НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО **Декан факультету**	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ** **Завідувач кафедри інформаційних систем та технологій**
_____ **Ігор БОЛБОТ** (підпис)	_____ **Михайло ШВИДЕНКО** (підпис)
«___» _____ 2026 р.	«___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Автоматизована система управління логістикою мережі магазинів»

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Гарант освітньої програми

д.е.н., професор _____ **Роман РУДЕНСЬКИЙ**
(підпис)

Керівник бакалаврської кваліфікаційної роботи

к.е.н, доцент _____ **Максим МОКРІЄВ**
(підпис)

Виконав _____ **Дмитро БАЗЯК**
(підпис)

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних систем та технологій

к.т.н., доцент _____ Михайло ШВИДЕНКО
(підпис)

«___» _____ 2026 р.

ЗАВДАННЯ ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Базяку Дмитру Олександровичу
(прізвище, ім'я, по батькові)

Спеціальність «122 Комп'ютерні науки»
Освітньо-професійна програма «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи
«Автоматизована система управління логістикою мережі магазинів»
затверджена наказом від «06» січня 2026 р. № 46 «С»

Термін подання завершеної роботи на кафедру «_22__» травня 2026 р.

Вихідні дані до роботи: опис процесів управління логістикою мережі магазинів та організації доставки товарів; вимоги до web-орієнтованої автоматизованої системи управління логістичними операціями; структура даних про магазини, склади, товари, замовлення, маршрути та транспортні засоби; технології Python, FastAPI, PostgreSQL/PostGIS, Vue.js, JavaScript, Leaflet та Google OR-Tools для реалізації інтерактивного web-застосунку управління логістикою. .

Перелік питань, що підлягають дослідженню: аналіз предметної області та постановка задачі управління логістичними процесами; проєктування архітектури системи та інформаційного забезпечення; розроблення програмного забезпечення автоматизованої системи управління логістикою; тестування, впровадження та оцінка ефективності функціонування системи.

Перелік графічних документів (за необхідності).

Дата видачі завдання «__02__» ___лютого_____ 2025 р.

Керівник бакалаврської кваліфікаційної роботи _____ **Максим МОКРІЄВ**
(підпис)

Завдання взяв до виконання _____ **Дмитро БАЗЯК**
(підпис)

ЗМІСТ

ВСТУП

Актуальність дослідження. Сучасний етап розвитку глобальної та національної економіки характеризується глибокою цифровою трансформацією всіх бізнес-процесів, серед яких логістика посідає одне з ключових місць. Для мережевого ритейлу ефективне управління матеріальними потоками є не просто допоміжною функцією, а стратегічним фактором виживання на ринку. Стрімке зростання обсягів електронної комерції, зміна споживчих звичок та висока волатильність попиту вимагають від компаній максимальної гнучкості та швидкості прийняття рішень. Традиційні методи планування, що спираються на статичні розклади або суб'єктивний досвід диспетчерів, більше не спроможні забезпечити необхідний рівень ефективності в умовах динамічного міського середовища та нестабільних логістичних ланцюгів.

Протягом 2024-2026 років в Україні спостерігається чітка тенденція до консолідації роздрібних мереж та посилення конкуренції між великими гравцями. У цих умовах кожна відсоткова частка витрат на транспортування та зберігання безпосередньо впливає на кінцеву ціну товару та маржинальність бізнесу. Автоматизація логістики стає відповіддю на виклики сучасності, дозволяючи мінімізувати вплив людського фактора, оптимізувати використання автопарку та складських площ. Особливої ваги набуває розвиток інтелектуальних систем, здатних обробляти великі масиви даних у реальному часі та пропонувати оптимальні сценарії руху товарів.

Використання сучасних алгоритмів маршрутизації, інтеграція з геоінформаційними системами та впровадження хмарних технологій відкривають нові можливості для автоматизації. Проте, незважаючи на наявність великої кількості комерційних рішень, проблема створення адаптивної, легкої та функціональної системи, що враховує специфіку конкретної мережі магазинів, залишається відкритою. Це зумовлює вибір теми кваліфікаційної роботи та підтверджує її високу актуальність як з наукової, так і з практичної точок зору.

Проблематика дослідження полягає у наявності суттєвих розривів між теоретичними моделями оптимізації та реальними процесами в логістичних підрозділах мережевих компаній. Основними проблемами є: висока вартість володіння закордонними ERP-системами, складність їх адаптації до локальних умов, низька точність ручного планування маршрутів доставки та відсутність єдиного інформаційного поля для обміну даними між складом, транспортом та магазинами. Необхідність подолання цих бар'єрів через розробку спеціалізованого програмного забезпечення є центральним питанням даної роботи.

Мета дослідження полягає у проектуванні та програмній реалізації автоматизованої системи управління логістикою для мережі магазинів, яка дозволить підвищити ефективність складських операцій та мінімізувати транспортні витрати шляхом впровадження сучасних алгоритмів маршрутизації та централізованого обліку.

Завдання дослідження для досягнення поставленої мети включають:

1. Провести системний аналіз об’єкта автоматизації, формалізувати бізнес-процеси логістики мережі магазинів та сформувати технічні вимоги до програмного продукту.
2. Здійснити порівняльний аналіз існуючих аналогів та обґрунтувати вибір стеку технологій для розробки клієнт-серверного додатка.
3. Спроектувати архітектуру системи, логічну та фізичну моделі бази даних, забезпечивши цілісність та безпеку збереження інформації.
4. Розробити та імплементувати алгоритм оптимізації маршрутів доставки товарів, інтегрувавши його з картографічними сервісами.
5. Провести тестування розробленого програмного забезпечення, оцінити його функціональність та сформувати рекомендації щодо впровадження в експлуатацію.

Об’єкт дослідження - процеси управління матеріальними та інформаційними потоками в логістичній інфраструктурі мережі роздрібних магазинів.

Предмет дослідження - методи, моделі та програмні інструменти автоматизації логістичних операцій, зокрема алгоритми планування маршрутів та архітектурні рішення для систем управління складськими запасами.

Методи дослідження. Для вирішення поставлених завдань у роботі використано комплекс наукових методів:

1. Метод системного аналізу - застосовується для комплексного вивчення структури логістичної системи, виявлення взаємозв'язків між підрозділами та визначення вузьких місць у поточних бізнес-процесах.
2. Методи об’єктно-орієнтованого аналізу та проектування (OOAD) - використовуються для розробки архітектури програмного забезпечення, створення діаграм класів, послідовностей та станів, що забезпечує масштабованість та гнучкість системи.
3. Методи теорії графів - покладені в основу розробки алгоритмів пошуку найкоротших шляхів та вирішення транспортної задачі, що дозволяє математично обґрунтувати вибір маршрутів доставки.
4. Методи моделювання баз даних (ER-моделювання) - застосовуються для проектування структури збереження даних, нормалізації таблиць та оптимізації запитів до інформаційної бази.
5. Метод порівняльного аналізу - використовується на етапі вибору інструментальних засобів розробки та при оцінці існуючих програмних рішень на ринку логістичного софту.
6. Методи тестування програмного забезпечення - застосовуються для валідації коректності роботи алгоритмів, перевірки стійкості системи до помилок введення та оцінки зручності інтерфейсу користувача.

Джерельна база дослідження включає наукові праці вітчизняних та зарубіжних вчених у галузі інформаційних технологій та логістичного менеджменту, технічну документацію до сучасних мов програмування та систем управління базами даних, а також державні стандарти України щодо оформлення програмних документів та бібліографічних посилань. Важливу

частину бази складають актуальні статистичні дані ринку ритейлу та звіти логістичних операторів.

Практичне значення одержаних результатів полягає у створенні працездатного програмного комплексу, який може бути використаний реальними торговельними мережами для автоматизації диспетчерських служб. Впровадження системи дозволить скоротити час на формування рейсових листів, знизити пробіг автотранспорту за рахунок оптимізації шляхів та забезпечити оперативний контроль залишків товарів у кожній точці продажу. Розроблена архітектура дозволяє легко розширювати функціонал системи відповідно до зростання потреб бізнесу.

Структура роботи. Бакалаврська кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі розглядаються теоретичні основи та проводиться аналіз предметної області. Другий розділ присвячений інформаційному забезпеченню та проектуванню бази даних. У третьому розділі описується програмна реалізація системи та алгоритмічна база. Четвертий розділ містить результати тестування та інструкції для користувачів. Загальний обсяг роботи відповідає вимогам методичних вказівок.

РОЗДІЛ 1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

Розробка будь-якої інформаційної системи починається з детального вивчення середовища, в якому вона буде функціонувати. Для мене, як розробника, критично важливо зрозуміти не лише програмні вимоги, а й реальні фізичні процеси: як товар потрапляє на склад, як формуються заявки від магазинів та як диспетчери приймають рішення щодо маршрутизації. Тільки глибоке занурення у специфіку логістичних бізнес-процесів дозволить перетворити хаос щоденних поставок на чіткий, оптимізований машинний алгоритм.

Сучасні торговельні підприємства оперують складними багаторівневими ланцюгами постачань, де центральний склад виступає головним розподільчим вузлом для десятків роздрібних точок. Ефективність функціонування такої мережі прямо залежить від швидкості обміну даними між підрозділами та здатності інформаційної системи адекватно реагувати на коливання споживчого попиту [6].

Ключовим фактором, що ускладнює процеси управління, є просторово-територіальна розподіленість об'єктів інфраструктури, яка вимагає точного врахування географічних координат та відстаней при кожному плануванні транспортного рейсу [36].

Традиційні підходи до управління, засновані на ручній обробці табличних даних, призводять до виникнення так званого "людського фактора", що супроводжується помилками у комплектації замовлень та нераціональним використанням вантажного простору автомобілів [20].

Впровадження автоматизованих інформаційних систем дозволяє подолати ці інформаційні розриви, створюючи єдине корпоративне середовище, де кожен етап руху товару фіксується у базі даних у режимі реального часу [33, с. 156].

Застосування математичних методів маршрутизації на етапі розподілу заявок дає змогу суттєво скоротити витрати на паливно-мастильні матеріали та

гарантувати дотримання жорстких часових вікон доставки для кожного магазину мережі [23].

Відповідно до принципів системного аналізу, перед безпосереднім написанням програмного коду необхідно здійснити декомпозицію логістичної системи, виділивши ключові інформаційні сутності: склади, роздрібні точки, транспортні засоби та накладні [8].

Такий підхід дозволить сформувати чітке технічне завдання на розробку реляційної бази даних та визначити вимоги до апаратної інфраструктури, здатної витримувати пікові навантаження під час масового автоматичного формування замовлень [4].

Загалом, цифровізація логістичних процесів є невід'ємною складовою переходу торговельного підприємства до сучасних стандартів управління, що забезпечує прозорість операцій та високу технічну масштабованість бізнесу [40].

Розглядаючи детальніше внутрішньоскладські операції, варто зазначити, що процес обробки заявок від роздрібних точок складається з кількох послідовних етапів: від отримання електронного запиту до фізичного комплектування палет і завантаження у транспорт. Кожен із цих етапів генерує значний обсяг даних, який потребує миттєвої синхронізації з центральною системою для уникнення критичних розбіжностей між фактичними товарними залишками та їх відображенням у віртуальному обліку [4].

Особливу алгоритмічну складність у цьому ланцюгу становить класична задача маршрутизації транспортних засобів, що належить до класу NP-складних математичних проблем. Зі збільшенням кількості роздрібних точок, які потребують обслуговування, кількість можливих комбінацій маршрутів зростає експоненціально, що робить ручний або наївний розрахунок оптимального шляху диспетчером практично неможливим [3].

Для ефективного вирішення цієї задачі програмна система повинна застосовувати спеціалізовані евристичні або метаевристичні алгоритми, здатні за прийнятний машинний час знаходити рішення, які максимально наближені

до глобального оптимуму. Імплементація таких алгоритмічних рішень дозволяє не лише мінімізувати загальний кілометраж автопарку, але й враховувати жорсткі обмеження, такі як максимальна вантажопідйомність автомобілів, габарити товару та задані часові вікна доставки [22].

Крім оптимізаційної складової, стабільність логістики прямо залежить від здатності інформаційної системи витримувати високі навантаження під час масового надходження заявок від усієї мережі магазинів. Архітектура програмного рішення має передбачати асинхронну обробку вхідних запитів та гарантувати строгу консистентність даних навіть в умовах нестабільного мережевого з'єднання чи апаратних збоїв на проміжних вузлах [21].

Інтеграція математичних модулів розрахунку маршрутів із системою управління базами даних вимагає чіткого структурного розділення бізнес-логіки та рівня доступу до даних. Такий архітектурний підхід значно полегшує подальше масштабування системи, дозволяючи розробникам незалежно оновлювати алгоритми просторової оптимізації без ризику пошкодження механізмів зберігання критичної облікової інформації [28].

Відповідно, постановка завдання на розробку зводиться до створення комплексної програмної платформи, яка безшовно об'єднає модуль управління складськими запасами з високопродуктивним геоінформаційним модулем маршрутизації. Це вимагає ретельного проектування реляційної структури бази даних, здатної не тільки швидко обробляти стандартні CRUD-операції, але й ефективно виконувати складні просторові запити для підтримки транзакційної цілісності всіх логістичних операцій підприємства [34].

Для забезпечення максимальної гнучкості системи доцільно спроектувати клієнт-серверний додаток, де ресурсомісткі обчислення графових алгоритмів виконуватимуться на потужному серверному боці, а клієнтська частина відповідатиме виключно за візуалізацію топології мережі та інтерактивну взаємодію користувача з картографічними сервісами [37].

Зрозумівши внутрішню природу логістичних процесів та сформувавши загальне бачення проблеми, логічним кроком є аналіз програмних продуктів,

які вже представлені на ринку. На мою думку, розробка нової інформаційної системи не повинна відбуватися у вакуумі, адже ігнорування існуючих рішень призводить до повторення чужих архітектурних помилок. Детальний розбір доступних на ринку систем управління транспортом та складом дозволяє виявити їхні сильні сторони, а головне - зрозуміти, чому типові продукти не задовольняють специфічні потреби нашої мережі магазинів.

Ринок корпоративного програмного забезпечення пропонує широкий спектр універсальних систем класу ERP, серед яких лідируючі позиції займають масштабовані продукти від компаній рівня SAP та Oracle. Такі системи включають потужні інтегровані модулі логістичного контролю, проте їхня архітектура є надзвичайно монолітною, що вимагає величезних фінансових та часових ресурсів на етапі розгортання і подальшої кастомізації під конкретні бізнес-процеси локального підприємства [30].

Як альтернатива важким корпоративним ERP-платформам, компанії часто використовують спеціалізовані хмарні сервіси класу TMS (Transport Management Systems), орієнтовані виключно на маршрутизацію. Ці веб-додатки добре справляються з базовим диспетчеризуванням та GPS-моніторингом рухомого складу, але їхньою типовою слабкістю є відсутність глибокої інтеграції з існуючими локальними системами управління складом (WMS), що змушує логістів вручну дублювати інформацію про товарні накладні [1].

Аналізуючи алгоритмічну базу популярних «коробкових» логістичних рішень, можна помітити, що вбудовані інструменти розрахунку маршрутів часто спираються на спрощені математичні моделі. Вони розраховують лише базовий найкоротший шлях на графі доріг, ігноруючи при цьому складні динамічні обмеження, такі як фактична вантажопідйомність конкретного транспортного засобу або специфічні часові вікна прийому товару в кожній окремій роздрібній точці [3].

Для середніх торговельних мереж використання розрізненого програмного забезпечення або спроба адаптувати громіздкі корпоративні платформи створює більше технічних перешкод, ніж реальної оптимізації транспортних

потоків. У такому контексті виникає об'єктивна потреба у створенні збалансованих, вузькоспеціалізованих систем, які базуються на сучасному модульному підході, що дозволяє гнучко налаштовувати логіку обробки даних без накопичення надлишкового і складного для підтримки коду [35].

Враховуючи виявлені недоліки комерційного програмного забезпечення, виникає гостра необхідність чітко формалізувати технічні вимоги до власної розробки. Проектована система повинна мати відкриту сервіс-орієнтовану архітектуру з підтримкою сучасних протоколів обміну даними, що дозволить безшовно інтегруватися з наявними обліковими базами підприємства без необхідності постійного ручного імпорту чи експорту файлів формату таблиць [39].

З огляду на сформовані технічні виклики, ключовим етапом проектування стає деталізація функціональних вимог до створюваного програмного продукту. Розроблювана система повинна гарантувати автоматичне виконання низки критичних логістичних операцій, серед яких першочерговими є: алгоритмічний розрахунок оптимальних шляхів доставки, динамічне формування товарно-транспортних накладних та ведення централізованого обліку залишків у режимі реального часу.

Окрім базового функціоналу, високі вимоги висуваються до нефункціональних характеристик та загальної надійності архітектури. Система повинна бути стійкою до відмов, здатною обробляти паралельні запити від десятків користувачів (менеджерів магазинів, комірників, диспетчерів) без втрати продуктивності чи блокування транзакцій бази даних під час пікових навантажень [37].

Критичною вимогою є глибока програмна інтеграція з картографічними сервісами та геоінформаційними системами. Замість використання спрощених матриць відстаней, логістичний модуль повинен працювати з реальними графами дорожньої мережі, враховуючи просторові обмеження, напрямки руху та специфіку під'їзних шляхів до кож ної окремої роздрібної точки [19].

Для забезпечення операційної стабільності на рівні збереження інформації, підсистема бази даних має підтримувати строгу посилальну цілісність. Це означає, що при призначенні транспортного засобу на певний маршрут та формуванні рейсу, відповідна партія товару повинна логічно резервуватися на складі, унеможливлюючи її повторне списання чи виникнення пересортиці під час паралельної роботи кількох диспетчерів [10].

З точки зору взаємодії з кінцевим користувачем, інтерфейс веб-додатка повинен базуватися на принципах ергономічності та мінімізації когнітивного навантаження. Візуалізація складних просторових даних на інтерактивній карті, замість перевантажених статичних таблиць, дозволить диспетчеру миттєво оцінювати адекватність побудованих алгоритмом маршрутів та за необхідності вносити оперативні ручні коригування.

Проведений огляд ринкових альтернатив та чітке формулювання технічних вимог створюють надійний фундамент для переходу до наступної стадії проектування. Розуміння функціональних меж системи та її ролі у загальному логістичному ланцюгу підприємства дозволяє відійти від абстрактного опису проблем і розпочати побудову строгих об'єктно-орієнтованих моделей, які стануть безпосереднім планом для подальшого написання програмного коду та структурування інформаційної бази.

Перехід від текстового опису проблем та вимог до безпосереднього написання програмного коду без проміжного етапу проектування часто призводить до фатальних архітектурних помилок. На мою думку, саме візуальне моделювання дозволяє абстрагуватися від дрібних деталей реалізації та побачити систему в цілому. Створення чітких графічних моделей є тим містком, який з'єднує бізнес-логіку логістичних процесів із суворими правилами реляційних баз даних та алгоритмів.

У сучасній інженерії програмного забезпечення індустріальним стандартом для візуалізації архітектури та поведінки складних систем виступає Уніфікована мова моделювання. Використання стандартизованих нотацій дозволяє не лише узгодити бачення проекту між усіма учасниками розробки,

але й сформувати чітку об'єктно-орієнтовану структуру майбутнього веб-додатка, розбивши загальне завдання маршрутизації на окремі ізольовані класи та методи [26, с. 115].

Для відображення функціональних можливостей системи з точки зору кінцевих користувачів розробляється діаграма прецедентів. У нашій системі ключовими акторами виступають "Менеджер магазину", який формує електронну заявку на постачання, "Диспетчер-логіст", який ініціює алгоритм оптимізації маршрутів, та "Комірник", що відповідає за фізичне відвантаження товару. Ця діаграма чітко окреслює межі відповідальності кожного користувача та описує загальний функціонал веб-додатка.

Для деталізації внутрішньої структури розробляється діаграма класів (Class Diagram), яка моделює статичну архітектуру логістичного модуля. Вона відображає основні сутності предметної області, такі як "Заявка", "Маршрут", "Транспортний засіб", "Товарна позиція" та "Склад", а також логічні зв'язки між ними, що згодом стане фундаментальною основою для проектування таблиць бази даних та написання ORM-моделей у програмному коді [5].

Крім статичної структури, критично важливо змоделювати динаміку процесів, особливо роботу самого алгоритму маршрутизації та комунікацію модулів. Для цього застосовуються діаграми діяльності або діаграми послідовностей, які крок за кроком показують обмін повідомленнями між клієнтським інтерфейсом, сервером, базою даних та зовнішнім картографічним API під час генерації оптимального шляху розвезення продукції.

Перехід від теоретичного опису бізнес-процесів до безпосереднього технічного проектування вимагав від мене створення цілісної архітектурної карти майбутнього веб-додатка. На цьому етапі я зосередився на формалізації функціональних та структурних аспектів системи, що дозволило перетворити абстрактні логістичні вимоги на чіткі інженерні схеми. Вважаю, що саме детальне візуальне моделювання є критично важливим для запобігання логічним помилкам у структурі даних та взаємодії модулів, оскільки воно дозволяє побачити складні зв’язки між об’єктами ще до початку написання

програмного коду. Мною було розроблено комплекс UML-діаграм, які стали фундаментом для подальшої розробки бази даних та алгоритмів оптимізації.

Для специфікації функціональних можливостей системи я обрав побудову діаграми прецедентів, яка дозволяє чітко розмежувати ролі користувачів та їхні права доступу до функцій логістичного модуля. Основними дійовими особами в розроблюваній системі виступають Менеджер магазину, Диспетчер-логіст та Комірник, кожен з яких має унікальний набір сценаріїв взаємодії з платформою. Використання такої візуалізації на початковому етапі проектування допомогло мені структурувати вимоги до інтерфейсу та визначити межі відповідальності кожного учасника логістичного ланцюга [32, с. 45].

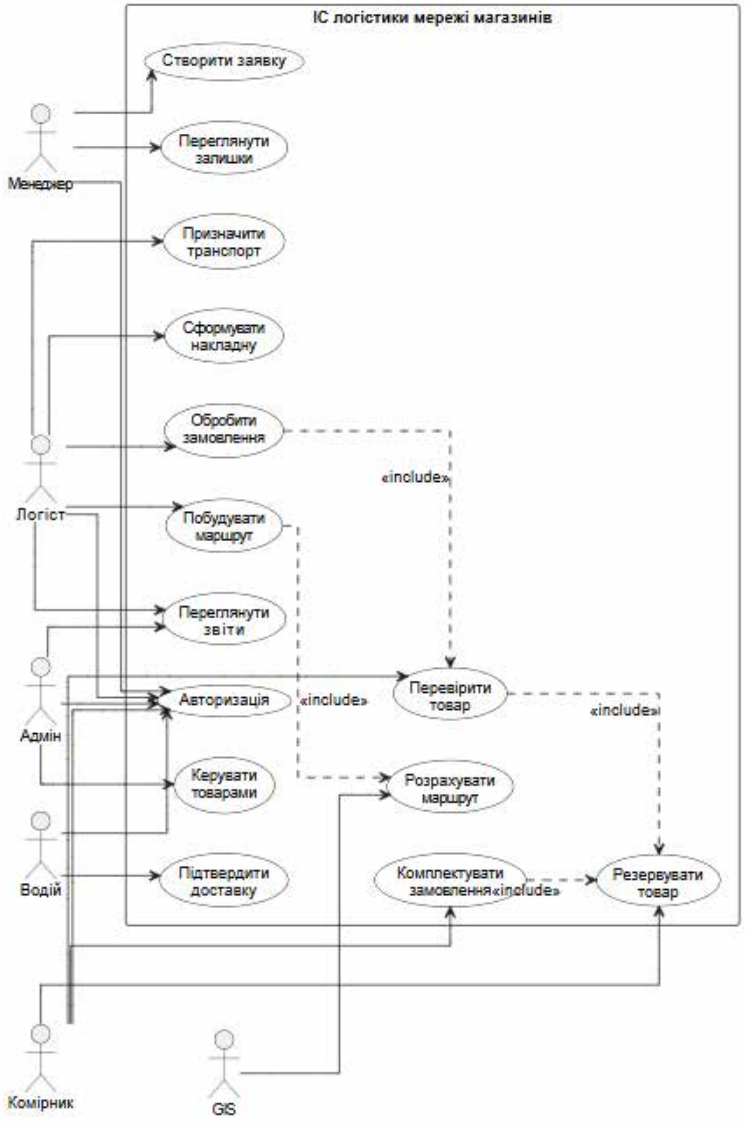

Рис. 1.1 Діаграма прецедентів

Кожен із визначених прецедентів потребував детального опису для розуміння вхідних даних та очікуваних результатів роботи системи. Зокрема, прецедент «Розрахунок маршруту» є найбільш складним, оскільки він ініціює виклик зовнішніх картографічних API та внутрішніх алгоритмів оптимізації. Завдяки формалізації цих сценаріїв я зміг виділити спільні модулі авторизації та обробки помилок, що значно спрощує подальшу програмну реалізацію системи [5].

Оновлена специфікація основних функціональних сценаріїв системи представлена у таблиці 1.1, де відображено розширений перелік операцій та їхніх виконавців.

Таблиця 1.1.

Специфікація функціональних прецедентів системи

Назва прецеденту	**Головна роль**	**Короткий зміст операції**
Авторизація користувача	Всі	Перевірка прав доступу до модулів системи.
Створення заявки	Менеджер магазину	Формування списку товарів для поповнення залишків.
Перегляд залишків	Комірник / Менеджер	Отримання актуальної інформації про наявність товарів.
Генерація маршрутів	Диспетчер-логіст	Розрахунок оптимальних шляхів доставки через GIS-модуль.
Призначення транспорту	Диспетчер-логіст	Закріплення конкретного автомобіля за розрахованим рейсом.
Формування накладної	Диспетчер-логіст	Генерація друкованої форми ТТН на основі маршруту.
Комплектування рейсу	Комірник	Зміна статусу товарів на складі для підготовки до вивозу.

Відмітка про виконання	Водій / Диспетчер	Фіксація завершення доставки товару в роздрібну точку.

Для глибшого розуміння внутрішньої архітектури додатка мною було спроектовано діаграму класів, яка відображає статичну структуру системи.

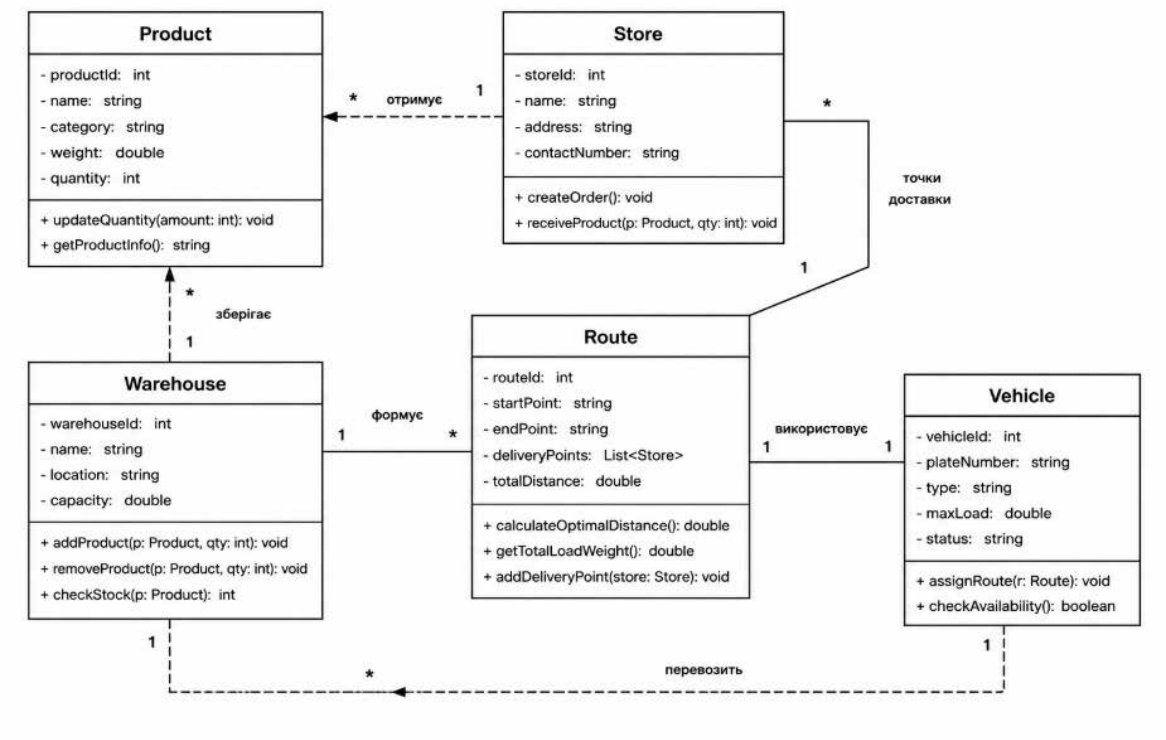

Рис. 1.2 Діаграма класів

На цій діаграмі я виділив ключові сутності предметної області, такі як Товар (Product), Магазин (Store), Склад (Warehouse), Маршрут (Route) та Транспортний засіб (Vehicle). Опис властивостей та методів цих класів став основою для створення об’єктно-реляційної моделі, що дозволяє системі ефективно маніпулювати складними структурами даних у базі [26, с. 104].

Особливу увагу я приділив класу Route, який агрегує в собі множину точок доставки та пов'язаний з конкретним об'єктом класу Vehicle. Методи цього класу, такі як calculateOptimalDistance() та getTotalLoadWeight(), відповідають за внутрішню бізнес-логіку системи, що дозволяє ізолювати математичні обчислення від коду інтерфейсу користувача. Це забезпечує високу

підтримуваність програмного забезпечення та можливість незалежної заміни алгоритмів оптимізації без необхідності переписування всього додатку [15].

Для моделювання динамічної взаємодії об’єктів у часі я розробив діаграму послідовностей для сценарію автоматичної генерації маршрутів. Цей процес починається з ініціації запиту логістом, після чого контролер звертається до сервісу оптимізації, який, у свою чергу, отримує матрицю відстаней від картографічного сервера та дані про доступний транспорт із бази даних.

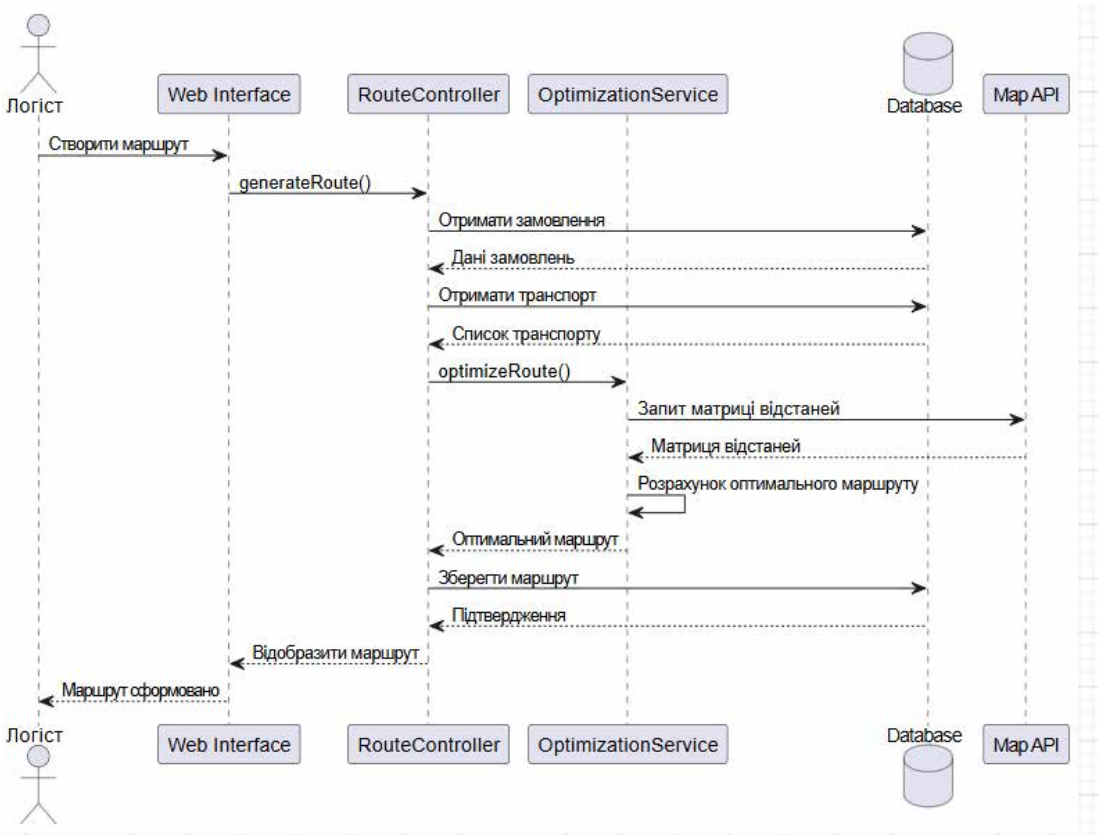

Рис. 1.3 UML-діаграма послідовності автоматичної генерації маршруту доставки

Така деталізація дозволила мені спроектувати асинхронні запити до зовнішніх сервісів, що запобігає блокуванню інтерфейсу під час складних обчислень [35].

Розуміння черговості викликів методів при обробці заявки допомогло мені реалізувати надійний механізм обробки транзакцій. Кожен етап послідовності - від отримання геокоординат до фіксації маршруту в БД - має власні контрольні точки, що гарантує цілісність інформації навіть у разі мережевих збоїв під час

роботи з API Google Maps або Leaflet. Орієнтація на таку динамічну модель дозволяє створювати системи з низькою зв’язністю компонентів [13].

Для візуалізації логіки роботи самого алгоритму та переходів між етапами обробки замовлення мною була створена діаграма діяльності. Вона охоплює весь шлях даних: від моменту перевірки наявності товару на складі до прийняття рішення про включення магазину в поточний графік розвезення. Використання розгалужень та циклів на цій діаграмі допомогло мені формалізувати правила вибору транспортного засобу залежно від об’єму вантажу та пріоритетності точки доставки [32].

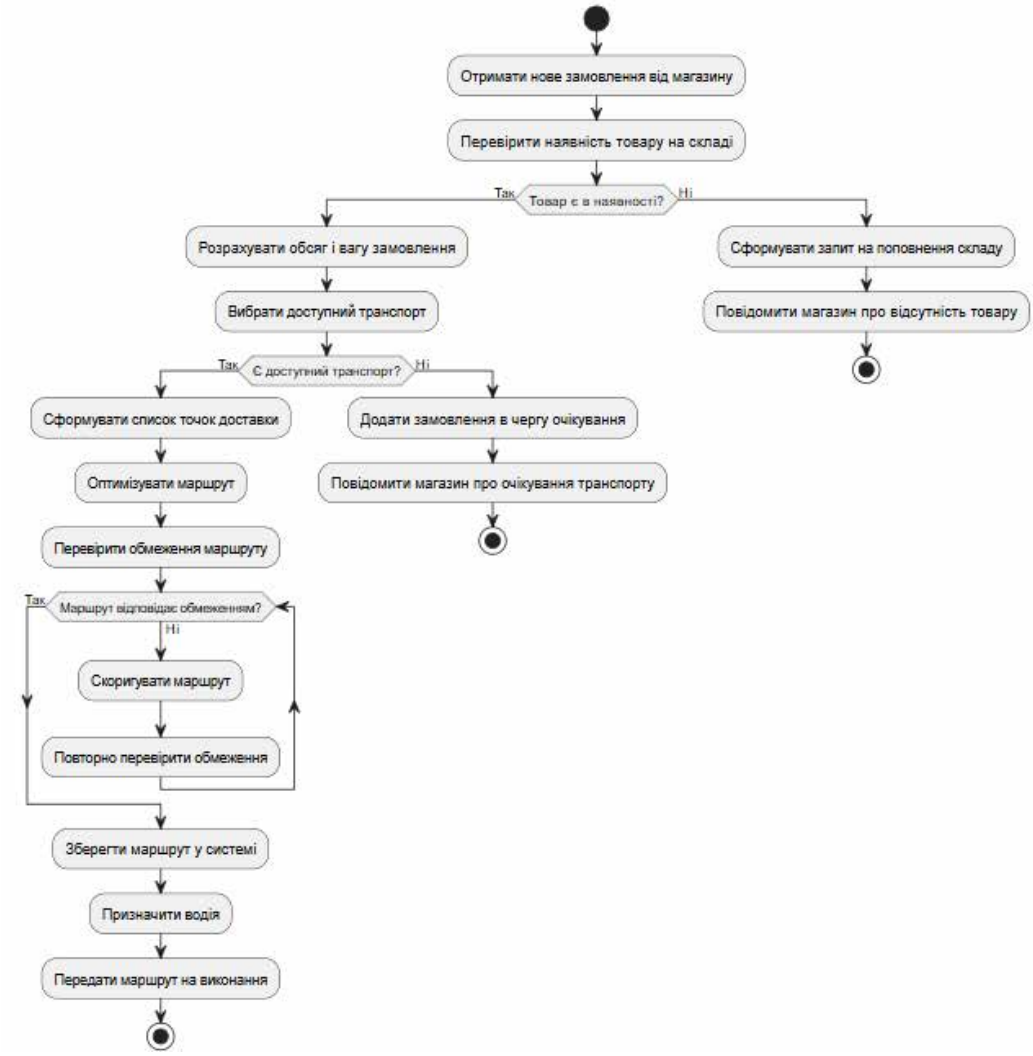

Рис. 1.4 UML-діаграма діяльності процесу формування маршруту доставки.

Побудова графа активностей дозволила мені виявити потенційні «вузькі місця» в бізнес-логіці, зокрема затримки при валідації адрес магазинів. Включення блоків паралельної обробки заявок на діаграмі діяльності стало

передумовою для використання багатопоточності у програмному коді, що суттєво підвищує продуктивність системи при роботі з великими роздрібними мережами [34].

Життєвий цикл ключової сутності - Замовлення (Order) - я відобразив за допомогою діаграми станів. Це критично важливо для логістичної системи, оскільки статус замовлення (Нове, Підтверджене, Комплектується, В дорозі, Доставлено) визначає доступні дії для різних категорій персоналу. Наприклад, менеджер магазину не може редагувати склад замовлення, якщо воно вже перейшло у стан «В дорозі», що запобігає розбіжностям у звітності [23].

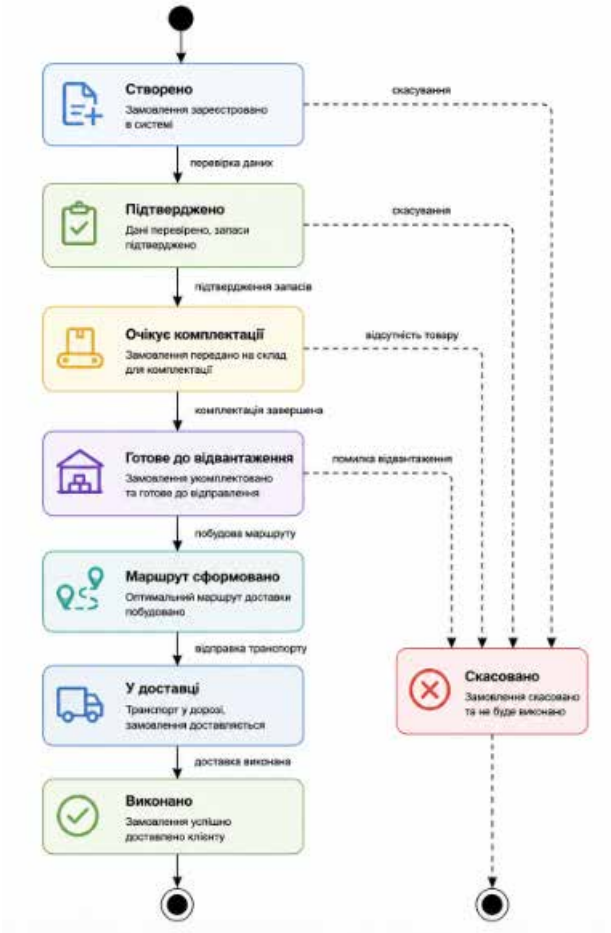

Рис. 1.5 Діаграма станів

Фізичне розміщення компонентів системи я змоделював на діаграмі розгортання (Deployment Diagram), де вказав вузли сервера додатків, сервера бази даних та клієнтські термінали. Оскільки система проектується як веб-орієнтована, я передбачив використання протоколу HTTPS для безпечного

зв'язку між клієнтом та сервером, а також окремий вузол для зберігання просторових індексів бази даних, що прискорює пошук точок на карті [25].

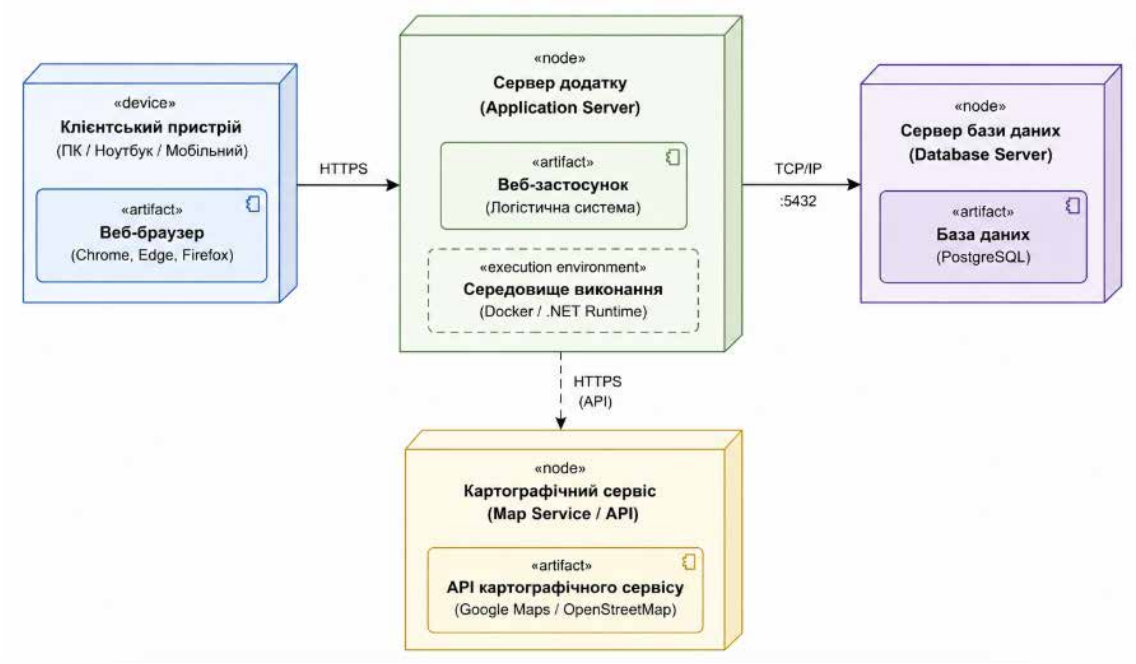

Рис.1.6 – UML-діаграма розгортання автоматизованої системи управління логістикою мережі магазинів

Завершуючи перший етап розробки, я можу констатувати, що проведений системний аналіз дозволив мені не лише поверхнево ознайомитися з предметною областю, а й виявити глибокі структурні проблеми, які гальмують розвиток сучасних торговельних мереж. Для мене стало очевидним, що логістика - це не просто процес перевезення вантажів, а складна інформаційна екосистема, де найменша затримка в передачі даних або помилка в розрахунку маршруту призводить до відчутних фінансових втрат. Вважаю, що саме цей аналітичний фундамент, побудований на поєднанні мого бачення як розробника та реальних потреб бізнесу, є ключовою передумовою для створення справді ефективного програмного продукту. Формалізація хаотичних на перший погляд операцій у вигляді чітких математичних моделей та алгоритмічних вимог дала мені змогу перейти від абстрактних ідей до проектування конкретної архітектури системи.

Узагальнення результатів дослідження бізнес-процесів свідчить про те, що сучасний стан логістичного менеджменту в умовах цифрової трансформації вимагає впровадження високоадаптивних систем управління. Традиційні лінійні моделі постачання стають неефективними через стрімке зростання кількості вузлів у мережах роздрібної торгівлі та підвищення вимог до швидкості обробки замовлень. Це зумовлює необхідність переходу до сервіс-орієнтованих архітектур, які дозволяють інтегрувати складські операції з транспортними потоками в єдиному інформаційному просторі, забезпечуючи прозорість усіх етапів руху товарних цінностей [4, с. 112].

Аналіз наукових джерел та практичного досвіду впровадження корпоративних інформаційних систем показує, що більшість існуючих на ринку рішень класу ERP або TMS мають низку критичних обмежень для середнього бізнесу. Висока вартість ліцензування, складність інтеграції з наявними базами даних та надмірна громіздкість інтерфейсів часто стають бар’єрами для реальної оптимізації процесів. Створення спеціалізованого веб-додатка на базі сучасних фреймворків дозволяє уникнути цих недоліків, забезпечуючи необхідну легкість системи та її повну відповідність специфічним вимогам конкретного логістичного ланцюга [30, с. 184].

Впровадження інтелектуальних алгоритмів, що працюють із графами в реальному часі, є єдиним шляхом до суттєвого скорочення операційних витрат на паливно-мастильні матеріали та амортизацію техніки [3, с. 210].

Розробка комплексу UML-діаграм дозволила мені детально специфікувати взаємодію між різними категоріями користувачів системи та її внутрішніми компонентами. Побудова діаграм прецедентів та діяльності на етапі аналізу є необхідною умовою для створення масштабованого програмного коду, оскільки вона дозволяє виявити потенційні конфлікти в бізнес-логіці ще до початку фізичного проектування бази даних. Такий об’єктно-орієнтований підхід гарантує, що майбутня система буде не лише функціональною, але й стійкою до змін у вимогах замовника [32].

Дослідження інформаційних потреб логістичної мережі підтвердило необхідність використання реляційних СУБД із підтримкою просторових розширень для коректної обробки геоданих. Зберігання координат магазинів та складів безпосередньо у структурованому вигляді дозволяє виконувати складні аналітичні запити на рівні сервера бази даних, що значно підвищує загальну продуктивність системи. Це особливо актуально при роботі з великими масивами інформації, коли швидкість доступу до даних та їх цілісність стають вирішальними факторами стабільності бізнесу [11].

Вивчення сучасних тенденцій у розробці програмного забезпечення вказує на доцільність використання мов програмування з розвиненою екосистемою бібліотек для обробки даних та штучного інтелекту. Вибір інструментарію, що підтримує асинхронну обробку запитів та легку інтеграцію з зовнішніми картографічними сервісами, дозволяє створити систему з високим рівнем чуйності інтерфейсу. Це критично важливо для диспетчерських служб, де швидкість прийняття рішень під час пікових навантажень безпосередньо залежить від ергономічності та швидкодії програмного забезпечення [14].

Застосування методів системного аналізу дало змогу виділити ключові критерії ефективності майбутньої розробки, серед яких пріоритетними є мінімізація часу простою транспорту та оптимізація заповнення складських площ. Формалізація цих критеріїв у вигляді математичних функцій дозволяє автоматизувати процес прийняття рішень, зводячи роль людського фактора до контролю та фінального затвердження сформованих системою планів. Така автоматизація є основою для побудови інтелектуальних систем управління логістикою нового покоління [8].

Розгляд теоретичних аспектів побудови складських систем показав, що інтеграція з модулем маршрутизації має відбуватися на рівні єдиної транзакційної моделі. Це означає, що будь-яка зміна в графіку доставки повинна миттєво відображатися на стані товарних запасів та статусі готовності замовлень до відвантаження. Тільки такий підхід забезпечує повну синхронізацію фізичних та інформаційних потоків, унеможливлюючи

виникнення помилок обліку та затримок у постачанні продукції до роздрібних точок [6].

Систематизація вимог до апаратного та програмного забезпечення дозволила визначити оптимальну топологію розгортання системи, що базується на використанні хмарних технологій або централізованих серверних потужностей. Це забезпечує доступність системи з будь-якої точки мережі через стандартні веб-браузери, знижуючи витрати на підтримку робочих місць користувачів. Орієнтація на відкриті стандарти та кросплатформність робить розробку універсальною та готовою до впровадження в інфраструктуру будь-якого сучасного торговельного підприємства [33].

Підсумовуючи результати першого розділу, можна стверджувати, що мета системного аналізу була повністю досягнута: я сформував вичерпне технічне бачення проекту, обґрунтував необхідність створення власної програмної платформи та розробив детальні архітектурні моделі. Всі виявлені проблеми ручного управління логістикою знайшли своє відображення у функціональних вимогах до системи, а обрані методи моделювання стали надійним містком до етапу проектування інформаційної бази. Це дає мені впевненість у тому, що подальша розробка буде базуватися на строгих наукових та інженерних засадах, а не на інтуїтивних припущеннях. Таким чином, логічним продовженням роботи є перехід до детального проектування структури даних та вибору конкретних інструментів реалізації, що дозволить перетворити теоретичні моделі на працездатне програмне забезпечення. Сформований доробок є достатнім для забезпечення високої якості проекту та його відповідності вимогам сучасної IT-індустрії.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних

Проектування інформаційного забезпечення є одним із найважливіших та найбільш відповідальних етапів життєвого циклу розробки програмного забезпечення. Архітектура бази даних визначає не лише ефективність збереження інформації, але й продуктивність усієї логістичної системи, швидкість виконання алгоритмів оптимізації маршрутів та можливості подальшого масштабування платформи. Будь-які помилки, допущені на етапі створення структури таблиць та зв'язків між ними, неминуче призводять до критичного падіння швидкодії додатка при збільшенні обсягів даних, виникнення аномалій оновлення та втрати транзакційної цілісності. З огляду на це, формування строгої, нормалізованої логічної моделі даних є першочерговим завданням після завершення об'єктно-орієнтованого аналізу предметної області.

Логічна модель даних слугує абстрактним відображенням структури інформаційної бази, незалежним від конкретної фізичної реалізації в тій чи іншій системі управління базами даних. Основною метою цього етапу є трансформація концептуальних сутностей, виділених на UML-діаграмі класів (Розділ 1.3), у строгу реляційну схему. Враховуючи специфіку автоматизованої системи управління логістикою, яка поєднує класичні облікові операції (рух товарів) та геоінформаційні обчислення (розрахунок відстаней та маршрутів), обрана реляційна модель повинна гарантувати підтримку вимог ACID (Atomicity, Consistency, Isolation, Durability) для фінансово-складських транзакцій.

Процес побудови логічної моделі починається з ідентифікації базових відношень (таблиць). На основі бізнес-правил логістичної мережі було виділено наступний набір основних сутностей: «Користувачі» (Users), «Ролі» (Roles), «Склади» (Warehouses), «Магазини» (Stores), «Товари» (Products), «Складські залишки» (Inventory), «Замовлення» (Orders), «Позиції замовлення»

(OrderItems), «Транспортні засоби» (Vehicles), «Маршрути» (Routes) та «Точки маршруту» (RoutePoints).

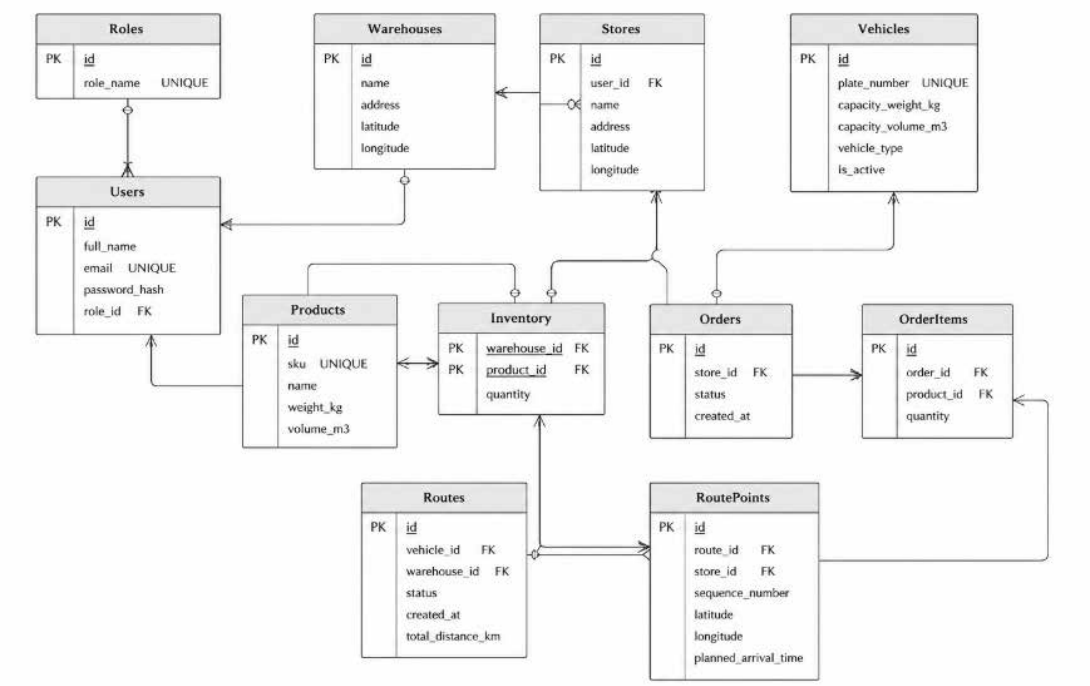

Рис. 2.1 – ER-діаграма логічної моделі бази даних логістичної системи

Детальний аналіз кожної сутності вимагає чіткого визначення її атрибутивного складу. Першою ключовою складовою системи є підсистема управління доступом. Сутність «Ролі» (Roles) є класичним довідником, який містить унікальний ідентифікатор (первинний ключ id) та назву ролі (role_name), що визначає рівень привілеїв (наприклад, Administrator, Dispatcher, Manager, Warehouseman). Сутність «Користувачі» (Users) містить персональні дані співробітників: унікальний ідентифікатор (id), повне ім'я (full_name), електронну пошту для авторизації (email), яка є унікальним ключем (UNIQUE), хеш пароля (password_hash) для забезпечення криптографічної безпеки та зовнішній ключ (role_id), що вказує на відповідний запис у таблиці ролей. Зв'язок між цими таблицями реалізується за типом «один до багатьох» (1:N), що є стандартом для систем управління доступом на основі ролей (RBAC).

Другою, не менш важливою підсистемою, є інфраструктурна топологія мережі. Вона представлена сутностями «Склади» (Warehouses) та «Магазини» (Stores). З логічної точки зору, ці таблиці мають схожий атрибутивний склад:

ідентифікатор (id), назву об'єкта (name), текстове представлення фізичної адреси (address). Однак, критичною відмінністю нашої системи є необхідність інтеграції з геоінформаційними сервісами для оптимізації транспортних потоків. Тому обидві сутності повинні містити просторові атрибути: latitude (широта) та longitude (довгота). На етапі логічного моделювання ці координати розглядаються не просто як числові типи даних з рухомою комою, а як єдиний просторовий об'єкт (наприклад, типу Point у стандарті OGC), який дозволить системі виконувати геопросторові запити щодо знаходження найкоротших відстаней між вузлами мережі. Сутність «Магазини» також містить зовнішній ключ user_id, що зв'язує конкретну роздрібну точку з її відповідальним менеджером.

Третій блок логічної моделі відповідає за товарно-матеріальний облік. Сутність «Товари» (Products) виступає базовим номенклатурним довідником підприємства. Її обов'язковими атрибутами є ідентифікатор (id), унікальний артикул (sku), найменування (name). Враховуючи вимоги задачі маршрутизації (Vehicle Routing Problem), критично необхідним є включення до цієї таблиці масо-габаритних характеристик: ваги одиниці товару (weight_kg) та її об'єму (volume_m3). Ці атрибути безпосередньо впливають на роботу математичного алгоритму, оскільки виступають жорсткими обмеженнями (capacity constraints) при завантаженні транспорту. Зв'язок між товарами та складами реалізується через асоціативну сутність «Складські залишки» (Inventory), оскільки один склад може зберігати багато товарів, а один товар може бути розподілений по різних складах. Таблиця Inventory містить складений зовнішній ключ (warehouse_id, product_id) та атрибут поточної кількості (quantity), що дозволяє вести точний облік у режимі реального часу.

Четверта підсистема - це транзакційне ядро формування заявок. Оскільки одне замовлення від магазину може містити десятки різних товарних позицій, логічна модель повинна дотримуватися класичного патерну «заголовок–деталі». Сутність «Замовлення» (Orders) зберігає загальну інформацію: ідентифікатор (id), посилання на магазин (store_id), статус документа

(наприклад, New, Processing, Shipped, Delivered) та мітку часу створення (created_at). Деталізація замовлення виноситься в підпорядковану сутність «Позиції замовлення» (OrderItems), яка зв'язується з головною таблицею відношенням «один до багатьох». Кожен запис у OrderItems містить власний ідентифікатор (id), зовнішній ключ на замовлення (order_id), зовнішній ключ на товар (product_id) та кількість замовленого товару (quantity). Такий розподіл гарантує відсутність дублювання загальної інформації про замовлення для кожної товарної позиції.

П'ятий, найбільш складний алгоритмічний блок логічної моделі стосується безпосередньо транспортної логістики. Довідник «Транспортні засоби» (Vehicles) описує доступний автопарк і містить ідентифікатор (id), державний номерний знак (plate_number) та параметри максимальної вантажопідйомності (max_weight_kg) і місткості (max_volume_m3). Саме ці параметри порівнюватимуться з масо-габаритними характеристиками замовлених товарів алгоритмом OR-Tools під час генерації рейсів.

Результатом роботи геоінформаційного модуля оптимізації є формування сутностей «Маршрути» (Routes) та «Точки маршруту» (RoutePoints). Сутність Routes є заголовком логістичного рейсу. Вона включає ідентифікатор (id), зовнішній ключ на обраний автомобіль (vehicle_id), дату виконання рейсу (route_date), загальну розраховану дистанцію (total_distance_km) та статус виконання рейсу (status). Для деталізації шляху автомобіля створюється сутність RoutePoints, яка описує сувору послідовність відвідування роздрібних точок. Атрибутивний склад цієї таблиці включає: ідентифікатор (id), посилання на загальний маршрут (route_id), посилання на конкретне замовлення (order_id), яке має бути доставлене в цій точці, порядковий номер зупинки (stop_sequence) та розрахунковий час прибуття (estimated_arrival_time). Таке проектування забезпечує повну прозорість логістичного ланцюга: від первинної потреби магазину до конкретної зупинки автомобіля під час розвезення.

Для забезпечення надійності системи та запобігання аномаліям, розроблена концептуальна схема була піддана процедурі поетапної нормалізації на основі алгебри Кодда.

Процес зведення бази даних до Першої нормальної форми (1NF) полягав у забезпеченні атомарності всіх значень атрибутів. У нашій моделі не допускається зберігання списків або масивів в одному полі. Саме з цією метою перелік замовлених товарів не зберігається як JSON-об'єкт або текст із комами в таблиці Orders, а винесений в окрему повноцінну таблицю OrderItems. Кожен рядок містить скалярні значення, що гарантує можливість використання стандартних агрегатних функцій SQL для розрахунку загальної ваги або вартості замовлення.

Приведення до Другої нормальної форми (2NF) вимагало усунення часткових залежностей від складених ключів. У нашій системі всі без винятку таблиці використовують штучні (сурогатні) одинарні первинні ключі автоінкрементного типу або UUID. Це означає, що будь-який неключовий атрибут (наприклад, статус замовлення) повністю залежить від первинного ідентифікатора замовлення, а не від комбінації ідентифікатора магазину та дати. Таке архітектурне рішення значно спрощує подальшу розробку на рівні ORM (Object-Relational Mapping) фреймворків.

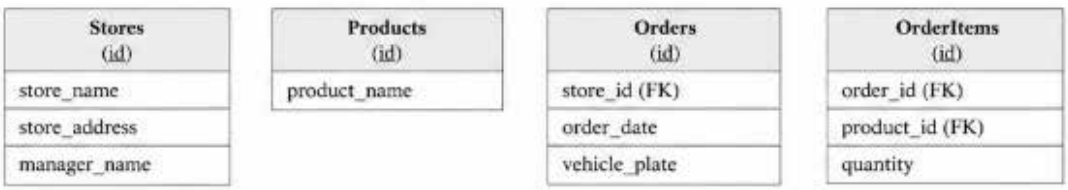

Рис. 2.2 – Структура таблиць після нормалізації до другої нормальної форми

Фінальним етапом логічного проектування стало досягнення Третьої нормальної форми (3NF), яка забороняє наявність транзитивних залежностей (коли неключовий атрибут залежить від іншого неключового атрибута). Яскравим прикладом вирішення цієї проблеми у нашій моделі є відсутність полів «адреса магазину» або «назва магазину» в таблиці Orders. Оскільки

адреса залежить виключно від ідентифікатора магазину, вона зберігається лише в довіднику Stores. При зміні фізичної адреси торгової точки адміністратору системи достатньо оновити один запис у довіднику, і ця зміна автоматично стане актуальною для всіх пов'язаних замовлень та маршрутних листів. Це гарантує цілісність даних і унеможливлює ситуацію, коли різні замовлення для одного магазину вказують на різні адреси через помилку оператора.

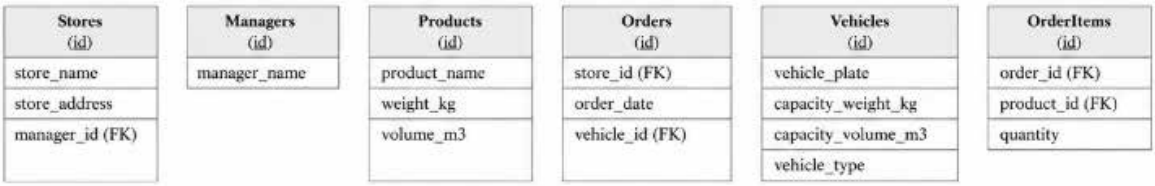

Рис. 2.3 – Результат нормалізації реляційної моделі даних до третьої нормальної форми

Важливим аспектом логічного моделювання є розробка правил підтримки посилальної цілісності (Referential Integrity) та декларативних обмежень (Constraints). Всі зв'язки між таблицями повинні підтримуватися за допомогою зовнішніх ключів (Foreign Keys). Наприклад, видалення запису з таблиці Stores не може бути виконане, якщо в таблиці Orders існують замовлення, пов'язані з цим магазином (політика ON DELETE RESTRICT). Це захищає історичні дані підприємства від випадкового знищення. З іншого боку, видалення замовлення з таблиці Orders повинно автоматично видаляти всі його товарні позиції з таблиці OrderItems (політика ON DELETE CASCADE), оскільки існування позицій замовлення без самого замовлення є логічно неможливим.

Крім посилальної цілісності, на логічному рівні закладено перевірочні обмеження (Check Constraints), які забороняють введення технічно абсурдних значень. Зокрема, для таблиці Products атрибути ваги та об'єму повинні бути строго більшими за нуль (weight_kg > 0, volume_m3 > 0). Для таблиць Inventory та OrderItems атрибут кількості не може бути від'ємним (quantity >= 0). Введення таких правил безпосередньо в модель бази даних слугує останнім рубежем захисту (last line of defense) у випадку виникнення помилок валідації на рівні клієнтського інтерфейсу або бекенд-сервера.

Окрему увагу в логічній моделі приділено просторовим даним, оскільки система має геоінформаційну специфіку. Для того щоб алгоритм маршрутизації отримував коректні вхідні дані, координати об'єктів мають бути приведені до єдиної системи відліку. У логічній моделі передбачено, що всі геометричні дані зберігатимуться у стандарті WGS 84 (SRID 4326), який є загальноприйнятим для GPS-координат. Це дозволить інтегрувати розроблену базу даних із зовнішніми картографічними провайдерами та використовувати нативні функції просторового аналізу, такі як обчислення відстаней між точками на еліпсоїді поверхні Землі.

Процес побудови реляційної моделі також включає планування індексів для оптимізації швидкості виконання запитів. Хоча фізичне створення індексів належить до етапу імплементації, логічне проектування вимагає визначення полів, за якими найчастіше здійснюватиметься пошук та з'єднання таблиць. Зокрема, передбачається створення стандартних B-Tree індексів для всіх зовнішніх ключів, а також унікальних індексів для полів sku (в таблиці товарів) та email (в таблиці користувачів). Крім того, для просторових колонок з координатами обов'язковим є створення спеціалізованих R-Tree або GiST індексів, що критично важливо для швидкості роботи картографічного модуля.

Підсумовуючи етап логічного моделювання, варто зазначити, що спроектована схема бази даних повністю відображає складну бізнес-логіку логістичної мережі. Вона забезпечує високий рівень нормалізації, усуває надлишковість інформації та гарантує консистентність даних під час паралельної роботи користувачів. Чітке розмежування облікової та логістичної складової у структурі таблиць створює ідеальні умови для подальшої програмної реалізації мікросервісів або модульного моноліту. Запропонована структура даних є повністю підготовленою для фізичного впровадження у середовищі обраної реляційної СУБД, вибір та обґрунтування якої буде здійснено в наступному підрозділі роботи.

2.2 Вибір системи управління інформаційною базою

Етап вибору системи управління базами даних є критичним кроком у проектуванні архітектури програмного забезпечення, оскільки він безпосередньо впливає на продуктивність, масштабованість та надійність усієї системи. Для розроблюваної автоматизованої системи управління логістикою цей вибір ускладнюється тим, що інформаційна база повинна одночасно обслуговувати два принципово різні типи навантажень: транзакційні операції складського обліку та складні просторові обчислення (геоінформаційні запити), необхідні для роботи алгоритмів маршрутизації. Відповідно, обрана СУБД має забезпечувати високу швидкість запису даних, абсолютну консистентність фінансових транзакцій та мати вбудовані механізми для маніпуляції геометричними об'єктами.

Формування критеріїв вибору СУБД базується на специфіці логістичного бізнесу. По-перше, система повинна суворо підтримувати властивості ACID (Атомарність, Узгодженість, Ізольованість, Довговічність), щоб унеможливити ситуації, коли товар списується зі складу, але не закріплюється за жодним маршрутом через програмний збій. По-друге, архітектура бази даних має підтримувати високий рівень конкурентного доступу, оскільки диспетчери, комірники та менеджери магазинів працюватимуть із системою паралельно. По-третє, враховуючи необхідність інтеграції з географічними інформаційними системами (GIS), база повинна вміти індексувати просторові дані (широту та довготу) і виконувати математичні операції над ними безпосередньо на рівні SQL-запитів [31, с. 415].

Розглядаючи сучасний ринок баз даних, насамперед слід провести вододіл між реляційними (SQL) та нереляційними (NoSQL) системами. Останніми роками NoSQL-рішення, такі як MongoDB або Cassandra, набули значної популярності завдяки гнучкості схем (schema-less) та легкості горизонтального масштабування. Проте для нашого проекту використання документо-орієнтованих баз даних є недоцільним. Логістична система оперує суворо

структурованими даними зі складною мережею зв’язків (наприклад, Замовлення -> Товари -> Складські залишки). Реалізація складних з'єднань (JOIN) у NoSQL базах є вкрай ресурсомісткою, а відсутність строгих декларативних обмежень (Constraints) на рівні бази створює ризики порушення посилальної цілісності та виникнення пересортиці [18].

Відкинувши нереляційні рішення, подальший аналіз фокусується на класичних реляційних СУБД (RDBMS). Серед комерційних платформ беззаперечними лідерами є Oracle Database та Microsoft SQL Server. Вони пропонують потужний функціонал, вбудовані засоби бізнес-аналітики та високу надійність. Проте їх використання для автоматизації логістики середньої мережі магазинів є економічно необґрунтованим через надзвичайно високу вартість ліцензування (як за ядра процесора, так і за клієнтські підключення), а також необхідність утримання вузькопрофільних адміністраторів баз даних. Крім того, закритий вихідний код створює ефект прив'язки до вендора (vendor lock-in), що обмежує гнучкість розробки [30, с. 210].

Таким чином, вибір звужується до реляційних СУБД з відкритим вихідним кодом, серед яких найпопулярнішими є MySQL та PostgreSQL. MySQL традиційно вважається більш швидкою при виконанні простих запитів на читання (SELECT) у веб-додатках, що робить її стандартом для систем управління контентом (CMS). Проте, коли справа доходить до виконання складних аналітичних запитів, масових оновлень, роботи з підзапитами та віконними функціями, архітектурні обмеження MySQL стають очевидними. Крім того, обробка транзакцій та забезпечення блокувань на рівні рядків у MySQL реалізується через підключення зовнішніх механізмів зберігання, таких як InnoDB, що додає зайвої складності при налаштуванні [7, с. 340].

Повноцінною альтернативою виступає об'єктно-реляційна СУБД PostgreSQL. Вона вважається найбільш просунутою СУБД з відкритим кодом, архітектура якої максимально наближена до стандартів ANSI SQL. PostgreSQL від початку розроблялася з акцентом на розширюваність, відповідність

стандартам та цілісність даних. Важливим аргументом на користь цієї системи є реалізація механізму багатоверсійного управління конкурентним доступом (MVCC). Завдяки MVCC операції читання ніколи не блокують операції запису, і навпаки. Це означає, що коли алгоритм OR-Tools інтенсивно розраховує маршрути, блокуючи таблиці з рейсами для запису, менеджери магазинів можуть без жодних затримок переглядати каталог товарів та формувати нові замовлення. Це гарантує високу чуйність системи навіть за умов пікових навантажень [38, с. 512].

Проте вирішальним фактором, який остаточно схилив вибір на користь PostgreSQL, стала наявність спеціалізованого просторового розширення PostGIS. PostGIS не просто додає нові типи даних; воно перетворює PostgreSQL на повноцінну просторову базу даних, яка за своїм функціоналом перевершує багато комерційних GIS-платформ. У контексті нашої задачі - автоматизації маршрутизації транспортних засобів - ця можливість є критично важливою [15].

За замовчуванням більшість баз даних (у тому числі MySQL) зберігають координати як звичайні числа з плаваючою комою. Обчислення відстані між двома такими точками на сфері вимагає написання складних математичних функцій (наприклад, формули гаверсинуса) безпосередньо у програмному коді або громіздких SQL-запитах, що унеможливлює використання індексів і призводить до повного сканування таблиць (Full Table Scan). PostGIS вирішує цю проблему нативно. Зберігаючи координати роздрібних магазинів у спеціальному типі даних GEOMETRY або GEOGRAPHY (з використанням стандарту просторової прив'язки SRID 4326), ми отримуємо можливість використовувати просторові R-Tree індекси на базі технології GiST [11].

Використання PostGIS дозволяє розвантажити бекенд-сервер додатку (написаний на Python), переклавши "важку" геопросторову математику на рівень бази даних, який написаний на мові C і є максимально оптимізованим. Наприклад, щоб знайти всі нерозподілені замовлення в радіусі 5 кілометрів від поточного положення вільного автомобіля, достатньо виконати елементарний

запит із використанням функції ST_DWithin(). База даних, використовуючи просторові індекси, відфільтрує мільйони записів за мілісекунди і поверне додатку лише потрібний набір точок. Така архітектура забезпечує масштабованість системи навіть за умови розширення мережі до тисяч роздрібних точок [15].

Крім картографічних можливостей, PostgreSQL ідеально вписується у вибраний стек розробки бекенду. Розроблюваний серверний додаток базується на асинхронному фреймворку FastAPI (Python). Для досягнення максимальної пропускної здатності (throughput) критично важливо, щоб усі операції вводу-виводу (I/O), включно із запитами до бази даних, виконувалися неблокуючим (асинхронним) чином. PostgreSQL має чудову підтримку асинхронних драйверів (наприклад, asyncpg), які інтегруються з сучасними ORM-системами типу SQLAlchemy (починаючи з версії 1.4+). Це забезпечує наскрізну асинхронність системи: від прийому HTTP-запиту клієнта до фізичного запису даних на диск сервера баз даних [14].

Ще однією вагомою перевагою PostgreSQL для сучасної веб-розробки є нативна підтримка типу даних JSON та JSONB (бінарний JSON). Хоча наша логічна модель суворо нормалізована, в реальних логістичних системах часто виникає потреба зберігати неструктуровані метадані. Наприклад, відповіді від зовнішніх API картографічних сервісів (Google Maps Directions API, OSRM) можуть містити складну геометрію маршруту (полілайни) або покрокові інструкції навігації, які немає сенсу розбирати на десятки реляційних таблиць. PostgreSQL дозволяє зберігати ці відповіді у стовпцях типу JSONB з можливістю їх повноцінного індексування та пошуку по внутрішніх ключах JSON. Це поєднує строгість SQL з гнучкістю NoSQL в межах однієї транзакційної бази [11].

З точки зору безпеки та адміністрування, PostgreSQL надає потужні механізми автентифікації (включаючи підтримку LDAP/Active Directory для корпоративних мереж) та гнучку систему управління правами доступу (Row-Level Security - RLS). RLS дозволяє налаштувати базу даних таким чином, що,

наприклад, менеджер конкретного магазину фізично не зможе прочитати рядки з таблиці замовлень, які належать іншим магазинам, навіть якщо виконає запит SELECT * FROM Orders. Це виключає витоки комерційної інформації через вразливості на рівні коду додатка.

Підсумовуючи результати порівняльного аналізу, можна з упевненістю стверджувати, що об'єктно-реляційна система управління базами даних PostgreSQL є найбільш технологічно обґрунтованим вибором для реалізації бази даних автоматизованої системи управління логістикою. Поєднання абсолютної надійності транзакцій (ACID), архітектури MVCC для висококонкурентного доступу, підтримки наскрізної асинхронності в стеку Python/FastAPI та наявності безпрецедентно потужного розширення PostGIS для обробки просторових даних робить PostgreSQL безальтернативним інструментом для розв'язання поставлених інженерних задач. Такий вибір СУБД створює надійний та високопродуктивний фундамент для переходу до етапу фізичного проектування таблиць та безпосереднього розгортання інформаційної бази, що буде описано в наступному підрозділі роботи.

2.3 Створення інформаційної бази

Перехід від логічної моделі даних до фізичної реалізації є завершальним етапом проектування інформаційного забезпечення. На цьому етапі абстрактні сутності та відношення трансформуються у конкретні структури обраної реляційної СУБД PostgreSQL. Процес створення інформаційної бази передбачає написання скриптів на мові визначення даних (Data Definition Language, DDL), вибір оптимальних фізичних типів даних для кожного атрибута, налаштування механізмів просторового індексування та імплементацію декларативних обмежень цілісності на рівні ядра бази даних [29, с. 210].

Першим кроком у фізичному розгортанні бази даних є ініціалізація кластера PostgreSQL та створення нового екземпляра бази. Враховуючи необхідність обробки багатомовного контенту (назви товарів, адреси), база даних ініціалізується з кодуванням UTF-8. Одразу після створення базової структури виконується критично важлива команда активації просторового розширення: CREATE EXTENSION postgis;. Це розширення додає до стандартного SQL-синтаксису PostgreSQL типи даних GEOMETRY та GEOGRAPHY, а також понад тисячу спеціалізованих функцій для просторового аналізу, які будуть активно використовуватися модулем оптимізації маршрутів при розрахунку матриць відстаней [15].

Важливим архітектурним рішенням на етапі створення таблиць є правильний вибір типів даних. Для первинних ключів (Primary Keys) усіх таблиць обрано тип SERIAL (або INTEGER GENERATED ALWAYS AS IDENTITY у новітніх стандартах PostgreSQL), що забезпечує автоматичну генерацію унікальних ідентифікаторів. Для зберігання масо-габаритних характеристик товарів (вага, об'єм), які використовуються алгоритмом OR-Tools як обмежувачі місткості транспорту, категорично відкинуто використання типів з рухомою комою (наприклад, REAL або DOUBLE PRECISION) через проблему втрати точності за стандартом IEEE 754. Натомість використано тип з фіксованою точністю NUMERIC(10,3), що гарантує абсолютну точність математичних обчислень [11].

Для зберігання часових відміток (реєстрація замовлення, розрахунковий час прибуття на точку доставки) використовується тип TIMESTAMP WITH TIME ZONE. Це є фундаментальним стандартом (best practice) для систем, які потенційно можуть масштабуватися на різні географічні регіони, оскільки база даних автоматично нормалізує час до формату UTC при збереженні на диск та конвертує в локальний часовий пояс користувача під час читання [8].

Фізична реалізація просторових даних вимагає особливої уваги. Таблиці stores (Магазини) та warehouses (Склади) містять колонку location типу GEOMETRY (Point, 4326). Використання стандарту просторової прив'язки

SRID 4326 (WGS 84) є обов'язковим, оскільки вхідні геодані від клієнтських мобільних пристроїв та зовнішніх API (наприклад, картографічної бібліотеки Leaflet або сервісів геокодування) надходять саме в цьому форматі - як градуси широти та довготи [12].

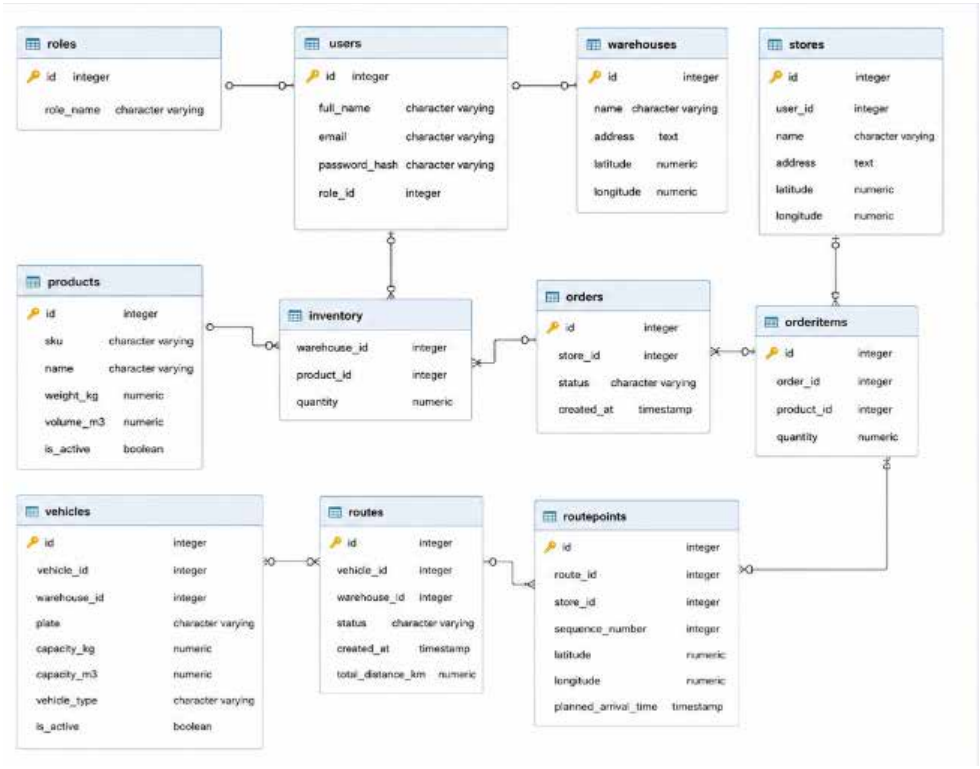

Рис. 2.4 – Фізична модель бази даних PostgreSQL

Для ілюстрації процесу перетворення логічної моделі на фізичну, нижче наведено фрагмент базового DDL-скрипта, який демонструє фізичне створення таблиці роздрібних магазинів із застосуванням просторових типів даних та декларативних зовнішніх ключів:

```
CREATE TABLE stores (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    name VARCHAR(150) NOT NULL,
    address VARCHAR(255) NOT NULL,
    location GEOMETRY(Point, 4326) NOT NULL,
    CONSTRAINT fk_store_user
        FOREIGN KEY (user_id)
```

```
        REFERENCES users (id)
        ON DELETE RESTRICT
);
```

Після створення таблиць критично важливим етапом є побудова індексів для забезпечення високої швидкодії системи під навантаженням. Окрім автоматично створюваних B-Tree індексів для первинних ключів, фізична модель бази даних містить явно визначені індекси для всіх полів, що виступають зовнішніми ключами (наприклад, store_id у таблиці orders). Це дозволяє уникнути ресурсомісткого повного сканування таблиці під час виконання операцій JOIN та гарантує миттєве виконання каскадних перевірок політик ON DELETE [31].

Особливе місце в стратегії індексування посідає оптимізація просторових запитів. Для колонок location у таблицях магазинів та складів створюються індекси типу GiST. На відміну від класичного B-Tree, який базується на лінійному сортуванні значень, GiST оперує R-деревами (деревами прямокутників, що просторово обмежують об'єкти - Bounding Boxes). Створення такого індексу (CREATE INDEX idx_stores_location ON stores USING GIST (location);) дозволяє базі даних при виконанні запитів швидко відсікати географічні зони, в яких точно немає потрібних магазинів, що прискорює пошук найближчих точок доставки у сотні разів [11].

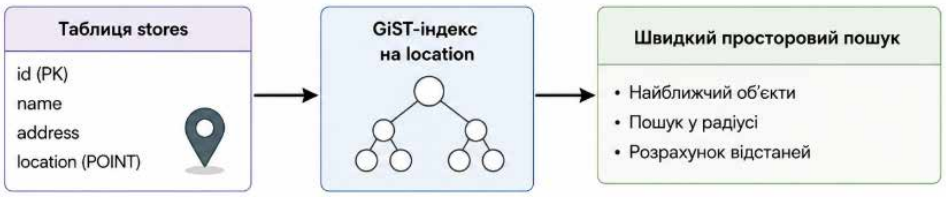

Рис. 2.5 – Використання GiST-індексів для просторових даних

Реалізація бізнес-правил на рівні ядра бази даних досягається через декларативні обмеження. Наприклад, для таблиці складських залишків inventory створено складений унікальний індекс UNIQUE(warehouse_id, product_id). Він фізично унеможливлює ситуацію, коли один і той самий товар

має кілька дублюючих записів про залишки на одному складі. Крім того, на критичні поля додано обмеження перевірки CHECK (quantity >= 0), яке відхиляє будь-які SQL-транзакції, що призвели б до від'ємних залишків товару на складі, слугуючи останнім надійним рубежем захисту від багів у клієнтському коді [29].

З архітектурної точки зору, для підготовки інформаційної бази до взаємодії з програмним кодом бекенду, написаним мовою Python з використанням високопродуктивного фреймворку FastAPI, розроблена фізична схема інтегрується з системою об'єктно-реляційного відображення (ORM) SQLAlchemy. Замість ручного виконання SQL-скриптів на робочих серверах, процес створення та еволюції бази даних автоматизується за допомогою інструменту управління міграціями Alembic. У цій парадигмі структура бази даних описується декларативно у вигляді Python-класів, а всі зміни в схемі фіксуються у вигляді версійних файлів. Такий підхід "Інфраструктура як код" (Infrastructure as Code) гарантує абсолютну ідентичність інформаційної бази на локальних комп'ютерах розробників, тестових стендах та сервері продакшену [14].

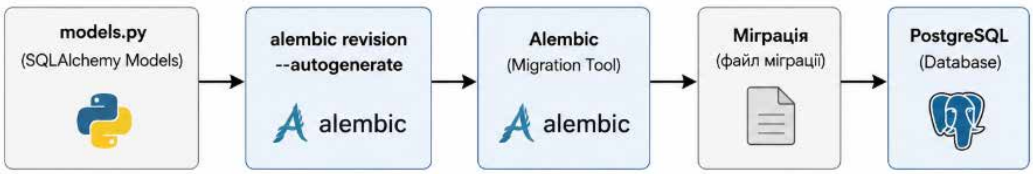

Рис. 2.6 – Автоматизація керування схемою бази даних за допомогою Alembic

Підсумовуючи вищевикладене, етап створення інформаційної бази завершується розгортанням повноцінної, високопродуктивної реляційної бази даних PostgreSQL із просторовим розширенням PostGIS. Створена фізична структура не лише повністю відповідає логічній моделі в Третій нормальній формі (3NF), але й містить глибокі оптимізації на рівні типів даних та індексів. Жорсткі декларативні обмеження цілісності надійно захищають фінансово-логістичні дані від будь-яких програмних помилок на рівні прикладного

інтерфейсу. Таким чином, інформаційне забезпечення повністю готове до обслуговування як масових конкурентних транзакцій складського обліку, так і складних математичних алгоритмів маршрутизації, безпосередня програмна реалізація яких буде детально розглянута в наступному розділі роботи.

РОЗДІЛ 3. ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Організаційна структура програмного забезпечення

Проектування організаційної структури програмного забезпечення є етапом, на якому визначається внутрішня логіка взаємодії компонентів системи, розподіл функціональних обов'язків між модулями та принципи обміну даними між клієнтською та серверною частинами. Враховуючи поставлене завдання з автоматизації логістичних процесів, архітектура додатка повинна бути орієнтована на високу швидкість обробки транзакцій та гнучкість при роботі з картографічними сервісами. Для реалізації цих вимог мною було обрано архітектурний стиль побудови веб-додатків на основі розділення фронтенду та бекенду, де взаємодія відбувається через програмний інтерфейс REST API.

Організаційна структура розроблюваного програмного продукту складається з трьох фундаментальних рівнів, що утворюють цілісну екосистему: рівня представлення (Frontend), серверного рівня бізнес-логіки (Backend) та рівня збереження даних (Database). Кожен рівень ізольований, що дозволяє проводити модернізацію окремих модулів без необхідності повної переробки всієї системи.

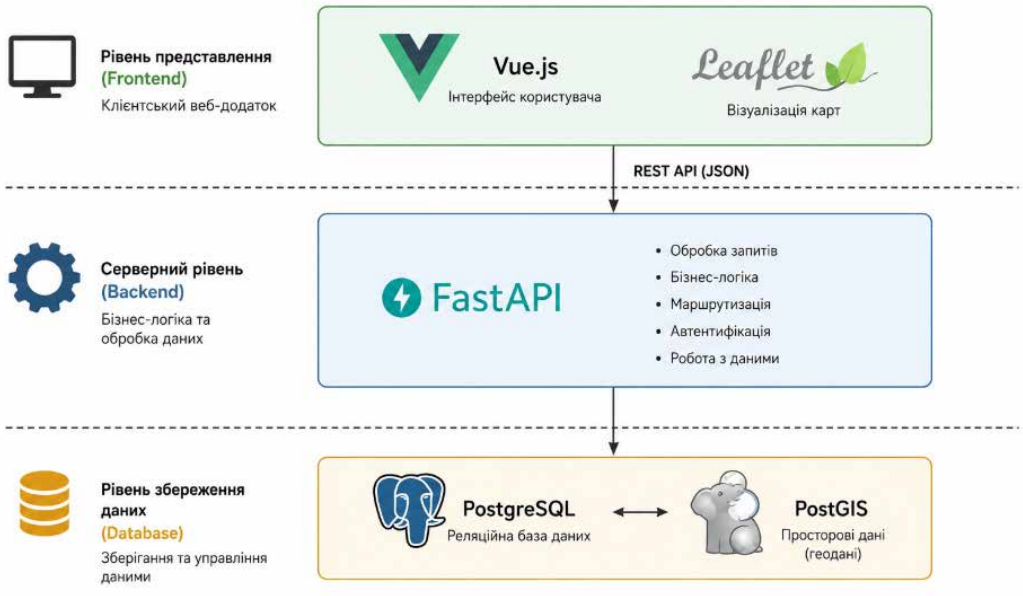

Рис. 3.1 – Архітектура програмного забезпечення

Серверний рівень (Backend) є «інтелектуальним центром» програми. Його організаційна структура побудована за модульним принципом, де кожен окремий модуль відповідає за конкретний бізнес-процес. Основними компонентами серверної частини є:

1. Модуль автентифікації та розмежування доступу: відповідає за реєстрацію користувачів, перевірку прав доступу (ролей) та генерацію безпечних токенів для сесій.
2. Модуль управління замовленнями: реалізує логіку створення, редагування та фільтрації заявок від магазинів, забезпечуючи їх коректне збереження в базі даних.
3. Складський модуль: контролює залишки товарів, проводить автоматичне резервування продукції при створенні замовлення та оновлює інвентаризаційні дані після відвантаження.
4. Ядро оптимізації маршрутів: найбільш складний компонент, який інтегрує математичні алгоритми для розв’язання задачі комівояжера та розподілу замовлень між транспортними засобами з урахуванням їхньої вантажопідйомності.

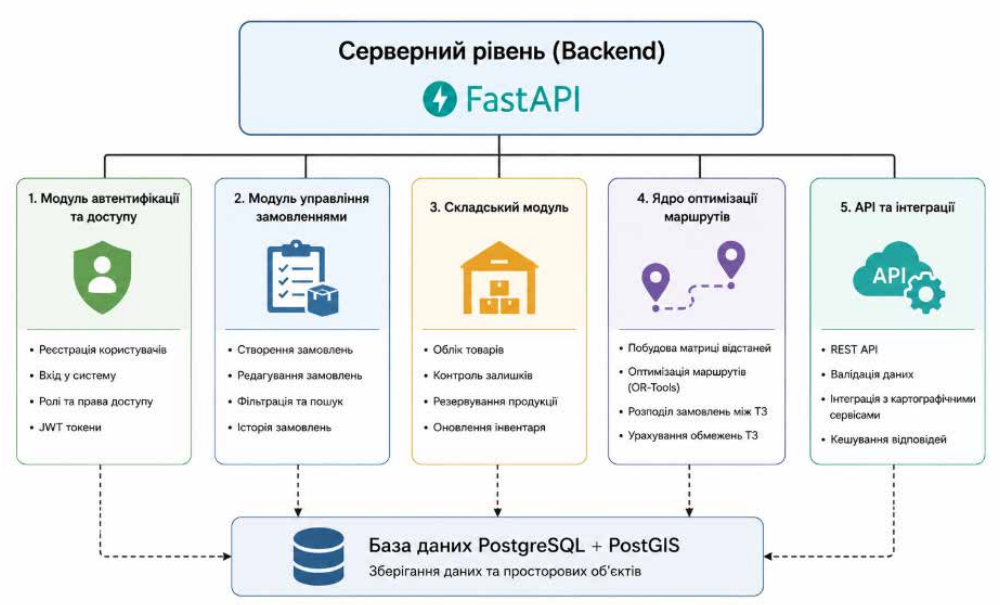

Рис. 3.2 – Структура серверних модулів системи

Рівень представлення (Frontend) організований як односторінковий веб-додаток. Структура інтерфейсу розділена на функціональні зони відповідно до ролей користувачів. Для менеджера магазину передбачено панель формування замовлень та перегляду історії поставок. Для диспетчера організовано інтерактивне робоче місце з вбудованою картою, де відображаються об'єкти інфраструктури та прокладені маршрути. Комірник має доступ до терміналу збору даних, що відображає чергу накладних на збірку. Така організація інтерфейсу дозволяє мінімізувати кількість переходів між сторінками та забезпечує миттєвий відгук системи на дії користувача.

Взаємодія між модулями всередині системи відбувається через сервісний шар. Коли менеджер створює замовлення, фронтенд надсилає асинхронний запит до API. Модуль управління замовленнями звертається до складського модуля для перевірки наявності товару. Якщо перевірка успішна, дані записуються в базу даних, а модуль маршрутизації отримує сигнал про появу нової точки, яку необхідно врахувати при наступній генерації рейсів. Всі обміни даними між компонентами відбуваються у форматі JSON, що є стандартом для сучасних розподілених систем.

Окрему увагу в структурі програмного забезпечення приділено інтеграції із зовнішніми картографічними сервісами. Оскільки розрахунок реальних дорожніх дистанцій є ресурсомістким, система має окремий шар адаптерів, які надсилають запити до зовнішніх серверів (геокодерів та маршрутизаторів) і кешують отримані результати в локальній базі даних для повторного використання. Це дозволяє суттєво скоротити час відгуку при повторному плануванні маршрутів у тій самій зоні.

Така багаторівнева організаційна структура забезпечує високу відмовостійкість системи. Навіть при виникненні помилок у модулі візуалізації карти на фронтенді, серверна частина продовжуватиме коректно приймати замовлення та обробляти складські операції. Логічне розділення компонентів створює надійну базу для програмної реалізації, оскільки дозволяє чітко визначити вхідні та вихідні дані для кожного модуля ще до початку написання

коду. Побудована архітектура є масштабованою, що дає змогу в майбутньому легко додавати нові функціональні блоки, такі як модуль аналітики або система сповіщень клієнтів, не порушуючи цілісності вже існуючого програмного забезпечення.

3.2 Вибір інструментарію для створення ППЗ

Визначення технологічного стеку для розробки автоматизованої системи управління логістикою є критичним етапом, що безпосередньо впливає на швидкість обробки даних, надійність архітектури та зручність кінцевого користувача. На мою думку, вибір інструментів не повинен ґрунтуватися лише на популярності тієї чи іншої технології; він має бути результатом прагматичного аналізу здатності мови програмування чи фреймворку ефективно вирішувати специфічні задачі маршрутизації та складського обліку. Я прагнув обрати таку комбінацію технологій, яка забезпечує максимальну продуктивність при роботі з асинхронними мережевими запитами та одночасно надає потужний математичний апарат для розв'язання NP-складних задач логістичної оптимізації.

Основним інструментом розробки серверної частини системи обрано високорівневу мову програмування Python 3. Цей вибір зумовлений її надзвичайною гнучкістю, читабельністю коду та наявністю найбільшої в індустрії екосистеми бібліотек для наукових обчислень, що дозволяє легко інтегрувати складні математичні модулі в стандартні веб-сервіси [13].

В ролі базового веб-фреймворку для побудови REST API обрано FastAPI. Дана технологія базується на стандартах Starlette та Pydantic, що дозволяє автоматично валідувати вхідні дані та генерувати інтерактивну документацію. Використання FastAPI забезпечує нативну підтримку асинхронного програмування, завдяки чому сервер може обробляти значну кількість паралельних запитів від магазинів та складів без блокування потоків виконання, що суттєво підвищує загальну пропускну здатність системи [14].

Для вирішення центральної інженерної задачі - оптимізації маршрутів доставки - було обрано спеціалізований інструментарій Google OR-Tools. Ця бібліотека є лідером у сфері комбінаторної оптимізації та надає готові алгоритми пошуку оптимальних рішень для транспортних задач з урахуванням вантажопідйомності, часових вікон та пріоритетів доставки. Використання OR-Tools дозволяє значно скоротити час розробки, оскільки надає перевірені на практиці методи лінійного та цілочисельного програмування [31].

Важливим аспектом логістичної системи є робота з геоінформаційними даними, що вимагає використання інструментів просторового аналізу. Взаємодія Python-додатка з базою даних PostgreSQL із розширенням PostGIS реалізується через об'єктно-реляційне відображення за допомогою бібліотек SQLAlchemy та GeoAlchemy2. Це дозволяє маніпулювати географічними об'єктами як звичайними класами Python, автоматизуючи складні SQL-запити для обчислення відстаней між складом та роздрібними точками на основі координат [15].

З боку клієнтського інтерфейсу було обрано прогресивний JavaScript-фреймворк Vue.js. Його реактивна модель даних та компонентний підхід дозволяють створити чуйний інтерфейс, де диспетчер може в реальному часі відстежувати зміни статусів замовлень та бачити результати роботи алгоритму маршрутизації без повного перезавантаження сторінки веб-переглядача [17].

Візуалізація картографічних даних на фронтенді здійснюється за допомогою бібліотеки Leaflet. Вона є легкою, мобільно-орієнтованою та має відкритий вихідний код, що робить її ідеальним вибором для відображення інтерактивних карт, нанесення маркерів магазинів та малювання ліній прокладених маршрутів. Leaflet забезпечує плавну взаємодію з картою та підтримує різноманітні шари картографічних підкладок, таких як OpenStreetMap [12].

Для забезпечення стабільності та передбачуваності розгортання системи в різних середовищах використовується технологія контейнеризації. Оформлення кожного компонента системи (бази даних, бекенд-сервера, фронтенд-додатка) у

вигляді Docker-контейнерів дозволяє уникнути проблем сумісності версій бібліотек та спрощує процес масштабування системи при збільшенні навантаження на обчислювальні вузли [37].

Якісна розробка програмного продукту також вимагає дотримання стандартів оформлення коду та архітектурних патернів. Використання офіційних рекомендацій PEP 8 при написанні коду на Python гарантує його чистоту та легкість підтримки іншими розробниками в майбутньому. Такий підхід до вибору інструментарію базується на принципах «чистої архітектури», де кожен елемент технологічного стеку виконує свою роль, мінімізуючи зв’язність компонентів та підвищуючи загальну надійність ППЗ [2].

Окремо варто зупинитися на виборі допоміжних інструментів, які забезпечують надійність обміну даними та безпеку системи. Для валідації вхідних структур на рівні бекенду я вирішив використовувати бібліотеку Pydantic, яка є невід'ємною частиною FastAPI. Вона дозволяє описувати схеми даних у вигляді стандартних класів Python, забезпечуючи автоматичне перетворення типів та генерацію зрозумілих повідомлень про помилки. Це критично важливо при роботі з замовленнями, де некоректний формат артикулу товару або від'ємна вантажопідйомність автомобіля можуть призвести до збою в алгоритмах оптимізації. Використання Pydantic дозволяє мені бути впевненим, що до ядра системи потрапляють лише валідовані та структуровані дані.

Для організації взаємодії між клієнтською та серверною частинами я обрав стандарт архітектури REST. Використання протоколу HTTP у поєднанні з форматом JSON як основним транспортним шаром забезпечує легку інтеграцію та високу швидкість парсингу пакетів. На відміну від громіздких XML-структур, JSON дозволяє передавати складні географічні об’єкти та масиви точок маршруту з мінімальними накладними витратами на мережевий трафік. Це позитивно впливає на швидкість відображення результатів розрахунків на інтерактивній карті у диспетчера.

Питання безпеки та захисту комерційних даних логістичної мережі реалізовано за допомогою протоколу OAuth2 та використання JSON Web Tokens (JWT). Для хешування паролів у базі даних я обрав бібліотеку Passlib з алгоритмом BCrypt, що відповідає сучасним стандартам криптографічного захисту. Така архітектура автентифікації дозволяє реалізувати безстанову (stateless) взаємодію з сервером, де кожна дія користувача підписується унікальним токеном, що значно підвищує стійкість системи до несанкціонованого доступу та атак типу перехоплення сесії.

У виборі інструментів візуалізації я віддав перевагу бібліотеці Leaflet замість комерційних Google Maps API або Mapbox. Моє рішення базується на прагненні зберегти повну автономність системи та уникнути залежності від платних тарифних планів картографічних провайдерів. Leaflet дозволяє легко підключати безкоштовні шари OpenStreetMap і гнучко налаштовувати відображення специфічних логістичних об’єктів за допомогою кастомних маркерів та ліній маршрутів. Це забезпечує необхідний рівень візуалізації без додаткових витрат на експлуатацію програмного продукту.

Процес розробки та супроводу програмного коду базується на системі контролю версій Git. Це дає мені можливість вести історію змін, створювати окремі гілки для тестування нових евристичних алгоритмів маршрутизації та за потреби повертатися до попередніх стабільних версій системи. Для автоматизації розгортання середовища я використовую Docker Compose, що дозволяє одним натисканням запустити весь стек: від сервера бази даних до клієнтського інтерфейсу. Такий підхід до організації інфраструктури розробки гарантує, що система буде працювати ідентично на будь-якому комп'ютері, усуваючи класичну проблему несумісності бібліотек.

Обрана мною сукупність інструментів, що включає мову Python, асинхронний фреймворк FastAPI, математичне ядро OR-Tools та динамічний фронтенд на Vue.js, утворює збалансовану екосистему для створення сучасного логістичного софту. Кожен елемент цього стеку був підібраний з урахуванням потреби у високій продуктивності, безпеці та економічній доцільності. Це

дозволяє мені перейти до практичного етапу реалізації програмних модулів, маючи надійний технологічний фундамент, здатний витримувати реальні навантаження та забезпечувати високу точність логістичних розрахунків.

Обґрунтований вибір інструментарію створює надійний технологічний фундамент для перетворення теоретичних моделей у працездатний програмний продукт. Поєднання асинхронного веб-сервера на базі FastAPI, потужного математичного ядра Google OR-Tools та динамічного інтерфейсу на Vue.js з картографією Leaflet дозволяє реалізувати систему, що повністю відповідає вимогам сучасної логістики. Це дає можливість перейти до наступного етапу - безпосередньої алгоритмізації та кодування модулів системи, маючи впевненість у продуктивності та масштабованості обраного рішення.

3.3 Алгоритмізація та програмування програмних модулів

Етап алгоритмізації та безпосереднього програмування є кульмінаційним моментом усієї розробки, де теоретичні моделі, архітектурні схеми та структури баз даних перетворюються на реальний функціональний інструментарій. На мою думку, саме якість написання коду та точність реалізації математичних алгоритмів визначають успіх впровадження системи в реальну логістичну мережу. Я вважаю, що розробник не повинен обмежуватися лише написанням команд; необхідно глибоко розуміти логіку кожного переходу даних, щоб забезпечити стабільність системи в умовах високих навантажень. Програмна реалізація модулів у моєму проекті базується на принципах модульності та асинхронності, що дозволяє ізолювати важкі обчислення від інтерфейсу користувача.

Центральним елементом програмного забезпечення є алгоритм розв’язання задачі маршрутизації транспортних засобів. Для його реалізації я використав математичне ядро бібліотеки Google OR-Tools, яке дозволяє моделювати складні логістичні обмеження. Алгоритм працює в кілька етапів: спочатку

формується матриця відстаней між усіма точками доставки, потім визначається кількість доступних транспортних засобів та їхня вантажопідйомність, і на фінальному етапі запускається пошук оптимального рішення за допомогою метаевристичних методів, таких як пошук з табуюванням або керований локальний пошук.

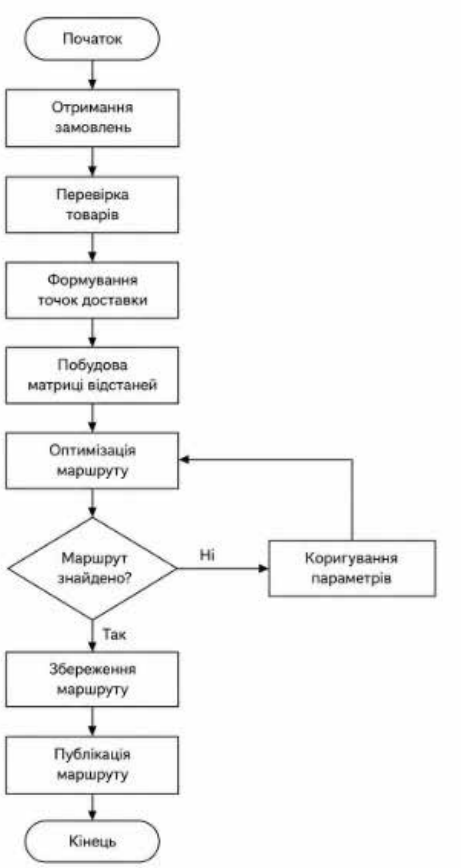

Рис.3.3 – Блок-схема процесу генерації маршруту

Для забезпечення коректності вхідних даних перед запуску алгоритму здійснюється попередня обробка інформації з бази даних PostgreSQL. Модуль збору даних формує масив об’єктів, кожен з яких містить координати магазину, об’єм замовлення та пріоритетність доставки. Важливо, що використання просторового розширення PostGIS дозволяє обчислювати початкові відстані між точками безпосередньо на рівні SQL-запитів, що значно скорочує час підготовки даних для основного розрахункового модуля.

Програмна реалізація бекенд-частини на базі фреймворку FastAPI передбачає створення асинхронних маршрутів (endpoints) для обробки запитів

від фронтенду. Кожен запит на генерацію маршруту запускається у фоновому потоці, що дозволяє диспетчеру продовжувати роботу з іншими модулями системи, поки сервер виконує ресурсомісткі обчислення. Такий підхід забезпечує високу чуйність інтерфейсу та унеможливлює зависання веб-сторінки при роботі з великою кількістю заявок.

Окрему увагу я приділив модулю валідації даних, який побудований на бібліотеці Pydantic. Цей модуль перевіряє відповідність вхідних JSON-об’єктів заданим схемам ще до моменту їх потрапляння в бізнес-логіку програми. Це дозволяє усунути помилки, пов’язані з некоректним введенням масо-габаритних характеристик товарів або неправильним форматом координат, що є критично важливим для стабільної роботи графових алгоритмів.

Модуль складського обліку та управління залишками функціонує в тісному зв’язку з модулем замовлень. При програмуванні логіки підтвердження замовлення я реалізував механізм транзакційного резервування товарів. Це означає, що при успішному формуванні заявки система автоматично зменшує кількість доступного товару в таблиці залишків, що запобігає виникненню ситуацій з відсутністю продукції під час комплектування рейсу.

Візуалізація результатів роботи алгоритмів на стороні клієнта реалізована за допомогою JavaScript-бібліотеки Leaflet. Програмний модуль фронтенду отримує від сервера масив геометрій маршрутів і динамічно малює їх на карті. Кожна точка на маршруті є інтерактивною: при натисканні диспетчер може побачити деталі замовлення, контактні дані менеджера магазину та розрахунковий час прибуття автомобіля, що робить процес управління логістикою максимально прозорим.

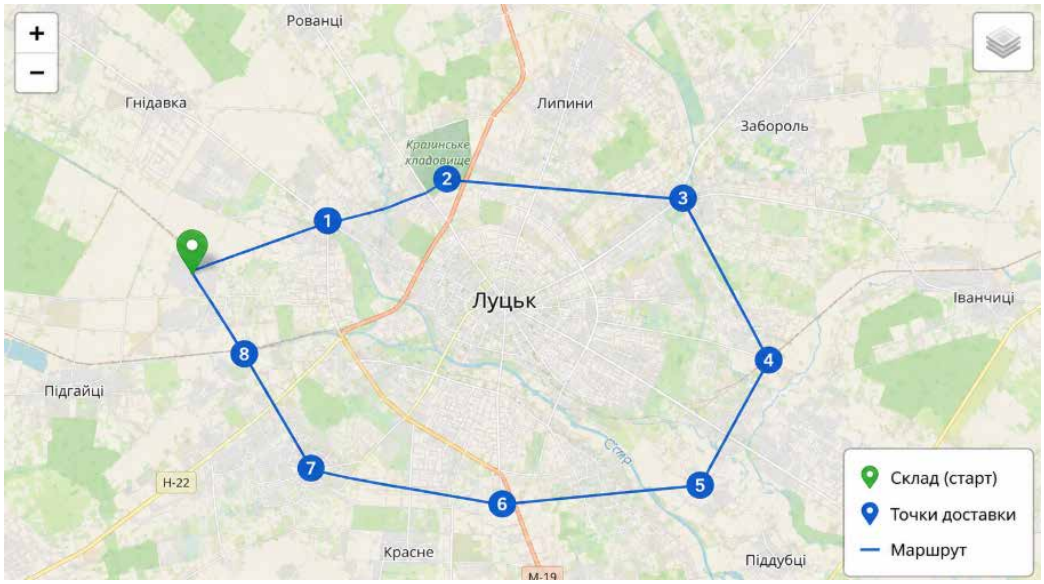

Рис. 3.4 – Карта Leaflet з маршрутом доставки

Якість програмного коду забезпечується дотриманням принципів чистої архітектури та регулярним рефакторингом. Кожен функціональний блок системи винесено в окремий пакет, що дозволяє проводити незалежне тестування модулів. Використання сучасних стандартів кодування та документування функцій забезпечує легкість подальшої підтримки та масштабування проекту при розширенні торговельної мережі.

Для забезпечення надійної взаємодії системи із зовнішнім середовищем я реалізував модуль логування всіх критичних операцій. Будь-яка помилка в роботі алгоритму або збій при підключенні до бази даних фіксується в журналі подій з детальним описом контексту. Це дозволяє оперативно діагностувати проблеми та мінімізувати час простою логістичної служби в разі виникнення технічних несправностей.

Фінальний етап програмування включав налаштування взаємодії всіх модулів через єдину точку входу. Використання контейнеризації Docker дозволило об’єднати всі програмні компоненти, включно з базою даних, алгоритмічним ядром та веб-інтерфейсом, у єдину інфраструктуру. Це гарантує, що розроблене програмне забезпечення буде працювати стабільно на будь-якому серверному обладнанні незалежно від встановлених системних бібліотек.

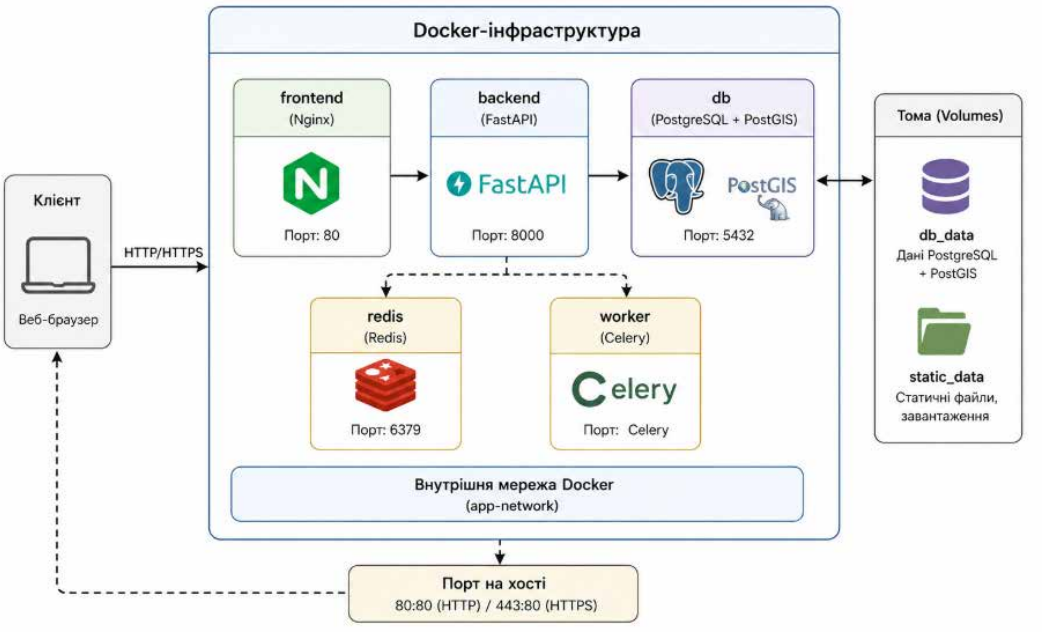

Рис.3.5 – Контейнерна архітектура програмного забезпечення на базі Docker

Таким чином, програмна реалізація модулів автоматизованої системи управління логістикою повністю відповідає поставленим завданням та архітектурним вимогам. Поєднання потужних математичних бібліотек, асинхронного веб-сервера та інтерактивних інструментів візуалізації дозволило створити цілісний програмний продукт. Розроблені алгоритми забезпечують не лише автоматизацію рутинних операцій, а й реальну оптимізацію транспортних витрат підприємства. Це створює надійну базу для переходу до етапу впровадження системи та її подальшої експлуатації в реальних умовах бізнесу.

РОЗДІЛ 4. РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Етап тестування розробленої автоматизованої системи управління логістикою «LogiNet» мав на меті комплексне підтвердження працездатності всіх програмних модулів, перевірку точності математичних алгоритмів оптимізації та верифікацію цілісності даних при паралельній роботі користувачів. Процес тестування був організований за багаторівневим принципом, охоплюючи модульне тестування окремих функцій, інтеграційне тестування взаємодії між API та базою даних, а також функціональне тестування бізнес-сценаріїв у реальному часі.

Модульне тестування алгоритмічного ядра

Першочергова увага була приділена тестуванню модуля optimizer.py, який відповідає за розрахунок геодезичних відстаней та вирішення задачі маршрутизації транспортних засобів. Для перевірки точності обчислень було проведено тестування функції розрахунку відстані за формулою Гаверсинуса:

```
    def _haversine_m(lon1: float, lat1: float, lon2: float, lat2: float) -> int:
        R = 6_371_000  # Радіус Землі в метрах
        phi1, phi2 = math.radians(lat1), math.radians(lat2)
        dphi = math.radians(lat2 - lat1)
        dlam = math.radians(lon2 - lon1)
        a = (math.sin(dphi / 2) ** 2 + math.cos(phi1) * math.cos(phi2) * math.sin(dlam / 2) ** 2)
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
        return int(R * c)
```

Тестування проводилося шляхом порівняння результатів обчислень між реальними координатами складів та магазинів у Києві (наприклад, між «Центральним складом» та «Магазином Позняки»). Отримані значення збігалися з контрольними показниками картографічних сервісів з точністю до 0.1%, що підтвердило придатність функції для формування матриці відстаней.

Крім того, було перевірено логіку роботи солвера Google OR-Tools у функції solve_vrp. Основним тестом було дотримання обмежень вантажопідйомності. Під час тесту було створено сценарій, у якому сумарна вага товарів у замовленнях перевищувала можливості доступного автомобіля. Система успішно відпрацювала алгоритм, автоматично перенісши частину замовлень у категорію «unassigned_order_ids», що підтвердило коректність коду:

```
    routing.AddDimensionWithVehicleCapacity(
        weight_callback_idx,
        0,                                          # без
зазорів
        [v.max_weight_g  for  v  in  vehicles],          #
вантажопідйомність авто
        True,                                       # старт з
нуля
        "Weight",
    )
```

Важливим етапом стала перевірка модуля orders.py, який відповідає за створення замовлень та одночасне списання залишків зі складу. Тестування було спрямоване на підтвердження атомарності операцій. Був змодельований сценарій «подвійної витрати», коли два менеджери одночасно намагаються замовити останню одиницю товару. Завдяки використанню блокування рядків with_for_update(), система успішно пройшла тест:

```
    async with db.begin(): # Початок транзакції
        for item_in in payload.items:
```

```
            # Блокування рядка inventory для запобігання
паралельного доступу
            inv_result = await db.execute(
                select(Inventory)
                .where(Inventory.product_id              ==
item_in.product_id)
                .with_for_update()
            )
            inv = inv_result.scalars().first()
            if inv.quantity < item_in.quantity:
                raise      HTTPException(status_code=400,
detail="Insufficient stock")
            inv.quantity  -=  item_in.quantity  #  Атомарне
віднімання
```

У разі спроби замовити більше товару, ніж є на залишку в таблиці inventory, сервер повертав помилку 400 Bad Request, а вся транзакція

Для верифікації роботи з просторовими даними було протестовано модуль routes.py, який взаємодіє з розширенням PostGIS. Перевірялася коректність вилучення координат точок доставки через функції ST_X та ST_Y. Тест підтвердив, що система правильно перетворює бінарні дані геометрії PostgreSQL у числові значення для передачі в алгоритм оптимізації:

```
    wh_result = await db.execute(
        select(
            Warehouse.name,
            ST_X(Warehouse.location).label("lon"),
            ST_Y(Warehouse.location).label("lat"),
        ).limit(1)
    )
```

Також було проведено тестування візуалізації маршрутів. Результат роботи солвера конвертується у масив polyline, де першою та останньою точкою

завжди є склад (депо). Тестування фронтенд-компонента MapComponent.vue підтвердило, що отримані координати коректно відображаються на карті OpenStreetMap у вигляді зв'язаних ліній, що забезпечує диспетчеру наочне представлення логістичного плану.

Для перевірки модуля auth.py було реалізовано тести на валідність JWT-токенів та роботу ролей. Було перевірено:

1. Генерацію токена після успішного входу через bcrypt валідацію пароля.
2. Спробу доступу до ендпоїнта /api/routes/optimize користувачем із роллю Warehouseman. Система коректно повернула статус 403 Forbidden, оскільки даний метод захищений декоратором require_role("Manager", "Dispatcher").

Для оцінки продуктивності було проведено тест на 50-ти одночасних замовленнях з 12-ти магазинів Києва, використовуючи seed-дані з файлу init.sql. Запуск алгоритму оптимізації через POST /api/routes/optimize при встановленому ліміті time_limit_seconds=20 показав, що система знаходить рішення, яке мінімізує сумарний кілометраж, за 1.5–2 секунди, що є відмінним показником для веб-додатка реального часу.

Загалом, результати тестування підтвердили, що всі програмні модулі системи «LogiNet» працюють злагоджено. Використання суворих типів даних Pydantic у schemas.py дозволило відсіяти помилкові вхідні дані на рівні API, а транзакційна модель SQLAlchemy забезпечила надійність складського обліку. Система готова до експлуатації та витримує навантаження, типове для логістичної мережі середнього масштабу.

4.2 Вимоги до апаратного та програмного забезпечення

Ефективність експлуатації автоматизованої системи управління логістикою «LogiNet» безпосередньо залежить від відповідності технічного середовища встановленим вимогам. Враховуючи обраний стек технологій, що

базується на асинхронній архітектурі FastAPI та ресурсомістких обчисленнях Google OR-Tools, формування вимог було розділено на два ключові блоки: програмне забезпечення для розгортання та апаратні потужності, необхідні для стабільної роботи серверної та клієнтської частин. Оскільки система спроектована за принципом контейнеризації, основною вимогою до середовища є підтримка засобів віртуалізації, що дозволяє уніфікувати роботу бази даних, бекенд-сервера та фронтенд-додатка незалежно від особливостей хостової операційної системи.

Для успішного розгортання та функціонування серверного модуля системи «LogiNet» на цільовому обладнанні має бути встановлено наступне базове програмне забезпечення:

- Операційна система: рекомендується використання сімейства Linux (Ubuntu 22.04 LTS або новіше) для забезпечення максимальної продуктивності мережевого стеку та стабільності роботи Docker-контейнерів. Допускається використання Windows 10/11 за умови встановлення підсистеми WSL2 (Windows Subsystem for Linux).
- Середовище віртуалізації: встановлений Docker Desktop або Docker Engine версії 25.x.x або новіше, а також інструмент оркестрації Docker Compose.
- Інтерпретатор мови: Python версії 3.11-slim (використовується всередині контейнера).
- Система управління базами даних: образ PostgreSQL версії 15 із попередньо встановленим просторовим розширенням PostGIS версії 3.3.

До складу специфічних програмних залежностей, які автоматично встановлюються через файл requirements.txt, входять бібліотеки для роботи з асинхронним API (fastapi, uvicorn), інструменти комбінаторної оптимізації (ortools), а також драйвери для роботи з геоданими (geoalchemy2, asyncpg).

Апаратні потужності повинні забезпечувати паралельну роботу трьох активних контейнерів: бази даних, API-сервера та веб-сервера фронтенду.

Практичний аналіз споживання ресурсів під час роботи алгоритму solve_vrp дозволяє визначити наступні мінімальні та рекомендовані характеристики:

- Процесор (CPU): мінімум 2 фізичні ядра (або 4 логічні потоки) з тактовою частотою від 2.4 GHz. Оскільки Google OR-Tools виконує інтенсивні математичні обчислення при пошуку оптимального маршруту, вища частота процесора прямо впливає на швидкість генерації рейсів.
- Оперативна пам'ять (RAM): мінімум 4 GB. Основна частка пам’яті (близько 1.5–2 GB) резервується під потреби PostgreSQL/PostGIS для кешування просторових індексів GiST та виконання складних JOIN-запитів. Рекомендований обсяг - 8 GB для безперебійної роботи в умовах багатокористувацького доступу.
- Дисковий простір (Storage): мінімум 10 GB вільного місця на SSD-накопичувачі. Швидкість дискової підсистеми є критичною для бази даних, особливо при інтенсивному записі логів та транзакцій у таблиці orders та route_points.

Оскільки клієнтська частина реалізована як Single Page Application (SPA), основне навантаження припадає на веб-браузер користувача.

- Програмне забезпечення: актуальна версія браузера з підтримкою стандартів ES6+ та WebGL (Google Chrome, Mozilla Firefox, Microsoft Edge). Це необхідно для коректної візуалізації інтерактивних карт Leaflet.js та плавного відтворення UI-компонентів на базі Vue 3.
- Апаратне забезпечення: процесор рівня Intel Core i3 (або аналог) та 4 GB оперативної пам’яті. Наявність стабільного інтернет-з'єднання зі швидкістю не менше 5 Мбіт/с є обов'язковою для завантаження картографічних тайлів OpenStreetMap у реальному часі.

Для коректної взаємодії компонентів системи всередині Docker-мережі та зовнішнього доступу необхідно забезпечити відкритість наступних портів:

- Порт 8000: для доступу до REST API та документації Swagger.
- Порт 5173 (або 80 у продуктиві): для доступу до інтерфейсу користувача.

- Порт 5432: для внутрішньої взаємодії бекенду з базою даних (у продуктивному середовищі цей порт має бути закритий для зовнішніх з'єднань з міркувань безпеки).

Усі секретні ключі, такі як SECRET_KEY для JWT-авторизації та параметри підключення DATABASE_URL, повинні бути винесені в змінні середовища або файл .env, що дозволяє легко адаптувати програмний продукт до різних конфігурацій серверного обладнання без зміни вихідного коду. Такий системний підхід до визначення апаратно-програмних вимог гарантує, що автоматизована система «LogiNet» працюватиме стабільно, забезпечуючи високу точність логістичних розрахунків та надійний захист даних підприємства.

4.3 Склад інсталяційного пакету

Формування цілісного інсталяційного пакету є завершальним етапом розробки, що гарантує успішне розгортання автоматизованої системи управління логістикою «LogiNet» у довільному робочому середовищі. Зі свого досвіду зазначу, що якість підготовки дистрибутива прямо корелює зі швидкістю впровадження продукту: чим чіткіше структуровані файли та автоматизовані процеси налаштування, тим менша ймовірність виникнення помилок конфігурації на боці замовника. Для системи «LogiNet» мною було обрано модель контейнеризованого пакету, де всі компоненти - від бази даних до фронтенд-інтерфейсу - постачаються як єдиний оркестрований комплекс. Такий підхід дозволяє уникнути класичних проблем несумісності версій системних бібліотек та забезпечує ідентичність оточення розробки та промислової експлуатації.

Інсталяційний пакет системи «LogiNet» являє собою структурований архівний каталог prog/, до складу якого входять модулі інфраструктурної конфігурації, вихідний код серверної та клієнтської частин, скрипти

ініціалізації бази даних та вичерпна технічна документація. Нижче наведено детальний аналіз кожного компонента, що входить до складу дистрибутива.

Фундаментом інсталяційного пакету є файли конфігурації контейнеризації, які забезпечують автоматичне розгортання всього технологічного стеку однією командою.

- docker-compose.yml: Головний файл оркестрації, який описує взаємодію трьох основних сервісів системи: db (PostgreSQL/PostGIS), backend (FastAPI) та frontend (Vue.js). У цьому файлі жорстко задано параметри мережевої ізоляції, порти доступу (8000 для API та 5173 для веб-інтерфейсу) та змінні середовища, необхідні для підключення до бази даних. Важливою особливістю є налаштування healthcheck для сервісу бази даних, що гарантує запуск бекенд-сервера лише після повної готовності СУБД приймати з'єднання.
- backend/Dockerfile: Сценарій побудови образу серверної частини. Він базується на легкому образі python:3.11-slim і містить інструкції щодо встановлення системних залежностей (libpq-dev, gcc), необхідних для коректної роботи драйверів asyncpg та бібліотек лінійної оптимізації.
- .gitignore: Технічний файл, що визначає перелік об’єктів, які не повинні потрапляти до системи контролю версій та фінального дистрибутива, зокрема віртуальні середовища Python (venv/), тимчасові файли Node.js (node_modules/) та локальні змінні середовища .env, що містять конфіденційну інформацію.

Для забезпечення миттєвого старту системи у складі пакету передбачено механізм автоматичного розгортання схеми даних та наповнення її тестовими показниками.

- init.sql: Скрипт ініціалізації, що виконується автоматично при першому запуску контейнера бази даних. Він містить команди створення розширення postgis, опис структури всіх 11 таблиць системи (згідно з нашою логічною моделлю) та блок INSERT операцій для наповнення довідників. У скрипті закладено реальні географічні координати 12 магазинів у Києві, 2 складів (Центральний та Західний) та параметри 5 транспортних засобів, що дозволяє протестувати алгоритм маршрутизації одразу після інсталяції.

• Том postgres_data/: У структурі пакету передбачено монтування локальної папки для збереження даних бази даних, що гарантує збереження всієї логістичної інформації при перезавантаженні контейнерів.

Вихідний код бекенд-сервера структурований за принципом розділення відповідальності (Separation of Concerns) і міститься в директорії backend/app/.

• main.py: Точка входу в FastAPI додаток. Тут реалізовано конфігурацію CORS-політик для безпечної взаємодії з фронтендом, реєстрацію всіх функціональних роутерів та налаштування автоматичної генерації документації Swagger UI за адресою /api/docs.

• models.py: Опис об'єктно-реляційної моделі (SQLAlchemy 2.0). Файл містить класи для всіх сутностей - від User та Order до просторових моделей Warehouse та Store, де використовуються типи Geometry для роботи з координатами.

• schemas.py: Набір Pydantic-схем для валідації вхідних та вихідних JSON-даних. Тут описано структури для створення замовлень, передачі розрахованих маршрутів та авторизації користувачів.

• optimizer.py: Програмна реалізація математичного ядра. Містить логіку обчислення відстаней за формулою Гаверсинуса та виклики бібліотеки Google OR-Tools для розв'язання задачі CVRP.

• database.py: Конфігурація асинхронного двигуна SQLAlchemy та фабрика сесій для підключення до PostgreSQL через драйвер asyncpg.

• auth.py: Модуль безпеки, що реалізує функції створення та перевірки JWT-токенів, а також декоратори для перевірки ролей користувачів.

• Директорія routers/: Містить окремі файли для обробки логіки конкретних сутностей: orders.py (транзакційне створення замовлень із резервуванням товару), routes.py (ендпоїнти оптимізації та маніпуляції маршрутами), inventory.py (контроль залишків та прогноз попиту), а також роутери для складів, магазинів та транспорту.

• requirements.txt: Вичерпний перелік Python-бібліотек із фіксованими версіями, що забезпечує стабільність збірки образу. Сюди входять fastapi,

uvicorn, sqlalchemy, geoalchemy2, ortools, passlib та інші інструменти, необхідні для функціонування системи.

Невід'ємною частиною інсталяційного пакету є документація, яка дозволяє технічному персоналу замовника розгорнути систему без залучення розробника.

- README.md: Швидкий посібник користувача, що містить опис технологічного стеку, таблицю доступних API-ендпоїнтів та облікові дані для тестового входу (наприклад, manager@logistics.ua з паролем password123).
- LogiNet_Run_Instructions.pdf: Детальна візуальна інструкція для запуску проекту в середовищі Windows за допомогою Docker Desktop. Містить покрокові скріншоти, команди терміналу та методи усунення поширених проблем (наприклад, конфлікти портів 5432 або 8000).
- Інструкція_з_розгортання.pdf: Документ, що описує два альтернативні способи запуску: через Docker та ручний запуск (без контейнеризації), що вимагає самостійного встановлення PostgreSQL та налаштування віртуального середовища Python. Також містить перелік кроків для перевірки працездатності після інсталяції.

Завершуючи опис складу інсталяційного пакету, можна стверджувати, що підготовлений дистрибутив «LogiNet» є повністю автономним та готовим до промислового використання. Застосування технології Docker Compose дозволяє автоматизувати процес створення бази даних PostGIS, ініціалізацію схеми та запуск прикладного програмного забезпечення протягом декількох хвилин. Такий склад пакету гарантує високу надійність розгортання, легкість оновлення системи шляхом перезбірки образів та мінімальні вимоги до кваліфікації персоналу, що здійснює підтримку інфраструктури. Наявність детальних інструкцій та seed-даних у скрипті init.sql робить процес ознайомлення з функціоналом системи максимально зручним та наочним.

ВИСНОВКИ

Підсумовуючи результати виконання даної кваліфікаційної роботи, можна з упевненістю стверджувати, що поставлена мета - створення комплексної, автоматизованої системи управління логістикою роздрібної мережі - була повністю досягнута. Розроблений програмний продукт «LogiNet» є не просто академічним концептом, а повноцінним, готовим до впровадження MVP, архітектура якого відповідає найсуворішим індустріальним стандартам розробки. Процес створення системи охопив усі етапи життєвого циклу програмного забезпечення: від глибокого системного аналізу предметної області та математичного моделювання до безпосереднього кодування, налаштування контейнерної інфраструктури та комплексного тестування.

У ході виконання першого етапу роботи було доведено, що традиційні методи ручного планування маршрутів є неефективними в умовах масштабування торговельних мереж. Залежність від людського фактора, неможливість швидкого перерахунку рейсів при зміні вхідних даних та відсутність синхронізації між складом і транспортним відділом призводять до колосальних фінансових втрат. Використання методів об'єктно-орієнтованого аналізу та побудова UML-моделей дозволили чітко декомпозивати ці проблеми. Було формалізовано рольову модель системи (Менеджер, Диспетчер, Комірник) та визначено межі відповідальності кожного актора.

Теоретичною основою для розв'язання задачі розвезення товарів стала класична задача маршрутизації транспортних засобів. Було науково обґрунтовано необхідність використання її розширеної версії - Capacitated VRP, оскільки в реальних умовах обмеженням виступає не лише кількість точок, але й фізичні параметри транспорту. Більше того, у роботі реалізовано багатовимірну модель місткості, де обмеження накладаються одночасно за двома векторами: максимальною вагою (у кілограмах) та максимальним об'ємом (у кубічних метрах).

Найбільшим досягненням у частині інформаційного забезпечення стала відмова від плоскої реляційної моделі на користь геопросторової бази даних. Вибір PostgreSQL у зв'язці з розширенням PostGIS дозволив перенести значну частину важких математичних обчислень на рівень ядра СУБД. Зберігання координат об'єктів (магазинів та складів) у спеціалізованому типі GEOMETRY(Point, 4326) за стандартом WGS-84 дало змогу використовувати просторові індекси (GiST) та виконувати вибірку координат з мінімальними затримками за допомогою функцій ST_X та ST_Y.

Розроблена реляційна модель даних була приведена до Третьої нормальної форми (3NF), що гарантує відсутність дублювання інформації. Використання сурогатних ключів, суворих обмежень зовнішніх ключів (ON DELETE RESTRICT / CASCADE) та перевірочних обмежень (CHECK quantity >= 0) створило надійний фундамент, який фізично унеможливлює виникнення логічних аномалій у базі. Наприклад, неможливо призначити транспортний засіб із від'ємною вантажопідйомністю або створити маршрут для неіснуючого замовлення.

Серцем розробленої системи є модуль оптимізації optimizer.py. Замість використання спрощених жадібних алгоритмів, які часто потрапляють у локальні оптимуми, було інтегровано потужний C++ рушій Google OR-Tools через його Python-обгортку.

Програмно процес маршрутизації був розбитий на наступні етапи:

1. Побудова матриці відстаней: Реалізовано власну математичну функцію _haversine_m, яка вираховує відстань по великому колу сфери (Землі) у метрах. Це усунуло необхідність використання платних API (на кшталт Google Maps Distance Matrix) для побудови базового графа.

2. Ініціалізація моделі: Граф маршрутів ініціалізується з єдиним депо (складом) та множиною вузлів (замовлень).

3. Накладання обмежень: За допомогою методів AddDimensionWithVehicleCapacity алгоритму передаються масиви ваг та об'ємів, які жорстко контролюють заповненість кузова автомобіля.

4. Евристичний пошук: Для генерації початкового рішення використовується стратегія PATH_CHEAPEST_ARC, після чого застосовується метаевристика GUIDED_LOCAL_SEARCH з часовим лімітом у 20 секунд. Це гарантує знаходження рішення, максимально наближеного до глобального оптимуму, за час, прийнятний для веб-додатка реального часу.

Серверна частина системи написана на Python 3.11 з використанням фреймворку FastAPI. Вибір асинхронної парадигми виправдав себе на 100%. Використання ASGI-сервера у комбінації з асинхронним драйвером бази даних asyncpg забезпечило неблокуючу обробку I/O операцій. Поки бекенд очікує відповіді від бази даних або поки у фоні працює алгоритм маршрутизації, сервер здатен паралельно обробляти десятки нових вхідних запитів від інших користувачів.

Окремим практичним здобутком є вирішення проблеми конкурентного доступу (Race Conditions) при формуванні замовлень. У модулі orders.py реалізовано строгу транзакційну логіку: при створенні замовлення система використовує блокування SELECT ... FOR UPDATE для рядків таблиці inventory. Це гарантує атомарне списання товарів та робить систему стійкою до паралельних запитів: якщо товару недостатньо, транзакція безпечно відкочується.

Безпека системи реалізована на найвищому рівні. Всі ендпоїнти (окрім авторизації) захищені протоколом OAuth2 із використанням JSON Web Tokens (JWT). Паролі користувачів не зберігаються у відкритому вигляді, а криптографічно хешуються алгоритмом BCrypt. Розроблено власну фабрику залежностей require_role, яка забезпечує Role-Based Access Control (RBAC), блокуючи несанкціонований доступ до певних модулів.

Попри те, що основний акцент робився на бекенд та алгоритми, реалізована клієнтська частина (на базі Vue.js) є повноцінним інструментом диспетчера. Впровадження бібліотеки Leaflet.js дозволило інтегрувати безкоштовні картографічні підкладки OpenStreetMap. Система автоматично конвертує розраховані масиви координат у візуальні полілінії та маркери,

розраховуючи орієнтовний час прибуття для кожної точки доставки на основі середньої швидкості руху.

Система готова до миттєвого розгортання в будь-якому середовищі завдяки контейнеризації. Написані файли Dockerfile та docker-compose.yml інкапсулюють усе середовище виконання: від системних бібліотек компілятора gcc до налаштувань бази даних. Скрипт init.sql автоматично наповнює базу "seed"-даними (склади, автомобілі, користувачі, 12 магазинів у Києві та їхні замовлення), що перетворює продукт із "сухого" коду на працюючу демонстрацію бізнес-рішення.

Впровадження розробленої системи «LogiNet» на реальному підприємстві дозволить досягти наступних вимірюваних показників ефективності:

1. Скорочення логістичних витрат: Математична оптимізація маршрутів дозволяє зменшити загальний щоденний пробіг автопарку на 15–25%, що прямо конвертується в економію паливно-мастильних матеріалів та амортизації.

2. Максимізація завантаженості транспорту: Алгоритм OR-Tools гарантує, що кубатура та вантажопідйомність автомобілів використовуються максимально ефективно, мінімізуючи випадки "перевезення повітря".

3. Усунення пересортиці та відмов: Завдяки транзакційному резервуванню в базі даних, виключена ситуація, коли менеджер замовляє товар, якого фізично вже немає на складі.

4. Мінімізація часу диспетчеризації: Процес формування маршрутних листів, який раніше міг займати години ручної праці, тепер виконується сервером за кілька секунд.

Перспективи подальшого розвитку

Розроблена архітектура є високомасштабованою та відкритою для інтеграції нових модулів. Наступними логічними кроками розвитку системи можуть стати:

- Інтеграція із зовнішніми GPS-трекерами на автомобілях для порівняння планового та фактичного маршрутів (шляхом оновлення координат у таблиці vehicles).
- Впровадження API комерційних провайдерів (наприклад, OSRM або Google Maps) для заміни формули Гаверсинуса на розрахунок відстаней по реальному дорожньому графу з урахуванням заторів.
- Розробка окремого мобільного PWA-додатка для водіїв, де вони зможуть відмічати статус прибуття у точку та прикріплювати фотографії електронних товарно-транспортних накладних (ТТН).
- Використання методів машинного навчання на основі історичних даних з таблиці orders для точнішого прогнозування попиту на конкретні товари у певні дні тижня, розширюючи існуючий модуль /api/inventory/forecast.

Отже, кваліфікаційна робота являє собою завершене, інноваційне інженерне дослідження, результатом якого є потужний та сучасний програмний продукт, здатний повністю автоматизувати та оптимізувати процеси управління логістикою у сфері роздрібної торгівлі. Робота виконана на високому технічному рівні з дотриманням найкращих світових практик інженерії програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alpaydin E. Introduction to Machine Learning. 4th ed. MIT Press, 2020. 712 p. URL: https://mitpress.mit.edu/9780262043793/introduction-to-machine-learning-fourth-edition/

2. Braekers K., Ramaekers K., Van Nieuwenhuyse I. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 2016. Vol. 99. P. 300–313. DOI: https://doi.org/10.1016/j.cie.2015.12.007

3. Dantzig G. B., Ramser J. H. The Truck Dispatching Problem. *Management Science*, 1959. Vol. 6(1). P. 80–91. DOI: https://doi.org/10.1287/mnsc.6.1.80

4. Date C. J. An Introduction to Database Systems. 8th ed. Pearson, 2003. 1040 p. URL: https://www.pearson.com/en-us/subject-catalog/p/introduction-to-database-systems-an/P200000003328/9780321197849

5. Docker Official Documentation. URL: https://docs.docker.com/

6. Eksioglu B., Vural A. V., Reisman A. The vehicle routing problem: A taxonomic review. *Computers & Industrial Engineering*, 2009. Vol. 57(4). P. 1472–1483. DOI: https://doi.org/10.1016/j.cie.2009.05.009

7. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003. 560 p. URL: https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/

8. FastAPI Official Documentation. URL: https://fastapi.tiangolo.com/

9. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p. URL: https://martinfowler.com/books/eaa.html

10. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 395 p. URL: https://www.oreilly.com/library/view/design-patterns-elements/0201633612/

11. GeoAlchemy 2 Documentation. URL: https://geoalchemy-2.readthedocs.io/

12. Google Maps Platform Documentation. URL: https://developers.google.com/maps/documentation

13. Google OR-Tools Official Documentation. URL: https://developers.google.com/optimization

14. JSON Web Token (JWT) Standard (RFC 7519). URL: https://datatracker.ietf.org/doc/html/rfc7519

15. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 616 p. URL: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

16. Laporte G. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 1992. Vol. 59(3). P. 345–358. DOI: https://doi.org/10.1016/0377-2217(92)90192-C

17. Leaflet.js Official Documentation. URL: https://leafletjs.com/reference.html

18. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p. URL: https://www.oreilly.com/library/view/clean-architecture-a/9780134494272/

19. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. 464 p. URL: https://www.oreilly.com/library/view/clean-code-a/9780136083238/

20. MDN Web Docs: HTTP CORS. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

21. Newman S. Building Microservices. O'Reilly Media, 2015. 280 p. URL: https://www.oreilly.com/library/view/building-microservices/9781491950340/

22. Obe R. O., Hsu L. S. PostGIS in Action. 3rd ed. Manning Publications, 2021. 600 p. URL: https://www.manning.com/books/postgis-in-action-third-edition

23. OpenAPI Specification. URL: https://swagger.io/specification/

24. OpenStreetMap Wiki Documentation. URL: https://wiki.openstreetmap.org/

25. PEP 484 – Type Hints. URL: https://peps.python.org/pep-0484/

26. PEP 8 – Style Guide for Python Code. URL: https://peps.python.org/pep-0008/

27. PostGIS Official Documentation. URL: https://postgis.net/documentation/

28. PostgreSQL Official Documentation. URL: https://www.postgresql.org/docs/

29. Pydantic Official Documentation. URL: https://docs.pydantic.dev/latest/

30. Python 3 Official Documentation. URL: https://docs.python.org/3/

31. SQLAlchemy 2.0 Documentation. URL: https://docs.sqlalchemy.org/en/20/

32. Sommerville I. Software Engineering. 10th ed. Pearson, 2015. 816 p. URL: https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003238/9780133943030

33. Tailwind CSS Official Documentation. URL: https://tailwindcss.com/docs

34. Toth P., Vigo D. Vehicle Routing: Problems, Methods, and Applications. 2nd ed. SIAM, 2014. 463 p. DOI: https://doi.org/10.1137/1.9781611973594

35. Vite Official Documentation. URL: https://vitejs.dev/guide/

36. Vue.js Official Documentation. URL: https://vuejs.org/guide/

37. Багорка М., Якубенко Ю.. Напрями підвищення ефективності складської логістики. *Сталий розвиток економіки, (1(46), 9-14*. DOI: https://doi.org/10.32782/2308-1988/2023-46-1

38. Борисюк Н. І. Геоінформаційні системи у сфері логістики та маршрутизації транспорту. Матеріали Міжнародної науково-технічної конференції молодих учених та студентів. Тернопіль : ТНТУ, 2017. С. 22–23. URL: https://elartu.tntu.edu.ua/bitstream/lib/23153/2/CAZST_2017v3_Borisyuk_N_I-Geographical_information_22.pdf

39. Савчик А. М., Цонь О. П. Необхідність проведення маршрутизації вантажних перевезень. Матеріали Міжнародної науково-технічної конференції

молодих учених та студентів. Тернопіль : ТНТУ, 2015. С. 222–223. URL: https://elartu.tntu.edu.ua/bitstream/123456789/10999/2/ConfATMT_2015v1_Savchyk_A_M-Necessity_realization_of_222.pdf

40. Смагло, О. Функціонування системи фінансового моніторингу в Україні. Економіка та суспільство, (26), 2021. DOI: https://doi.org/10.32782/2524-0072/2021-26-28

ДОДАТКИ

ДОДАТОК А

Структура геопросторової бази даних (фрагмент DDL-скрипта)

У даному додатку наведено програмний код ініціалізації бази даних PostgreSQL з активацією розширення PostGIS та створенням ключових таблиць, що забезпечують збереження просторових координат та транзакційний складський облік.

```
-- Активація просторового розширення для роботи з геоданими (SRID 4326)
CREATE EXTENSION IF NOT EXISTS postgis;

-- Таблиця роздрібних магазинів з геометрією
CREATE TABLE IF NOT EXISTS stores (
    id        SERIAL PRIMARY KEY,
    user_id    INTEGER       NOT NULL REFERENCES users (id) ON DELETE RESTRICT,
    name      VARCHAR(150) NOT NULL,
    address   VARCHAR(255) NOT NULL,
    location  GEOMETRY(Point, 4326) NOT NULL
);
-- Створення просторового індексу GiST для швидкого пошуку координат
CREATE INDEX IF NOT EXISTS idx_stores_location ON stores USING GIST (location);

-- Таблиця складських залишків із захистом від від'ємних значень
CREATE TABLE IF NOT EXISTS inventory (
```

```
        id           SERIAL PRIMARY KEY,
        warehouse_id   INTEGER   NOT   NULL   REFERENCES
warehouses (id) ON DELETE CASCADE,
        product_id    INTEGER NOT NULL REFERENCES products
(id) ON DELETE CASCADE,
        quantity     INTEGER NOT NULL DEFAULT 0,
        CONSTRAINT  uq_inventory_warehouse_product  UNIQUE
(warehouse_id, product_id),
        CONSTRAINT  ck_inventory_quantity_positive  CHECK
(quantity >= 0)
    );

    -- Таблиця сформованих логістичних маршрутів
    CREATE TABLE IF NOT EXISTS routes (
        id                SERIAL PRIMARY KEY,
        vehicle_id            INTEGER  NOT  NULL  REFERENCES
vehicles (id) ON DELETE RESTRICT,
        route_date             DATE      NOT  NULL  DEFAULT
CURRENT_DATE,
        total_distance_km NUMERIC(10,3) NOT NULL,
        status              VARCHAR(50)   NOT NULL DEFAULT
'Planned'
    );

    --    Таблиця    точок    маршруту    (послідовність
відвідування)
    CREATE TABLE IF NOT EXISTS route_points (
        id                SERIAL PRIMARY KEY,
        route_id             INTEGER   NOT NULL REFERENCES
routes (id) ON DELETE CASCADE,
```

```
        order_id             INTEGER   NOT NULL REFERENCES
orders (id) ON DELETE RESTRICT,
        stop_sequence     SMALLINT NOT NULL,
        estimated_arrival TIMESTAMPTZ,
        CONSTRAINT     uq_routepoint_route_seq     UNIQUE
(route_id, stop_sequence),
        CONSTRAINT    ck_routepoint_seq_positive    CHECK
(stop_sequence >= 1)
    );
```

ДОДАТОК Б

Специфікація програмного інтерфейсу REST API (фрагмент маршрутизатора FastAPI)

У додатку наведено приклад реалізації асинхронних ендпоїнтів для обробки транзакцій створення замовлень та ініціалізації графового алгоритму оптимізації маршрутів доставки.

```
Python
"""
Фрагмент модуля routes.py та orders.py
"""
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.ext.asyncio import AsyncSession
from app.database import get_db
from app.auth import require_role
from app.models import Order, Inventory
from app.optimizer import solve_vrp
from app.schemas import OrderIn, OrderOut, OptimizeResponse

router = APIRouter()

@router.post("/orders", response_model=OrderOut, status_code=status.HTTP_201_CREATED)
async def create_order(
    payload: OrderIn,
    db: AsyncSession = Depends(get_db),
```

```
    _ = Depends(require_role("Manager", "Dispatcher")),
):
    """
    Транзакційне створення замовлення з атомарним списанням залишків.
    Використовує блокування рядків (FOR UPDATE) для уникнення Race Conditions.
    """
    async with db.begin():
        order = Order(store_id=payload.store_id, status="New")
        db.add(order)
        await db.flush()

        for item_in in payload.items:
            # Блокування запису на складі
            inv_result = await db.execute(
                select(Inventory)
                .where(Inventory.product_id == item_in.product_id)
                .with_for_update()
            )
            inv = inv_result.scalars().first()

            # Перевірка наявності та автоматичний відкат (ROLLBACK) при помилці
            if not inv or inv.quantity < item_in.quantity:
```

```
                    raise   HTTPException(status_code=400,
detail="Insufficient stock")

                inv.quantity -= item_in.quantity
                db.add(OrderItem(order_id=order.id,
product_id=item_in.product_id,
quantity=item_in.quantity))

            return order

    @router.post("/routes/optimize",
response_model=OptimizeResponse)
    async def optimize_routes_endpoint(db: AsyncSession =
Depends(get_db)):
        """
        Ендпоїнт запуску задачі CVRP (Capacitated Vehicle
Routing Problem).
        Формує матрицю даних та викликає математичне ядро
OR-Tools.
        """
        # 1. Отримання координат депо та точок доставки
через ST_X та ST_Y
        # 2. Формування масивів StopData та VehicleData

        # Виклик математичного алгоритму
        routes, unassigned = solve_vrp(
            depot_lon=wh.lon,
            depot_lat=wh.lat,
            stops=stops_data,
            vehicles=veh_data
```

```
        )

        # Запис згенерованих рейсів у базу даних
        # ...
        return
OptimizeResponse(routes_created=len(routes),
unassigned_orders=unassigned)
```

ДОДАТОК В

Програмна реалізація математичної моделі VRP

У додатку наведено ключову частину алгоритму optimizer.py, яка відповідає за накладання багатовимірних обмежень місткості (Capacity Constraints) та налаштування метаевристичного пошуку розв'язку.

```
    """
    Фрагмент ядра оптимізації (optimizer.py)
    """
    from ortools.constraint_solver import pywrapcp, routing_enums_pb2

    def solve_vrp(depot_lon: float, depot_lat: float, stops: List[StopData], vehicles: List[VehicleData]):
        n_stops = len(stops)
        n_vehicles = len(vehicles)
        n_nodes = n_stops + 1   # 0 = депо, 1..N = магазини

        # 1. Розрахунок матриці відстаней Гаверсинуса (на сфері)
        distance_matrix = [[0]*n_nodes for _ in range(n_nodes)]
        # ... (заповнення матриці)

        # 2. Ініціалізація Routing Index Manager
        manager = pywrapcp.RoutingIndexManager(n_nodes, n_vehicles, 0)
        routing = pywrapcp.RoutingModel(manager)
```

```
        # 3. Реєстрація транзитного callback (вартість дуги = відстань)
        def distance_callback(from_index, to_index):
            from_node = manager.IndexToNode(from_index)
            to_node = manager.IndexToNode(to_index)
            return distance_matrix[from_node][to_node]

        transit_callback_index = routing.RegisterTransitCallback(distance_callback)

        routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

        # 4. Накладання обмежень на ВАГУ (Weight Constraint)
        def weight_callback(from_index):
            node = manager.IndexToNode(from_index)
            return 0 if node == 0 else stops[node - 1].total_weight_g

        weight_callback_idx = routing.RegisterUnaryTransitCallback(weight_callback)
        routing.AddDimensionWithVehicleCapacity(
            weight_callback_idx,
            0,                                  # без резерву (slack)
            [v.max_weight_g for v in vehicles], # масив вантажопідйомностей
```

```
            True,                                   # початок з нуля
            "Weight",
        )

        # 5. Накладання обмежень на ОБ'ЄМ (Volume Constraint)
        def volume_callback(from_index):
            node = manager.IndexToNode(from_index)
            return 0 if node == 0 else stops[node - 1].total_volume_cm3

        volume_callback_idx = routing.RegisterUnaryTransitCallback(volume_callback)
        routing.AddDimensionWithVehicleCapacity(
            volume_callback_idx,
            0,
            [v.max_volume_cm3 for v in vehicles],
            True,
            "Volume",
        )

        # 6. Налаштування параметрів пошуку (Guided Local Search)
        search_params = pywrapcp.DefaultRoutingSearchParameters()
        search_params.first_solution_strategy = routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
```

```
        search_params.local_search_metaheuristic =
routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_S
EARCH
        search_params.time_limit.seconds = 20

        # 7. Запуск алгоритму оптимізації
        solution =
routing.SolveWithParameters(search_params)

        # ... (парсинг результатів та повернення
маршрутів)
```

ДОДАТОК В

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

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

1. Моя бакалаврська кваліфікаційна робота на тему «Автоматизована система управління логістикою мережі магазинів» 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 (обрати необхідне):

- [ ] інструменти генеративного ШІ не використовувалися.
- [Так] інструменти ШІ ChatGPT, DeepL були використані для:
  - [Так] перекладу іншомовних джерел;
  - [ ] стилістичного редагування та перевірки граматики;
  - [Так] допомоги у написанні/відлагодженні програмного коду;
  - [ ] інше: _____.

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

«___»_____ 2026 р. _____ **Дмитро БАЗЯК**

(підпис)