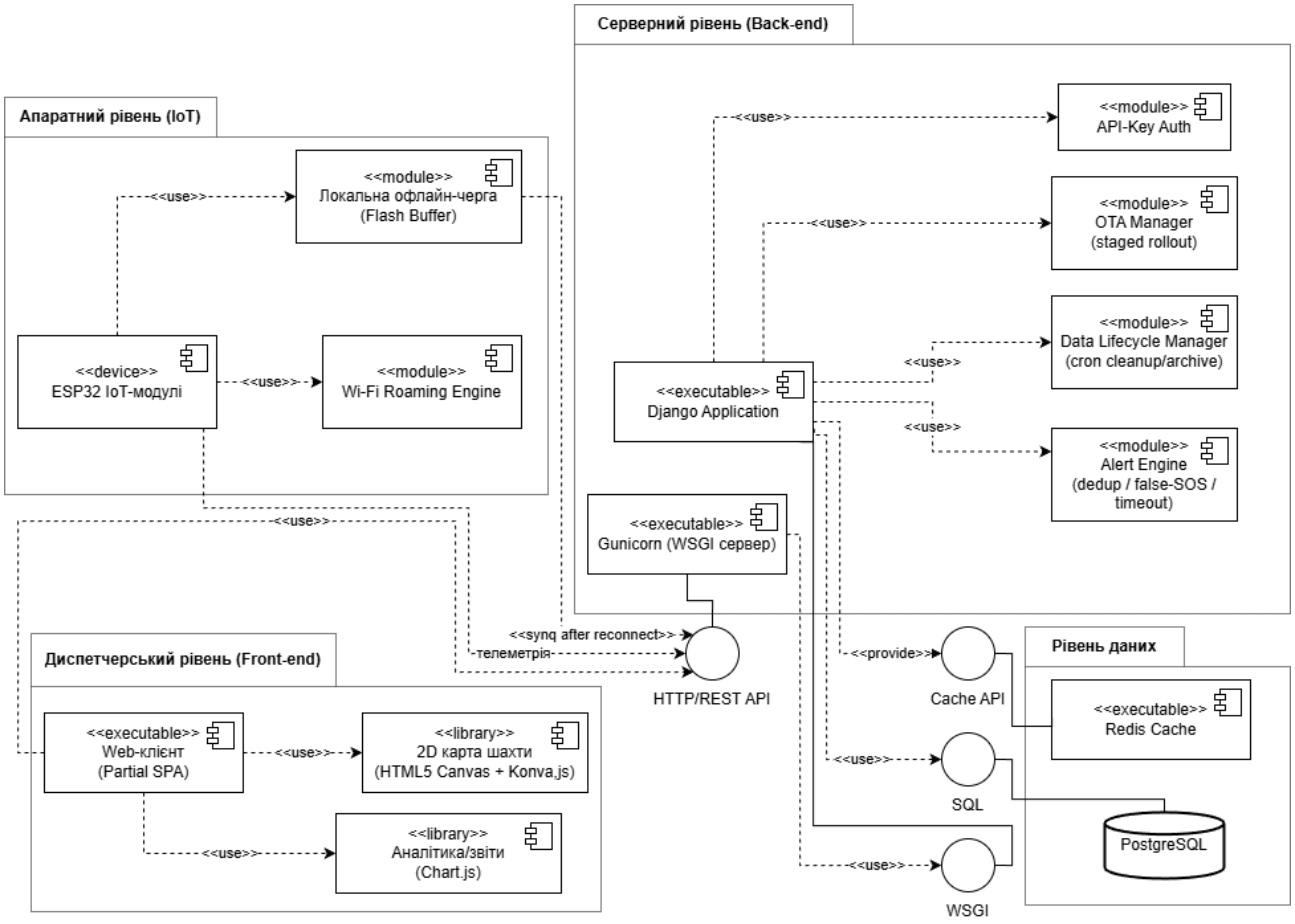

Рис. 2.3 UML-діаграма компонентів та логічної структури взаємодії модулів системи моніторингу

Логіка роботи системи поділяється на чотири ізольовані рівні. Взаємодія між ними відбувається через стандартизовані інтерфейси (позначені на діаграмі у вигляді «lollipop»-нотації). Це забезпечує слабку зв'язність (loose coupling) та високу масштабованість.

1. **Апаратний рівень (IoT-рівень).** Базовими компонентами рівня є фізичні пристрої <<device>> ESP32 IoT-модулі. Для безперервної роботи під землею до них підключено програмні модулі бізнес-логіки:

- Wi-Fi Roaming Engine – забезпечує безшовне перемикання між базовими станціями (репітерами) без втрати з'єднання;
- Локальна офлайн-черга (Flash Buffer) – модуль енергонезалежної пам'яті, який буферизує телеметрію при обриві мережі та ініціює синхронізацію з сервером після відновлення зв'язку.

2. **Серверний рівень (Back-end).** Реалізований як виконуваний компонент <<executable>> Django Application, що працює під управлінням WSGI-сервера Gunicorn. Сервер надає точки входу через інтерфейс HTTP/REST API. До складу серверного ядра входять такі авторські модулі:

- API-Key Auth – підсистема криптографічної перевірки доступу апаратних клієнтів;
- OTA Manager – підсистема управління життєвим циклом прошивок (з підтримкою поступового розгортання staged rollout);
- Alert Engine – інтелектуальний рушій обробки тривог, який виконує дедуплікацію повідомлень, фільтрацію хибних SOS-сигналів та контроль тайм-аутів;
- Data Lifecycle Manager – cron-модуль для архівації та очищення застарілих даних.

3. **Рівень даних.** Відповідає за консистентність та швидкість доступу до інформації. Містить основну реляційну СУБД PostgreSQL (взаємодія через інтерфейс SQL) та in-memory кеш-сервер <<executable>> Redis Cache (взаємодія через Cache API). Redis використовується для швидкої віддачі «гарячих» даних телеметрії.

4. **Диспетчерський рівень (Front-end).** Представлений компонентом Web-клієнт, розробленим за методологією часткового SPA (Single Page Application). Він асинхронно отримує телеметрію через HTTP/REST API та відображає її за допомогою спеціалізованих бібліотек <<library>>: 2D-карта

шахти (побудована на HTML5 Canvas із власним рендерингом на Canvas 2D) та аналітика/звіти (на базі Chart.js).

Використання UML-стереотипів (<<module>>, <<library>>, <<executable>>) дає змогу чітко розмежувати власні програмні рішення та сторонні інструменти. Це також підкреслює модульність розробленого програмного забезпечення.

2.3 Проєктування алгоритмів роботи програмних модулів

Кіберфізична система моніторингу «Глибина 4.0» передбачає не лише збір сирих даних, а й їх своєчасну та інтелектуальну обробку. У гірничодобувному підприємстві одночасно працюють сотні IoT-пристроїв, які генерують безперервний потік телеметрії. Без проміжних алгоритмів аналізу це неминуче призвело б до інформаційного перевантаження диспетчера (так званого «спаму тривог») та затримок в ухваленні критичних рішень.

Для розв'язання цієї проблеми спроєктовано та реалізовано програмний модуль розумних тривог – Smart Alert Engine. Його головні завдання: валідація вхідних даних, каскадний аналіз рівня небезпеки, дедуплікація аварійних повідомлень та контроль життєвого циклу статусу працівника.

Блок-схему розробленого алгоритму обробки телеметрії наведено на рис. 2.4. Логіку роботи алгоритму можна розділити на п'ять послідовних етапів.

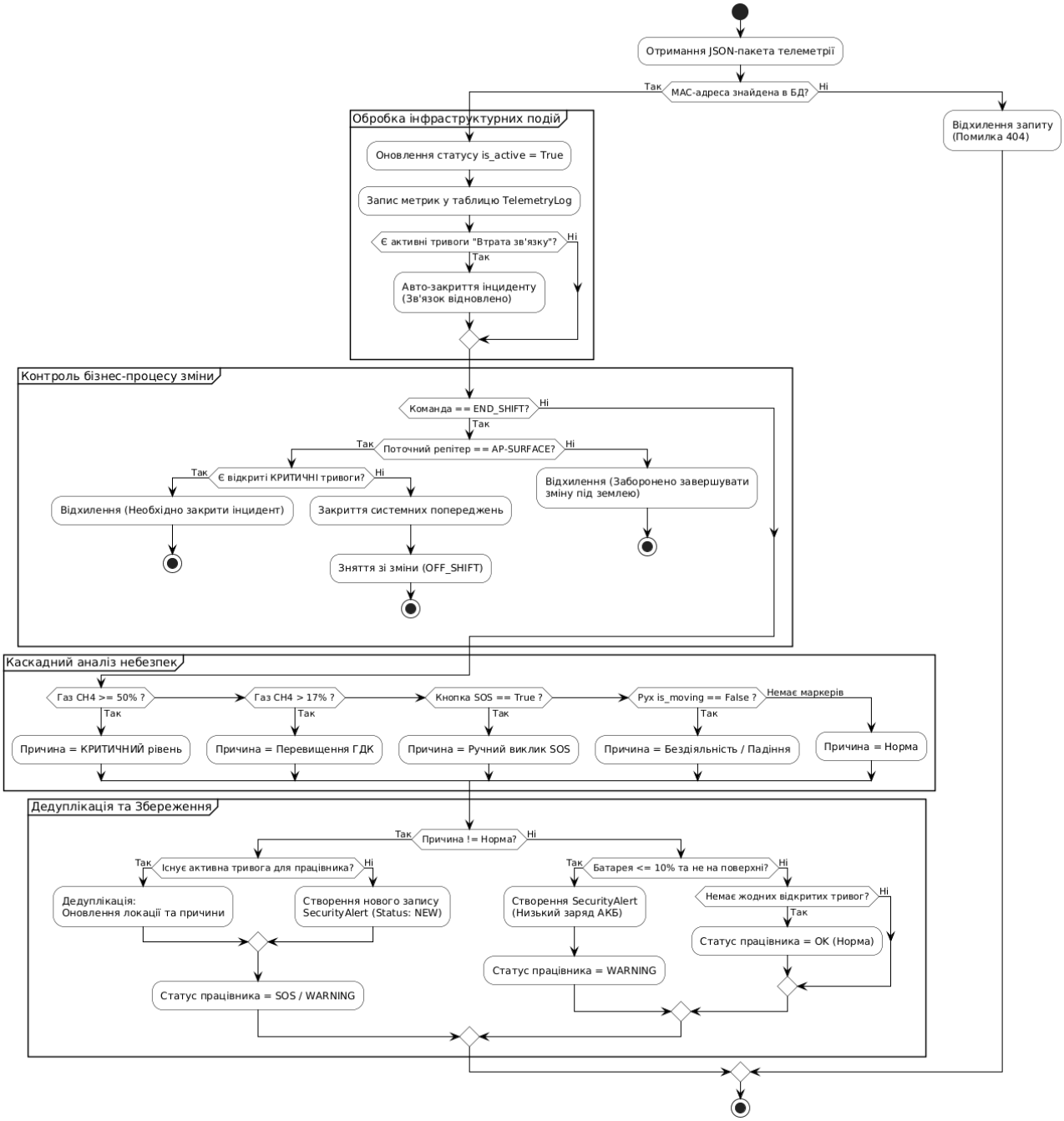

Рис. 2.4 Блок-схема алгоритму обробки телеметрії (Smart Alert Engine)

Етап 1 – ідентифікація та валідація. При надходженні JSON-пакета від IoT-модуля через протокол HTTP/REST система спочатку шукає пристрій у базі даних за його унікальною MAC-адресою. Якщо пристрій не зареєстрований або не закріплений за конкретним працівником (для мобільних модулів), транзакція відхиляється. Присвоєння статусу is_active = True сигналізує про успішне встановлення сеансу зв'язку.

Етап 2 – обробка інфраструктурних подій. Алгоритм перевіряє наявність відкритих тривог щодо втрати зв'язку (понад 3 хвилини) для цього працівника. Оскільки пакет телеметрії успішно доставлено, система ініціює автозакриття таких тривог і фіксує в журналі час відновлення з'єднання. Це реалізує принцип самовідновлення системи (Self-Healing).

Етап 3 – контроль бізнес-процесу зміни (Hardware Shift Control). Алгоритм перевіряє наявність команди END_SHIFT (завершення зміни). Важливою особливістю є апаратний контроль локації: система дозволяє перевести працівника у статус відпочинку (OFF_SHIFT) лише тоді, коли пакет даних передано через базову станцію на поверхні (ідентифікатор репітера AP-SURFACE) та пристрій підключено до зарядної док-станції. Спроба завершити зміну під землею блокується на рівні сервера.

Етап 4 – каскадний аналіз небезпек. На цьому етапі параметри телеметрії проходять багаторівневий фільтр пріоритетів:

- **Критична небезпека:** якщо концентрація метану (CH_4) досягає або перевищує 50% нижньої межі вибуховості (LEL), генерується вимога негайної евакуації.
- **Підвищена небезпека:** концентрація метану понад 17% LEL розцінюється як перевищення гранично допустимої концентрації (ГДК).
- **Ручний виклик:** аналізується стан апаратної кнопки SOS.
- **Іммобілізація:** якщо датчик вібрації пристрою фіксує відсутність руху (is_moving = False), формується підозра на падіння або втрату свідомості гірника.
- **Технічне попередження:** аналіз рівня заряду акумулятора (якщо заряд $\leq 10\%$).

Етап 5 – інтелектуальна дедуплікація. Це ключовий механізм оптимізації бази даних. Якщо алгоритм виявляє стан небезпеки, він перевіряє, чи не створено вже активну тривогу для цього працівника. Якщо тривога вже існує (наприклад, гірник натиснув SOS, а за хвилину підвищився рівень газу), система не створює новий запис у базі даних, а оновлює наявний – змінює причину на критичнішу та

фіксує нову локацію. Це гарантує, що диспетчер бачить лише одну, найактуальнішу картку інциденту для кожної людини.

Якщо жодних маркерів небезпеки немає і всі попередні інциденти закрито, система автоматично переводить маркер безпеки працівника у стан норми (OK). На диспетчерській панелі це візуально відображається зеленим кольором.

Розроблений алгоритм виконується швидко – всі операції обгорнуто в атомарні транзакції СУБД. При цьому повністю виключається вплив людського фактора на етапі первинної реєстрації аварії.

2.4 Проєктування об’єктно-орієнтованої структури (UML – діаграма класів)

Серверна частина системи «Глибина 4.0» реалізована з використанням об'єктно-орієнтованого підходу (ООП) на базі фреймворку Django. Відповідно до архітектурного патерну MVT (Model-View-Template), управління даними здійснюється через моделі. Вони реалізують патерн Active Record та інкапсулюють як структуру даних, так і бізнес-логіку їх обробки.

Для візуалізації структури розроблених класів, їх атрибутів, методів та зв'язків побудовано UML-діаграму класів (рис. 2.5).

Основними класами предметної області є такі:

1. **MineMap** – клас просторової топології шахти. Містить векторні дані *map_data* у форматі JSON для побудови 2D-карти у вебінтерфейсі.
2. **Employee** – клас, що описує працівника. Включає атрибути персональних даних, табельний номер (жетон) та поточний статус безпеки. Містить перевизначений метод *save*().
3. **MinerDevice** – клас апаратного IoT-вузла. Зберігає MAC-адресу, мережеві налаштування та координати для стаціонарних пристроїв (*static_x, static_y*). Містить метод валідації *clean*().

4. InfrastructureDevice – клас об'єктів мережевої інфраструктури (репітерів). Інкапсулює метод просторового обчислення *location_in_mine*() для позиціювання на 2D-карті.

5. TelemetryLog – клас реєстрації телеметричних показників. Містить атрибути мікроклімату, статусів руху, заряду батареї та причину генерації SOS-сигналу (sos_*reason*).

6. SecurityAlert – клас управління інцидентами. Зберігає стан тривоги, причину генерації, нотатки диспетчера про вжиті заходи, а також часові мітки створення (*created*_at) та закриття інциденту (*resolved*_at).

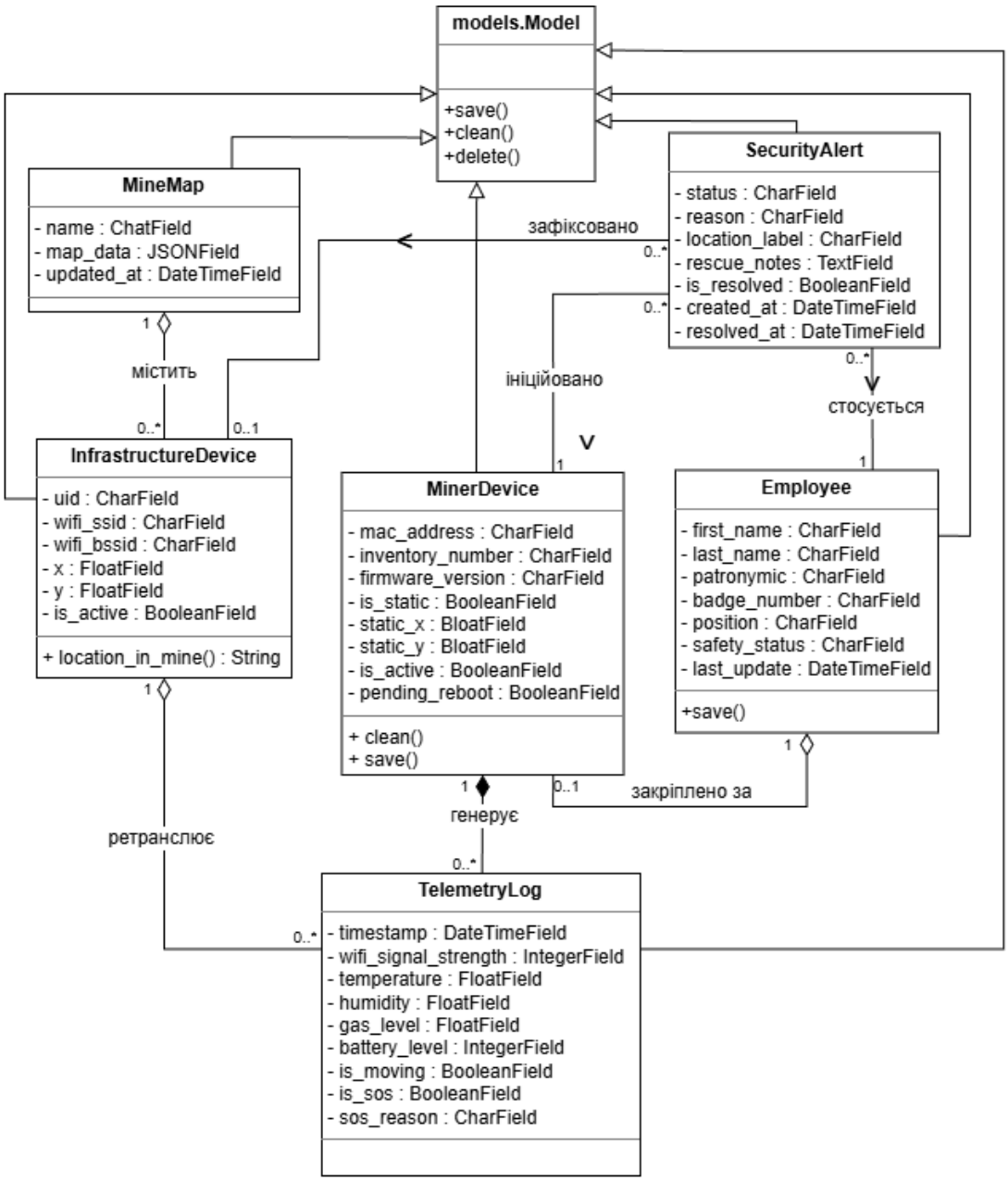

Рис. 2.5 UML-діаграма класів програмної системи

Усі розроблені класи предметної області пов'язані відношенням **узагальнення (Generalization)** з базовим класом django.db.models.Model. Це означає, що вони успадковують властивості батьківського класу, отримуючи готовий інтерфейс для взаємодії з реляційною базою даних через ORM (Object-Relational Mapping).

Між собою класи взаємодіють за допомогою таких структурних відношень:

- **Композиція (Composition):** Відображає жорстку залежність «ціле-частина». Застосовано між класами *MinerDevice* та *TelemetryLog*. Оскільки логи генеруються конкретним пристроєм, при його видаленні з системи всі пов'язані лог-записи також знищуються (механізм каскадного видалення *CASCADE*).
- **Агрегація (Aggregation):** Відображає відношення «ціле-частина», де частина може існувати самостійно. Наприклад, *MineMap* агрегує *InfrastructureDevice* (репітери розміщені на карті). Клас *Employee* агрегує *MinerDevice* (пристрій закріплюється за працівником лише на час зміни). Крім того, *InfrastructureDevice* агрегує *TelemetryLog* (при втраті зв'язку з репітером лог-запис не знищується, а лише втрачає прив'язку до нього).
- **Спрямована асоціація (Directed Association):** Відображає семантичний зв'язок (відношення використання) між об'єктами. Клас *SecurityAlert* має асоціативні зв'язки з *Employee*, *MinerDevice* та *InfrastructureDevice*, оскільки кожна аварійна тривога посилається на те, з яким працівником, пристроєм та в зоні якого репітера трапився інцидент.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління інформаційною базою

Для реалізації логічної моделі даних системи «Глибина 4.0» необхідно було обрати систему управління базами даних (СУБД). Специфіка розробленого програмного забезпечення полягає в роботі в умовах високого навантаження (HighLoad). IoT-пристрої (модулі ESP32) генерують безперервний потік часових рядів (телеметрію), який потрібно швидко записувати. Водночас диспетчерський вебінтерфейс кожні кілька секунд опитує сервер для оновлення стану персоналу на карті. Використання єдиної класичної реляційної СУБД для всіх операцій призвело б до вузького місця (bottleneck) у дисковій підсистемі.

Для розв'язання цієї проблеми ухвалено рішення використати гібридну (поліглотну) архітектуру збереження даних, яка поєднує класичну реляційну СУБД та NoSQL in-memory сховище.

Вибір основної реляційної СУБД.

Як основне сховище постійних даних обрано об'єктно-реляційну СУБД PostgreSQL. Для обґрунтування цього вибору виконано порівняльний аналіз PostgreSQL з іншою популярною реляційною СУБД – MySQL. Результати наведено в таблиці 3.1.

Таблиця 3.1 – Порівняльний аналіз СУБД PostgreSQL та MySQL

Характеристика/Критерій	PostgreSQL	MySQL
Підтримка складних типів даних	Повна підтримка JSONB, масивів (Array) та словників (HStore).	Обмежена підтримка JSON, відсутність нативних масивів.
Індексація часових рядів (IoT)	Наявність просторових та блокових індексів (BRIN, GiST) [12].	Тільки класичні B-Tree індекси.
Механізм паралельного доступу	Високоефективний MVCC (Multi-Version Concurrency Control).	Схильність до блокувань таблиць при масових INSERT-запитах.

Характеристика/Критерій	PostgreSQL	MySQL
Інтеграція з фреймворком Django	Максимальна (наявність спеціалізованого модуля django.contrib.postgres).	Базова (лише стандартний ORM).
Складність адміністрування	Висока, потребує тонкого налаштування пулів з'єднань.	Середня, легша у базовому розгортанні.

Як видно з таблиці 3.1, PostgreSQL має беззаперечні переваги для розроблюваної системи. Вирішальним фактором стала інтеграція PostgreSQL із фреймворком Django – наявність пакета django.contrib.postgres дозволяє використовувати специфічні функції бази даних без написання «сирих» SQL-запитів.

Це дало змогу:

1. Зберігати складну ієрархічну топологію карти шахти (координати штреків та вузлів) у форматі JSONB в моделі MineMap – сервер швидко парсить карту.
2. Застосувати BRIN-індекс (Block Range Index) для поля часу (timestamp) у таблиці TelemetryLog. Оскільки IoT-дані додаються послідовно в часі, BRIN-індекси займають у десятки разів менше пам'яті, ніж стандартні B-Tree індекси MySQL.
3. Гарантувати строгу транзакційність під час роботи модуля Smart Alert Engine, коли одночасно виконуються кілька операцій запису від різних датчиків.

Вибір in-memory бази даних (Redis)

Диспетчер потребує миттєвої реакції системи. Під час тестування з'ясувалося: поки Django ORM формує ресурсомісткий SQL-запит до PostgreSQL, карта «зависає». Тому обрано Redis. Він тримає «гарячі» дані позиціювання безпосередньо в оперативній пам'яті – час відповіді зменшився до 50 мс.

Redis працює за принципом «ключ-значення» (Key-Value) і зберігає всі дані в RAM сервера. Документування REST API системи реалізовано за стандартом OpenAPI, інтерфейс якого (Swagger UI) наведено на рис. 3.1.

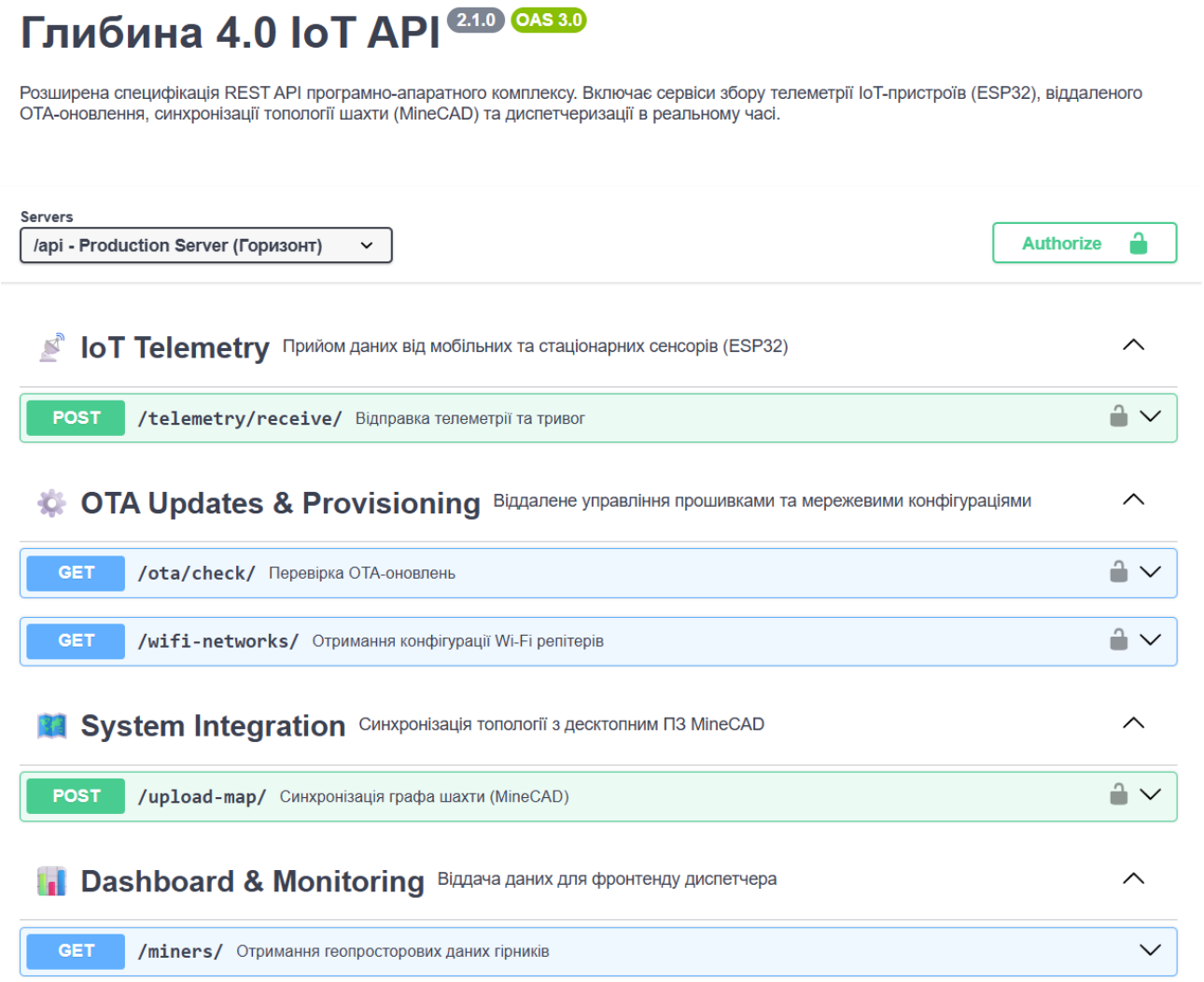

Рис. 3.1 Інтерактивна документація REST API системи у форматі Swagger UI

У розробленій системі Redis виконує роль проміжного кешувального брокера:

1. API-ендпоінти (dashboard_stats_api, api_active_miners) кешують результати агрегації мікроклімату та позиціювання персоналу на короткий проміжок часу (3–5 секунд).

2. Якщо кілька диспетчерів одночасно відкривають карту шахти, сервер не виконує важкі SQL-запити з об'єднанням таблиць (JOIN), а миттєво віддає готову JSON-відповідь з Redis.

3. Кешування статичних об'єктів (векторної карти шахти) виконується з тайм-аутом 1 година з можливістю примусової валідації кешу при внесенні змін адміністратором.

Поєднання надійності PostgreSQL для довгострокового зберігання та швидкості Redis для оперативного кешування дозволяє інформаційній базі системи ефективно масштабуватися при підключенні нових гірничих ділянок без деградації продуктивності.

3.2 Розробка інформаційної бази

Відповідно до обраного технологічного стека, перехід від логічної моделі даних (ER-діаграми) до фізичної структури бази даних PostgreSQL виконується не через написання прямих SQL-запитів (DDL), а за допомогою механізму об'єктно-реляційного відображення (ORM), вбудованого у фреймворк Django.

ORM дає змогу проєктувати фізичну схему бази даних за допомогою об'єктно-орієнтованого підходу мовою Python. Кожна таблиця БД представлена як клас, що успадковується від *django.db.models.Model*, а кожна колонка – як екземпляр класу відповідного типу поля (*CharField*, *IntegerField*, *FloatField*).

Особливу увагу під час розробки фізичної бази даних приділено таблиці збереження телеметрії, оскільки вона зазнає найбільшого навантаження. У лістингу 3.1 наведено фрагмент коду моделі TelemetryLog, який описує фізичну структуру таблиці часових рядів.

Лістинг 3.1 Фрагмент коду моделі TelemetryLog для створення таблиці в PostgreSQL

```
class TelemetryLog(models.Model):
    device = models.ForeignKey(MinerDevice,
on_delete=models.CASCADE, related_name='logs')
    timestamp = models.DateTimeField(auto_now_add=True)
    connected_repeater = models.ForeignKey(InfrastructureDevice,
on_delete=models.SET_NULL, null=True, blank=True)
    wifi_signal_strength = models.IntegerField(default=0,
verbose_name=«RSSI»)
    temperature = models.FloatField(null=True)
    humidity = models.FloatField(null=True)
```

```
    gas_level = models.FloatField(default=0, verbose_name=«Gas %
LEL»)
    battery_level = models.IntegerField(default=100)
    is_moving = models.BooleanField(default=True)
    is_sos = models.BooleanField(default=False, db_index=True)
    sos_reason = models.CharField(max_length=255, default='NONE')

    class Meta:
        ordering = ['-timestamp']
        indexes = [
            models.Index(fields=['device', '-timestamp']),
            BrinIndex(fields=['timestamp']),
        ]
```

Як видно з лістингу 3.1, фізична схема бази даних містить зовнішні ключі (*ForeignKey*) з правилами каскадного видалення (*on_delete*=models.CASCADE) для збереження референційної цілісності даних.

На рівні фізичної БД реалізовано HighLoad-оптимізацію за допомогою двох типів індексів у підкласі Meta:

1. **Складений B-Tree індекс** за полями ['device', '-timestamp'] – для миттєвого пошуку актуального місцеперебування конкретного працівника на карті.

2. **Просторовий індекс** BrinIndex (Block Range Index) для поля timestamp. Оскільки сенсори генерують дані безперервно і послідовно, BRIN-індекс не зберігає значення кожного окремого рядка, а фіксує мінімальні та максимальні значення часу для цілих блоків пам'яті на диску. Це дозволило зменшити обсяг індексу в оперативній пам'яті сервера більш ніж у 20 разів порівняно з традиційним B-Tree.

Для керування даними на рівні адміністратора системи використовується вбудована панель адміністрування фреймворку Django з розширення unfold (рис. 3.2).

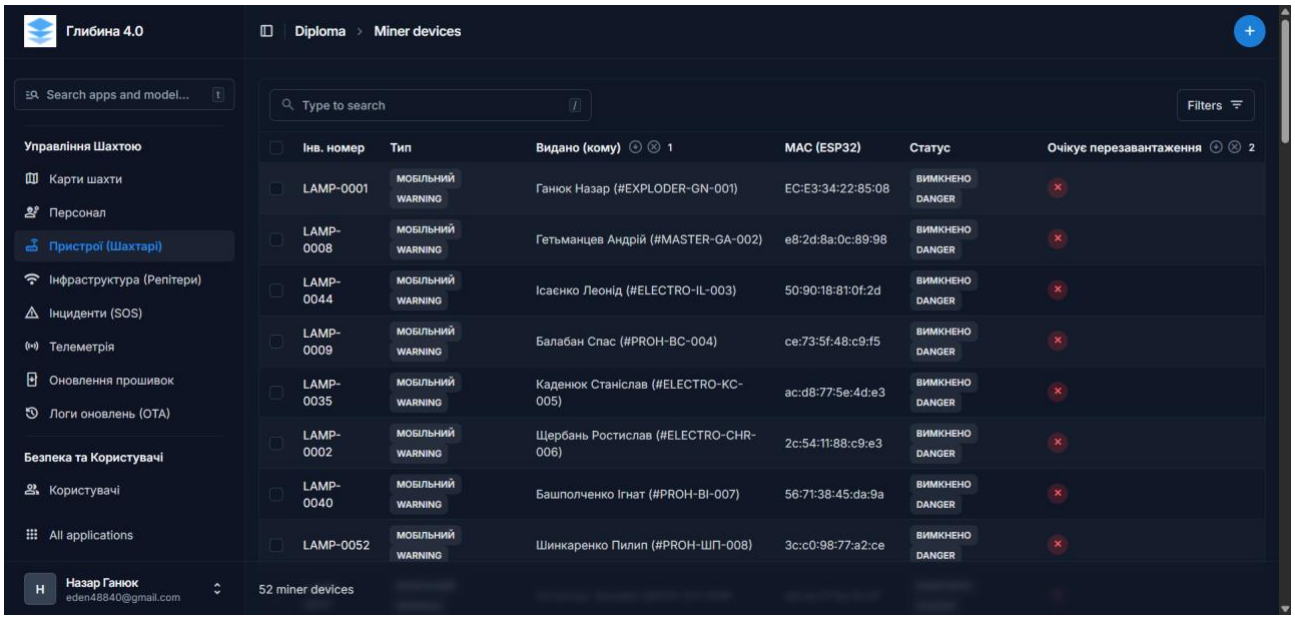

Рис. 3.2 Інтерфейс панелі адміністрування бази даних системи

Після визначення моделей у Python їх фізичне перетворення в таблиці PostgreSQL виконувалося за допомогою підсистеми міграцій фреймворку (команди makemigrations та migrate). Цей інструментарій автоматично аналізує зміни в класах і генерує відповідні SQL-скрипти: створення таблиць (CREATE TABLE), зміни колонок (ALTER TABLE) та побудови індексів (CREATE INDEX). Такий підхід гарантує відсутність синтаксичних помилок і повну сумісність із діалектом SQL обраної СУБД.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Кіберфізична система «Глибина 4.0» вимагає комплексу інструментальних засобів, оскільки програмне забезпечення розділене на ізольовані рівні: серверну частину (Back-end), клієнтську частину диспетчера (Front-end), підсистему генерації звітності та апаратний IoT-рівень. Вибір мов програмування та фреймворків виконано з урахуванням вимог до продуктивності, безпеки та можливості швидкого масштабування.

Інструментарій серверної частини (Back-end).

Як основну мову програмування серверної частини обрано Python (версії 3.10+). Цей вибір обґрунтований читабельністю, наявністю потужної екосистеми для обробки великих масивів даних (що важливо для аналізу часових рядів телеметрії) та широкою підтримкою мережевих протоколів.

Для побудови бізнес-логіки обрано вебфреймворк Django. На відміну від мікрофреймворків (Flask або FastAPI), Django реалізує принцип «batteries included» [6] (усе включено), що прискорило розробку. Основні переваги Django для цього проєкту:

1. Вбудована ORM з нативною підтримкою специфічних функцій PostgreSQL (наприклад, BrinIndex та JSONField).
2. Потужна підсистема аутентифікації та управління сесіями, що відповідає вимогам корпоративної безпеки (захист від брутфорс-атак, валідація паролів).
3. Система маршрутизації та вбудований механізм кешування, що дозволило налаштувати взаємодію з Redis для оптимізації HighLoad-запитів.

Інструментарій клієнтської частини (Front-end)

Диспетчерський вебінтерфейс розроблено з використанням класичного стека: HTML5, CSS3 (фреймворк Bootstrap 5) та JavaScript (ES6+). Для оновлення даних без перезавантаження сторінки використано технологію асинхронних запитів AJAX (Fetch API), що наближає систему до парадигми SPA.

Особливим викликом стала реалізація 2D-карти шахти (MineMap). На карті одночасно можуть відображатися десятки або сотні об'єктів (штреки, репітери, персонал). Використання стандартних засобів роботи з DOM або формату SVG призвело б до падіння частоти кадрів (FPS). Тому обрано технологію HTML5 Canvas (2D context) із власним шаром рендерингу.

Основні особливості модуля MineMap (HTML5 Canvas API):

- **Кластеризація об'єктів при малому масштабі** – близькі маркери групуються в кластер, що зменшує кількість примітивів для рендерингу.
- **Перевірка попадання (hit-testing)** – координати екрана перетворюються в координати карти з урахуванням scale та offset, після чого визначається об'єкт взаємодії (працівник/кластер/інфраструктура).
- **Панорамування та масштабування (pan/zoom)** – навігація реалізована через зміну offsetX/offsetY і scale зі збереженням точки фокусу.
- **Кінематографічний «переліт» камери (flyTo)** – автоматичне фокусування на об'єкті з плавною ease-in-out інтерполяцією масштабу і зсуву.
- **Анімації в циклі** requestAnimationFrame – візуальні ефекти тривог (пульсація) виконуються в окремому циклі, який активується лише за наявності тривог.
- **Асинхронне оновлення даних** – координати та статуси отримуються періодичними запитами (Fetch/AJAX), після чого виконується швидке перемальовування карти.

Для побудови аналітичних дашбордів (модуль статистичних звітів) використано легковагову JavaScript-бібліотеку *Chart.js*, яка рендерить адаптивні графіки на базі Canvas з мінімальним споживанням ресурсів.

Інтерфейс десктопного додатка для проєктування топології шахти представлено на рис. 3.3. Він дозволяє будувати карти шахт, взаємодіяти з усіма об’єктами на карті та перевіряти рух тестових шахтарів по штрекам.

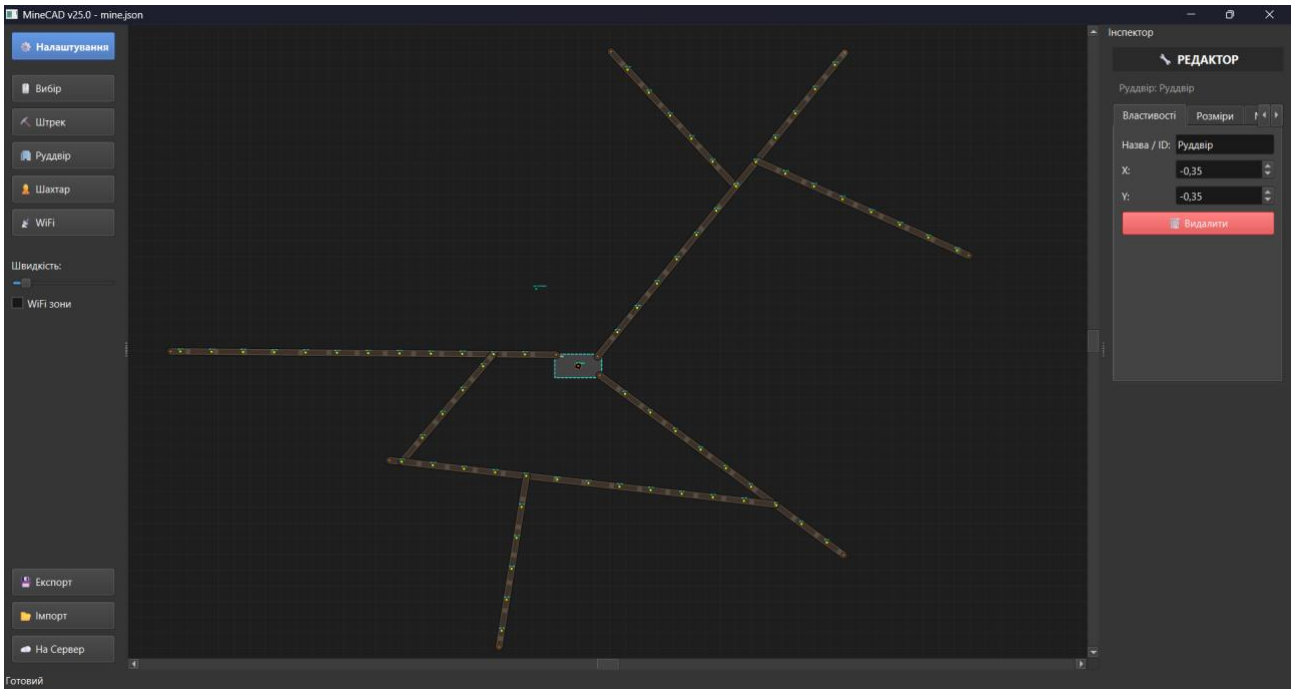

Рис. 3.3 Інтерфейс десктопного додатка «MineCAD» для проєктування топології шахти

Інструментарій забезпечення безпеки

Для захисту публічних точок входу (ендпоінтів авторизації) від автоматизованих брутфорс-атак та скрапінгу інтегровано технологію Cloudflare Turnstile [4]. На відміну від застарілих CAPTCHA, Turnstile працює у фоновому режимі, використовуючи неінтерактивні криптографічні виклики (Proof-of-Work) та аналіз телеметрії браузера. Це дозволяє надійно відсіювати бот-трафік, зберігаючи зручність інтерфейсу (UX).

Паролі користувачів зберігаються у вигляді криптографічних хешів відповідно до стандартних механізмів Django, що унеможливлює їх відновлення у відкритому вигляді. Додатково на рівні політик безпеки браузера застосовано Content Security Policy (CSP), яка обмежує виконання небезпечних скриптів і зменшує ризики XSS-атак.

На рис. 3.4 представлено вікно авторизації з захистом Turnstile від Cloudflare.

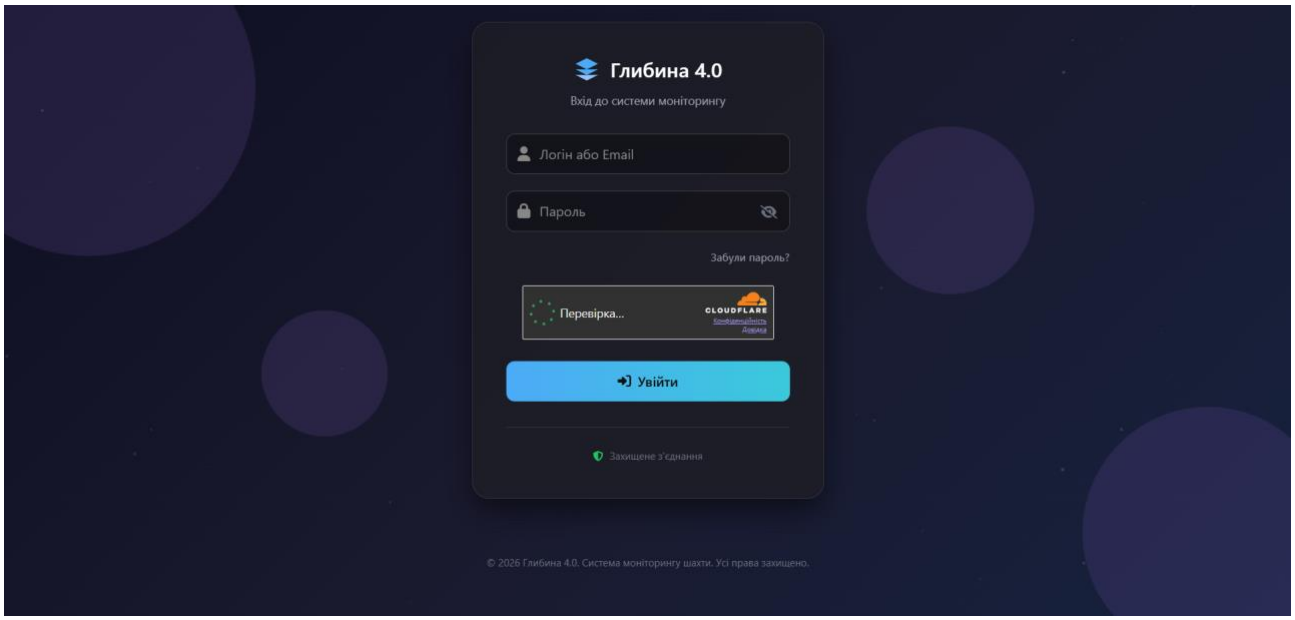

Рис. 3.4 Сторінка безпечної авторизації диспетчера з інтегрованим захистом Turnstile

Взаємодія IoT-модулів із сервером здійснюється через HTTP API, доступ до якого обмежено ключем (API-Key) у заголовку запиту. Це відокремлює «машинний» доступ пристроїв від доступу користувачів. Для аудиту подій передбачено журналювання – зберігаються записи телеметрії, факти спрацювання тривог та зміни статусів інцидентів, що забезпечує відтворюваність ситуацій для подальшого аналізу.

Інструментарій для формування звітності та утиліт

Для генерації офіційних зведень у форматі PDF (наприклад, під час роботи модуля Data Lifecycle Management) застосовано бібліотеку **ReportLab** [16] для Python. Вона дозволяє створювати PDF-документи безпосередньо з коду (через пряме малювання на координатному полотні), що працює швидше за конвертацію HTML у PDF, і підтримує інтеграцію кастомних шрифтів (наприклад, DejaVu Sans) для коректного відображення кирилиці.

Для підготовки картографічних даних (топології штреків) та проведення навантажувального тестування (симуляції руху гірників) розроблено окремий

десктопний інструмент **«MineCAD»** мовою Python з використанням графічного фреймворку PyQt6.

Апаратний рівень (IoT)

Програмування мікроконтролерів ESP32 (мобільних модулів та стаціонарних датчиків) виконано мовами C/C++ у середовищі Arduino / ESP-IDF. Це забезпечує низькорівневий доступ до пам'яті (Flash Buffer), керування периферією (Wi-Fi трансівер, сенсори газу MQ-серії, датчик руху, buzzer, DHT22).

Таким чином, обраний поліглотний стек технологій покриває всі архітектурні потреби кіберфізичної системи та гарантує її ефективну роботу в умовах промислової експлуатації.

3.4 Алгоритмізація та програмування програмних модулів

На етапі програмування реалізовано бізнес-логіку системи у вигляді набору модулів, що взаємодіють на базі фреймворку Django. Для забезпечення модульності та зручності супроводу коду логіку обробки даних відділено від логіки їх представлення. Найбільш критичними з точки зору архітектури є модулі обробки телеметрії, управління життєвим циклом даних та підсистема кібербезпеки.

Програмування модуля Smart Alert Engine.

Модуль розумних тривог (Smart Alert Engine) відповідає за приймання вхідних пакетів від IoT-пристроїв, їх валідацію та прийняття рішень щодо генерації аварійних сповіщень. Ключове завдання алгоритму – інтелектуальна дедуплікація, тобто запобігання створенню дублювальних записів про одну й ту саму небезпеку.

У лістингу 3.2 наведено фрагмент коду (метод обробки POST-запиту), що ілюструє логіку дедуплікації.

Лістинг 3.2 Фрагмент програмного коду алгоритму дедуплікації Smart Alert Engine

```
def process_telemetry_alert(device, employee, data,
current_repeater):
    ««««Каскадний аналіз небезпек та дедуплікація тривог.»»««
    alert_reason = None
    alert_level = 'WARNING'

    # Каскадний аналіз пріоритетів небезпеки
    if data.get('gas_level', 0) >= 50.0:
        alert_reason = «КРИТИЧНИЙ РІВЕНЬ МЕТАНУ (Евакуація)»
        alert_level = 'NEW'
    elif data.get('is_sos') is True:
        alert_reason = «Ручний виклик SOS»
        alert_level = 'NEW'
    elif data.get('is_moving') is False:
        alert_reason = «Підозра на падіння (бездіяльність)»

    # Якщо виявлено маркер небезпеки
    if alert_reason:
        # Пошук існуючої активної тривоги для дедуплікації
        active_alert = SecurityAlert.objects.filter(
            employee=employee,
            is_resolved=False
        ).first()

        if active_alert:
            # Оновлення існуючої тривоги (без створення дубліката в
БД)
            active_alert.reason = f»Оновлено: {alert_reason}»
            active_alert.connected_repeater = current_repeater
            active_alert.save(update_fields=['reason',
'connected_repeater'])
        else:
            # Генерація нової тривоги
            SecurityAlert.objects.create(
                employee=employee,
                device=device,
                connected_repeater=current_repeater,
                reason=alert_reason,
                status=alert_level
            )
            # Оновлення статусу працівника
            employee.safety_status = 'SOS' if alert_level == 'NEW'
else 'WARNING'
            employee.save(update_fields=['safety_status'])
```

Використання методу update_fields в ORM Django (лістинг 3.2) дозволяє виконувати легковагові SQL-запити UPDATE лише для вказаних колонок. Це зменшує навантаження на дискову підсистему при частих оновленнях локації гірника.

Програмування модуля Data Lifecycle Management

Оскільки IoT-пристрої безперервно надсилають телеметрію, таблиця TelemetryLog швидко зростає. Для запобігання деградації продуктивності PostgreSQL розроблено модуль управління життєвим циклом даних, який автоматично архівує застарілу інформацію. У лістингу 3.3 показано фрагмент командного скрипта (Management Command), що видаляє записи старші за 30 днів.

Лістинг 3.3 Фрагмент коду модуля архівації застарілих даних

```
from django.core.management.base import BaseCommand
from django.utils import timezone
from datetime import timedelta
from diploma.models import TelemetryLog

class Command(BaseCommand):
    help = 'Архівація та очищення телеметрії старшої за 30 днів'

    def handle(self, *args, **kwargs):
        cutoff_date = timezone.now() - timedelta(days=30)

        # Вибірка застарілих даних (ігноруючи записи, пов'язані з
інцидентами)
        old_logs = TelemetryLog.objects.filter(
            timestamp__lt=cutoff_date,
            is_sos=False
        )

        logs_count = old_logs.count()
        if logs_count > 0:
            # Використання оптимізованого методу _raw_delete для
швидкого очищення
            old_logs._raw_delete(old_logs.db)
            self.stdout.write(self.style.SUCCESS(f'Успішно видалено
{logs_count} старих записів.'))
        else:
            self.stdout.write('Немає даних для архівації.')
```

Програмування підсистеми кібербезпеки та захисту облікових записів

Відповідно до вимог індустріальних систем, диспетчерський інтерфейс має високий рівень захисту від несанкціонованого доступу. Для аудиту безпеки облікових записів використано механізм «сигналів» (Signals) у Django. Алгоритм автоматично відстежує IP-адресу та User-Agent при кожній спробі входу.

Лістинг 3.4 Фрагмент програмного коду підсистеми кібербезпеки та захисту облікових записів

```
@receiver(user_logged_in)
def check_new_device_login(sender, user, request, **kwargs):
    """Перевірка, чи IP та Браузер нові для цього користувача."""

    # Визначення реальної IP-адреси через проксі-сервер (наприклад,
Cloudflare)
    ip_address = request.META.get('HTTP_CF_CONNECTING_IP') or
request.META.get('REMOTE_ADDR')
    if ip_address and ',' in ip_address:
        ip_address = ip_address.split(',')[0].strip()

    user_agent = request.META.get('HTTP_USER_AGENT',
'Unknown')[:255]

    # Спроба отримати або створити запис про пристрій
    device, created = UserDevice.objects.get_or_create(
        user=user,
        ip_address=ip_address,
        user_agent=user_agent
    )

    # Якщо пристрій новий - надсилаємо E-mail попередження
    if created and user.email:
        subject = '⚠ Новий вхід у ваш обліковий запис - Глибина
4.0'
        html_message =
render_to_string('diploma/auth/new_device_email.html', {
            'user': user,
            'ip_address': ip_address,
            'user_agent': user_agent,
            'time': timezone.now()
        })
        plain_message = strip_tags(html_message)
        try:
            send_mail(
                subject, plain_message, settings.DEFAULT_FROM_EMAIL,
                [user.email], html_message=html_message,
fail_silently=True
            )
```

```
        except Exception:
            pass
```

Реалізований алгоритм (лістинг 3.4) не лише записує цифрові відбитки сесій, але й асинхронно надсилає користувачу електронний лист із деталями входу (IP-адреса, час, браузер). Конструкція fail_silently=True забезпечує відмовостійкість: якщо SMTP-сервер тимчасово недоступний, виняток не призведе до збою авторизації.

Програмування апаратних модулів IoT-рівня (ESP32)

Найважливішою складовою системи є програмне забезпечення мікроконтролерів ESP32, яке працює в умовах екстремального середовища та постійних перебоїв зв'язку. Прошивку реалізовано мовою C++ з використанням неблокуючих алгоритмів (non-blocking) та асинхронного виконання задач.

Для практичної реалізації IoT-модуля було розроблено електричну принципову схему збору телеметрії. Як видно з рис. 3.5, сенсорна периферія підключається до мікроконтролера ESP32.

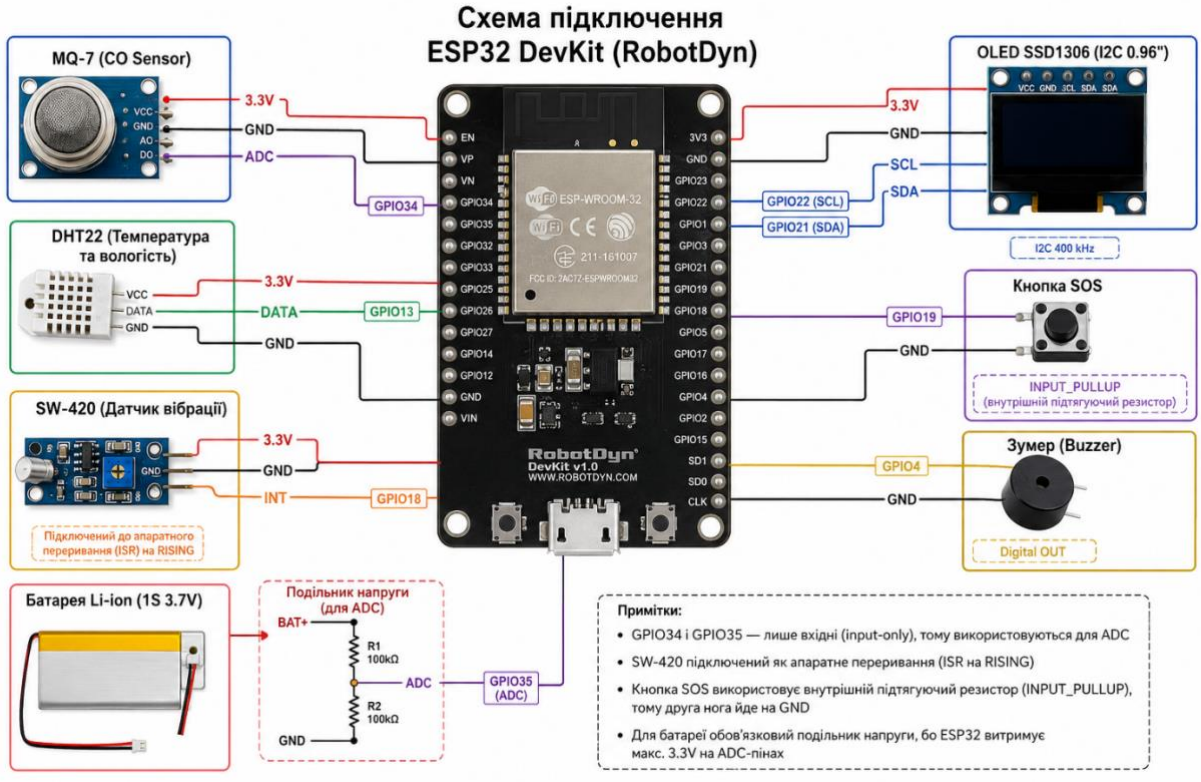

Рис. 3.5 Електрична схема мобільного IoT-модуля гірника

Для забезпечення нульової втрати критичних даних під час обриву зв'язку розроблено алгоритм локальної офлайн-черги на базі std::queue. У поєднанні з асинхронним скануванням радіоефіру пристрій виконує безшовний роумінг між базовими станціями.

У лістингу 3.5 наведено фрагмент коду мікроконтролера, що ілюструє логіку буферизації телеметрії та відновлення з'єднання.

Лістинг 3.5 Фрагмент коду реалізації Wi-Fi роумінгу та офлайн-черги мікроконтролера ESP32

```
// Тут був баг: якщо вимкнути репітер, ESP зависає на HTTP.begin.
// Вирішив обгорнути чергою. Якщо немає Wi-Fi, просто кладемо пакет
у telemetryQueue.
// Важливо: MAX_QUEUE_SIZE = 30, бо пам'яті на ESP32 всього 520Кб,
більше не вивезе.
std::queue<String> telemetryQueue;
const size_t MAX_QUEUE_SIZE = 30; // Зберігаємо до 30 записів (~5
хвилин офлайну)
unsigned long lastQueueProcessMs = 0;

// ========================================
// НАДСИЛАННЯ ДАНИХ АБО БУФЕРИЗАЦІЯ ПРИ ВТРАТІ ЗВ'ЯЗКУ
// ========================================
void sendTelemetry(bool isSos, String reason = «Normal») {
  StaticJsonDocument<384> doc;
  doc[«mac_address»] = WiFi.macAddress();
  doc[«battery»] = getBatteryPct();
  doc[«gas_level»] = getCleanGas();
  doc[«is_sos»] = isSos;
  doc[«reason»] = reason;

  String json;
  serializeJson(doc, json);

  // Якщо немає мережі -> додаємо в офлайн-чергу і виходимо
  if (WiFi.status() != WL_CONNECTED) {
    if (telemetryQueue.size() < MAX_QUEUE_SIZE) {
        telemetryQueue.push(json);
    }
    return;
  }

  HTTPClient http;
  http.setTimeout(HTTP_TIMEOUT_MS);
  http.begin(String(SERVER_ADDRESS) + TELEMETRY_ENDPOINT);
  http.addHeader(«X-API-Key», API_KEY);
  http.addHeader(«Content-Type», «application/json»);
```

```
  int httpCode = http.POST(json);
  if (httpCode != 200 && httpCode != 201) {
    // Якщо сервер повернув помилку, зберігаємо дані
    if (telemetryQueue.size() < MAX_QUEUE_SIZE)
telemetryQueue.push(json);
  }
  http.end();
}

// ==========================================
// СИНХРОНІЗАЦІЯ ЧЕРГИ ПІСЛЯ ВІДНОВЛЕННЯ ЗВ'ЯЗКУ
// ==========================================
void processTelemetryQueue() {
  if (WiFi.status() != WL_CONNECTED || telemetryQueue.empty())
return;

  String json = telemetryQueue.front();
  HTTPClient http;
  http.setTimeout(HTTP_TIMEOUT_MS);
  http.begin(String(SERVER_ADDRESS) + TELEMETRY_ENDPOINT);
  http.addHeader(«X-API-Key», API_KEY);
  http.addHeader(«Content-Type», «application/json»);

  int httpCode = http.POST(json);
  if (httpCode == 200 || httpCode == 201) {
    telemetryQueue.pop(); // Видаляємо успішно відправлене
повідомлення з черги
  }
  http.end();
}

// ==========================================
// АСИНХРОННЕ СКАНУВАННЯ МЕРЕЖ (Wi-Fi Roaming)
// ==========================================
void performWifiRoaming() {
  bool wifiLost = (WiFi.status() != WL_CONNECTED);
  unsigned long scanInterval = wifiLost ? 3000UL :
WIFI_SCAN_INTERVAL_MS;

  // Запуск фонового сканування (не блокує цикл loop)
  if (millis() - lastWifiScanMs > scanInterval) {
    if (!wifiScanActive) {
      WiFi.scanNetworks(true); // true = асинхронно
      wifiScanActive = true;
    }
  }

  // Обробка результатів сканування
  if (wifiScanActive) {
    int n = WiFi.scanComplete();
    if (n >= 0) {
```

```
      int bestRssi = -1000;
      String bestSsid = "";
      String bestBssid = "";

      // Пошук найсильнішої точки доступу
      for (int i = 0; i < n; ++i) {
        String ssid = WiFi.SSID(i);
        int rssi = WiFi.RSSI(i);
        if (ssid == baseSsid && rssi > bestRssi) {
          bestRssi = rssi;
          bestSsid = baseSsid;
          bestBssid = WiFi.BSSIDstr(i);
        }
      }

      int currentRssi = WiFi.RSSI();
      bool shouldRoam = wifiLost || (bestBssid != WiFi.BSSIDstr() &&
bestRssi > currentRssi + 10);

      if (shouldRoam && bestSsid != "") {
        WiFi.disconnect();
        WiFi.begin(bestSsid.c_str(), basePass.c_str());
      }
      WiFi.scanDelete();
      wifiScanActive = false;
      lastWifiScanMs = millis();
    }
  }
}
```

Як видно з лістингу 3.5, алгоритм повністю уникає блокуючих функцій (наприклад, delay()). Сканування Wi-Fi мереж виконується асинхронно (WiFi.scanNetworks(true)) [7], що дозволяє мікроконтролеру продовжувати опитування датчиків газу та вібрації навіть під час пошуку репітера.

Якщо з'єднання втрачено, метод sendTelemetry не блокує виконання тривалими спробами підключення, а миттєво серіалізує JSON та розміщує його в черзі telemetryQueue. Після відновлення зв'язку метод processTelemetryQueue забезпечує поступову відправку накопичених пакетів на сервер, звільняючи пам'ять мікроконтролера.

На рис. 3.6 представлено загальний вигляд зібраного лабораторного прототипу пристрою під час роботи. На OLED-дисплеї відображається поточний статус підключення та показники газу.

Рис. 3.6 Лабораторний прототип апаратного IoT-модуля збору телеметрії

Така архітектура IoT-рівня робить систему «Глибина 4.0» максимально адаптивною до реальних фізичних умов вугледобувного підприємства та гарантує консистентність історичних даних.

3.5 Конструктивне виконання та інтеграція апаратного модуля

Для забезпечення практичного використання розробленої системи «Глибина 4.0» у реальних умовах шахти, було розроблено концепцію інтеграції апаратної частини у стандартне екіпірування гірника. Основним принципом розробки стала модернізація (upgrade) серійної шахтарської коногонки (індивідуального світильника), що дозволяє зберегти ергономіку та надійність перевіреного часом обладнання, додаючи йому функціонал «розумного» пристрою.

Конструктивне рішення базується на заміні штатної верхньої кришки на спеціально розроблений корпус. Новий корпус повністю сумісний із оригінальним нижнім кейсом акумулятора, що дозволяє уникнути повної заміни дороговартісного обладнання (рис. 3.7).

Рис. 3.7 Оригінальний вигляд коногонки та модернізована версія з інтегрованим модулем «Глибина 4.0»

Розроблений корпус має такі конструктивні особливості:

1. **Модульність:** Внутрішній простір кришки спроєктований таким чином, щоб надійно зафіксувати мікроконтролер ESP32, газоаналізатор MQ-7, датчик температури та вологості DHT22 та модуль акселерометра.

2. **Інтерфейс користувача:** На верхній панелі передбачено отвори для OLED-дисплея та тактової кнопки (SOS-сигнал), що забезпечує швидкий доступ до інформації та екстрених функцій.

3. **Вентиляційна решітка:** Оскільки датчики газу та мікроклімату потребують постійного контакту з навколишнім середовищем, у корпусі реалізовано технологічні прорізи для вільної циркуляції повітря, захищені від прямого потрапляння великих часток пилу.

4. **Збереження основного функціоналу:** Конструкція передбачає вихід для оригінального проводу, що веде до наголовного ліхтаря, забезпечуючи безперебійне освітлення.

Інтеграція живлення та збірка: Замість прямого паяння до акумулятора, для підвищення ремонтопридатності та надійності системи, було обрано роз'ємне з'єднання. Живлення системи здійснюється від оригінального літій-іонного акумулятора коногонки (3.7V). Процес підключення включає наступні кроки:

- Встановлення швидкороз’ємних конекторів (типу JST або аналогічних) на виводи оригінального акумулятора.
- Підключення живлення до відповідних пінів ESP32 через дільник напруги для моніторингу рівня заряду.
- Фіксація кришки на штатні кріплення нижнього кейсу, що забезпечує механічну міцність конструкції.

Такий підхід до проєктування дозволяє значно знизити вартість впровадження системи на підприємстві, оскільки не потребує закупівлі нових корпусів та акумуляторів – модернізується лише верхня частина пристрою, яка перетворює звичайний ліхтар на повноцінний вузол безпеки IoT.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування є важливим етапом життєвого циклу розробки ПЗ, мета якого – перевірка відповідності продукту початковим вимогам та виявлення дефектів. Оскільки система «Глибина 4.0» належить до класу Safety-Critical Systems (систем гарантування безпеки) і працює в умовах високого навантаження (HighLoad), процес тестування розділено на два етапи: функціональне тестування бізнес-логіки та навантажувальне стрес-тестування.

Функціональне тестування апаратної та програмної логіки.

Для забезпечення якості програмно-апаратного комплексу та формалізації перевірки бізнес-логіки розроблено специфікацію тестових сценаріїв (Test Cases). Функціональне тестування виконано методом «чорного ящика» (Black-Box Testing) з імітацією нестандартних та аварійних ситуацій.

У таблицях 4.1–4.3 наведено деталізовані кроки перевірки найбільш критичних функцій: роботи офлайн-черги ESP32, алгоритму дедуплікації Smart Alert Engine та апаратного контролю завершення зміни.

Таблиця 4.1 Тестовий сценарій №1 (ТС-01): Перевірка алгоритму локальної офлайн-черги ESP32

Крок	Дія користувача / системи	Очікуваний результат	Фактичний результат	Статус
1	Імітація обриву зв'язку (знеструмлення базової станції Wi-Fi).	IoT-модуль фіксує втрату мережі. На дисплеї з'являється попередження «ВТРАТА ЗВ'ЯЗКУ». Активується зумер.	Дисплей оновлено, модуль перейшов у режим фонового сканування ефіру.	Passed
2	Імітація аварії в офлайн-режимі (натискання кнопки SOS або перевищення рівня газу).	Дані не втрачаються, а серіалізуються у JSON-пакет і поміщаються в локальну структуру telemetryQueue (Flash-буфер).	Пакет збережено в оперативній пам'яті ESP32.	Passed

Крок	Дія користувача / системи	Очікуваний результат	Фактичний результат	Статус
3	Відновлення живлення репітера (поява мережі Wi-Fi).	Модуль автоматично підключається до репітера та ініціює асинхронну надсилання накопичених пакетів на сервер.	З'єднання відновлено. Черга пакетів поступово передана на REST API.	Passed
4	Перевірка цілісності даних у СУБД PostgreSQL.	Сервер реєструє тривогу з історичною міткою часу (timestamp), втрат даних не зафіксовано.	Тривога відобразилася на панелі диспетчера.	Passed

Таблиця 4.2 Тестовий сценарій №2 (TC-02): Перевірка дедуплікації тривог (Smart Alert Engine)

Крок	Дія користувача / системи	Очікуваний результат	Фактичний результат	Статус
1	Генерація події «SOS» від працівника (натискання апаратної кнопки на модулі).	Сервер створює новий запис у таблиці SecurityAlert зі статусом NEW. Працівник підсвічується червоним кольором на 2D-карті.	Тривога успішно створена, диспетчер отримав звукове сповіщення.	Passed
2	Генерація події «Критичний метан» від того ж самого працівника через 1 хвилину.	Сервер не створює дублювальний запис. Механізм дедуплікації знаходить активну тривогу та оновлює причину на більш критичну.	Запис у базі оновлено (update_fields). Дублікатів не виявлено.	Passed
3	Диспетчер закриває інцидент у вебінтерфейсі (статус RESOLVED).	Маркер небезпеки знімається. Статус працівника на карті змінюється на «ОК» (зелений колір).	Інцидент переміщено до архіву, статус оновлено миттєво через кеш Redis.	Passed

Таблиця 4.3 Тестовий сценарій №3 (TC-03): Апаратний контроль завершення зміни (Shift Control)

Крок	Дія користувача / системи	Очікуваний результат	Фактичний результат	Статус
1	Надсилання команди END_SHIFT (імітація здачі пристрою), знаходячись у зоні дії підземного репітера.	Сервер відхиляє транзакцію. Перехід у статус відпочинку заборонено.	Відмова у знятті статусу безпеки. Гірник залишається активним на зміні.	Passed
2	Переміщення працівника на поверхню (підключення модуля	Система фіксує зміну BSSID та дозволяє ініціацію завершення зміни.	Модуль успішно розпізнав базову станцію на поверхні.	Passed

Крок	Дія користувача / системи	Очікуваний результат	Фактичний результат	Статус
	до мережі AP-SURFACE).			
3	Підключення пристрою до зарядної док-станції.	Аналіз рівня заряду (АЦП) підтверджує заряджання. Сервер знімає працівника зі зміни (OFF_SHIFT).	Пристрій перейшов у режим CHARGING_SLEEP, сервер закрив зміну.	Passed

Модульне тестування програмного коду (Unit Testing)

Крім функціонального тестування, для перевірки критичних компонентів бізнес-логіки на рівні вихідного коду застосовано автоматизоване модульне тестування (методологія «білого ящика»). Для цього використано вбудований фреймворк *django.test* на базі стандартної бібліотеки *unittest* мови Python.

Особливу увагу приділено перевірці алгоритму дедуплікації Smart Alert Engine. Від коректної роботи цього модуля залежить сповіщення диспетчера, тому необхідно було програмно довести, що багаторазові критичні пакети від одного пристрою не дублюють записи в БД.

У лістингу 4.1 наведено розроблений тестовий клас *AlertEngineTestCase*, який імітує надходження послідовних аварійних пакетів та перевіряє стан бази даних.

Лістинг 4.1 Тестовий клас для перевірки дедуплікації тривог

```
from django.test import TestCase
from diploma.models import Employee, MinerDevice, SecurityAlert
from diploma.api.views import process_telemetry_alert

class AlertEngineTestCase(TestCase):
    def setUp(self):
        """Ініціалізація тестового середовища (Mock-дані)"""
        self.employee = Employee.objects.create(
            first_name="Іван",
            last_name="Іванов",
            badge_number="001",
            safety_status="OK"
        )
        self.device = MinerDevice.objects.create(
            mac_address="00:11:22:33:44",
            assigned_to=self.employee
        )
```

```
    def test_deduplication_logic(self):
        """Перевірка об'єднання двох послідовних тривог у єдиний
інцидент"""

        # Крок 1: Імітація першого пакету (високий рівень газу)
        telemetry_data_1 = {'gas_level': 55.0, 'is_sos': False,
'is_moving': True}
        process_telemetry_alert(self.device, self.employee,
telemetry_data_1, current_repeater=None)

        # Перевіряємо, що в БД створено рівно 1 тривогу
        self.assertEqual(SecurityAlert.objects.count(), 1)
        alert = SecurityAlert.objects.first()
        self.assertEqual(alert.status, 'NEW')
        self.assertIn("МЕТАНУ", alert.reason)

        # Крок 2: Імітація другого пакету через кілька секунд
(працівник натиснув SOS)
        telemetry_data_2 = {'gas_level': 55.0, 'is_sos': True,
'is_moving': True}
        process_telemetry_alert(self.device, self.employee,
telemetry_data_2, current_repeater=None)

        # Перевірка дедуплікації: кількість інцидентів має
залишитися 1
        self.assertEqual(SecurityAlert.objects.count(), 1)

        # Перевіряємо, що існуюча тривога оновила причину
        updated_alert = SecurityAlert.objects.first()
        self.assertIn("Ручний виклик SOS", updated_alert.reason)

        # Перевірка оновлення статусу працівника
        self.employee.refresh_from_db()
        self.assertEqual(self.employee.safety_status, 'SOS')
```

Метод setUp створює ізольовану тестову базу даних (in-memory SQLite) для незалежності тестів. Метод test_deduplication_logic генерує дві аварійні події поспіль: спочатку перевищення метану, потім ручний виклик SOS. За допомогою assertEqual система перевіряє, що кількість записів SecurityAlert не перевищує 1. Успішне проходження тесту підтверджує працездатність та надійність алгоритму дедуплікації перед розгортанням на робочому сервері (Production).

Навантажувальне та стрес-тестування (Load Testing)

Враховуючи, що в реальній шахті одночасно працюють сотні датчиків, які щосекундно генерують телеметрію, виконано комплексне навантажувальне тестування архітектури:

1. **Стрес-тестування API за допомогою Locust.** Для імітації висококонкурентного трафіку використано фреймворк Locust [10] мовою Python. Розроблено сценарій, який імітував роботу сотень одночасно підключених IoT-вузлів, що безперервно надсилали POST-запити на REST API. Моніторинг показав: завдяки WSGI-серверу Gunicorn та оптимізованим транзакціям Django ORM сервер витримав потік запитів без помилок 5xx. Результати стрес-тестування тестових API за допомогою інструменту Locust відображено на графіках (рис. 4.1).

2. **Тестування бази даних та підсистеми кешування.** За допомогою консольних команд (зокрема скрипта seed_load_test.py) у PostgreSQL згенеровано 5 000 тестових записів телеметрії та імітовано 200 віртуальних пристроїв і співробітників. Тестування підтвердило ефективність BRIN-індексів: швидкість вибірки історії переміщень залишилася стабільно високою, а використання Redis дозволило знизити час віддачі даних для вебклієнта до <50 мс.

3. **Тестування картографії та клієнтської частини (Front-end).** Для створення тестової топології шахти (штреки, базові станції, руддвір) та симуляції руху використано десктопну утиліту «MineCAD» на PyQt6. Перевірку продуктивності вебінтерфейсу виконано безпосередньо в браузері. Завдяки HTML5 Canvas та оптимізованому рендерингу (кластеризація, відмальовування в одному canvas, анімації через requestAnimationFrame) частота кадрів (FPS) залишалася на рівні 60 кадрів/с навіть при одночасному переміщенні десятків працівників на 2D-карті.

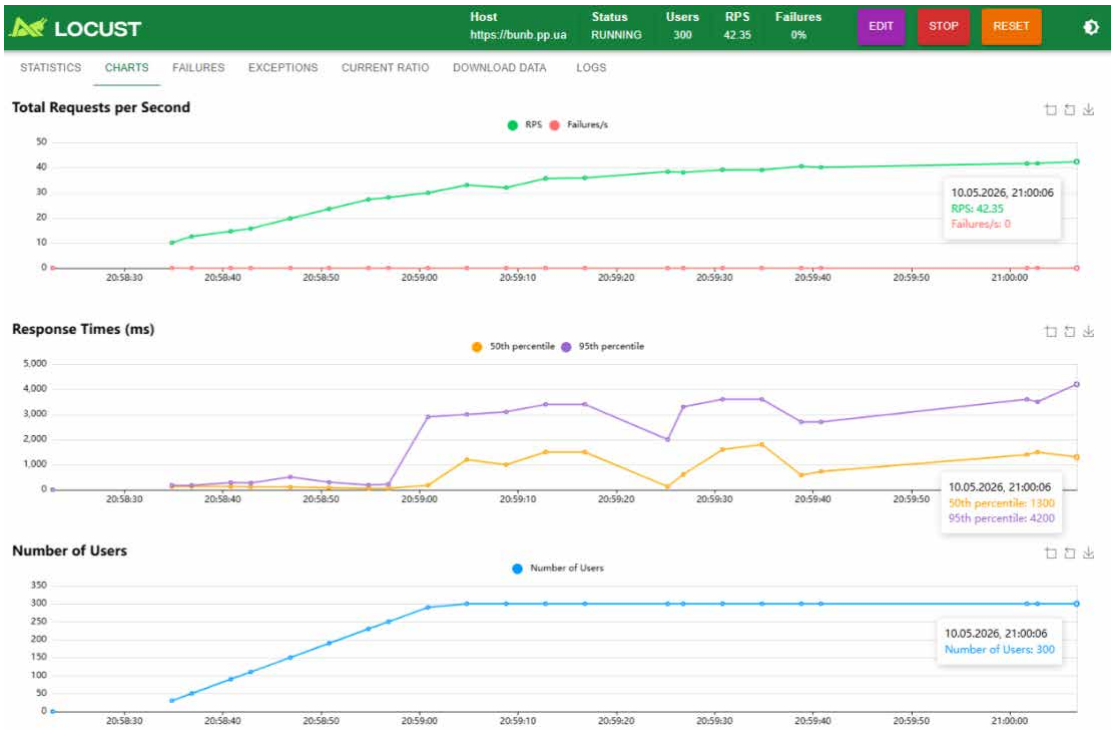

Рис. 4.1 Результати навантажувального стрес-тестування API в інтерфейсі Locust

Результати функціонального та навантажувального тестування підтверджують правильність архітектурних рішень та готовність програмно-апаратного комплексу до дослідно-промислової експлуатації.

4.2 Вимоги до апаратного та програмного забезпечення

Ефективна робота системи «Глибина 4.0» в умовах промислової експлуатації вимагає дотримання вимог до апаратного (Hardware) та програмного (Software) забезпечення. Зважаючи на багаторівневу розподілену архітектуру, вимоги логічно поділено на три категорії: серверна інфраструктура, робоче місце диспетчера та апаратний IoT-рівень.

Вимоги до серверної інфраструктури.

Серверна частина відповідає за прийом телеметрії, збереження даних у реляційній базі та виконання бізнес-логіки обробки тривог в умовах високого

навантаження (HighLoad). Мінімальні та рекомендовані вимоги до сервера наведено в таблиці 4.4.

Таблиця 4.4 Вимоги до серверної інфраструктури

Компонент	Вимоги	Обґрунтування / Примітка
Процесор (CPU)	x86_64, мінімум 4 фізичні ядра (рівня Intel Xeon або AMD EPYC)	Необхідно для паралельної обробки потоків WSGI-сервера (Gunicorn).
Оперативна пам'ять (RAM)	Не менше 8 ГБ	Для забезпечення швидкої роботи In-memory СУБД Redis та кеш-буферів PostgreSQL.
Дискова підсистема	SSD або NVMe, від 100 ГБ	Висока швидкість IOPS критично важлива для безперервного запису часових рядів телеметрії.
Операційна система	Linux (Ubuntu Server 22.04 LTS або Debian 12)	Забезпечує найвищу стабільність та безпеку для розгортання вебдодатків.
Додаткове ПЗ	Docker Engine 24.0+, Docker Compose	Для ізольованого контейнерного розгортання всіх мікросервісів системи.

Вимоги до робочого місця диспетчера

Основне обчислювальне навантаження лежить на сервері, але клієнтська частина використовує HTML5 Canvas для рендерингу та анімації переміщення персоналу на карті в реальному часі. Цей процес виконується на стороні клієнта. Вимоги наведено в таблиці 4.5.

Таблиця 4.5 Вимоги до клієнтського робочого місця

Компонент	Вимоги	Обґрунтування / Примітка
Операційна система	Windows 10/11 Pro або Linux (Ubuntu 22.04/24.04 LTS)	Забезпечення кросплатформності. Ubuntu LTS рекомендовано завдяки стабільній нативній підтримці пропрієтарних драйверів (NVIDIA/AMD) для апаратного прискорення графіки.
Процесор (CPU)	Intel Core i3 / AMD Ryzen 3 (або краще)	Забезпечення плавної роботи скриптів браузера (JavaScript ES6+).
Відеокарта (GPU)	Інтегрована або дискретна з підтримкою WebGL	Делегування графічних розрахунків Canvas на відеокарту для утримання 60 FPS.
Оперативна пам'ять	Від 4 ГБ	Для рендерингу векторної топології штреків без зависань.
Дисплей	Роздільна здатність 1920×1080 (Full HD)	Необхідно для зручного огляду розгалуженої карти шахти та аналітичних дашбордів на одному екрані.

Компонент	Вимоги	Обґрунтування / Примітка
Веббраузер	Google Chrome 100+, Mozilla Firefox 90+, Edge	Обов'язкова підтримка технологій AJAX, WebSocket та увімкнене апаратне прискорення.

Вимоги до апаратного рівня (IoT-пристроїв)

Мобільні модулі персоналу та стаціонарні датчики мікроклімату повинні гарантувати надійний збір і передачу даних навіть за нестабільного зв'язку. Апаратні вимоги до вузлів наведено в таблиці 4.6.

Таблиця 4.6 Вимоги до апаратного забезпечення IoT-рівня

Компонент	Вимоги	Обґрунтування / Примітка
Мікроконтролер	ESP32 (Xtensa Dual-Core 32-bit LX6)	Наявність 2 ядер та вбудованого Wi-Fi модуля дозволяє паралельно обробляти сенсори і підтримувати мережеве з'єднання.
Модулі сенсорів	Датчик газу (MQ-7), датчик мікроклімату (DHT22), датчик вібрації (SW-420)	Забезпечують безперервний вимір температури, вологості, наявності небезпечних газів та фіксацію нерухомості (імобілізації) працівника.
Периферія вводу/виводу	Дисплей OLED 128x64 (I2C), активний зумер (Buzzer), тактова кнопка	Необхідні для візуального та звукового сповіщення гірника про небезпеку, статус мережі, а також для подачі ручного сигналу SOS.
Бездротовий зв'язок	Wi-Fi 802.11 b/g/n (2.4 ГГц)	Забезпечення зв'язку з базовими станціями (репітерами) у штреках та підтримка Wi-Fi роумінгу.
Живлення та АЦП	Акумулятор 18650 або Li-Po (3.7V) з BMS, дільник напруги	Автономна робота протягом зміни. Дільник напруги підключений до АЦП (ADC) мікроконтролера для моніторингу відсотка заряду.
Пам'ять (Flash)	Не менше 4 МБ	Використовується для організації локальної офлайн-черги при втраті з'єднання та збереження бінарників для OTA-оновлень.

4.3 Склад інсталяційного пакету

Враховуючи сучасні стандарти інженерії ПЗ та мікросервісну архітектуру розробленого комплексу, класичне поняття «інсталяційний файл» замінено на інфраструктуру як код (Infrastructure as Code, IaC).

Інсталяційний пакет системи «Глибина 4.0» – це єдиний репозиторій вихідного коду (розміщений на GitHub) у поєднанні з конфігураційними файлами для Docker [6]. Такий підхід вирішує проблему сумісності залежностей («працює на моєму комп'ютері») і дозволяє розгорнути систему на будь-якому сервері за лічені хвилини.

Структура інсталяційного пакета (репозиторію).

Коренева директорія містить такі логічні блоки:

1. *diploma/* – головний додаток бізнес-логіки (моделі БД, Smart Alert Engine, API-контролери, підсистема OTA-оновлень).
2. *myproject/* – конфігурація ядра Django (налаштування безпеки, маршрутизація, інтеграція з БД).
3. *templates/* та *static/* – файли рівня представлення (HTML-шаблони, CSS, JavaScript-модулі: mine_map.js для Canvas-карти та Chart.js для дашбордів).
4. *requirements.txt* – специфікація версій усіх Python-бібліотек.
5. *Dockerfile* та *docker-compose.yml* – маніфести для збирання та оркестрації серверного середовища.
6. *.env.example* – шаблон файлу змінних середовища для безпечного зберігання секретних ключів.

Оркестрація контейнерів (Docker Compose).

Основа інсталяційного пакета – маніфест docker-compose.yml [6]. Він описує конфігурацію інфраструктури з трьох ізольованих контейнерів (сервісів), об'єднаних у спільну віртуальну мережу:

- **Сервіс *db*** – розгортає офіційний образ PostgreSQL. Контейнер автоматично конфігурує БД, користувача та пароль на основі змінних середовища. Дані зберігаються у Docker Volume, що гарантує збереження телеметрії навіть при повному видаленні контейнера.
- **Сервіс *redis*** – розгортає in-memory СУБД для кешування.

- **Сервіс *web*** – збирає образ на базі Linux та Python 3.10+, встановлює залежності з requirements.txt і запускає Django під управлінням production-сервера Gunicorn.

Алгоритм розгортання та конфігурації на новому сервері.

Процес інсталяції максимально автоматизований і складається з таких кроків:

1. **Клонування репозиторію** – адміністратор завантажує актуальну версію пакета з GitHub на цільовий сервер.
2. **Конфігурація безпеки** – на основі .env.example створюється файл .env, де адміністратор задає унікальні параметри (паролі до БД, SECRET_KEY для Django, ESP32_API_KEY для захисту IoT-трафіку, ключі SMTP для email-сповіщень). Жодні паролі не зберігаються у відкритому коді.
3. **Ініціалізація інфраструктури** – командою docker compose up -d --build адміністратор запускає завантаження образів і створення віртуальної мережі.
4. **Підготовка бази даних** – після старту контейнерів у сервісі web виконується python manage.py migrate, яке створює всі таблиці (зокрема для телеметрії) та індекси (BRIN) у PostgreSQL.
5. **Створення суперкористувача** – адміністратор генерує перший обліковий запис диспетчера для доступу до вебінтерфейсу.

Завдяки ізольованим Docker-контейнерам інсталяція системи не порушує роботу інших сервісів на сервері підприємства, а оновлення ПЗ зводиться до перезапуску контейнера з новою версією коду.

Автоматизація процесів тестування та розгортання (CI/CD)

Для надійності коду та унеможливлення потрапляння помилок на робочий сервер (Production) процес оновлення доповнено конвеєром безперервної інтеграції та доставки (CI/CD).

Оскільки код розміщено на GitHub, для автоматизованого конвеєра використано вбудований інструментарій **GitHub Actions**. У лістингу 4.2 наведено фрагмент конфігураційного файлу (Workflow) у форматі YAML, який запускається автоматично при кожному злитті (push/pull request) коду в гілку main.

Лістинг 4.2 Фрагмент конфігураційного файлу CI/CD конвеєра GitHub Actions

```yaml
name: Django CI/CD Pipeline

on:
  push:
    branches: [ «main» ]

jobs:
  test_and_build:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_DB: mine_db
          POSTGRES_USER: admin
          POSTGRES_PASSWORD: secretpassword
        ports:
          - 5432:5432
      redis:
        image: redis:latest
        ports:
          - 6379:6379

    steps:
    - name: Checkout repository code
      uses: actions/checkout@v3

    - name: Set up Python 3.10
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'
        cache: 'pip'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
```

```
    - name: Run Unit Tests (Alert Engine)
      env:
        DJANGO_SETTINGS_MODULE: myproject.settings
        DATABASE_URL:
postgres://admin:secretpassword@localhost:5432/mine_db
      run: |
        python manage.py test diploma.tests

    - name: Build Docker Image
      if: success()
      run: docker build -t mine-safety-system:latest .
```

Розроблений CI/CD конвеєр (лістинг 4.2) складається з кількох автоматизованих етапів. На першому етапі система розгортає віртуальне середовище Ubuntu, піднімає тестові сервіси PostgreSQL та Redis, встановлює залежності з requirements.txt.

Ключовий етап – **тестування** (Run Unit Tests), на якому конвеєр автоматично запускає модульні тести (зокрема AlertEngineTestCase з підрозділу 4.1). Якщо хоча б один тест завершується невдало (наприклад, порушено логіку дедуплікації), виконання конвеєра переривається, а розробник отримує сповіщення. Виключно після успішного проходження всіх тестів (if: success()) система переходить до компіляції оновленого Docker-образу.

Впровадження CI/CD дозволило виключити вплив людського фактору на етапі тестування нових версій ПЗ, що є обов'язковою вимогою стандартів якості для систем індустріальної безпеки.

Скріншоти основних екранів системи наведено у Додатку Б

ВИСНОВКИ

У кваліфікаційній бакалаврській роботі вирішено актуальне науково-практичне завдання – розроблено програмно-апаратне забезпечення для кіберфізичної системи моніторингу параметрів безпеки шахти «Глибина 4.0». За результатами виконання роботи можна зробити такі висновки.

1. Системний аналіз предметної області. Виявлено, що більшість наявних систем безпеки на гірничодобувних підприємствах мають закриту пропрієтарну архітектуру (vendor lock-in) та не забезпечують безперервного персонального газового контролю. На основі виявлених недоліків сформовано вимоги до системи та спроєктовано багаторівневу розподілену клієнт-серверну архітектуру.

2. Інформаційне забезпечення. Створено нормалізовану об'єктно-реляційну модель БД у PostgreSQL. Для вирішення проблеми надмірного споживання оперативної пам'яті при збереженні великих масивів часових рядів телеметрії застосовано BRIN-індекси. Це забезпечило оптимізацію БД для роботи в умовах високого навантаження (HighLoad).

3. Апаратний рівень та IoT-алгоритми. На базі ESP32 створено клієнтські мобільні та стаціонарні модулі, інтегровані з датчиками метану (MQ-7), мікроклімату (DHT22) та вібрації (SW-420). Реалізовано алгоритм локальної енергонезалежної офлайн-черги, який буферизує телеметрію у Flash-пам'яті при обриві Wi-Fi, гарантуючи нульову втрату критичних даних. Крім того, впроваджено підсистему бездротового оновлення прошивок (OTA Updates) з підтримкою поступового розгортання.

4. Серверна інфраструктура бізнес-логіки. На базі Django реалізовано ядро системи та REST API. Розроблено модуль Smart Alert Engine, який виконує каскадний аналіз небезпек, дедуплікацію аварійних повідомлень та запобігає інформаційному перевантаженню диспетчера. Для зменшення навантаження на дискову підсистему при частих запитах застосовано in-

memory кешування через Redis. Також створено модуль автоматичної архівації застарілих даних (Data Lifecycle Management).

5. **Диспетчерський вебінтерфейс.** Розроблено клієнтський додаток за принципами часткового SPA. Візуалізацію топології шахти та позиціювання персоналу виконано з використанням HTML5 Canvas API (2D context) та оптимізованого власного рендерингу (кластеризація, hit-testing, анімації через requestAnimationFrame, навігація flyTo). Це забезпечує стабільну продуктивність при одночасному відображенні динамічних об'єктів. Додатково реалізовано підсистему генерації звітів у форматі PDF (ReportLab).

6. **Кібербезпека та надійність.** Інтегровано захист від автоматизованих брутфорс-атак (Cloudflare Turnstile) та розроблено алгоритм відстеження несанкціонованого доступу з нових пристроїв із системою email-сповіщень.

7. **Тестування та інсталяційний пакет.** Навантажувальне тестування (Locust, скрипти генерації даних) підтвердило стабільність роботи API та СУБД при одночасній обробці запитів від сотень пристроїв. Розгортання програмного комплексу автоматизовано за допомогою Docker (Infrastructure as Code), що забезпечує кросплатформність та швидкість впровадження.

Обмеження розробки. Якість моніторингу в підземних умовах залежить від стабільності Wi-Fi покриття. Позиціювання за RSSI/BSSID є наближеним і потребує адаптації під конкретну топологію шахти. Газові датчики потребують періодичного калібрування. Повністю виключити хибні спрацювання неможливо, тому критичні події мають супроводжуватися підтвердженням диспетчера та аналізом історії телеметрії.

Отримані результати підтверджують, що розроблена система «Глибина 4.0» виконує всі поставлені завдання, відрізняється відмовостійкістю, масштабованістю та готова до пілотного впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гірничий закон України : Закон України від 06.10.1999 № 1127-XIV. URL: https://zakon.rada.gov.ua/laws/show/1127-14 (дата звернення: 01.05.2026).

2. Правила безпеки у вугільних шахтах : Наказ Міністерства енергетики та вугільної промисловості України від 12.03.2010 № 62. URL: https://zakon.rada.gov.ua/laws/show/z0236-10 (дата звернення: 01.05.2026).

3. Соммервілл І. Інженерія програмного забезпечення / пер. з англ. Київ : Основи, 2003. 512 с.

4. Cloudflare Turnstile Documentation. *Cloudflare Docs*. URL: https://developers.cloudflare.com/turnstile/ (дата звернення: 01.05.2026).

5. Django documentation. *Django Project*. URL: https://docs.djangoproject.com/ (дата звернення: 01.05.2026).

6. Docker Compose Overview. *Docker Documentation*. URL: https://docs.docker.com/compose/ (дата звернення: 01.05.2026).

7. ESP32 Wi-Fi API Guides. *Espressif Documentation*. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html (дата звернення: 02.05.2026).

8. ESPHome 2026.1.0: Improved Security, Wi-Fi Roaming for ESP32 Telemetry. *Hackster.io*. URL: https://www.hackster.io/news/esphome-2026-1-0-brings-improved-security-wi-fi-roaming-and-far-better-support-for-the-esp8266-7c37a060e96a (дата звернення: 15.04.2026).

9. Konva.js: 2D HTML5 Canvas framework for desktop and mobile applications. *Konva.js Official Site*. URL: https://konvajs.org/ (дата звернення: 15.04.2026).

10. Locust: An open source load testing tool. *Locust*. URL: https://locust.io/ (дата звернення: 02.05.2026).

11. PostgreSQL 16 Documentation: BRIN Indexes. *PostgreSQL*. URL: https://www.postgresql.org/docs/16/brin.html (дата звернення: 01.05.2026).

12. PostgreSQL 16 Documentation: JSON Types. *PostgreSQL*. URL: https://www.postgresql.org/docs/16/datatype-json.html (дата звернення: 01.05.2026).

13. Powering Django: A Deep Dive into CPU Cores, Threads, and Gunicorn Workers for High-Performance Deployment. *Chadura*. URL: https://chadura.com/blogs/powering-django-a-deep-dive-into-cpu-cores-threads-gunicorn-workers-and-unix-sockets-for-high-performance-deployment/ (дата звернення: 15.04.2026).

14. Rathi B. [та ін.]. Realizing the potential of Internet of Things (IoT) in Industrial applications. *Springer*. URL: https://link.springer.com/article/10.1007/s43926-025-00141-5 (дата звернення: 15.04.2026).

15. Redis Documentation. *Redis*. URL: https://redis.io/docs/ (дата звернення: 01.05.2026).

16. ReportLab PDF Library. *ReportLab*. URL: https://www.reportlab.com/opensource/ (дата звернення: 02.05.2026).

ДОДАТОК А

ФРАГМЕНТИ ВИХІДНОГО КОДУ ПРОГРАМНОЇ СИСТЕМИ

Лістинг коду прошивки мікроконтролера ESP32 (апаратний рівень)

```
/*

================================================================================
===
  ПРОЄКТ: ГЛИБИНА 4.0 (LOGICAL STARTUP EDITION) - АДАПТОВАНА ВЕРСІЯ
  ESP32 + MQ-7 + DHT22 + SW-420 + OLED 128x64

================================================================================
===
*/

#include <WiFi.h>
#include <WebServer.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <DHT.h>
#include <Preferences.h>
#include <nvs_flash.h>
#include <esp_wifi.h>
#include <vector>
#include <HTTPUpdate.h>
#include <queue>

// --- НАЛАШТУВАННЯ СЕРВЕРА ---
const char* SERVER_ADDRESS = «http://bunb.pp.ua»;
const char* TELEMETRY_ENDPOINT = «/diploma/api/telemetry/»;
const char* WIFI_API_ENDPOINT = «/diploma/api/wifi-networks/»;
const char* OTA_CHECK_ENDPOINT = «/diploma/api/ota/check/»;
const char* OTA_LOG_ENDPOINT = «/diploma/api/ota/log/»;
const char* API_KEY = «EXAMPLE_KEY»;  // Ключ доступу до API сервера

#define FIRMWARE_VERSION «1.0.1»

// --- ТАЙМЕРИ ТА ІНТЕРВАЛИ ---
#define DATA_SEND_INTERVAL_MS 10000UL
#define NO_MOTION_TIMEOUT_MS 10000UL
#define SOS_GRACE_PERIOD_MS 5000UL
#define LONG_PRESS_SOS_MS 5000UL
#define WARMUP_DURATION_MS 120000UL
#define WIFI_SCAN_INTERVAL_MS 10000UL

// --- ПІНИ ---
```

```
#define PIN_MQ7_ANALOG 34
#define PIN_BUTTON_SOS 19
#define PIN_SW420 18
#define PIN_BUZZER 4
#define PIN_DHT 13
#define PIN_BATTERY_ADC 35

// --- СТАНИ СИСТЕМИ ---
enum SystemState {
  PROVISIONING,
  ASK_CALIBRATION,
  WARMING_UP,
  CONNECT_CHECKPOINT,
  READY,
  SOS_MODE,
  CHARGING_SLEEP
};
SystemState currentState;

// --- ОБ'ЄКТИ ---
Adafruit_SSD1306 display(128, 64, &Wire, -1);
DHT dht(PIN_DHT, DHT22);
Preferences preferences;

// Тут був баг: якщо вимкнути репітер, ESP зависає на HTTP.begin.
// Вирішив обгорнути чергою. Якщо немає Wi-Fi, просто кладемо пакет у
telemetryQueue.
// Важливо: MAX_QUEUE_SIZE = 30, бо пам'яті на ESP32 всього 520Кб, більше
не вивезе.
std::queue<String> telemetryQueue;
const size_t MAX_QUEUE_SIZE = 30;

// Глобальні змінні
unsigned long lastVibrationMs = 0;
unsigned long lastDataSendMs = 0;
bool isSosActive = false;
String activeSosReason = «NONE»;
volatile bool motionDetected = false;

// Апаратне переривання (Рух)
void IRAM_ATTR motionISR() {
  motionDetected = true;
}

// Зчитування газу (Deadband 100)
int getCleanGas() {
  int raw = analogRead(PIN_MQ7_ANALOG);
  return (raw <= 100) ? 0 : (raw - 100);
}

// Відсоток заряду
int getBatteryPct() {
```

```
  int raw = analogRead(PIN_BATTERY_ADC);
  if (raw < 1000) return 100;
  return constrain(map(raw, 3100, 4095, 0, 100), 0, 100);
}

// Надсилання даних на сервер (із тайм-аутом та офлайн-чергою)
void sendTelemetry(bool isSos, String reason = «Normal») {
  StaticJsonDocument<384> doc;
  doc[«mac_address»] = WiFi.macAddress();
  doc[«ap_uid»] = WiFi.BSSIDstr();
  doc[«battery»] = getBatteryPct();
  doc[«gas_level»] = getCleanGas();
  doc[«is_sos»] = isSos;
  doc[«reason»] = reason;
  doc[«rssi»] = WiFi.RSSI();
  doc[«fw_version»] = FIRMWARE_VERSION;
  doc[«is_moving»] = (millis() - lastVibrationMs < NO_MOTION_TIMEOUT_MS);

  String json;
  serializeJson(doc, json);

  if (WiFi.status() != WL_CONNECTED) {
    if (telemetryQueue.size() < MAX_QUEUE_SIZE) telemetryQueue.push(json);
    return;
  }

  HTTPClient http;
  http.setTimeout(3000);
  http.begin(String(SERVER_ADDRESS) + TELEMETRY_ENDPOINT);
  http.addHeader(«X-API-Key», API_KEY);
  http.addHeader(«Content-Type», «application/json»);
  int httpCode = http.POST(json);

  if (httpCode == 200 || httpCode == 201) {
    lastDataSendMs = millis();
    String payload = http.getString();
    StaticJsonDocument<128> respDoc;
    if (!deserializeJson(respDoc, payload)) {
      if (respDoc[«command»] == «REBOOT») {
        esp_restart();
      }
    }
  } else {
    if (telemetryQueue.size() < MAX_QUEUE_SIZE) telemetryQueue.push(json);
  }
  http.end();
}

// Головний цикл
void loop() {
  // Обробка переривання від датчика руху
  if (motionDetected) {
```

```
    motionDetected = false;
    lastVibrationMs = millis();
  }

  // Обробка черги телеметрії
  if (WiFi.status() == WL_CONNECTED && !telemetryQueue.empty()) {
    processTelemetryQueue();
  }

  // Автоматичне завершення зміни (Зарядка в ламповій)
  int bat = getBatteryPct();
  static int lowestBat = bat;
  if (bat >= lowestBat + 3 && WiFi.SSID() == «AP-SURFACE») {
    sendTelemetry(false, «END_SHIFT»);
    WiFi.disconnect(true);
    currentState = CHARGING_SLEEP;
  }
}
```

Лістинг програмного модуля обробки телеметрії та розумних тривог (Smart Alert Engine)

```
@csrf_exempt
@api_key_required
@transaction.atomic
def api_receive_telemetry(request):
    ««««API для прийому POST-запитів від ESP32 з дедуплікацією тривог.»»»»
    if request.method == 'POST':
        try:
            data = json.loads(request.body)
            mac_address = data.get('mac_address')
            ap_uid = data.get('ap_uid', '')
            battery = int(data.get('battery', 100))
            gas_level = float(data.get('gas_level', data.get('gas_co',
0)))
            is_sos = data.get('is_sos', False)
            reason_text = data.get('reason', 'Normal')
            rssi = int(data.get('rssi', 0))
            temperature = float(data.get('temperature', 0.0))
            humidity = float(data.get('humidity', 0.0))
            fw_version = data.get('fw_version')
            is_moving = data.get('is_moving', True)

            if not mac_address:
                return JsonResponse({'status': 'error', 'message': 'MAC-
адреса не надана'}, status=400)

            device =
MinerDevice.objects.select_related('assigned_to').filter(mac_address__iexa
ct=mac_address).first()
            if not device:
```

```
                return JsonResponse({'status': 'error', 'message':
'Пристрій не зареєстровано'}, status=404)

            command = «REBOOT» if device.pending_reboot else None
            if device.pending_reboot:
                device.pending_reboot = False
            if not device.is_active:
                device.is_active = True
            device.save()

            employee = device.assigned_to
            ap =
InfrastructureDevice.objects.filter(Q(wifi_bssid__iexact=ap_uid) |
Q(uid__iexact=ap_uid)).first()

            # Збереження логу телеметрії (Time-Series)
            TelemetryLog.objects.create(
                device=device,
                connected_repeater=ap,
                battery_level=battery,
                gas_level=gas_level,
                wifi_signal_strength=rssi,
                is_sos=is_sos,
                temperature=temperature,
                humidity=humidity,
                is_moving=is_moving
            )

            # АВТО-ЗАКРИТТЯ тривоги про «Втрату зв'язку», якщо пристрій
знову вийшов на зв'язок
            offline_alerts =
SecurityAlert.objects.filter(employee=employee, is_resolved=False,
reason__icontains=«Втрата зв'язку»)
            if offline_alerts.exists():
                offline_alerts.update(
                    status='RESOLVED',
                    is_resolved=True,
                    resolved_at=timezone.now(),
                    rescue_notes=«Автоматично: Зв'язок відновлено.»
                )

            # ЛОГІКА ЗАВЕРШЕННЯ ЗМІНИ (Лампова / Док-станція)
            if reason_text == 'END_SHIFT':
                if ap and ap.uid == 'AP-SURFACE':
                    has_critical =
SecurityAlert.objects.filter(employee=employee,
is_resolved=False).exclude(status='WARNING').exists()
                    if has_critical:
                        return JsonResponse({'status': 'error', 'message':
'Є відкрита КРИТИЧНА тривога!'}, status=400)
```

```
                    warnings =
SecurityAlert.objects.filter(employee=employee, is_resolved=False,
status='WARNING')
                    if warnings.exists():
                        warnings.update(status='RESOLVED',
is_resolved=True, resolved_at=timezone.now())

                    if employee.safety_status != 'OFF_SHIFT':
                        employee.safety_status = 'OFF_SHIFT'
                        employee.save()
                    device.is_active = False
                    device.save()
                    return JsonResponse({'status': 'success', 'message':
'Зміну завершено.', 'command': command})
                else:
                    return JsonResponse({'status': 'error', 'message':
'Завершення зміни можливе ТІЛЬКИ в ламповій!'}, status=400)

            # ЛОГІКА СТВОРЕННЯ ТРИВОГ (Каскадний аналіз)
            alert_reason = None
            if gas_level >= 50:
                alert_reason = f'КРИТИЧНИЙ рівень CH4: {gas_level}% LEL
(Негайна евакуація!)'
            elif gas_level > 17:
                alert_reason = f'Перевищення ГДК CH4: {gas_level}% LEL'
            elif is_sos:
                alert_reason = 'Ручний виклик SOS'
            elif not is_moving:
                alert_reason = 'Бездіяльність (Немає руху / Можливе
падіння)'

            if alert_reason:
                # Дедуплікація тривог
                active_alert =
SecurityAlert.objects.filter(employee=employee,
is_resolved=False).exclude(status='WARNING').first()
                if not active_alert:
                    SecurityAlert.objects.create(
                        employee=employee, device=device,
connected_repeater=ap,
                        reason=alert_reason, status='NEW'
                    )
                else:
                    active_alert.connected_repeater = ap
                    if «КРИТИЧНИЙ» in alert_reason:
                        active_alert.reason = alert_reason
                    active_alert.save()

                new_status = 'SOS' if (is_sos or gas_level >= 50) else
'WARNING'
                if employee.safety_status != new_status:
                    employee.safety_status = new_status
```

```
                    employee.save()
            else:
                if not SecurityAlert.objects.filter(employee=employee,
is_resolved=False).exists():
                    if employee.safety_status != 'OK':
                        employee.safety_status = 'OK'
                        employee.save()

            return JsonResponse({'status': 'success', 'command': command})

        except Exception as e:
            return JsonResponse({'status': 'error', 'message': str(e)},
status=400)
    return JsonResponse({'status': 'invalid_method'}, status=405)
```

ДОДАТОК Б

Скріншоти інтерфейсу кіберфізичної системи моніторингу «Глибина 4.0»

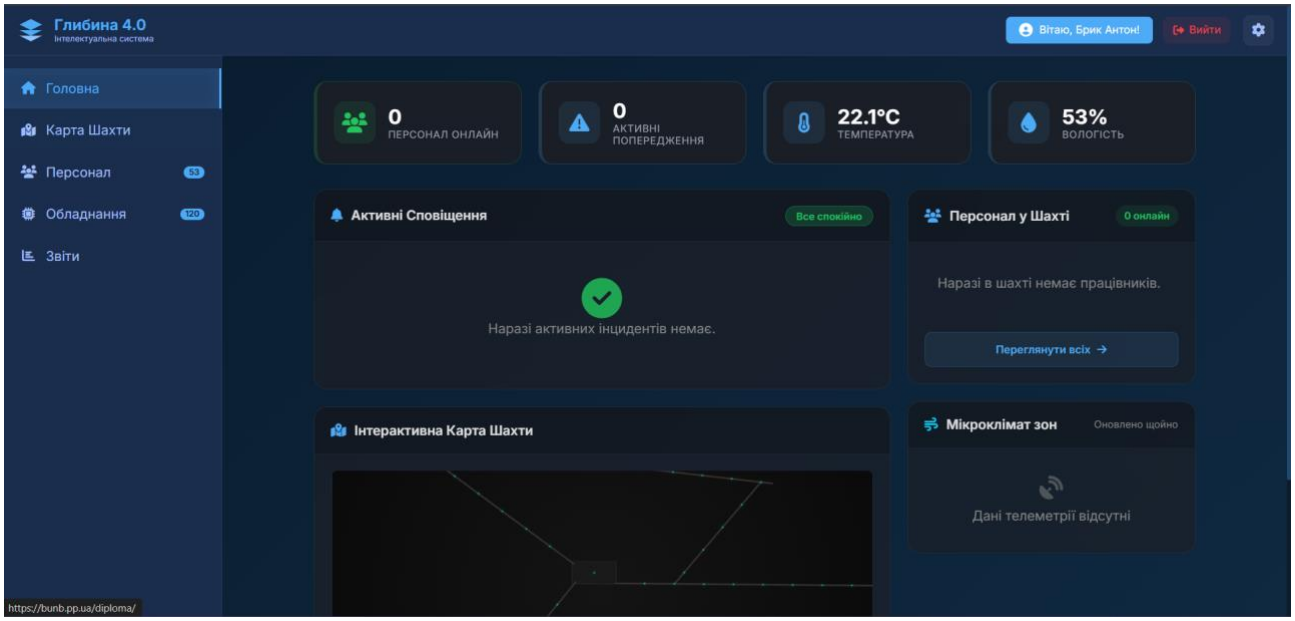

Рис. Б.1 Головний дашборд диспетчера

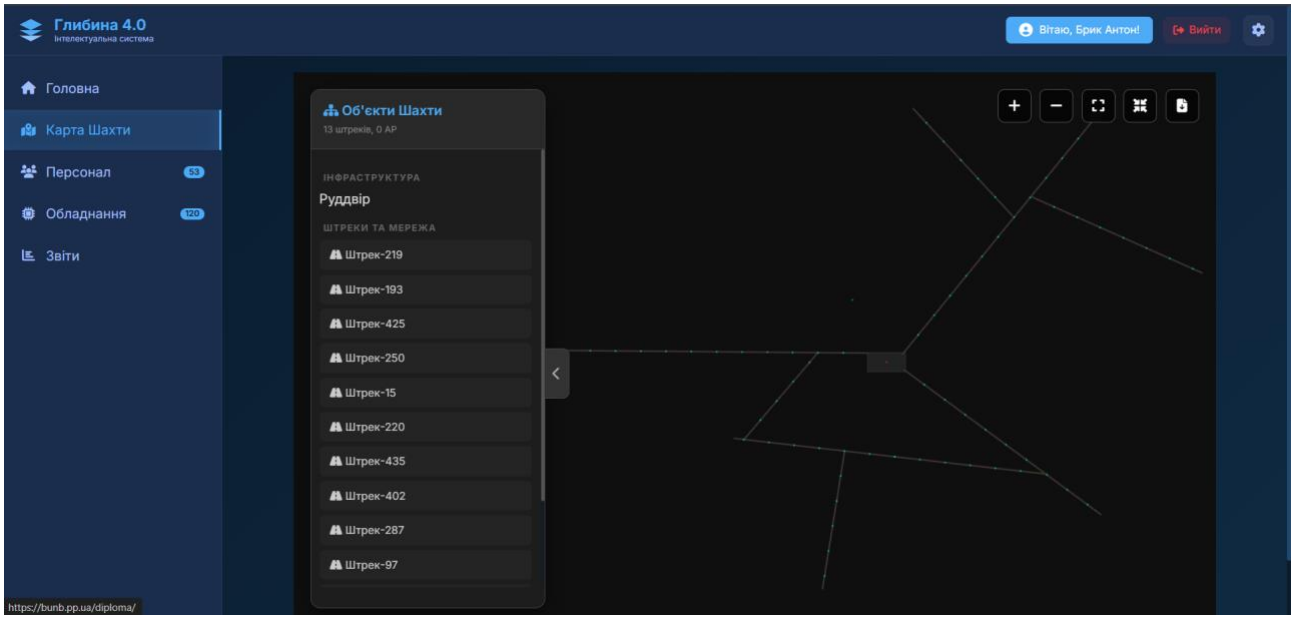

Рис. Б.2 2D-карта шахти з відображенням персоналу

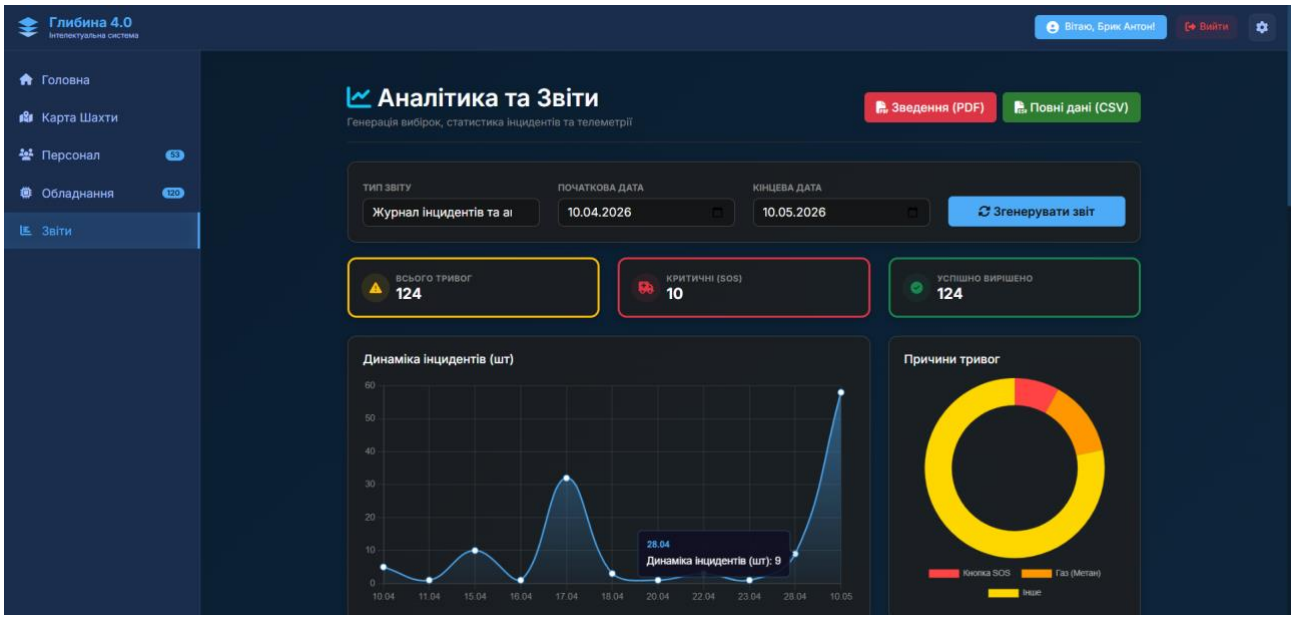

Рис. Б.3 Сторінка генерації аналітичних звітів

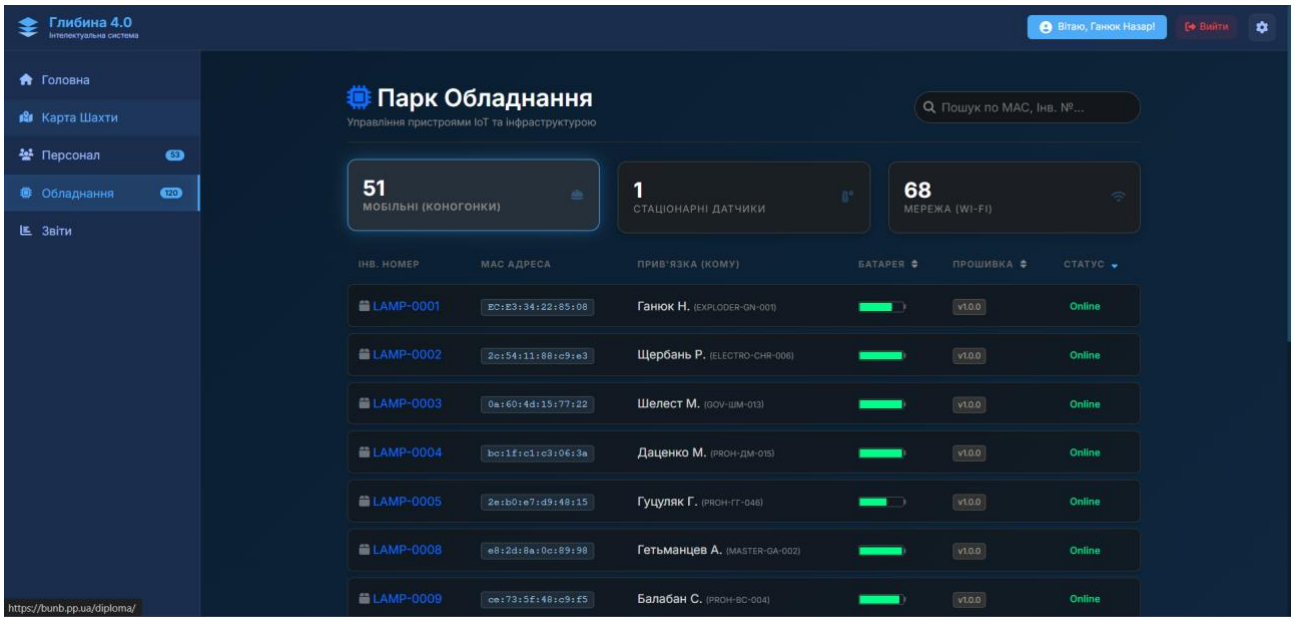

Рис. Б.4 Панель моніторингу та управління обладнанням

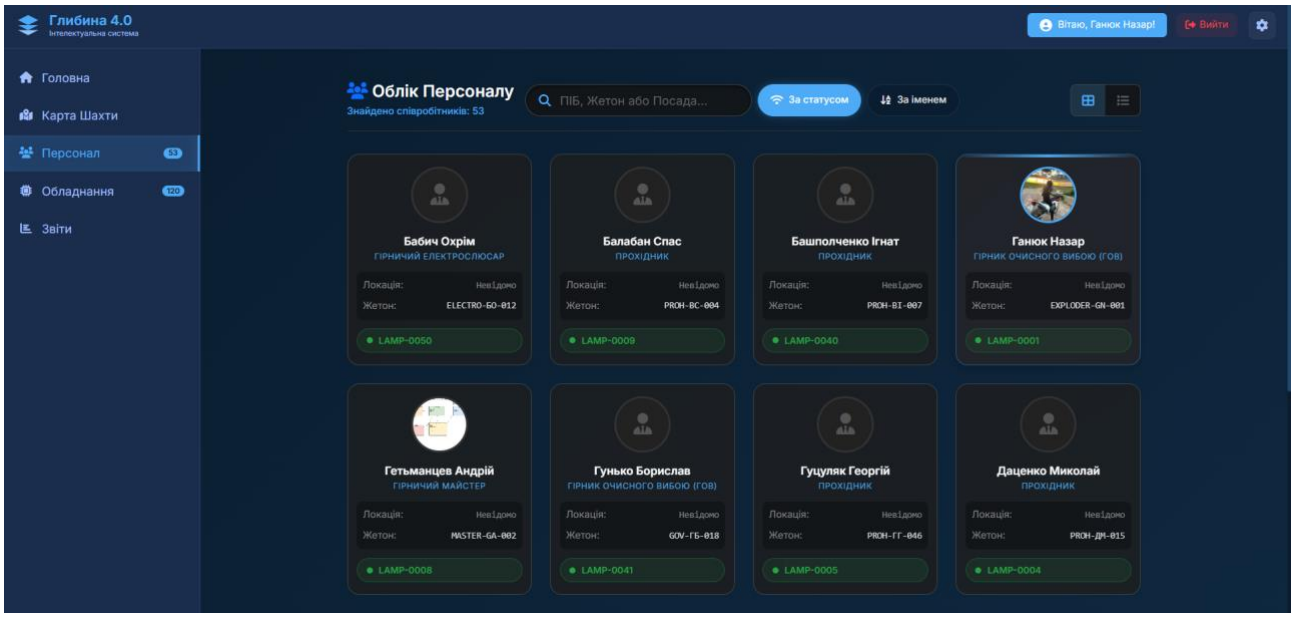

Рис. Б.5 Реєстр персоналу та оперативний статус безпеки

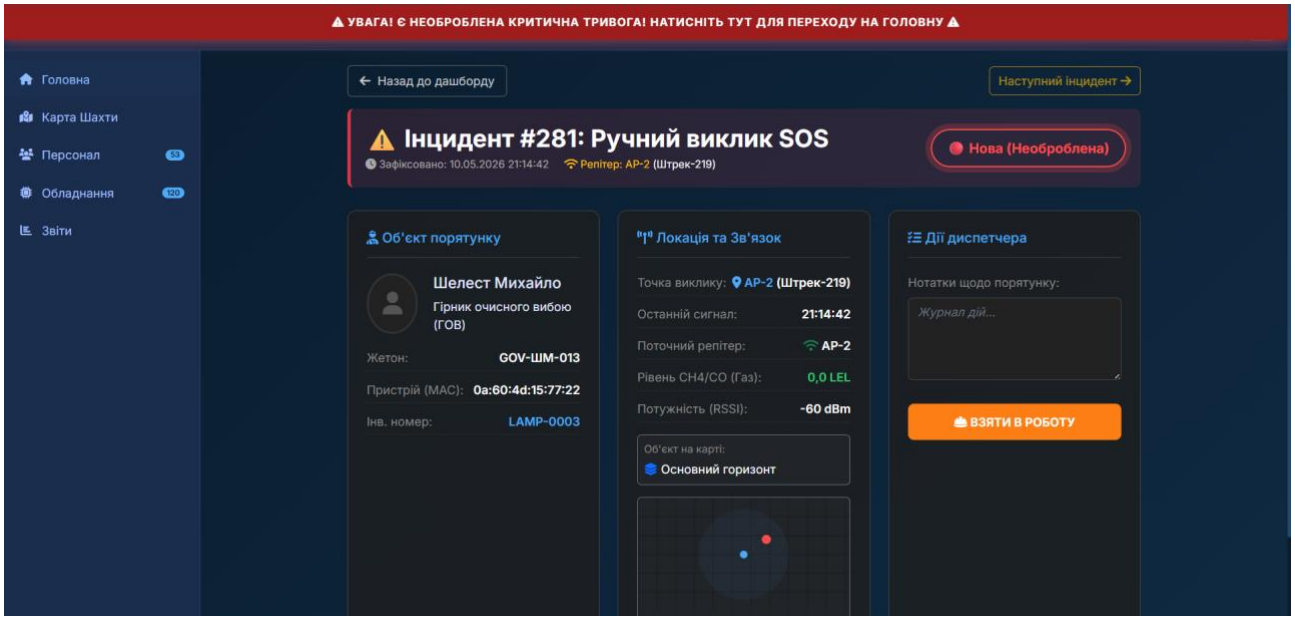

Рис. Б.6 Детальна картка інциденту та панель прийняття рішень

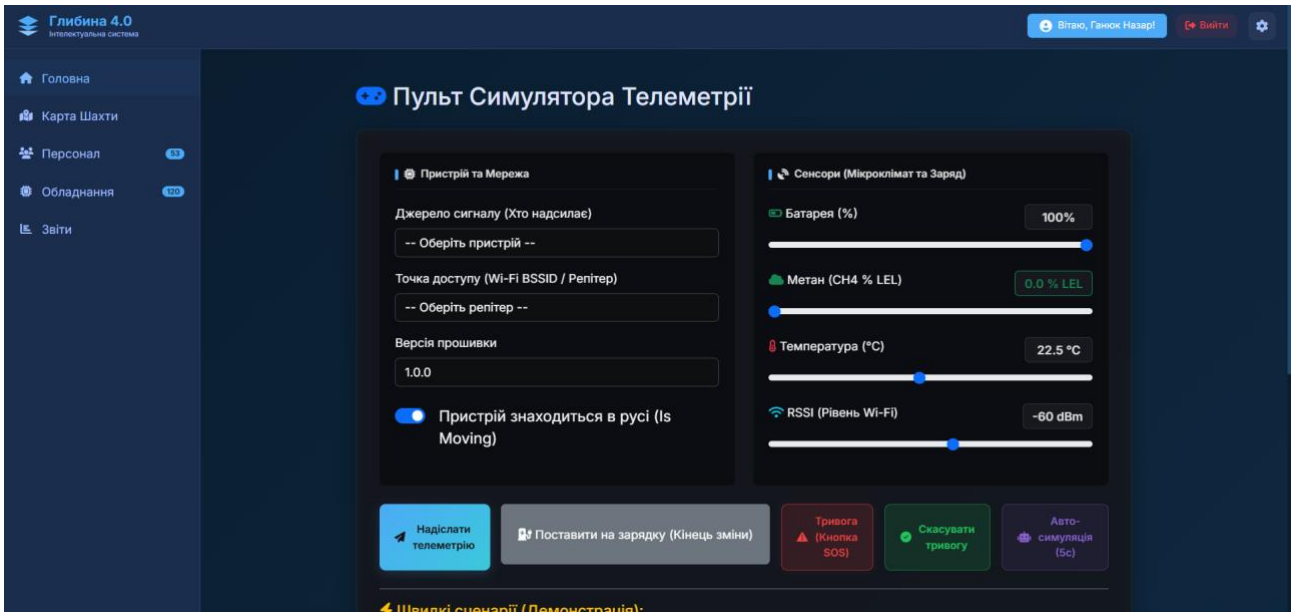

Рис. Б.7 Інтерфейс симулятора параметрів безпеки для демонстрації

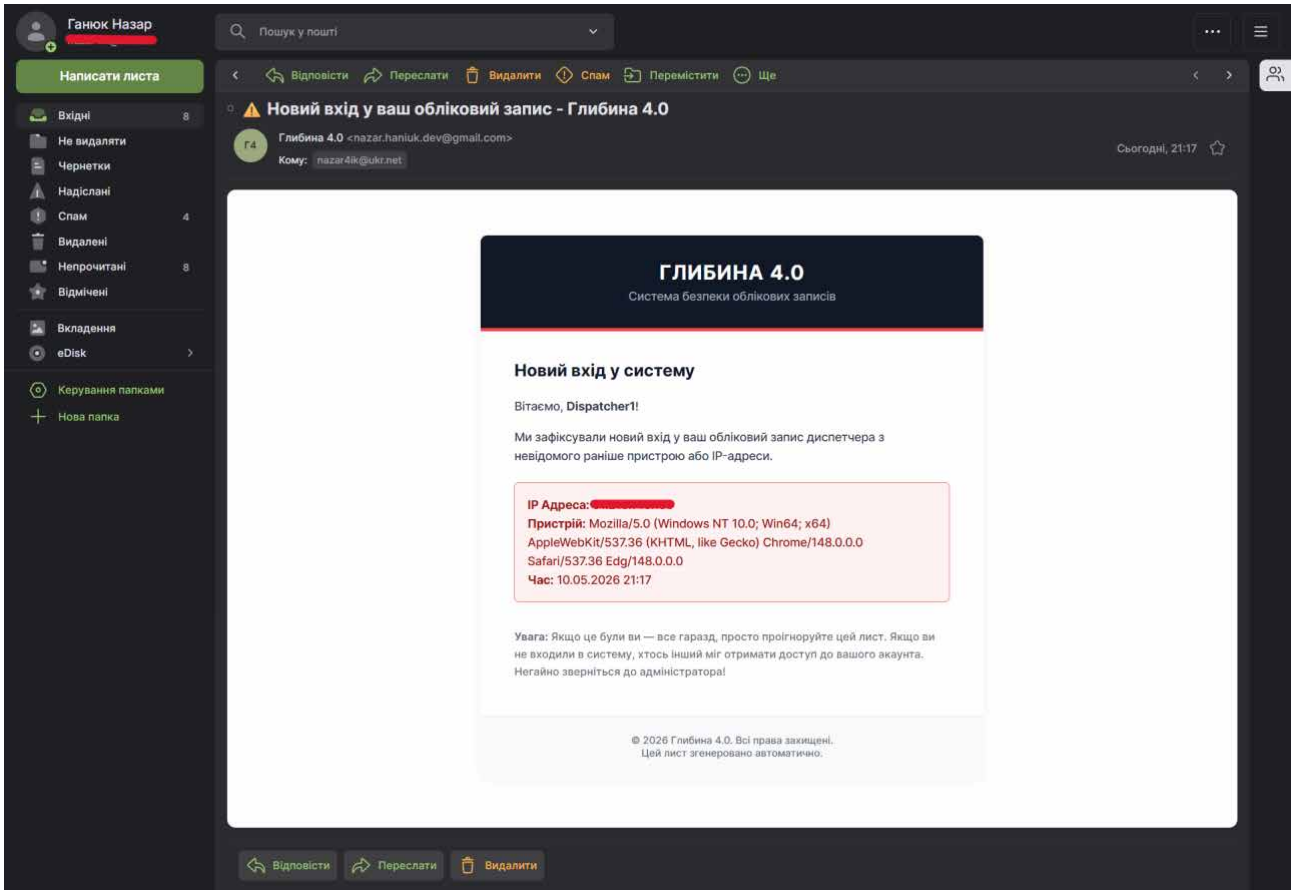

Рис. Б.8 Приклад автоматичного Email-сповіщення підсистеми кібербезпеки

ДОДАТОК В

Посилання на GitHub репозиторій проєкту – https://github.com/Nazoferon/Integrated-mine-monitoring-and-safety-system

ДОДАТОК Д

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

ДЕКЛАРАЦІЯ ЗДОБУВАЧА

Я, Ганюк Назар Олександрович, здобувач НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота (проєкт) на тему «Програмне забезпечення системи моніторингу параметрів безпеки шахти» є результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що:
    - [ ] інструменти генеративного ШІ не використовувалися.
    - [✓] інструменти ШІ (Gemini, DeepL) були використані для:
        - [✓] перекладу іншомовних джерел;
        - [ ] стилістичного редагування та перевірки граматики;
        - [✓] допомоги у написанні/відлагодженні програмного коду;
        - [✓] інше: генерація зображень.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___» _____ 2026 р. _____**Назар ГАНЮК**
(підпис)