Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка архітектури та алгоритмів сервісу автоматизованої системи обліку фінансів»

Виконав: студент 2 курсу, групи 1АКІТР-24м спеціальності 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка.
(шифр і назва спеціальності)

Натан МАМАЛИГА
(ПІБ студента)

Керівник: к.т.н., професор кафедри АІІТ
Євген ПАЛАМАРЧУК
(науковий ступінь, вчене звання / посада, ПІБ керівника)

«12» грудня 2025 р.

Опонент: д.т.н., професор кафедри КСУ
Володимир ДУБОВОЙ
(науковий ступінь, вчене звання / посада, ПІБ опонента)

«12» грудня 2025 р.

Допущено до захисту
Завідувач кафедри АІІТ
д.т.н., проф. Олег БІСІКАЛО
(науковий ступінь, вчене звання)

«12» грудня 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій
Освітньо-кваліфікаційний рівень II-ий (магістерський)
Галузь знань – Електроніка, автоматизація та електронні комунікації
Спеціальність – 174 - Автоматизація, комп'ютерно-інтегровані технології та робототехніка
Освітньо-професійна програма – Інтелектуальні комп'ютерні системи

ЗАТВЕРДЖУЮ
Завідувач кафедри АІІТ
д.т.н., проф. О.В. Бісікало
«26» 09 2025 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Мамализі Натану Євгенійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка архітектури та алгоритмів сервісу автоматизованої системи обліку фінансів
Керівник проекту (роботи) к.т.н., професор кафедри АІІТ Паламарчук Є. А.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджена наказом вищого навчального закладу від 24 « вересня »
2025 року № 313

2. Строк подання студентом роботи: до «10» грудня 2025 року

3. Вихідні дані до роботи) серверна і клієнтська частини системи обліку фінансів. Вимоги до техніки: ОС Ubuntu 20.04, Процесор Intel Core i5 або вище. Об'єм оперативної пам'яті 8 ГБ або більше. Жосткий диск 120 GB Sata2 або вище.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розкрити) актуальність системи, порівняльна характеристика існуючих систем, вибір технологій для реалізації системи, розробка програмного забезпечення, економічний розділ, висновки, список використаних джерел, додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) ER-модель схеми бази даних, Схема роботи серверної частини , Схема взаємодії систем, ER-модель схеми рольово-дозвільної моделі, ER-модель схеми технічних таблиць, Моделювання залежності величини внутрішньої дохідності потенційних інвестицій від рівня інфляції в країні.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання отримав
1-3	Євген ПАЛАМАРЧУК, професор каф. АІІТ	25.09.2025	05.12.2025
4	Володимир КОЗЛОВСЬКИЙ, к.е.н, професор каф. ЕП та ВМ	08.11.2025	13.11.2025

7. Дата видачі завдання «25» 09 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ етапу	Назва етапів дипломної роботи	Строк виконання етапів роботи (проекту)	Примітка
1	Вибір, узгодження та затвердження теми МКР	25.09.2025 – 27.09.2025	вик.
2	Огляд сервісів автоматизованої системи обліку фінансів	27.09.2025 – 30.09.2025	вик.
3	Огляд та вибір технологій	30.09.2025 - 02.10.2025	вик
4	Побудова і організація роботи системи управління	02.10.2025 – 25.10.2025	вик
5	Розробка документів на супроводження програмного забезпечення	26.10.2025 – 01.10.2025	вик
6	Тестування програмного забезпечення	01.11.2025 – 07.11.2025	вик
7	Підготовка економічного розділу	09.11.2025 – 13.11.2025	вик
8	Оформлення матеріалів до захисту	14.11.2025 – 17.11.2025	вик
9	Попередній захист	02.12.2025	вик
10	Остаточний захист	15.12.2025	вик

Студент _____ Натан МАМАЛИГА
(підпис) (прізвище та ініціали)

Керівник роботи (проекту) _____ Євген ПАЛАМАРЧУК
(підпис)

УДК 004.
Мамалиг
истеми обліку
– Автоматизац
рофесійна пр
2025. 114 с.
На укр. м
У більш
над надходже
щоденно пере
не лише потр
пов'язані з мо
клієнт-сервер
фінансового
точність і дос
Запрова
інформації, ф
роботу як дл
потреба у ру
за різними па
Така с
оперативно
некоректним
впроваджен

Анотація

УДК 004.4:004.42:004.692

Мамалига Н.Є. Розробка архітектури та алгоритмів сервісу автоматизованої системи обліку фінансів. Магістерська кваліфікаційна робота зі спеціальності 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка, освітньо-професійна програма – Інтелектуальні комп'ютерні системи. Вінниця: ВНТУ, 2025. 109 с.

На укр. мові. Бібліогр.: 58 назв; рис.: 9; табл.: 9.

У більшості організацій із великим обсягом фінансових операцій контроль над надходженнями та витратами покладається на окремих працівників, які щоденно перевіряють платежі, ведуть облік і стежать за витратами. Такий підхід не лише потребує значних трудових ресурсів, а й створює додаткові ризики, пов'язані з можливими неточностями через людський фактор. Натомість сучасні клієнт-серверні технології дозволяють автоматизувати ключові процеси фінансового обліку, зменшуючи навантаження на персонал та підвищуючи точність і достовірність даних.

Запровадження системи, що забезпечує централізований доступ до інформації, формування статистики та онлайн-оплати, дозволяє значно спростити роботу як для працівників, так і для керівництва. Завдяки автоматизації зникає потреба у ручному внесенні даних, фінансові потоки можна швидко аналізувати за різними параметрами, а ризик помилок суттєво зменшується.

Така система підвищує прозорість фінансової діяльності, дає можливість оперативно реагувати на зміни та запобігати проблемам, пов'язаним із некоректним обліком чи затримками у платежах, що в сучасних умовах робить її впровадження особливо актуальним.

Abstract

Mamaliga N.Ye. Development of the Architecture and Algorithms of a Service for an Automated Financial Accounting System. Master's Qualification Thesis in specialty 174 – Automation, Computer-Integrated Technologies and Robotics, Educational and Professional Program – Intelligent Computer Systems. Vinnytsia: VNTU, 2025. 109 p.

In Ukrainian. Bibliography: 58 sources; figures: 9; tables: 9.

In many organizations with a large volume of financial operations, control over revenues and expenditures is assigned to individual employees who conduct daily payment checks, maintain records, and monitor expenses. Such an approach requires significant human resources and involves additional risks associated with potential inaccuracies caused by the human factor. In contrast, modern client-server technologies enable the automation of key financial accounting processes, reducing staff workload while improving the accuracy and reliability of financial data.

Implementing a system that provides centralized access to information, generates statistical reports, and supports online payments significantly simplifies the work of both employees and management. Automation eliminates the need for manual data entry, allows financial flows to be analyzed quickly across various parameters, and substantially reduces the likelihood of errors.

Such a system enhances the transparency of financial operations, enables timely responses to changes, and helps prevent issues related to incorrect accounting or payment delays, making its introduction particularly relevant in today's conditions.

ЗМІСТ

ВСТУП

Актуальність роботи. В умовах стрімкого зростання обсягів фінансових даних, що генеруються різними відомствами та регіональними підрозділами, традиційні методи їхнього збирання та обробки (ручні реєстри, розрізнені таблиці Excel, паперові звіти) дедалі більше виявляють свою неефективність. Наявність централізованої, автоматизованої системи контролю й обліку фінансів дозволяє:

- Підвищити оперативність прийняття рішень. Автоматизоване збирання й узагальнення даних у реальному часі дає змогу керівникам швидко реагувати на зміни бюджетних показників та перерозподіляти ресурси.

- Зменшити ймовірність помилок і шахрайства. Єдина прозора платформа із розмежуванням доступу знижує ризики некоректного або навмисного спотворення даних та підвищує відповідальність виконавців.

- Оптимізувати витрати на адміністрування. Виключення дублювання операцій і мінімізація ручної праці скорочують час і ресурси, необхідні для ведення фінансового обліку та звітності.

- Забезпечити багатовимірний аналіз і прогнозування. Інтеграція дашбордів і гнучких фільтрів дає змогу вивчати показники за регіонами, періодами та категоріями витрат, що сприяє точнішому бюджетному плануванню й оцінці ефективності програм.

Таким чином, розробка автоматизованої системи контролю та обліку фінансів по регіонах є надзвичайно актуальною з огляду на потребу державних та приватних організацій у сучасних IT-інструментах, які забезпечать оперативну, достовірну й безпечну обробку фінансової інформації для прийняття обґрунтованих управлінських рішень.

Метою даної магістерської роботи є створення покращеної автоматизованої системи контролю та обліку фінансів, яка забезпечить централізоване збирання, обробку й аналіз фінансових даних у режимі реального

часу з високим рівнем достовірності та безпеки, а також надасть користувачам інструменти для формування наочних звітів і дашбордів для обґрунтованого прийняття управлінських рішень.

Задачі досліджень магістерської кваліфікаційної роботи:

- Провести аналіз вимог до системи контролю та обліку фінансів по регіонах;
- Спроєктувати архітектуру клієнт-серверної системи з урахуванням масштабованості та безпеки;
- Розробити та реалізувати модулі збирання й зберігання фінансових даних у реляційній СУБД;
- Гарантувати безпеку системи та захист конфіденційних даних

Об’єктом дослідження є процес збору, зберігання, обробки та представлення регіональних фінансових даних в автоматизованій інформаційнійсистемі.

Предметом дослідження є методи та технології проектування й реалізації модулів автоматизованого збирання, обробки, валідації й візуалізації регіональних фінансових даних у клієнт-серверній SPA-системі.

Методи дослідження. У роботі використано методи ідентифікації, збору і обробки даних, які отримуються від користувачів, та використання цих даних для контролю і аналізу регіональної ситуації.

Науково-технічний результат полягає в розробці інтегрованої SPA-системи на базі Vue 3 та Laravel для централізованого контролю й обліку фінансових даних по регіонах у режимі реального часу, забезпечує багатовимірний аналітичний двигун з інтерактивними дашбордами для формування довільних комбінацій звітів, підтримує наживо розширювану багатомовність інтерфейсу, закладає архітектуру з можливістю поступового переходу до мікросервісів через чітке розділення модулів і стандартизований REST-API, а також інтегрує високий рівень безпеки з використанням JWT-

автентифікації, розширеної моделі ролей і прав, детального аудиторного логування, поєднуючи при цьому функціонально-структурний аналіз для оптимізації продуктивності під реальні навантаження. [1]

Практична цінність проекту полягає в тому, що розроблена система дозволяє організаціям оперативно збирати та аналізувати фінансові дані з різних регіонів у єдиному інтерфейсі, істотно скорочуючи час підготовки звітів і знижуючи витрати на ручну обробку інформації. Завдяки модулю автоматичного імпорту та валідації даних виключається ризик помилок і дублювання записів, що підвищує достовірність показників, а інтерактивні дашборди сприяють швидкому прийняттю управлінських рішень. Багатомовність та гнучка система ролей забезпечують адаптивність рішення для різних користувачів і організаційних структур, а закладена архітектура з можливістю поступового переходу до мікросервісів гарантує масштабованість і довготривалу підтримку проекту. Впровадження високого рівня безпеки й детального аудиту підвищує рівень захисту конфіденційної інформації та сприяє виконанню нормативних вимог щодо прозорості фінансових процесів.

Апробація та публікації матеріалів досліджень. Основні результати виконання магістерської кваліфікаційної роботи були опубліковані в матеріалах Всеукраїнської науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи» 6 (Вінниця, ВНТУ, 2025 рр.). [2]

1 ЗАГАЛЬНІ ВІДОМОСТІ

1.1 Об’єкт автоматизації

У сучасних умовах цифрової трансформації економіки обсяг фінансової інформації зростає у геометричній прогресії. Дані формуються не лише центральними відомствами, а й численними регіональними підрозділами, територіальними громадами, підприємствами та організаціями різних рівнів. Традиційні підходи до ведення фінансового обліку - паперові звіти, розрізнені електронні таблиці, локальні бази даних - дедалі частіше демонструють свою низьку ефективність. Вони ускладнюють оперативний аналіз, створюють ризики помилок і шахрайських дій, призводять до дублювання інформації та затримок у прийнятті управлінських рішень.

У цьому контексті впровадження автоматизованої системи контролю та обліку фінансів є надзвичайно актуальним, оскільки вона дозволяє:

- Підвищити швидкість і якість управлінських рішень. Автоматизоване збирання та консолідація даних у режимі реального часу забезпечує керівництво актуальною інформацією. Це дозволяє своєчасно реагувати на відхилення у бюджетних показниках, здійснювати перерозподіл ресурсів та коригувати фінансові стратегії.

- Зменшити кількість помилок і ризик зловживань. Використання єдиної централізованої платформи з чітко визначеними правами доступу мінімізує людський фактор, унеможливлює несанкціоновані зміни у звітності та підвищує прозорість усіх фінансових операцій.

- Оптимізувати витрати на адміністрування. Виключення дублювання операцій, автоматична генерація звітів та зменшення ручної праці знижують часові та фінансові витрати, пов’язані з обліком і контролем.

- Розширити аналітичні можливості. Інтеграція інтерактивних дашбордів, візуалізацій, гнучких фільтрів і систем прогнозування дозволяє отримувати багатовимірну картину стану фінансів. Це відкриває нові можливості для стратегічного планування, оцінки ефективності використання

коштів та прогнозування майбутніх фінансових показників.

- Забезпечити відповідність принципам відкритості та підзвітності. Єдина автоматизована система робить процес розподілу коштів більш прозорим і зрозумілим для всіх зацікавлених сторін. Це, у свою чергу, сприяє підвищенню довіри громадськості, поліпшенню інвестиційного клімату та гармонізації з європейськими практиками публічного управління.

Таким чином, розробка системи автоматизованого обліку та контролю фінансів є не лише питанням технічної модернізації. Це необхідна умова формування ефективної, прозорої й підзвітної фінансової політики на всіх рівнях управління. Її впровадження відповідає сучасним вимогам інформаційного суспільства, сприяє цифровізації управлінських процесів і створює передумови для підвищення конкурентоспроможності як державних, так і приватних структур.

1.2 Порівняльна характеристика систем обліку фінансів

В наш час існує багато систем контролю і обліку фінансів. Вони пропонують різний набір можливостей та функцій. Розглянемо декілька з них, а саме Sap, Odoo та BAS/1C. Більшість з них – це великі enterprise рішення для B2B сегменту.

1.2.1 Sap

Серед сучасних корпоративних інформаційних систем особливе місце займають рішення компанії SAP, зокрема продукти SAP S/4HANA та SAP Business One. Дані програмні комплекси відносяться до класу ERP (Enterprise Resource Planning) і орієнтовані на комплексне управління фінансово-господарською діяльністю підприємств. Архітектурно вони побудовані на основі in-memory технологій (база даних HANA), що забезпечує високу

швидкість обробки великих обсягів фінансової інформації у реальному часі. Системи SAP підтримують багатомовність, мультивалютність, інтеграцію з міжнародними стандартами бухгалтерського обліку (IFRS, GAAP), а також пропонують широкий спектр інструментів бізнес-аналітики та прогнозування.

Разом із тим, аналіз практики впровадження таких систем показує наявність суттєвих обмежень у контексті завдань регіонального фінансового обліку. По-перше, вартість ліцензування та консалтингових послуг є надзвичайно високою, що робить SAP малодоступним для державних установ і організацій середнього рівня. По-друге, тривалість впровадження (від кількох місяців до кількох років) ускладнює адаптацію системи до швидко змінюваних умов фінансового середовища. По-третє, локалізація під українське законодавство та бухгалтерські практики потребує залучення сторонніх інтеграторів, що збільшує витрати та залежність від постачальників.

У порівнянні з SAP, розроблювана автоматизована система обліку та контролю фінансів на базі Laravel 11 [4] та Vue 3 демонструє інші характеристики (табл. 1.1)

Таблиця 1.1 - Порівняння можливостей SAP та розроблюваної системи

	SAP (S/4HANA, Business One)	Розроблювана система
Цільовий сегмент	Великі транснаціональні корпорації, підприємства з багаторівневою структурою управління	Державні та муніципальні організації, середні підприємства з регіональними підрозділами
Вартість володіння	Дуже висока (ліцензії, впровадження, консалтинг)	Контрольована, на базі open-source інструментів
Локалізація	Реалізується через партнерів, висока вартість адаптації	Вбудована підтримка мов, адаптація під місцеві вимоги
Аналітичні можливості	Потужні BI-рішення (SAP Analytics Cloud, HANA)	інтерактивні дашборди у Vue;

Продовження таблиці 1.1

Швидкість впровадження	Від кількох місяців до кількох років	Гнучке впровадження власними силами протягом коротких термінів
Інтеграції	Широкі, але переважно через платні конектори	API-first підхід, підтримка webhook, інтеграція з платіжними системами
Масштабування	Високе, але потребує значних інвестицій	Horizon [5] + Redis, оптимізоване під середній рівень навантажень

Таким чином, можна зробити висновок, що хоча системи SAP відзначаються високою функціональністю, надійністю та зрілістю, їх використання у контексті завдань регіонального фінансового контролю є надмірним за масштабом і фінансово невиправданим. Розроблювана система, навпаки, орієнтована на оперативний збір, обробку та аналіз даних по регіонах, забезпечує нижчу вартість володіння, простоту розгортання і гнучку адаптацію під локальні умови, що робить її більш доцільною у зазначеній предметній області.

1.2.2 Odoo

Odoo є однією з найпоширеніших у світі open-source ERP-систем, яка завдяки модульному принципу побудови охоплює широкий спектр бізнес-процесів: від фінансового та бухгалтерського обліку до CRM, логістики, управління проєктами та електронної комерції. Архітектура Odoo передбачає використання сотень модулів, розроблених як офіційними постачальниками, так і спільнотою, що забезпечує високу гнучкість та масштабованість.

Перевагами Odoo є:

- відкритий код і відсутність жорсткої ліцензійної залежності;

- наявність великої кількості готових модулів для різних сфер діяльності;
- можливість інтеграції з іншими системами через API;
- активна міжнародна спільнота та швидкий розвиток екосистеми.

Разом з тим, практичне використання Odoo у сфері фінансового обліку супроводжується низкою обмежень. По-перше, якість сторонніх модулів є нерівномірною, що створює ризики для стабільності та безпеки системи. По-друге, адаптація до українських стандартів бухгалтерського обліку та податкового законодавства часто потребує суттєвого доопрацювання. По-третє, через універсальність архітектури Odoo система може виявитися надмірно громіздкою для організацій, яким потрібен вузьконаправлений функціонал регіонального фінансового контролю.

У порівнянні з Odoo, розроблювана система на базі Laravel 11 та Vue 3 пропонує більш сфокусований підхід (табл. 2.1).

Таблиця 1.2 - Порівняння можливостей Odoo та розроблюваної системи

	Odoo	Розроблювана система
Модель розповсюдження	Open-source, комерційна підтримка (Odoo Enterprise)	Open-source стек (Laravel, Vue), кастомна розробка
Модульність	Висока: сотні готових модулів для різних сфер	Орієнтація на фінанси та аналітику, розширюваність через власні модулі
Локалізація	Частково доступна, потребує доопрацювань для UA	Вбудована підтримка української мови та специфіки обліку
Аналітичні можливості	Базові фінансові дашборди, сторонні BI-модулі	Vue-компоненти з інтерактивними зрізами по регіонах
Простота впровадження	Висока крива навчання, залежність від налаштувань модулів	Швидке впровадження у визначеній предметній області
Інтеграції	API + велика кількість сторонніх конекторів	API-first підхід, інтеграція з платіжними системами та системами моніторингу

Таким чином, Odoo є гнучкою платформою з великим потенціалом розширення, однак її універсальність у поєднанні з різнорідністю модулів може бути надмірною для завдань вузької спеціалізації. Розроблювана система натомість є сфокусованим рішенням для фінансового обліку та контролю по регіонах. Отже, у контексті завдання, яке полягає у створенні централізованої та прозорої системи управління фінансовими даними на регіональному рівні, власна розробка має перевагу над універсальними open-source ERP-рішеннями, такими як Odoo.

1.2.3 BAS/1C

Програмні продукти лінійки BAS (Business Automation Software), які є спадкоємцями системи «1С:Підприємство», традиційно займають провідні позиції на українському ринку бухгалтерських та фінансових рішень. Їхня популярність пояснюється широким поширенням, наявністю готових конфігурацій для ведення бухгалтерського обліку відповідно до українського законодавства, а також розгалуженою мережею інтеграторів та консультантів.

Разом із тим, аналіз BAS/1C виявляє низку обмежень у контексті побудови сучасних автоматизованих систем контролю та обліку фінансів. По-перше, технологічна основа платформи є застарілою: система переважно працює в архітектурі «клієнт-сервер», має обмежену веб-функціональність та слабку інтеграцію із сучасними DevOps та BI інструментами. По-друге, можливості аналітики обмежені класичними звітами і візуалізаціями, що недостатньо для комплексного багатовимірного аналізу даних у реальному часі. По-третє, масштабованість і продуктивність системи у випадку роботи з великими обсягами даних та розподіленими підрозділами є невисокою.

У цьому контексті розроблювана система на базі Laravel 11, Vue 3 та Docker-інфраструктури має низку переваг (табл. 1.3).

Таблиця 1.3 - Порівняння можливостей 1С та розроблюваної системи

	BAS/1C	Розроблювана система
Відповідність законодавству	Висока: повна підтримка українських стандартів бухгалтерського обліку	Часткова: можливість адаптації під локальні вимоги через конфігурацію
Аналітичні можливості	Класичні звіти, базові дашборди	Vue-дашборди з багатовимірним аналізом
Технологічна основа	Клієнт-серверна архітектура, обмежена веб-функціональність	Сучасний веб-стек (Laravel + Vue), мікросервісна архітектура на Docker
Масштабованість	Обмежена, зниження продуктивності при зростанні обсягів даних	Redis + Horizon, можливість горизонтального масштабування
Інтеграції	Обмежені, здебільшого через сторонніх розробників	API-first, інтеграція з платіжними системами, сервісами моніторингу
Простота впровадження	Висока, завдяки готовим конфігураціям	Гнучке, але потребує початкової розробки і налаштувань
Сприйняття користувачами	Звичне серед бухгалтерів	Сучасний інтерфейс, але потребує навчання персоналу

Системи BAS/1C залишаються ефективним інструментом для класичного бухгалтерського обліку та формування звітності відповідно до українських норм, проте вони є обмеженими у сфері сучасної фінансової аналітики та регіонального контролю. Їхня технологічна архітектура не відповідає вимогам високої масштабованості та інтеграції з сучасними інформаційними екосистемами.

Натомість розроблювана система орієнтована на оперативний аналіз фінансових даних у реальному часі, підтримує дашбордний підхід до управління, забезпечує гнучку кастомізацію та інтеграцію з сучасними сервісами моніторингу і візуалізації. Це робить її більш придатною для вирішення завдань, пов’язаних із контролем фінансів по регіонах, прозорістю використання коштів та побудовою сучасної аналітичної інфраструктури.

1.3 Висновки до розділу

Проведений аналіз існуючих інформаційних систем для автоматизації фінансового обліку та контролю показав, що на ринку представлений широкий спектр рішень - від високофункціональних корпоративних ERP-систем (SAP), до модульних open-source платформ (Odoo) та локалізованих бухгалтерських програм (BAS/1C).

Системи SAP характеризуються високою надійністю, багатофункціональністю та потужними аналітичними можливостями, проте їх впровадження супроводжується значними фінансовими витратами, тривалими термінами налаштування та складністю локалізації під українські умови. Odoo завдяки відкритому коду та гнучкій модульній архітектурі є універсальним інструментом для автоматизації бізнес-процесів, однак нерівномірна якість сторонніх модулів і відсутність готової відповідності національним стандартам обліку знижують її ефективність у специфічних завданнях регіонального фінансового контролю. Своєю чергою, BAS/1C повністю відповідає вимогам українського законодавства і широко використовується для бухгалтерського обліку, але її застаріла технологічна основа та обмежені аналітичні можливості не дозволяють забезпечити гнучку інтеграцію з сучасними IT-інструментами та ефективне масштабування.

Узагальнюючи результати порівняння, можна зробити висновок, що жодне з існуючих рішень не задовольняє у повній мірі потреби у централізованій, прозорій та масштабованій системі контролю фінансів по регіонах. Це обґрунтовує актуальність розробки власної автоматизованої системи на базі сучасних веб-технологій, наприклад Laravel 11, Vue 3, Docker.

2 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ СИСТЕМИ

2.1 Вибір мови програмування для реалізації серверної частини

Під час розробки автоматизованої клієнт–серверної системи фінансового обліку постала задача вибору програмної платформи, здатної забезпечити стабільність, безпеку та гнучкість при реалізації бізнес-логіки і взаємодії з користувачем у реальному часі. Враховуючи необхідність побудови сучасного вебзастосунку з чітким розділенням функцій між клієнтською та серверною частинами, було обрано платформу Laravel як основу серверного середовища та Vue.js як фреймворк для клієнтського інтерфейсу. Фреймворк Laravel вирізняється розвиненою екосистемою, підтримкою архітектури MVC, наявністю вбудованих механізмів автентифікації, черг, кешування та REST API, що суттєво скорочує час розробки й підвищує рівень безпеки. У свою чергу, Vue.js забезпечує реактивність інтерфейсу, компонентну структуру та високу продуктивність при обробці великих обсягів динамічних даних. Для забезпечення відтворюваності середовища, спрощення процесу розгортання і тестування обрано Docker, який дозволяє контейнеризувати всі сервіси системи та підтримувати узгоджене середовище виконання на різних етапах життєвого циклу програмного продукту. Сукупність зазначених технологій створює надійну, масштабовану та гнучку платформу, що повністю відповідає вимогам до сучасних інформаційних систем фінансового спрямування.

2.1.1 Порівняння мов програмування та вибір основної мови

На сучасному етапі розвитку інформаційних технологій для створення веборієнтованих інформаційних систем найчастіше використовуються мови програмування PHP, Python, JavaScript (Node.js), Java, C# та Ruby. Кожна з них має власні переваги та недоліки, які необхідно враховувати під час розробки автоматизованої системи обліку фінансів.

Python характеризується зрозумілим синтаксисом і широкою екосистемою бібліотек, що робить його популярним у сфері аналізу даних, штучного інтелекту та машинного навчання. Однак у сфері веброзробки його продуктивність поступається сучасним версіям PHP, а також вимагає значних ресурсів для масштабування при великому навантаженні.

JavaScript (Node.js) дозволяє використовувати єдину мову на клієнтській і серверній частині, що спрощує взаємодію в команді та пришвидшує розробку. Проте Node.js орієнтований насамперед на системи реального часу (чати, потокова передача даних) і виявляє певні обмеження при реалізації складної бізнес-логіки з великою кількістю транзакційних операцій.

Java традиційно застосовується у великих корпоративних рішеннях, де важливі стабільність і масштабованість. Водночас її використання потребує більше часу на розробку та суттєвих ресурсів, що робить її менш зручною для створення систем середнього масштабу з обмеженими строками реалізації.

C# (.NET Core) демонструє високу продуктивність та глибоку інтеграцію з екосистемою Microsoft, але часто застосовується у вузькому сегменті корпоративних замовлень, що обмежує його універсальність у порівнянні з PHP.

Ruby (Ruby on Rails) має високу швидкість створення прототипів, однак його продуктивність нижча за PHP та Node.js, а популярність у світовій спільноті останніми роками суттєво зменшилася, що ускладнює пошук кваліфікованих розробників.

На цьому тлі PHP зберігає стійкі позиції завдяки своїй спеціалізації у вебсфері, широкій підтримці баз даних, наявності розвиненої інфраструктури фреймворків (Laravel, Symfony), а також активній спільноті. Важливо зазначити, що сучасні версії PHP (починаючи з 7.x і особливо 8.x) отримали суттєві покращення продуктивності, зокрема завдяки впровадженню JIT-компіляції, що дозволяє цій мові конкурувати з JavaScript та Python у швидкодії. Крім того, PHP залишається одним із найпоширеніших інструментів для створення систем класу ERP та CRM, що підтверджує його доцільність у розробці автоматизованої системи обліку фінансів.

Таким чином, у контексті поставленого завдання вибір PHP як основної мови програмування є обґрунтованим завдяки поєднанню високої продуктивності, наявності зрілої інфраструктури, простоти інтеграції з іншими вебтехнологіями та значної кількості готових рішень, які можна адаптувати для потреб фінансового обліку.

З огляду на наведене порівняння мов програмування доцільним є уточнення щодо конкретної версії PHP, обраної для реалізації системи. У межах даної роботи використано PHP 8.2, що зумовлено низкою причин як технічного, так і методологічного характеру.

По-перше, PHP 8.2 є однією з найновіших стабільних версій мови, яка підтримується спільнотою та забезпечує актуальність у середньостроковій перспективі. Використання сучасної версії знижує ризики, пов’язані з безпекою, адже вона містить регулярні оновлення, спрямовані на усунення вразливостей.

По-друге, у PHP 8.2 реалізовано низку суттєвих удосконалень мови, що напряму впливають на якість і продуктивність коду. Зокрема:

- JIT-компіляція (Just-In-Time) забезпечує помітне підвищення швидкодії у складних обчисленнях, що може бути корисним при фінансових розрахунках і багатовимірній аналітиці.
- Розширена типізація (наприклад, підтримка intersection types, readonly-класів) сприяє зменшенню кількості помилок на етапі розробки та робить програмний код більш передбачуваним і підтримуваним.
- Оптимізація роботи з пам’яттю дозволяє ефективніше обробляти великі масиви даних, що є характерним для систем фінансового обліку.

По-третє, вибір PHP 8.2 має й організаційні переваги: дана версія є рекомендованою для більшості сучасних фреймворків, зокрема Laravel, який розглядається як основний каркас для побудови системи. Це забезпечує сумісність, спрощує інтеграцію сторонніх бібліотек та підвищує загальну стабільність проєкту.

Таким чином, застосування PHP 8.2 у проєкті зумовлене прагненням поєднати сучасні можливості мови з потребами фінансової системи в

надійності, продуктивності та довгостроковій підтримці. Це рішення дозволяє створити гнучку й масштабовану платформу, здатну відповідати як поточним, так і перспективним вимогам у сфері автоматизації фінансового обліку.

2.1.2 Порівняння фреймворків та вибір основного фреймворка

Як писалось вище - було обрано фреймворк Laravel, що працює поверх мови програмування PHP. Вибір саме цього інструмента зумовлений як технічними характеристиками, так і практичними перевагами, які він забезпечує в порівнянні з іншими поширеними PHP-фреймворками. Далі буде наведений список та конкурентний аналіз популярних фреймворків.

Symfony є одним із найстаріших та найбільш функціональних фреймворків на PHP. Його часто використовують у великих корпораціях завдяки гнучкій архітектурі та можливості застосування окремих компонентів у сторонніх проєктах. Серед безперечних переваг Symfony можна назвати стабільність, надійність і багатий набір інструментів для складних застосунків. Однак він вимагає значних знань від розробників і має високий поріг входження. У результаті початкові витрати часу на проєктування й налаштування системи зростають, що робить Symfony менш зручним для проєктів, де важливі швидкі результати.

CodeIgniter, навпаки, позиціонується як надзвичайно простий і легкий фреймворк. Його основною перевагою є мінімалістичний підхід, що дозволяє швидко стартувати навіть початківцю. Він не потребує складної конфігурації та забезпечує високу продуктивність на рівні простих застосунків. Проте цей фреймворк практично не розвивається останніми роками, має відносно невелику спільноту і обмежені можливості інтеграції із сучасними бібліотеками чи мікросервісними архітектурами. Використання CodeIgniter у проєкті, орієнтованому на довготривалу перспективу, може призвести до технологічного глухого кута.

Yii2 - ще один популярний фреймворк у світі PHP, що відомий своєю

високою продуктивністю та наявністю зручного генератора коду Gii. Він добре підходить для швидкої розробки прототипів та має досить потужний ORM. Водночас Yii2 менш популярний у комерційних розробках порівняно з Laravel чи Symfony. Його спільнота значно менша, а кількість готових пакетів та інтеграцій обмежена, що створює певні ризики у випадку масштабування системи або необхідності впровадження сучасних підходів (наприклад, мікросервісної архітектури чи інтеграції з DevOps-середовищем).

Таким чином, кожен із зазначених фреймворків має як сильні сторони, так і недоліки. Symfony вирізняється масштабованістю та надійністю, проте занадто складний. CodeIgniter швидкий і простий, але застарілий. Yii2 продуктивний, проте менш популярний і має обмежену екосистему.

На цьому тлі Laravel займає збалансовану позицію, поєднуючи простоту використання з високим рівнем функціональності. Основними перевагами є:

- Зручний синтаксис та низький поріг входження. Laravel надає зрозумілий, «людяний» синтаксис, що робить процес розробки значно простішим та швидшим у порівнянні з Symfony.

- Розвинена екосистема. У Laravel інтегровано власний ORM (Eloquent), систему міграцій, шаблонізатор Blade, засоби тестування, підтримку черг та планувальників завдань. Додатково фреймворк має офіційні пакети для авторизації, управління підписками, побудови API тощо.

- Широка спільнота та документація. Laravel є найпопулярнішим PHP-фреймворком у світі, що забезпечує величезну кількість навчальних матеріалів, пакетів і готових рішень. Це спрощує як навчання нових учасників команди, так і підтримку довготривалих проєктів.

- Гнучкість і сучасність. Laravel постійно оновлюється та враховує сучасні підходи розробки - від підтримки REST API та GraphQL до інтеграції з контейнерами Docker та системами CI/CD.

- Вбудовані інструменти моніторингу. Такі модулі, як Horizon і Telescope, забезпечують контроль за чергами, продуктивністю й відлагодженням коду. Це критично важливо для систем фінансового обліку, де стабільність та прозорість роботи є ключовими.

2.1.3 Сторонні пакети та їх доцільність

Також важливу роль грають сторонні бібліотеки проєкту, які необхідні для пришвидшення розробки та зменшенню кількісті помилок:

- archtechx/enums - допоміжна бібліотека для роботи з переліками (enums): чистіші моделі, явні значення, менше «магічних рядків» у логіці.
- laravel/horizon [5] - інтерфейс та моніторинг для Redis-черг Laravel; дозволяє управляти й спостерігати фонові задачі (retries, throughput, job failures).
- laravel/sanctum [6] - легковагова система аутентифікації для SPA та API (cookie-based / token); зручна для захищених односторінкових інтерфейсів.
- laravel/tinker [37] - інтерактивна REPL-консоль для відлагодження й швидкого тестування логіки додатку (корисно під час розробки й експериментів з моделями).
- promphp/prometheus_client_php [36] - клієнт для експорту метрик у Prometheus; дозволяє збирати технічні метрики (черги, latencies, counts) для подальшої візуалізації в Grafana.
- spatie/laravel-permission [10]- готова реалізація ролей і прав доступу (RBAC); спрощує управління правами, перевірки доступу й аудит ролей у системі.
- spatie/laravel-translatable [11] - інструмент для мультимовності полів у моделях (зберігання перекладів прямо в атрибутах моделі), корисний для локалізації інтерфейсу та даних.
- tymon/jwt-auth [12]- пакет для JWT-аутентифікації; зручний, коли потрібно видавати й перевіряти токени для мобільних клієнтів/зовнішніх інтеграцій.
- barryvdh/laravel-ide-helper [13] - генерує phpdoc/файли для IDE; покращує автодоповнення та навігацію по кодовій базі (швидше розробляти й менше помилок).

- fakerphp/faker [38] - генератор фейкових даних для сідів і тестів (імена, дати, суми тощо), спрощує наповнення тестової БД.
- laravel/pail [39] - утиліта/скриптовий набір для допоміжних dev-операцій (локальний workflow, скрипти збірки/деплою); використовується для стандартизації повторюваних задач під час розробки.
- laravel/pint [8] - офіційний інструмент форматування коду (style/lint) для Laravel-проєктів; забезпечує єдиний стиль коду в команді.
- laravel/sail [9]- офіційне легке Docker-середовище для локальної розробки Laravel.
- mockery/mockery [40] - бібліотека для створення mock-об’єктів у тестах; дозволяє ізолювати юніти й ефективно перевіряти поведінку сервісів.
- nunomaduro/collision [41] - дружнє форматування помилок у консольному режимі (кращі стеки помилок під час запуску тестів/artisan-команд).
- phpunit/phpunit [14] - стандартна тестова платформа для PHP; необхідна для модульного й інтеграційного тестування логіки застосунку.

2.2 Вибір мови програмування для реалізації клієнтської частини

2.2.1 Порівняння мов програмування та вибір основної мови

У процесі розробки клієнтської частини автоматизованої системи обліку фінансів важливим є вибір мови програмування, яка б забезпечувала високу інтерактивність користувацького інтерфейсу, надійну взаємодію з серверною частиною та довгострокову підтримуваність коду. У цьому контексті було обрано комбінацію JavaScript та TypeScript, що дозволяє поєднати гнучкість і універсальність першої мови з надійністю та передбачуваністю другої.

JavaScript є нативною мовою браузера, що визначає його як де-факто стандарт для розробки веб-інтерфейсів. Він забезпечує можливість створення

односторінкових застосунків (SPA), у яких навігація між екранними формами здійснюється без повного перезавантаження сторінки. Це суттєво підвищує зручність і швидкодію роботи користувачів, що є критичним для систем фінансового обліку, де операції часто здійснюються з великими обсягами даних. Додатковою перевагою є широка екосистема бібліотек та інструментів (axios, бібліотеки для роботи з таблицями, компонентні UI-елементи), що дозволяє зменшити час розробки та використовувати перевірені рішення для типових завдань.

Разом із тим, використання лише JavaScript має обмеження, пов’язані з динамічною типізацією, яка підвищує ризик появи помилок у процесі виконання. Для системи, що працює з фінансовими даними, такі ризики є неприйнятними. Саме тому доцільним є застосування TypeScript — надбудови над JavaScript, яка запроваджує статичну типізацію. Це дає змогу здійснювати виявлення значної кількості помилок ще на етапі компіляції, підвищує якість і передбачуваність програмного коду та полегшує його довгострокове супроводження.

Використання TypeScript забезпечує такі переваги:

- формування типобезпечних контрактів для обміну даними з серверною частиною (Data Transfer Objects), що знижує ймовірність некоректної інтерпретації даних;
- можливість здійснення безпечних рефакторингів, що критично важливо для еволюції системи в умовах розширення функціоналу;
- зручність у роботі команди завдяки автодоповненню коду та інструментам статичного аналізу;
- підвищення узгодженості та консистентності даних, особливо у модулях, пов’язаних з обліком фінансових показників, дат, валют чи регіонів.

Крім того, застосування TypeScript має прямий вплив на нефункціональні характеристики системи:

- надійність підвищується завдяки мінімізації помилок у бізнес-логіці;
- безпека посилюється через уніфіковані механізми контролю доступу на рівні інтерфейсу;

- продуктивність користувацького інтерфейсу забезпечується завдяки точному визначенню структур даних і скороченню кількості повторних перевірок у рантаймі.

Таким чином, поєднання JavaScript як універсальної мови веброзробки та TypeScript як інструменту інженерної дисципліни дозволяє досягти балансу між гнучкістю та надійністю. Це рішення є оптимальним для системи, що функціонує у сфері фінансового обліку, оскільки воно забезпечує високу якість коду, знижує ризики помилок при обробці чутливих даних і сприяє масштабованості та довготривалій підтримуваності проєкту.

2.2.2 Порівняння фреймворків та вибір основного фреймворка

Для реалізації клієнтської частини системи було обрано фреймворк Vue.js (версія 3), який поєднує у собі переваги реактивної моделі програмування, компонентно-орієнтованої архітектури та зручності впровадження. Застосування Vue.js дозволяє створювати односторінкові застосунки з високим рівнем інтерактивності, що є особливо важливим для автоматизованих систем фінансового обліку, де користувач працює з великими обсягами даних у вигляді таблиць, графіків і звітів, а швидкість реакції інтерфейсу безпосередньо впливає на ефективність роботи.

Ключовою інновацією у Vue.js 3 є Composition API, що сприяє більшій структурованості бізнес-логіки, зручнішому повторному використанню коду та підвищенню тестованості компонентів. Це дозволяє розробникам створювати модулі, які легко масштабуються, та застосовувати їх у різних частинах системи, забезпечуючи узгодженість інтерфейсу. У випадку фінансових систем така архітектура є надзвичайно важливою, адже користувачі часто одночасно працюють із багатьма реєстрами, формами й фільтрами. Додатково Vue.js характеризується високою продуктивністю, низьким порогом входження та чіткою документацією, що скорочує час на підготовку нових розробників і полегшує підтримку довготривалих проєктів.

Порівняння з найближчими конкурентами демонструє збалансованість вибору. React має значну популярність і величезну екосистему бібліотек, проте для реалізації базових можливостей (керування станом, маршрутизація, інтернаціоналізація) він потребує використання сторонніх інструментів, що ускладнює архітектуру та підвищує витрати часу на інтеграцію. Це робить React менш оптимальним для проєктів середнього масштабу, де важлива швидка адаптація до предметної області.

Angular, навпаки, надає вбудований набір інструментів «з коробки» та дозволяє розробляти масштабні корпоративні системи, однак його громіздка архітектура, складність конфігурації та високий поріг входження роблять його надмірним для завдань, де на першому плані стоїть гнучкість і швидкість впровадження.

У цьому контексті Vue.js 3 займає проміжну позицію, поєднуючи простоту використання з достатньою глибиною для реалізації масштабованих застосунків. Він є легшим для освоєння, ніж Angular, і менш перевантаженим залежностями, ніж React, що дозволяє зберігати чистоту архітектури та знижувати загальну складність підтримки системи. Vue.js також має розвинену екосистему офіційних рішень для керування станом (Pinia), маршрутизації (vue-router), багатомовності (vue-i18n), що забезпечує узгодженість і скорочує потребу у використанні сторонніх бібліотек сумнівної якості.

Таким чином, вибір Vue.js як основного фреймворку для фронтенд-частини автоматизованої системи фінансового обліку є обґрунтованим. Він дозволяє забезпечити високу реактивність і інтерактивність інтерфейсу, ефективну організацію бізнес-логіки, просту інтеграцію з необхідними інструментами та швидку адаптацію під специфічні вимоги. Завдяки своїй збалансованості Vue.js створює оптимальні умови для побудови сучасного інтерфейсу, де продуктивність, зручність користування та надійність функціонування мають вирішальне значення.

2.2.3 Сторонні пакети та їх доцільність

У клієнтській частині системи пропонується застосування низки сторонніх бібліотек, кожна з яких виконує специфічні функції та сприяє підвищенню ефективності розробки:

- @bhplugin/vue3-datatable [24] — використовується для відображення та обробки великих табличних наборів даних із підтримкою сортування, пошуку, фільтрації та пагінації.
- @vueuse/head [23] — забезпечує динамічне керування метаданими сторінок (title, meta-теги), що є необхідним для коректної побудови SPA.
- Moment [42] — застосовується для форматування та маніпуляцій із датами й часом, зокрема у фінансових звітах та аналітичних модулях.
- Pinia [17] — сучасний менеджер стану, який забезпечує централізоване зберігання даних інтерфейсу, їхню узгодженість та типобезпечність у зв’язці з TypeScript.
- sweetalert2 [43] — використовується для реалізації модальних вікон, діалогів підтвердження та системи повідомлень.
- Vue [15] — основний фреймворк для побудови реактивного інтерфейсу, що забезпечує компонентно-орієнтовану архітектуру клієнтської частини.
- vue-axios [21] — інтеграція бібліотеки axios у Vue-застосунок для організації взаємодії з API.
- vue-height-collapsible [41] — забезпечує створення динамічних блоків інтерфейсу з анімованим розгортанням та згортанням.
- vue-i18n [18]— надає інструменти для організації багатомовного інтерфейсу, що особливо актуально для застосунків із підтримкою кількох мов.
- vue-router [16]— реалізує маршрутизацію у SPA та підтримує lazy loading модулів.
- vue3-perfect-scrollbar [45] — використовується для керування прокруткою у складних макетах, забезпечує зручність взаємодії з великими

таблицями та списками.

- vue3-popper [46]— забезпечує створення адаптивних спливаючих підказок і меню.
- @intlify/unplugin-vue-i18n [47] — оптимізує роботу локалізацій, забезпечує статичний імпорт перекладів і зменшення розміру бандла.
- @rollup/plugin-alias [48] — дозволяє організувати зручні шляхи імпорту та уникнути надмірної вкладеності у структурі проєкту.
- @tailwindcss/forms [49] — спрощує стилізацію форм, надаючи готові утиліти для єдиної візуальної мови інтерфейсу.
- @tailwindcss/typography [50] — забезпечує зручне оформлення текстового контенту, що актуально для звітів та аналітичних сторінок.
- @types/node [51] — додає визначення типів для Node.js, що необхідно у поєднанні з TypeScript.
- @vitejs/plugin-vue [52] — інтеграція Vue у збирач Vite для коректної роботи компонентів.
- Autoprefixer [53] — автоматично додає вендорні префікси до CSS для забезпечення кросбраузерної сумісності.
- axios [21] — HTTP-клієнт для виконання запитів до серверної частини системи, використовується у поєднанні з vue-axios.
- Concurrently [54] — дозволяє одночасно запускати кілька процесів у середовищі розробки (наприклад, бекенд та фронтенд).
- laravel-vite-plugin [55] — інтегрує процес збірки Vite із Laravel, що забезпечує цілісність фронтенду та бекенду.
- postcss [56] — інструмент для трансформації CSS, використовується разом із TailwindCSS.
- tailwindcss [21] — утилітарний CSS-фреймворк, який забезпечує швидку розробку адаптивних і уніфікованих інтерфейсів.
- typescript [25] — надбудова над JavaScript, що забезпечує статичну типізацію та зменшує кількість помилок у коді.
- vite [57] — сучасний високопродуктивний збирач, що пришвидшує розробку завдяки HMR та інкрементальній компіляції.

- vue-tsc [58] — статичний аналізатор для Vue-компонентів у поєднанні з TypeScript, що забезпечує додаткову перевірку типів.

Застосування цих бібліотек дозволяє створити стабільний, масштабований та багатофункціональний інтерфейс, який відповідає вимогам предметної області та сучасним стандартам веброзробки.

2.3 Вибір технологій збереження даних

У сучасних інформаційних системах, зокрема тих, що функціонують у сфері фінансового обліку, вибір системи керування базами даних (СКБД) є визначальним чинником, що безпосередньо впливає на продуктивність, надійність та масштабованість усього комплексу. База даних у такій системі виконує роль централізованого сховища, у якому зберігаються фінансові транзакції, відомості про бюджети, дані користувачів, а також численні проміжні результати обчислень. Отже, вимоги до СКБД охоплюють не лише обробку великих обсягів даних, але й забезпечення високої доступності, відмовостійкості та суворої відповідності принципам ACID.

Серед найбільш поширених систем, що застосовуються у веборієнтованих застосунках, слід виокремити PostgreSQL, MariaDB та MySQL. Кожна з цих СКБД має власні архітектурні особливості, історію розвитку та цільові сценарії застосування, що обумовлює необхідність їх порівняльного аналізу.

PostgreSQL традиційно розглядається як найбільш «просунута» з відкритих реляційних СКБД. Вона підтримує розширені можливості роботи з транзакціями, складними запитами та об’єктно-реляційними механізмами. Серед переваг PostgreSQL слід відзначити повну відповідність стандарту SQL:2016, можливість створення власних типів даних, розвинену систему розширень (наприклад, PostGIS для роботи з геопросторовими об’єктами), а також високу гнучкість налаштувань. Це робить PostgreSQL надзвичайно потужним інструментом у випадках, коли необхідна складна бізнес-логіка на

рівні бази даних. Проте використання PostgreSQL потребує більш високої кваліфікації від розробників та адміністраторів, має складніший механізм конфігурації та, як правило, вимагає більших ресурсів для досягнення порівнянної продуктивності у простих сценаріях. У випадку побудови фінансової системи середнього масштабу це може призвести до зростання вартості володіння та ускладнити подальшу підтримку.

MariaDB була створена як форк MySQL після придбання останньої компанією Oracle. Вона успадкувала більшість механізмів архітектури MySQL, проте розвивається у напрямку більшої відкритості та підтримки додаткових рушіїв зберігання (наприклад, Aria, ColumnStore). Серед її переваг — сумісність з MySQL на рівні SQL-діалекту та драйверів, що дозволяє легко мігрувати між цими СКБД. Водночас відмінності у дорожній карті розвитку поступово зменшують рівень синхронізації між MySQL і MariaDB, що може створювати труднощі при використанні сторонніх бібліотек або інструментів, орієнтованих на MySQL як «де-факто стандарт». Для локальних або невеликих проєктів MariaDB є привабливим рішенням, проте в умовах великої інтеграційної екосистеми MySQL має ширшу підтримку та стабільніший розвиток.

MySQL належить до числа найпопулярніших СКБД у світі, що зумовлено її багаторічною історією, відкритістю та підтримкою з боку провідних компаній. У промислових проєктах вона застосовується як основна база даних для систем управління контентом, електронної комерції, корпоративних порталів і фінансових застосунків. MySQL підтримує повний набір механізмів транзакційності (ACID), оптимізовану індексацію, реплікацію, кластеризацію та відмовостійкі конфігурації. Окремо слід відзначити InnoDB як основний рушій зберігання, що забезпечує підтримку зовнішніх ключів, багатовимірної індексації та механізмів блокувань на рівні рядків. Це дозволяє гарантувати цілісність даних навіть у випадках конкурентних доступів, що є принципово важливим у системах фінансового обліку.

Порівняння продуктивності показує, що MySQL демонструє високу

швидкість виконання типових запитів у середовищах із великою кількістю читання та помірною кількістю записів. Саме такий сценарій характерний для більшості фінансових систем, де формування звітів та аналітичних вибірок переважає над створенням нових записів. Крім того, MySQL має надзвичайно широку підтримку у хмарних платформах (AWS RDS, Google Cloud SQL, Azure Database for MySQL), що дозволяє легко масштабувати систему та забезпечувати її відмовостійкість.

З практичної точки зору вибір MySQL у даному проєкті обґрунтовується такими факторами:

- інтеграційна сумісність із фреймворком Laravel, що забезпечує швидку розробку та мінімізацію накладних витрат на конфігурацію;
- стабільність та перевіреність у промислових сценаріях, що знижує ризики при роботі з фінансово критичною інформацією;
- масштабованість через вбудовані механізми реплікації та можливість побудови кластерних конфігурацій;
- доступність інструментів адміністрування (phpMyAdmin, MySQL Workbench, інструменти моніторингу продуктивності), що скорочує витрати на супровід;
- велика спільнота та кількість навчальних матеріалів, що полегшує підготовку нових спеціалістів та забезпечує швидке вирішення проблем.

Крім того, важливим аргументом є відповідність MySQL вимогам до безпеки. Завдяки гнучким механізмам керування доступом, підтримці ролей, аудиту та шифрування з’єднань, ця СКБД дозволяє будувати захищені системи з високим рівнем довіри до даних. У фінансовій сфері, де відсутність або спотворення даних може призвести до значних збитків, ці характеристики мають першочергове значення.

Таким чином, проведений аналіз дозволяє зробити висновок, що серед розглянутих альтернатив саме MySQL є найбільш доцільним вибором для автоматизованої системи обліку фінансів. Вона поєднує простоту впровадження з високим рівнем надійності, продуктивності та масштабованості, що робить її оптимальною платформою для реалізації

завдань у сфері фінансового управління.

2.4 Вибір протоколу взаємодії між клієнтською і серверною частинами

У процесі проєктування архітектури клієнт-серверної взаємодії важливим етапом є вибір протоколу обміну даними та механізму автентифікації. У сучасних вебзастосунках найбільш поширеними підходами є використання протоколів REST або GraphQL, які функціонують поверх HTTP/HTTPS та забезпечують стандартизований спосіб взаємодії між фронтендом і бекендом. У даній системі обрано REST-парадигму у поєднанні з передачею облікових даних за допомогою JWT-токенів (JSON Web Token).

Основною причиною вибору REST є його простота, сумісність і зрозумілість. Цей протокол базується на стандартних методах HTTP (GET, POST, PUT, DELETE), що дозволяє легко організовувати логіку CRUD-операцій над фінансовими даними. REST широко підтримується як у фреймворку Laravel, так і в екосистемі Vue, а також має значну кількість інструментів для тестування, моніторингу та документування (Postman, Swagger, Laravel API Resources). Це дає змогу скоротити час розробки й забезпечити легке масштабування системи.

Для автентифікації та авторизації користувачів було обрано JWT як найбільш гнучкий та універсальний механізм у SPA-застосунках. JSON Web Token є компактним форматом передачі інформації у вигляді підписаного токена, що складається з трьох частин: заголовка, корисного навантаження (payload) та підпису. Такий підхід має низку переваг:

- безстанова автентифікація (stateless): сервер не зберігає інформацію про сесії, що зменшує навантаження на бекенд і спрощує масштабування;
- швидкість обробки: токени перевіряються шляхом криптографічного підпису без звернення до бази даних для кожного запиту;
- гнучкість у використанні: JWT може передавати не лише ідентифікатор користувача, але й додаткові атрибути (ролі, права доступу,

регіональні налаштування), що дозволяє будувати багаторівневу систему контролю доступу;

- сумісність з різними клієнтами: токен можна передавати у заголовках HTTP-запитів і використовувати як у вебінтерфейсі, так і у мобільних чи сторонніх інтеграціях.

Важливим фактором на користь JWT стало також його поєднання зі сторонніми бібліотеками, зокрема tymon/jwt-auth для Laravel, що спрощує реалізацію автентифікації, та інтерцепторами у Vue (axios), які автоматично додають токени до запитів і здійснюють їхнє оновлення у випадку закінчення терміну дії. Це дозволяє досягти балансу між безпекою та зручністю використання системи.

У порівнянні з альтернативами, такими як сесійна автентифікація або OAuth 2.0, обраний підхід має низку переваг у контексті предметної області. Сесійна модель передбачає збереження стану на сервері, що ускладнює масштабування у випадку горизонтального розподілу навантаження. OAuth 2.0, у свою чергу, більше орієнтований на інтеграцію із зовнішніми сервісами та вимагає складнішої конфігурації, яка у даному проєкті є надлишковою. JWT забезпечує достатній рівень безпеки для внутрішнього обміну даними та водночас залишається легким в реалізації й підтримці.

Таким чином, використання REST поверх HTTPS із застосуванням JWT-токенів є оптимальним рішенням для автоматизованої системи фінансового обліку. Обраний підхід поєднує у собі простоту інтеграції, високу продуктивність, масштабованість і відповідність вимогам безпеки, що дозволяє створити ефективний механізм взаємодії між клієнтською та серверною частинами системи.

2.5 Вибір технологій для контейнеризації і оркестрації

У процесі створення клієнт–серверної системи постала необхідність забезпечити стабільне, відтворюване та масштабоване середовище виконання

для всіх її компонентів. Через складність архітектури, яка включає серверну частину на Laravel, клієнтський застосунок на Vue.js, базу даних MariaDB, систему кешування та черг Redis, модулі асинхронної обробки Horizon, а також підсистеми моніторингу Prometheus і Grafana, використання традиційного підходу до розгортання стало б недостатньо гнучким і ресурсомістким. Саме тому контейнеризація на базі Docker була обрана як ключова технологія, що дозволяє ізолювати кожен сервіс у власному середовищі, уніфікувати конфігурації та виключити залежність роботи системи від особливостей операційної системи чи налаштувань сервера.

Використання засобів оркестрації, зокрема Docker Compose, забезпечує автоматизоване керування контейнерами, опис інфраструктури у вигляді декларативної конфігурації та можливість одночасного запуску декількох логічно пов’язаних сервісів. Це дозволяє не лише спростити розгортання та супровід системи, а й підвищити її надійність, оскільки кожний компонент може масштабуватися незалежно від інших, швидко перезапускатися у разі збою, а сама архітектура — динамічно адаптуватися до навантаження. Контейнеризація та оркестрація створюють основу для подальшого переходу до більш складних інфраструктурних рішень, таких як Kubernetes, і зберігають гнучкість системи на всіх етапах її життєвого циклу, що є критично важливим для сучасних фінансових вебзастосунків.

2.5.1 Вибір основної технології для контейнеризації

У сучасній розробці програмного забезпечення однією з ключових проблем є різнорідність середовищ виконання. Відмінності між локальними налаштуваннями розробників, тестовими серверами та промисловим розгортанням призводять до появи помилок, які важко відтворити та усунути. Традиційні підходи, засновані на віртуальних машинах, забезпечують ізоляцію, проте мають суттєві обмеження: значне споживання ресурсів, складність адміністрування та низьку швидкість запуску.

Для вирішення цих проблем у проєкті було обрано технологію Docker, що реалізує принцип контейнеризації. Контейнер є ізольованим середовищем, яке містить застосунок разом з усіма його залежностями, але використовує ядро операційної системи спільно з іншими контейнерами. Такий підхід забезпечує високу портативність та передбачуваність: контейнер, створений і протестований у середовищі розробника, гарантовано працюватиме так само на сервері розгортання. Додатковою перевагою Docker є економне використання ресурсів, оскільки відсутня необхідність дублювати цілу операційну систему для кожного застосунку, як це відбувається у випадку віртуалізації.

Разом із Docker використовується Docker Compose — інструмент оркестрації контейнерів на одному хості. Compose дозволяє описати конфігурацію усієї системи у вигляді декларативного YAML-файла, де визначаються сервіси, їхні залежності, мережі та томи. Це спрощує як процес локальної розробки, так і промислове розгортання, оскільки вся інфраструктура може бути піднята або відтворена однією командою. У випадку розроблюваної системи Docker Compose використовується для координації роботи таких сервісів, як вебсервер, база даних, кеш, інструменти моніторингу та допоміжні утиліти.

Важливою складовою архітектури є використання мереж Docker, які дозволяють ізолювати взаємодію між сервісами. У проєкті створено окремі мережі для застосунку, бази даних, Redis та інструментів моніторингу. Це підвищує безпеку, оскільки кожен контейнер має доступ лише до тих сервісів, які йому необхідні. Наприклад, база даних доступна для бекенд-застосунку, але не для зовнішніх клієнтів напряму.

Ще однією перевагою контейнеризації є використання томів, які забезпечують збереження даних незалежно від життєвого циклу контейнера. Це критично важливо для фінансової системи, де втрата даних є неприйнятною. У проєкті томи застосовуються для збереження даних MariaDB, конфігурацій Redis Insight та бази Grafana. Такий підхід дозволяє безпечно оновлювати або перезапускати контейнери, не втрачаючи

накопиченої інформації.

Таким чином, вибір Docker і Docker Compose як основи інфраструктури зумовлений їхньою здатністю забезпечити ізольованість, повторюваність, портативність та керованість середовищ. У поєднанні з механізмами мереж і томів ці інструменти формують гнучку та надійну платформу для розгортання комплексної системи, що обробляє чутливі фінансові дані. Використання даного підходу мінімізує ризики, пов’язані з людським фактором під час конфігурації середовищ, скорочує витрати часу на адміністрування та створює основу для подальшого масштабування і розвитку системи.

2.5.2 Вибір пакетів для контейнерів Docker

У складі системи ключовим контейнером є серверна частина застосунку, реалізована на основі фреймворку Laravel у середовищі PHP 8.2. Винесення застосунку в окремий контейнер дозволяє ізолювати бізнес-логіку від інших сервісів, зменшити залежність від конфігурації операційної системи та спростити процес розгортання. Такий підхід забезпечує однакову поведінку застосунку на локальних машинах розробників і у промисловому середовищі, а також створює передумови для горизонтального масштабування у разі зростання навантаження.

Для маршрутизації HTTP-запитів використовується вебсервер Nginx, що працює як зворотний проксі та забезпечує ефективний розподіл навантаження між контейнерами. Його вибір обумовлений високою продуктивністю, низьким споживанням ресурсів і гнучкістю конфігурації. Nginx також відповідає за обслуговування статичних ресурсів, що знижує навантаження на бекенд, і є галузевим стандартом для сучасних вебсистем.

Система зберігання даних побудована на основі реляційної СУБД MariaDB версії 10.7. Ця технологія є відкритим відгалуженням MySQL, що поєднує стабільність, продуктивність та сумісність зі стандартними інструментами розробки. У контексті автоматизованої фінансової системи

MariaDB виступає надійним рішенням для роботи з транзакційними даними, забезпечуючи їхню цілісність і підтримку складних аналітичних запитів. Для зручності адміністрування інтегровано вебінструмент phpMyAdmin, який надає можливість переглядати структуру таблиць, виконувати SQL-запити та здійснювати управління даними безпосередньо через браузер.

Для підвищення продуктивності та забезпечення асинхронної обробки завдань у системі використовується Redis — високошвидкісне in-memory сховище ключ-значення. Воно застосовується для кешування результатів обчислень, тимчасового зберігання даних і роботи з чергами завдань. Використання Redis дозволяє значно зменшити затримки у відгуках системи та розвантажити основну базу даних. Для діагностики та моніторингу Redis застосовується RedisInsight, що забезпечує зручний графічний інтерфейс для аналізу збережених ключів, моніторингу продуктивності та відстеження можливих проблем.

Окремо у контейнерах розгорнуті компоненти, які забезпечують керування внутрішніми процесами Laravel. Horizon використовується для моніторингу та управління чергами, надає вебінтерфейс для контролю стану завдань, перегляду невдалих спроб і вимірювання продуктивності. Це створює прозорий механізм асинхронної обробки даних, що особливо актуально для фінансових систем з високою інтенсивністю операцій. Scheduler, реалізований за допомогою Supervisor, відповідає за виконання запланованих завдань, таких як формування періодичних звітів або автоматичне очищення даних. Ізоляція цих процесів у окремі контейнери підвищує стабільність та контрольованість роботи системи.

Моніторинг інфраструктури та застосунку реалізовано через зв’язку Prometheus та Grafana. Prometheus виконує збір і збереження метрик продуктивності, дозволяє відстежувати використання ресурсів і поведінку сервісів у динаміці. Grafana інтегрується з Prometheus і забезпечує візуалізацію даних у вигляді інтерактивних дашбордів. Така інтеграція створює умови для оперативного реагування на проблеми, аналізу ефективності роботи та планування масштабування інфраструктури.

Усі перелічені сервіси взаємодіють між собою через окремі віртуальні мережі Docker, що підвищує безпеку та дозволяє ізолювати внутрішню логіку від зовнішнього середовища. Для збереження даних використовуються персистентні томи, завдяки чому навіть у випадку перезапуску або оновлення контейнерів фінансова інформація, аналітичні дані та конфігурації залишаються незмінними.

Таким чином, архітектура контейнеризації у даній системі охоплює всі ключові аспекти: виконання бізнес-логіки, зберігання даних, кешування, асинхронну обробку, адміністрування та моніторинг. Кожен компонент обраний з урахуванням його зрілості, популярності та перевіреної надійності у промислових умовах. Це дозволяє сформувати комплексне середовище, здатне забезпечити безперебійну роботу автоматизованої фінансової системи та її подальший розвиток.

2.6 Висновки до розділу

У результаті аналізу та порівняння можливих підходів було сформовано комплексне технологічне рішення, яке найбільш повно відповідає вимогам до автоматизованої системи обліку фінансів. Вибір кожної складової архітектури здійснювався з урахуванням балансу між продуктивністю, надійністю, гнучкістю та перспективами подальшого розвитку. Загальна мета полягала в тому, щоб створити платформу, здатну не лише виконувати базові функції обліку, а й бути готовою до масштабування, інтеграції з іншими системами та довготривалої підтримки.

На рівні серверної частини було обрано мову програмування PHP, яка впродовж багатьох років залишається стандартом де-факто у веброзробці. Свою популярність ця мова здобула завдяки простоті синтаксису, широкому спектру застосування та наявності великої кількості готових рішень для інтеграції з базами даних, системами авторизації, засобами кешування тощо. У проєкті використовується сучасна версія PHP 8.2, яка має суттєві

покращення продуктивності, розширені можливості роботи з типами та підтримку нових мовних конструкцій, що підвищує надійність коду й полегшує його супровід. На базі PHP обрано фреймворк Laravel, що зарекомендував себе як гнучкий і потужний інструмент для побудови складних вебсистем. Він поєднує в собі багатий набір вбудованих можливостей (автентифікація, авторизація, управління чергами, система маршрутизації, підтримка REST API) та величезну екосистему сторонніх бібліотек, які прискорюють розробку. Завдяки цьому Laravel є оптимальним вибором для побудови надійної та безпечної бізнес-логіки фінансової системи.

Фронтенд-середовище було вибудуване на основі комбінації JavaScript і TypeScript, що поєднують універсальність першої мови та строгість другої. JavaScript є єдиною нативною мовою виконання у браузерах і, відповідно, єдиним раціональним вибором для побудови динамічних інтерфейсів. Його доповнення TypeScript надає додатковий рівень захисту завдяки статичній типізації, що особливо важливо у фінансових системах, де навіть незначні помилки можуть мати суттєві наслідки. Для організації клієнтської частини було обрано фреймворк Vue.js, який поєднує простоту освоєння з потужністю інструментарію. Він дозволяє створювати масштабовані односторінкові застосунки, підтримує реактивність даних та має широку екосистему офіційних рішень для управління станом, маршрутизації та інтернаціоналізації. Порівняно з альтернативами Vue.js займає збалансовану позицію: він менш складний у конфігурації, ніж Angular, та менш перевантажений додатковими залежностями, ніж React. Це робить його доцільним вибором для побудови інтерфейсу системи, орієнтованої на роботу з великими обсягами даних у режимі реального часу.

Для зберігання фінансової інформації було обрано MariaDB, яка є стабільним і надійним рішенням серед реляційних СУБД. Її вибір обґрунтовується поєднанням продуктивності, сумісності з MySQL та відкритої моделі розвитку, що гарантує довгострокову підтримку й адаптивність. MariaDB забезпечує виконання транзакцій із високим рівнем цілісності даних, підтримує складні аналітичні запити та масштабування, що є

критичним у контексті фінансових систем. Для зручності адміністрування інтегровано phpMyAdmin, який дозволяє виконувати базові операції через графічний інтерфейс, що полегшує роботу як розробників, так і адміністраторів.

Взаємодія між клієнтською та серверною частинами організована на основі REST API, із використанням формату обміну даними JSON. Такий підхід відповідає сучасним стандартам побудови вебсистем, забезпечує простоту інтеграції та універсальність при роботі з різними клієнтами — як вебзастосунками, так і мобільними платформами. REST-архітектура забезпечує чітке відокремлення бізнес-логіки від користувацького інтерфейсу, що сприяє масштабованості й гнучкості системи у майбутньому.

Важливим компонентом архітектури є Docker у поєднанні з Docker Compose, які використовуються для контейнеризації та оркестрації сервісів. Завдяки цьому кожен елемент системи функціонує у власному ізольованому середовищі, що усуває конфлікти залежностей і підвищує передбачуваність роботи. Використання віртуальних мереж і персистентних томів гарантує безпеку та збереження даних незалежно від життєвого циклу контейнерів. Окремі контейнери відведено для вебсервера Nginx, бази даних MariaDB, кешу Redis, а також для інструментів моніторингу й керування процесами (Prometheus, Grafana, Horizon, Scheduler). Така модульність дозволяє масштабувати окремі сервіси незалежно один від одного, спрощує їх обслуговування та підвищує загальну надійність платформи.

Загалом вибір описаного комплексу технологій дозволяє сформувати архітектуру, яка відповідає ключовим критеріям сучасних інформаційних систем: продуктивності, безпеці, масштабованості та зручності супроводу. Кожен компонент було підібрано не лише з огляду на його популярність, а й з урахуванням практичної доцільності саме у сфері фінансового обліку. Такий підхід гарантує, що система зможе ефективно виконувати поставлені завдання, залишаючись водночас гнучкою для адаптації під нові виклики.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Розробка серверної частини системи

Розробка серверної частини системи є ключовим етапом у створенні повноцінного клієнт–серверного програмного забезпечення, оскільки саме на цьому рівні зосереджено обробку бізнес-логіки, забезпечення безпеки, цілісності даних та інтеграцію з зовнішніми сервісами. Серверна частина виконує роль центрального ядра системи: приймає та обробляє запити клієнтського застосунку, керує доступом користувачів, здійснює операції з базою даних, забезпечує роботу черг, планувальника задач та системи моніторингу. Враховуючи вимоги до надійності, масштабованості та продуктивності фінансових вебсистем, у роботі було обрано фреймворк Laravel 11, який поєднує сучасну архітектуру MVC, розвинену інфраструктуру для REST API, вбудовані засоби аутентифікації, підтримку асинхронної обробки через черги та інтеграцію з інструментами спостережуваності. Такий підхід дозволив створити гнучку та розширювану серверну платформу, яка забезпечує стабільну взаємодію між клієнтським інтерфейсом і базою даних та відповідає сучасним вимогам до розробки інформаційних систем фінансового спрямування.

3.1.1 Вибір архітектурного підходу

У процесі розроблення серверної частини системи було обрано архітектурний підхід, заснований на шаблоні Model–View–Controller (MVC), який лежить в основі фреймворку Laravel. Застосування цього шаблону дозволяє забезпечити структурну впорядкованість коду, чітке розмежування функціональних обов’язків між складовими частинами програми та створити гнучку архітектуру, придатну до масштабування й подальшого розвитку.

Сутність шаблону MVC полягає у поділі програмного забезпечення на

три взаємопов’язані компоненти: модель, контролер і подання. Модель відображає бізнес-логіку та структуру даних, подання відповідає за їх відображення, а контролер виконує роль посередника, що керує потоком даних між користувачем і системою. Такий розподіл сприяє високій зрозумілості коду, спрощує внесення змін і підвищує рівень тестованості програми.

У межах даної системи роль моделей реалізовано через ORM-шар Eloquent. Кожна модель відповідає окремій таблиці бази даних і відображає її структуру, забезпечуючи доступ до записів через об’єктно-орієнтований інтерфейс. Наприклад, модель Payment визначає основні атрибути платіжних операцій, а також зв’язки з іншими сутностями системи, зокрема користувачами. Завдяки цьому логіка роботи з даними зосереджується виключно на рівні моделей, не потрапляючи у контролери чи сервіси. Модельний шар відповідає за збереження, пошук, оновлення та видалення даних, а також інкапсулює бізнес-правила, що стосуються конкретної сутності.

Контролери в Laravel відіграють роль центральної ланки у взаємодії між клієнтом і внутрішніми механізмами системи. Кожен HTTP-запит, що надходить до сервера, спочатку потрапляє до маршрутизатора, який визначає відповідний контролер і метод для його обробки. Контролери виконують базову логіку — валідацію даних, виклик відповідного сервісу, формування відповіді та передачу результату клієнтові. У даному проєкті контролери не містять безпосередньої логіки роботи з базою даних, а взаємодіють із сервісами, які інкапсулюють функціональність бізнес-рівня. Такий підхід відповідає принципу єдиної відповідальності, коли кожен компонент системи вирішує лише вузьке коло завдань.

Особливістю цієї реалізації є те, що рівень подання представлено не у вигляді традиційних HTML-шаблонів, а через JSON-ресурси. Оскільки клієнтська частина системи побудована як односторінковий застосунок (SPA) на базі Vue.js, поданням для користувача виступає інтерфейс, який взаємодіє із сервером через REST-API. Кожен запит до API повертає строго

структуровану JSON-відповідь, сформовану за допомогою ресурсних класів Laravel. Такий підхід уможливлює повну незалежність клієнтського інтерфейсу від внутрішньої архітектури серверної частини, а також спрощує подальшу розробку мобільних або сторонніх клієнтів, які можуть використовувати ті самі API-контракти.

Таким чином, класична трирівнева структура MVC у межах цього застосунку зазнає розширення і поступово трансформується у більш сучасну архітектурну модель. Окрім моделей, подань і контролерів, у системі виокремлено сервісний шар, який реалізує основну бізнес-логіку. Наприклад, сервіси AuthService, PaymentService, UsefullLinkService або PrometheusService беруть на себе обробку прикладних процесів — від генерації JWT-токенів до формування метрик для моніторингу. Така структура дозволяє мінімізувати дублювання коду, забезпечити повторне використання логіки у різних частинах проєкту й створює передумови для майбутнього переходу до мікросервісної архітектури.

Важливою характеристикою MVC-підходу у Laravel є наявність проміжного рівня middleware, який обробляє запити до потрапляння їх у контролери. Зокрема, у системі реалізовано перевірку автентичності користувача через JWT-механізм, локалізацію запитів, обмеження доступу за ролями та перевірку політик. Це забезпечує додатковий рівень безпеки та ізоляцію технічних аспектів від бізнес-логіки.

Під час обробки запиту система діє за сталою послідовністю: вхідний HTTP-запит спочатку перевіряється проміжним ПЗ, далі передається у відповідний контролер, який, у свою чергу, звертається до сервісів і моделей, після чого результат формується у вигляді ресурсу й повертається користувачеві у форматі JSON. Такий механізм відповідає концепції розділення обов’язків та забезпечує прозорий потік даних від клієнта до бази даних і назад.

Завдяки застосуванню архітектурного підходу MVC вдалося досягнути високої впорядкованості проєкту, логічної розмежованості його частин і зручності підтримки. Контролери залишаються мінімалістичними, моделі —

цілісними, а рівень подання — універсальним. Це створює умови для стабільного розвитку системи, спрощує впровадження нових функцій та підвищує якість програмного коду. Реалізований у Laravel підхід до MVC не лише зберігає традиційні принципи архітектури, а й органічно поєднує їх із сучасними інженерними практиками, такими як використання сервісного шару, Data Transfer Object-підходів, черг і асинхронних процесів.

Таким чином, архітектура серверної частини системи демонструє збалансоване поєднання класичної моделі MVC з елементами чистої архітектури, що дозволяє забезпечити гнучкість, модульність і масштабованість програмного рішення. У результаті застосунок зберігає простоту й зрозумілість структури, властиву фреймворку Laravel, одночасно відповідаючи вимогам сучасних корпоративних систем щодо безпеки, ефективності та розширюваності.

3.1.2 Структура серверної частини системи

Структура серверної частини системи побудована на базі фреймворку Laravel, який організовує програмний код за чітко визначеними логічними рівнями та папками. Такий підхід забезпечує порядок у проєкті, зрозумілість взаємодії між компонентами та можливість ізолювати бізнес-логіку від інфраструктурного шару. Уся архітектура побудована відповідно до концепції «розділення обов’язків», коли кожен елемент проєкту відповідає лише за одну конкретну функцію. Схема роботи серверної частини зображена на рис. 3.1.

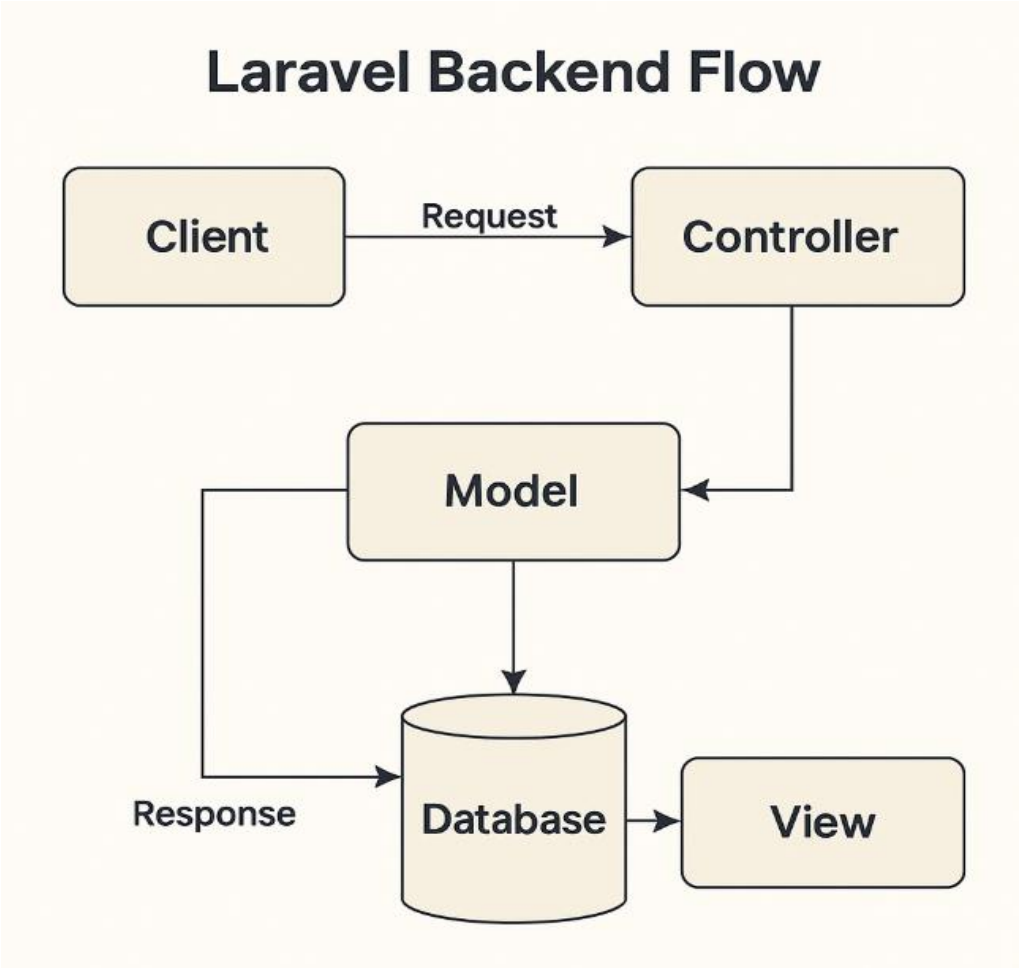


Рисунок 3.1 – Схема роботи серверної частини

Основу проєкту становить каталог app, у якому зосереджено ключову логіку системи. Саме тут містяться всі головні елементи — контролери, моделі, сервіси, обробники запланованих завдань, ресурси API, валідатори, middleware, черги та команди консолі. Фактично ця директорія є ядром застосунку, яке описує його поведінку та правила взаємодії з користувачами, базою даних і зовнішніми сервісами.

Підкаталог Http об’єднує усі елементи, що безпосередньо взаємодіють із зовнішнім світом через HTTP-протокол. Тут розташовані контролери, проміжне програмне забезпечення та класи для валідації запитів. Контролери виконують роль точок входу, через які система приймає запити від клієнтської частини. Кожен контролер відповідає за певний функціональний домен - наприклад, автентифікацію користувача, обробку платежів або надання аналітичних даних. Вони не містять складної логіки, а лише координують роботу між сервісами, моделями та шаром подання. Завдяки цьому контролери

залишаються простими, короткими й легко підтримуваними.

У цьому ж просторі розміщені класи проміжного програмного забезпечення (middleware), які перехоплюють запити до їх обробки контролерами. Вони реалізують перевірку прав доступу, автентифікацію за допомогою токенів JWT, контроль локалізації, логування або захист від повторних запитів. Завдяки цьому кожен запит до системи проходить через єдину перевірку безпеки, що гарантує стабільність і надійність роботи застосунку.

Окрему роль у структурі проєкту відіграють класи Requests, які визначають правила валідації даних, що надходять від користувачів. Вони замінюють ручну перевірку у контролерах і забезпечують централізовану логіку перевірки форм. У результаті будь-які помилки вводу, некоректні формати або відсутні обов’язкові поля перехоплюються ще до моменту виконання основної бізнес-логіки.

Базовий рівень даних представлений у підкаталозі Models, який містить класи, що описують основні сутності системи. У проєкті виділено моделі користувачів, платежів, аналітичних посилань та інших об’єктів, які відображають відповідні таблиці бази даних. Кожна модель реалізує зв’язки між сутностями, інкапсулює бізнес-правила та описує поведінку об’єкта на рівні даних. Використання ORM Eloquent дає змогу працювати з базою не через прямі SQL-запити, а через об’єктно-орієнтований інтерфейс, що значно підвищує безпеку й зручність розробки.

Над моделями розташований сервісний рівень, що представлений каталогом Services. Саме тут знаходиться основна прикладна логіка, яка описує, яким чином система виконує бізнес-операції. Наприклад, сервіси AuthService та UserService керують процесами реєстрації, входу користувачів та оновлення токенів, тоді як PaymentService реалізує механізми створення, підтвердження та моніторингу платежів. Окремий модуль PrometheusService відповідає за підготовку метрик для моніторингу, а UsefullLinkService — за роботу з інформаційними посиланнями. Винос логіки у сервіси дозволяє зберігати контролери чистими та спрощує повторне використання функцій у

різних частинах системи. У рамках тих самих сервісів розміщені інтеграційні модулі з платіжними провайдерами, наприклад LiqPayService або PaysystemManager, що інкапсулюють взаємодію із зовнішніми API та приховують усі деталі реалізації від решти програми.

Для стандартизації відповідей клієнтам використовується каталог Http/Resources, який реалізує шар подання у вигляді API-ресурсів. Кожен ресурс визначає формат, у якому дані від моделей або сервісів повертаються користувачу. Наприклад, PaymentResource або UserResource формують акуратні JSON-відповіді з необхідними атрибутами, виключаючи внутрішні службові поля. Такий підхід забезпечує стабільність API і незалежність клієнтської частини від змін у структурі бази даних.

Файли у каталозі Enums містять переліки фіксованих значень, таких як статуси платежів, типи операцій, ролі користувачів або типи відповідей. Використання енумів дозволяє уникати помилок, пов’язаних із «магічними рядками», робить код більш зрозумілим і підвищує його стійкість до змін.

Каталог Jobs відповідає за реалізацію відкладених і фонових завдань. Наприклад, відправлення листа після реєстрації користувача, експорт статистичних метрик чи створення інвойсу можуть виконуватись асинхронно через чергу. Це дає змогу не блокувати основний потік запитів і забезпечує високу швидкодію системи. Робота черг контролюється інструментом Laravel Horizon, який у даному проєкті запускається як окремий контейнер у Docker-середовищі.

У структурі також присутній каталог Console, де розміщено власні команди Artisan для обслуговування системи. Серед них є команди для керування чергами, очищення кешу, виконання розкладу завдань і надсилання повідомлень під час деплою. Завдяки цьому адміністратор або DevOps-інженер може виконувати необхідні дії безпосередньо через термінал.

Важливим компонентом архітектури є Handlers, що містять логіку планувальника задач. Через цей механізм система виконує регулярні операції, наприклад оновлення метрик або автоматичну генерацію звітів. Планувальник тісно взаємодіє з контейнером “scheduler”, який у Docker-середовищі запускає

процеси згідно з конфігурацією Supervisor.

Конфігураційні файли розташовані в каталозі config і визначають параметри підключення до бази даних, Redis, Horizon, Prometheus, системи черг і логування. Їх структура дозволяє централізовано керувати поведінкою системи в різних середовищах — локальному, тестовому або продуктивному. Змінні середовища зберігаються у файлі .env, який використовується під час ініціалізації контейнерів.

Додатково проєкт містить каталог routes, де оголошено всі маршрути веб- і API-рівнів. Вони згруповані за просторами імен і версіями API, що дозволяє підтримувати декілька паралельних версій без конфліктів. Кожен маршрут пов’язаний із відповідним контролером і middleware, яке перевіряє права доступу чи наявність токена. Така організація сприяє ясності структури запитів і забезпечує стабільну роботу системи навіть при розширенні її функціональності.

Усі компоненти взаємодіють між собою через контейнер залежностей Laravel, що дозволяє автоматично створювати й інжектувати потрібні класи без ручного керування залежностями. Завдяки цьому проєкт зберігає низький рівень зв’язності, а кожен елемент може бути легко замінений або протестований незалежно від решти системи.

Таким чином, структура серверної частини системи побудована логічно та послідовно. Вона поєднує класичні підходи фреймворку Laravel з елементами сервісно-орієнтованої архітектури, що забезпечує високу надійність, зручність супроводу та легкість розширення. Кожен компонент має власну чітко окреслену роль: контролери приймають і координують запити, моделі оперують даними, сервіси реалізують бізнес-процеси, ресурси формують відповіді, а middleware та черги гарантують безпеку й ефективність роботи. Така організація коду дозволяє підтримувати цілісність системи, забезпечує стабільність під час розробки нових функцій і створює основу для подальшого масштабування проєкту як повноцінного багатокомпонентного програмного комплексу.

3.2 Розробка структури бази даних системи

Опис бази даних подається у контексті обраної СУБД MariaDB 10.7, з урахуванням реальних міграцій проєкту та ролі БД у загальній архітектурі. На рисунку 3.1 зображена ER-модель бази даних.

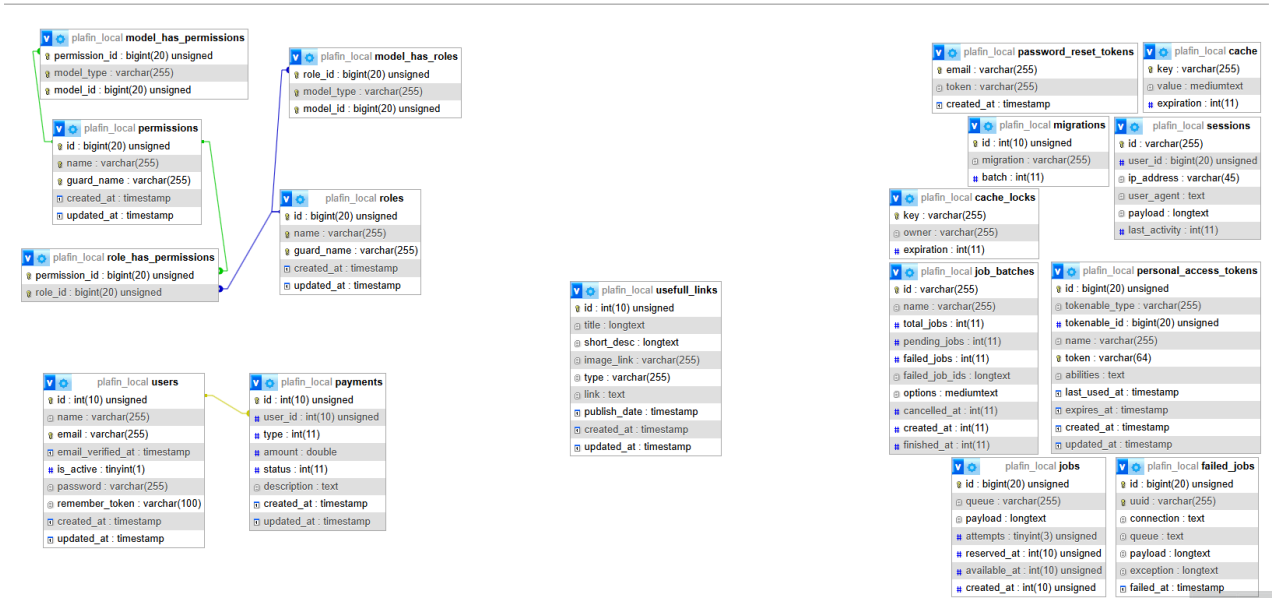


Рисунок 3.2 – ER-модель схеми бази даних

Схема побудована за принципом виокремлення доменних сутностей (користувачі, платежі, інформаційні посилання) від інфраструктурних таблиць (черги, кеш, сесії, персональні токени доступу, ACL), що забезпечує стабільність прикладної логіки і прозоре масштабування сервісів. Центральною сутністю виступає таблиця користувачів, яка містить ідентифікатор, облікові поля, прапорець активності та часові мітки; додатково підтримуються таблиці токенів для відновлення пароля і таблиця сесій із ключем первинної ідентифікації та індексованими атрибутами користувача та часу активності. Такий розподіл дозволяє відокремити довготривалі облікові дані від короткоживучих сесій і процедур безпеки, не перевантажуючи основну модель користувача.

Домен платежів репрезентовано окремою таблицею з цілочисельним

первинним ключем, зовнішнім ключем на користувача та індексом за цим зв’язком, грошовою сумою, типом і статусом операції, описами та часовими атрибутами. Зовнішній ключ визначено з каскадним оновленням і видаленням, що гарантує цілісність при операціях над обліковими записами; при видаленні користувача пов’язані платежі видаляються автоматично, виключаючи «висячі» записи. Стани та типи платежів у прикладній логіці представлені енумами, тому на рівні БД вони зберігаються як типи з обмеженим словником значень, керованим кодом застосунку; поєднання перевірок на рівні СУБД (NOT NULL, зовнішні ключі, індекси) і на рівні валідаторів FormRequest формує двоконтурний контроль якості даних. Операції над платежами часто ініціюються або завершуються асинхронно, тому таблиці черг відіграють інфраструктурну роль у гарантуванні доставлення та повторів: основна таблиця jobs зберігає навантаження, чергу, лічильники спроб і часові мітки доступності та створення; супровідні таблиці job_batches та failed_jobs фіксують групові виконання і виняткові ситуації з повним контекстом помилки для подальшого аналізу. Вибір такої схеми типовий для Laravel-екосистеми і узгоджується з використанням Redis як брокера, а також Horizon як диспетчера воркерів, що зменшує затримки в HTTP-шарі й підвищує відмовостійкість.

Додатковим прикладним доменом є інформаційні посилання; їх таблиця включає автоінкрементний ідентифікатор, JSON-поля для багатомовних назв і скорочених описів, посилання на зображення, тип, саму URL-адресу та дату публікації. Використання JSON для текстових атрибутів забезпечує зручний механізм локалізації без роздування схеми великою кількістю колонок, а також сумісність із пакетом багатомовності у прикладному шарі. Часові мітки дозволяють вибудовувати відбір за періодами і підтримувати публікаційний життєвий цикл. Для читання з цієї таблиці характерні фільтри за датою та типом, тоді як для платежів типовими є фільтри за користувачем, статусом та періодом; відповідні індекси закладено у міграціях, що покращує селективність запитів і контроль плану виконання.

Питання автентифікації та авторизації вирішено комбінацією двох

механізмів. Таблиця персональних токенів забезпечує зберігання токенів доступу з поліморфним посиланням на «власника» токена, унікальним хешем і набором здібностей, датами використання та завершення строку дії; така модель підтримує сценарії інтеграцій і машинного доступу, не пов'язаного з cookie-сесіями. Рольово-дозвільна модель реалізована через набір таблиць Spatie Permission: довідники прав і ролей із унікальними комбінаціями імен і guard-ів, з'єднувальні таблиці «модель-має-право» та «модель-має-роль», а також «роль-має-право». Схема передбачає індекси для морфних ключів і каскадні видалення з довідників прав і ролей; у підсумку авторизаційні перевірки виконуються у прикладному шарі без потреби додаткових джоїнів до доменних таблиць, що знижує зв'язність і спрощує аудит доступу. На рисунку 3.2 зображена ER-модель схеми рольово-дозвільної моделі.

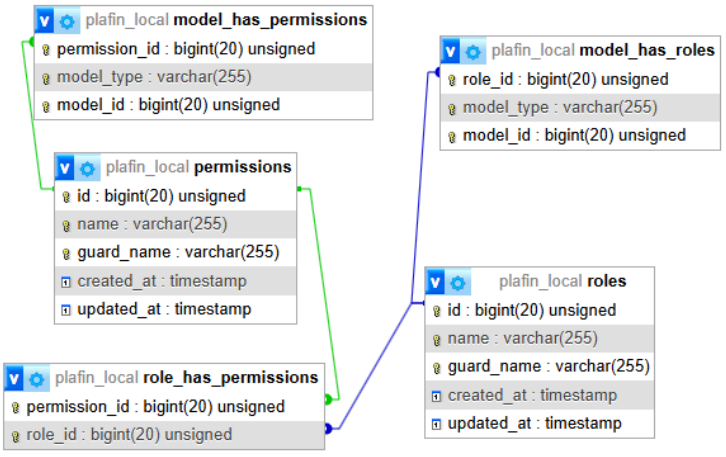


Рисунок 3.3 – ER-модель схеми рольово-дозвільної моделі.

Інфраструктурні аспекти оптимізації доповнено таблицями кешу та блокувань кешу, які дозволяють матеріалізовувати результати обчислювально дорогих операцій і синхронізувати конкурентний доступ до розділюваних ресурсів. Аналогічно, таблиця сесій зберігає серверний стан для користувачів із можливістю пошуку за ідентифікатором користувача та часом останньої активності; це дає змогу виконувати адміністративні операції на кшталт примусового виходу або інвентаризації активних сесій. Наявність цих таблиць

у структурі не прив’язує застосунок до конкретного способу зберігання стану, але створює надійний бекенд для сценаріїв, де cookie-сесії або веб-міст між SPA і приватними ресурсами вимагають перевірки часу життя та узгодження з політиками безпеки. На рисунку 3.3 зображена ER-модель схеми технічних таблиць.

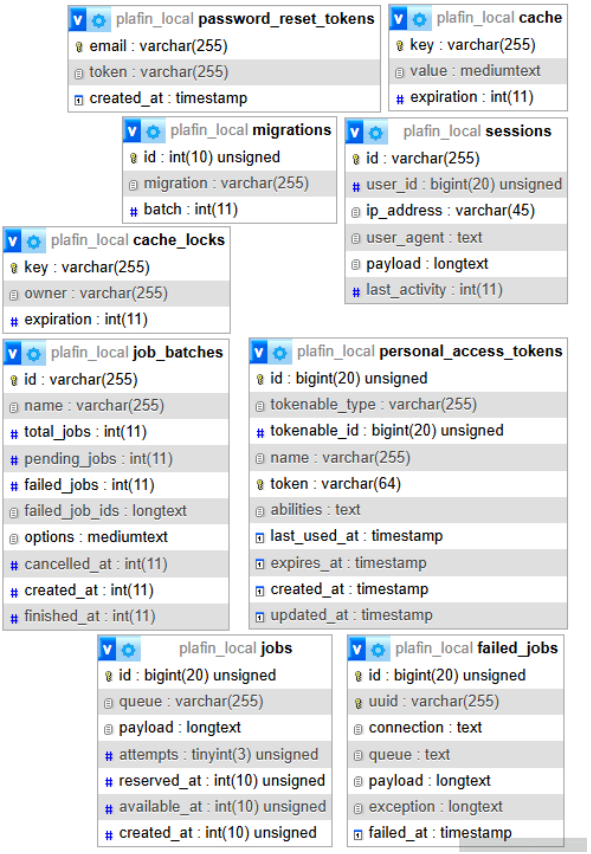


Рисунок 3.4 – ER-модель схеми технічних таблиць.

Навантажувальний профіль схеми передбачає продукційне використання індексів на зовнішніх ключах і часових полях, що дозволяє ефективно виконувати агрегації для метрик і аналітики. Так, кількість користувачів із платежами у стані «очікує» може бути отримана запитом, що використовує індекс за полем статусу та вибірку по користувачах з унікалізацією; ці значення надалі експортуються у реєстр метрик, звідки їх зчитує Prometheus. Подібні індикатори не вимагають змін у схемі БД, адже ґрунтуються на вже визначених ключах і індексах, що є ознакою коректного первинного моделювання.

З точки зору еволюції, схема зберігає баланс між нормалізацією та практичністю. Ключова бізнес-сутність «платіж» має мінімально необхідний набір атрибутів із чіткими посиланнями на користувача; можливе розширення через додаткові таблиці для провайдер-специфічних реквізитів, історії зміни статусів або збереження webhook-подій із часовими мітками та підписами постачальника. Облікові дані користувача залишаються відокремленими, тоді як авторизаційні таблиці кампанують право доступу незалежно від доменних даних. Локалізований контент уважно ізольовано в JSON-полях із строгим контролем з боку прикладного рівня, що спрощує додавання нових мов без міграцій. Асинхронні процеси мають власний цикл життя у таблицях черг, не змішуючись із транзакційними таблицями домену.

Узгодженість цієї схеми з архітектурою застосунку проявляється у прозорих кордонах між доменом, безпекою та інфраструктурою. Контролери і сервіси працюють із мінімальною кількістю таблиць, необхідних для виконання бізнес-операцій; решта механізмів — кеш, черги, сесії, ACL — залишаються універсальними компонентами, які не потребують спеціальної вбудованої логіки у доменні моделі. Такий поділ знижує зв’язність, полегшує тестування та створює надійний фундамент для подальшого масштабування: додавання нових платіжних провайдерів, розширення аналітики, уведення політик збереження історії чи побудова архівних шарів не вимагатимуть руйнівних перетворень існуючих таблиць, а відбуватимуться через додаткові, чітко сфокусовані міграції.

Загалом, представлена модель бази даних відповідає вимогам сучасних веб-систем із підвищеними вимогами до безпеки, спостережуваності та експлуатаційної простоти. Вона поєднує коректну підтримку цілісності через зовнішні ключі та каскадні політики, адекватну індексацію під найпоширеніші запити, ізоляцію авторизаційної логіки від доменних даних, а також надає необхідні інфраструктурні таблиці для ефективної роботи механізмів кешування, черг і сесій. Таке проєктування узгоджено з практиками Laravel і забезпечує стійку основу для подальшого розвитку системи у межах обраної технологічної платформи.

3.3 Розробка клієнтської частини системи

Клієнтська частина системи реалізована як односторінковий застосунок, інтегрований у Laravel у директорії resources, з повним циклом збирання через Vite та плагін Laravel Vite Plugin. Такий спосіб розміщення дозволяє підтримувати єдине середовище виконання, зберігати конгруентність змінних середовища та скоротити операційні витрати на розгортання: фронтенд і бекенд будуються синхронно, а артефакти збірки публікуються безпосередньо у public із коректною версією та хешуванням. Конфігурація Vite забезпечує швидкий HMR у локальному середовищі та мінімізацію продукційних бандлів, включно з tree-shaking, розділенням коду та дедуплікацією залежностей. Типізація на рівні TypeScript унеможливлює значний клас рантайм-помилок: типи DTO, інтерфейси API-відповідей і сигнатури методів сервісів формують контракт між клієнтом і сервером, що підвищує надійність інтеграції та спрощує рефакторинг.

Архітектура застосунку тримається на центральному вході, де ініціалізуються екземпляри Vue, Pinia, Router та i18n. На ранній фазі старту застосунок виконує «теплий» запит ідентифікації користувача, якщо у локальному сховищі наявний токен, що дозволяє відразу сформувати коректний набір маршрутів і доступів без мерехтіння інтерфейсу. Маршрутизатор організовано навколо декларативних конфігурацій, доповнених метаданими доступу, і супроводжується охоронцями навігації. Саме охоронці виконують перевірку токена, ініціалізацію профілю та ролей і вирішують питання допуску до закритих розділів. Важливим є те, що логіка доступу зосереджена не в окремих компонентах, а на рівні маршрутизатора та сервісного шару; завдяки цьому компоненти подання залишаються чистими й не містять умовної логіки безпеки, що зменшує зв’язність і ризик розсинхрону політик. Для незареєстрованих відвідувачів передбачено коректну поведінку редиректів і «м’яке» приземлення на форми автентифікації, тоді як для авторизованих — запобігання доступу до сторінок входу та реєстрації шляхом автоматичної переадресації на робочий стіл.

Шар стану реалізовано через Pinia з доменною сегрегацією: автентифікаційний стан зберігає профіль, ролі, прапор авторизованості та механізми роботи з токеном; бізнес-стани структуруються за предметними областями — платежі користувача, адміністративні реєстри, службові налаштування застосунку. Системні гетери інкапсулюють похідні дані — наприклад, обрахунок фільтрів або агрегацій для таблиць — і відокремлюють їх від уявлення. Дії сто́рів виконують виклики мережевих сервісів, уніфікуючи обробку помилок, показ повідомлень та індикатор завантаження. У підсумку компоненти працюють з уже підготовленими структурами, реагуючи лише на зміни стану, що відповідає реактивній природі Vue та спрощує тестування. Для сценаріїв з високою частотою взаємодії застосовано локальні кеші з TTL на рівні сто́ру, що зменшує кількість запитів до API і створює відчуття «миттєвості» інтерфейсу.

Мережевий шар представлений фасадом над Axios із централізованим налаштуванням заголовків і перехоплювачів. У фазі запиту відбувається ін’єкція заголовка авторизації формату Bearer, а у фазі відповіді — нормалізація помилок до єдиного словника. Це дозволяє зводити обробку винятків у компонентах до коротких сценаріїв типу «успіх/невдача», а деталізацію повідомлень, включно зі специфікою HTTP-статусів і кодів доменних помилок, виконувати на рівні одного сервісу. У випадку 401-відповідей реалізовано «м’яке відновлення» сеансу: якщо доступний рефреш-механізм, виконується повторна автентифікація без втручання користувача; якщо ні — здійснюється повне завершення сеансу з подальшим редиректом на сторінку входу. Чітке розділення приватного та адміністративного префіксів /api і /api/admin унеможливлює змішування прав доступу і підвищує прозорість трасування запитів на рівні DevTools і серверних логів.

Підсистема інтернаціоналізації побудована на vue-i18n і забезпечує повний цикл локалізації UI, включно з множинами, форматами дат і чисел, а також контекстно-залежними повідомленнями. Мовні ресурси розміщені у структурі, що відповідає доменам застосунку, що спрощує масштабування словників і залучення не технічних фахівців для редагування перекладів.

Локаль зберігається у загальному сховищі та синхронізується з маршрутизацією, що дозволяє будувати language-aware навігацію без дублювання сторінок. Наявність fallback-механізму унеможливлює порожні ключі, а централізоване налаштування i18n виключає контекстні конфлікти.

Візуальна система проєкту базується на Tailwind CSS з кастомізованими шарами утиліт і компонентними патернами, що зменшує залежність від великих UI-бібліотек і дозволяє зберігати сувору контрольованість дизайну. Окремі стилі для таблиць, форм і редакторів допомагають забезпечувати однакову типографіку, відступи, стани наведення та фокусів, що безпосередньо впливає на доступність. Для таблиць даних застосовано адаптивні підходи до пагінації та віртуалізації там, де це доцільно, а також чітке розділення відповідальності між шаром даних (Pinia) і шаром відображення. Важливу роль відіграють skeleton-компоненти і спінери, які заповнюють проміжки завантаження, формуючи коректні очікування користувача і відчуття швидкодії.

Логіка форм вибудувана навколо явних моделей даних і валідації як на клієнті, так і на сервері. На клієнті валідація виконується легкими композиціями, що покривають форматні обмеження, прості інваріанти та залежні поля, тоді як остаточну перевірку здійснює бекенд через FormRequest. Такий двоконтурний підхід поєднує оперативність інтерфейсу з гарантіями цілісності на сервері. Для критичних дій — створення платежу чи зміна ролей користувача — передбачено явні підтвердження і повторні перевірки станів, що унеможливлює конфлікти конкурентних змін.

Система повідомлень уніфікована через фасад поверх SweetAlert, завдяки чому успіхи, помилки і попередження демонструються консистентно з урахуванням локалі, таймінгів автозакриття та фокусу. Для довготривалих операцій реалізовано «неблокуючі» індикатори з можливістю відміни або переходу до історії подій. Повідомлення не дублюються у компонентах: вони надходять із сервісів або сто́рів як наслідок виконання дій, завдяки чому вдається уникати розсинхрону форматів і стилів.

З точки зору продуктивності застосовано кілька тактик. По-перше,

маршрути динамічно імпортують сторінки (code splitting), що скорочує розмір початкового бандла. По-друге, таблиці та списки використовують «розумну» пагінацію на сервері з кешуванням останньої сторінки і параметрів пошуку, що позитивно впливає на UX при поверненні на попередній екран. По-третє, загальні бібліотеки підтягуються один раз і не дублюються у чанках, тоді як рідкісні модулі — наприклад, редактор або спеціалізовані віджети — завантажуються на вимогу. По-четверте, для часто відвідуваних екранів застосовується prefetch посилань і lazy hydration, що покращує перший час відгуку без видимого побічного ефекту.

Безпека на клієнті розглядається як частина загального контуру. Токени не зберігаються у доступних для сторонніх сайтів cookie; у локальному сховищі вони використовуються виключно через контрольований сервіс, який, за потреби, виконує повну очистку сховища при завершенні сеансу. Усі потенційно небезпечні вставки контенту проходять санітизацію, а HTML-відображення, якщо воно дозволене, здійснюється через обмежені обгортки. Маршрути, що вимагають підвищених прав, мають додатковий рівень перевірки на клієнті, проте саме сервер залишається джерелом істини, і будь-які критичні операції підтверджуються бекендом.

Розробницький досвід підтримується завдяки послідовній структурі каталогів, чітким назвам файлів і стандартизованим хелперам. Компоненти залишаються дрібнозернистими, але не розпадаються на мікроскопічні одиниці, що ускладнюють навігацію; композиції узагальнюють повторювану поведінку, а сервіси приховують інтеграційні деталі. Статичний аналіз TypeScript і лінтинг стилів зупиняють типові помилки на етапі збірки, а локальний HMR робить цикл правок і перевірок майже миттєвим. Завдяки інтеграції з Docker фронтенд отримує однакові версії Node та інструментарію на всіх машинах, що мінімізує «дрейф середовищ» і спрощує онбординг.

Усі ці рішення в сумі визначають характер клієнтської частини як чітко структурованого, типобезпечного та інтернаціоналізованого SPA, що коректно застосовує політики доступу, уніфікує мережеву взаємодію й повідомлення, а також дотримується вимог доступності та продуктивності.

Користувач отримує передбачуваний інтерфейс із швидким відгуком і стабільною поведінкою, а команда розробки — архітектуру, яку нескладно розширювати: додавання нових платіжних сценаріїв, адміністративних панелей чи локалей зводиться до локалізованих змін у сто́рах, маршрутах і ресурсах перекладів, без необхідності переглядати базові механізми застосунку. Саме така організація відповідає якості програмного забезпечення: відокремлення обов'язків, контроль варіантів на межах шарів, спостережуваність поведінки та доведена здатність системи масштабуватися без руйнівних змін.

3.4 Розробка архітектури та взаємодії системи між серверною і клієнт-серверною частинами

Архітектура системи спроєктована як сукупність відокремлених, але тісно узгоджених підсистем, де клієнтський односторінковий застосунок на Vue 3 взаємодіє з сервером на Laravel через стабільні REST-контракти. На рівні інфраструктури обидві частини розгортаються в контейнеризованому середовищі, де nginx виконує роль фронтового проксі й статичної роздачі зібраних фронтенд-артефактів, а php-fpm обробляє HTTP-запити до API. Сервісна шина для черг і метрик побудована навколо Redis, тоді як транзакційна складова опирається на MariaDB, а спостережуваність забезпечується через експозицію показників у форматі Prometheus з подальшою візуалізацією у Grafana. Така композиція дозволяє розділити обчислювальні та мережеві обов'язки між спеціалізованими контейнерами, зберігаючи при цьому простоту операційного керування за рахунок єдиної docker-мережі та декількох логічних сегментів. Схема роботи зображена на рис. 3.5.

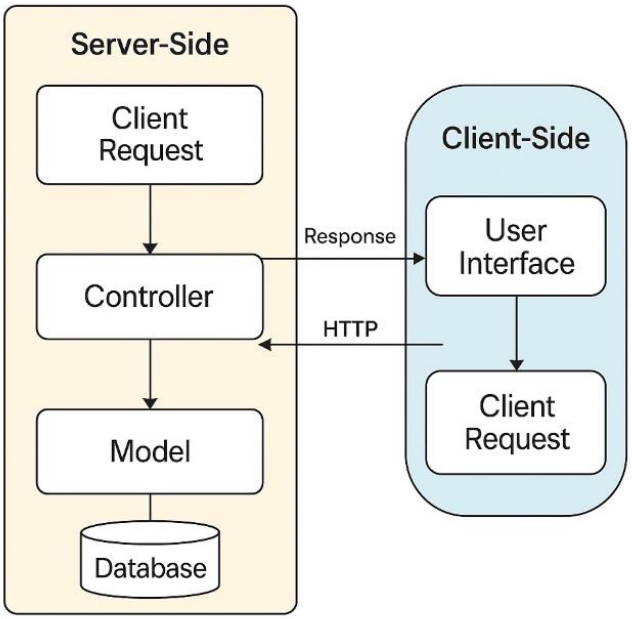


Рисунок 3.5 – Схема взаємодії систем.

Взаємодія між клієнтом і сервером побудована на строгій ролі серверної частини як джерела істини. Будь-яке змінення стану — від створення платежу до зміни ролей користувача — ініціюється викликом API і завершується відповіддю в уніфікованому JSON-форматі. Клієнт не зберігає дублікати критичних інваріантів і не виконує рішень, що мають безпекові наслідки; він лише відображає результати перевірених і підтверджених сервером операцій. Сталість контрактів підтримується ресурсним шаром Laravel: дані моделі ніколи не віддаються напряму, а серіалізуються через ресурси, які фіксують структуру полів і приховують службові атрибути. Таким чином, навіть при еволюції внутрішніх таблиць відповіді лишаються сумісними з клієнтом.

Контур автентифікації реалізовано як типова для SPA пара JWT-токенів. Після успішного логіна сервер повертає access-токен, який клієнт ін’єктує у заголовок Authorization у кожному подальшому запиті. Редиректи й допуск на маршрути керуються доотриманням профілю користувача і його ролей перед першою навігацією; клієнт не покладається на евристики інтерфейсу, а кожного разу підтверджує права на основі даних, що повернув сервер. Для захисту приватних розділів адміністративного інтерфейсу використовується окремий префікс маршрутизації й відокремлені політики доступу. Усі запити клієнта позначені Accept: application/json, що спрощує уніфікацію помилок і

обробку граничних станів, включно з простроченими токенами та невідповідністю дозволів.

Проєктна логіка побудована так, щоб зменшити латентність і при цьому не компрометувати консистентність. Операції, для яких час відповіді користувачу не критичний або результат може бути отриманий асинхронно, відвантажуються у черги: надсилання листів після реєстрації, формування агрегованих показників, технічні оновлення метрик для експорту Prometheus. Клієнт у таких сценаріях отримує миттєве підтвердження прийняття операції, тоді як справжнє виконання відбувається під наглядом Horizon, з політикою повторних спроб і фіксацією невдалих виконань у спеціалізованих таблицях. Це розвантажує HTTP-шар, стабілізує час відгуку і дає змогу масштабувати обробку вшир без втручання у клієнтські сценарії.

Взаємодія з платіжними провайдерами є показовим прикладом розмежування синхронних і асинхронних процесів. Клієнт формує запит на створення платежу, сервер перевіряє дані, мапує внутрішні статуси на загальні переліки й, за потреби, ініціює зовнішню транзакцію. Подальше надходження повідомлень від провайдера (успіх, помилка, колбек підтвердження) обробляється вебхуками і відправляється у чергу, де відбувається узгодження внутрішнього статусу, консистентне оновлення суміжних таблиць і, за потреби, нотифікація користувача. Клієнт бачить лише уніфікований результат через власні екрани історії та деталізації; він не взаємодіє напряму з третьою стороною, а всі переходи між станами верифікуються сервером. Така організація спрощує облік збоїв і повторів, а також робить поведінку інтерфейсу передбачуваною.

Рівень мережевих протоколів і політик доступу до API не допускає неоднозначностей. CORS налаштовано таким чином, щоб фронтенд, зібраний Vite та розміщений під тим самим доменом у production-режимі, взаємодіяв без зайвих preflight-витрат; у development-режимі дозволені походження відповідають локальним доменам і портам, визначеним у конфігураціях. Дроселювання запитів і захист від брутфорсу здійснюються на API-рівні; клієнт коректно інтерпретує статуси 429, 401 і 403 та відображає локалізовані

повідомлення без розкриття внутрішньої діагностики. Прикладні помилки і попередження уніфікуються ResponseService, завдяки чому клієнт оперує стабільним словником кодів і не будує логіку на рядкових повідомленнях.

Спостережуваність і керованість системи є частиною її архітектурних гарантій. Серверна частина експонує метрики у форматі Prometheus; серед них — показники черг, кількість користувачів у певних бізнес-станах, тривалість окремих операцій і частота помилок. Збирання метрик відділене від HTTP-викликів і відбувається планово або за подією, зберігаючи продуктивність API. Клієнтська частина при цьому залишається тонким клієнтом: вона не обчислює агрегати, а лише відображає дані, що отримує з API, або за потреби звертається до підготовлених серверних представлень. Це дозволяє не «вшивати» бізнес-аналітику в інтерфейс і не робити його залежним від структури внутрішніх таблиць.

Розмежування середовищ і версій також інтегроване в модель взаємодії. Внутрішня нотація версій для API (простір імен v1 у контролерах) дозволяє еволюціонувати контракти без руйнівних змін клієнтського коду, а фронтенд завдяки Vite отримує контент-хешовані артефакти, що усуває колізії кешів при оновленнях. Конфігурація доменів і базових шляхів до приватних і адміністративних ресурсів винесена у централізовані модулі клієнта, завдяки чому переключення між dev, staging і production не потребує правок у компонентах і зводиться до зміни змінних середовища на етапі збірки.

З точки зору цілісності даних обрана стратегія узгоджується з ACID-властивостями транзакційної СУБД і водночас допускає eventual consistency там, де користувач очікує швидкої реакції інтерфейсу, а не моментального агрегату. Для критичних шляхів — створення користувача, оформлення платежу, зміна дозволів — серверні операції атомарні й перевірені обмеженнями схеми і прикладними валідаторами; клієнт відображає проміжні стани лише як візуальні індикатори, не змінюючи фактичний стан без підтвердження з сервера. Для довідкових і аналітичних екранів допускається короткочасна розсинхронізація із подальшим автооновленням; це забезпечує чутливість інтерфейсу без компромісів із достовірністю.

Нарешті, важливою властивістю цієї архітектури є її прогнозована масштабованість. Збільшення навантаження на читання обслуговується горизонтальним масштабуванням веб-вузлів під nginx і реплікацією БД для read-only сценаріїв; зростання фонових задач — масштабуванням Horizon-пулів і черг на Redis. Клієнтська частина від таких змін не залежить: її взаємодія обмежена стабільними URL і однаковими форматами відповідей. Додавання нових доменних можливостей відбувається шляхом введення нових ресурсів і контролерів на сервері та локалізованих сторінок і сто́рів на клієнті; кордони між шарами не перетинаються, що зменшує вартість супроводу й ризик регресій.

У підсумку система демонструє цілісну архітектуру клієнт-серверної взаємодії, де фронтенд виступає керованим презентером, а бекенд — детермінованим виконавцем і арбітром бізнес-правил. Контракти чітко зафіксовані ресурсним шаром, безпека забезпечується JWT-контуром і політиками доступу, продуктивність — асинхронізацією довгих операцій, а спостережуваність — стандартизованими метриками. Таке поєднання гарантує передбачувану поведінку під навантаженням, простоту еволюції і наочність для подальшого академічного аналізу та практичного масштабування.

3.5 Розробка Docker середовища системи

Наведена система розгортається у контейнеризованому середовищі, де кожен технічний аспект — від веб-фронту до моніторингу — відокремлено у власний сервіс, але всі вони взаємодіють через заздалегідь визначені мережі та точки інтеграції. Базовим елементом є застосунок на Laravel, упакований у власний образ, що збирається з контексту проєкту та Dockerfile у каталозі docker/php/. Такий спосіб дає можливість зафіксувати версії PHP-розширень, керувати параметрами OPCache, включати потрібні системні пакети, а також акуратно під’єднати Composer ще на етапі білду, зменшуючи час «холодного»

старту контейнера. У процесі запуску середовища в контейнер застосунку примаплюється робочий каталог проєкту, а також підкладається .env із гілки docker/.env.docker безпосередньо у корінь Laravel. Це забезпечує розділення змінних середовища між локальним та продукційним режимами, зберігає відтворюваність збірки та унеможливлює випадкове змішування конфігурацій.

Взаємодія з користувачем відбувається через nginx, що працює у виділеному контейнері. Він виступає фронтовим проксі та HTTP-сервером статичних активів, а також термінальною точкою для маршруту /metrics. Конфігурація nginx примонтована з репозиторію, що дозволяє керувати правилами кешування артефактів Vite, маршрутизацією до PHP-FPM у контейнері застосунку та доступом до службових ендпойнтів без перезбирання образу. На рівні портів використано нестандартні значення (зокрема, 8081 для HTTP), що уникає конфліктів із системними службами Windows і спрощує паралельну роботу кількох проєктів. Усі HTTP-запити до API надходять у nginx і далі передаються сокетами до PHP-FPM у контейнері застосунку; SPA-артефакти фронтенду роздаються тим самим nginx, забезпечуючи єдиний домен і коректну роботу CORS.

Транзакційна складова покладена на MariaDB, що працює у власному контейнері з персистентним томом для збереження даних між перезапусками. Винос БД у окремий контейнер спрощує контроль ресурсів та ізолює можливі джерела деградації продуктивності. Для оперативної діагностики додано phpMyAdmin, який доступний на окремому порту й працює у тій самій мережі, що і MariaDB; ця ізоляція на рівні мережі дає змогу залишити панель керування поза загальнодоступним периметром, але водночас мати зручний інструмент для локальної розробки. Сеанси, кеш і черги використовують Redis у виділеному контейнері; такий поділ зменшує навантаження на БД, зокрема для короткоживучих даних і чергових повідомлень. Для спостереження за Redis піднято RedisInsight із власним томом — це забезпечує інспекцію ключів, TTL і продуктивності без проникнення в контейнери застосунку.

Асинхронні задачі реалізовано на Laravel Queues з бекендом Redis, а

диспетчеризацію та моніторинг виконує Horizon, запущений як окремий контейнер на тому ж образі, що і застосунок. Такий підхід дозволяє конфігурувати воркер-пули, пріоритети та окремі черги для різних доменів (платежі, метрики, пошта) без впливу на HTTP-процеси. Планувальник завдань винесено в окремий контейнер «scheduler», де Supervisor керує періодичним викликом php artisan schedule:run та іншими сервісними процесами. Це усуває залежність від cron хоста і робить планувальник відтворюваним у будь-якому середовищі. Взаємодія між цими контейнерами здійснюється виключно через внутрішні мережі Docker, що знижує поверхню атаки та унеможливлює випадкові зовнішні звернення до технічних портів.

Спостережуваність системи забезпечується через Prometheus і Grafana, розгорнуті окремими контейнерами у мережі моніторингу. Застосунок експонує метрики у форматі Prometheus на маршруті /metrics; nginx пропускає цей маршрут напряму до PHP-FPM або, залежно від конфігурації, роздає з кеша зі зваженим TTL, якщо метрики важкі для формування. Файл конфігурації Prometheus примонтований із репозиторію і містить джоб із таргетом на сервіс nginx; таким чином, пошуковик метрик регулярно збирає показники додатка, Horizon, кешів і бізнес-агрегатів. Grafana використовує власний том для збереження дашбордів і джерел даних, що дає змогу відтворювати панелі між машинами без повторної ручної конфігурації. Формування метрик на боці Laravel здійснюється поза контекстом HTTP-запитів через окремі джоби планувальника, що зберігає стабільність часу відгуку API та унеможливлює випадкове «провисання» інтерфейсу користувача.

Логічна сегментація мереж реалізована через чотири містки: додаткова мережа для застосунку й nginx, окрема — для БД і служб керування БД, окрема — для Redis і пов’язаних інструментів, і окрема — для моніторингу. Така топологія не лише поліпшує безпеку за рахунок мінімально необхідної видимості між сервісами, а й спрощує трасування збоїв. Наприклад, якщо спостерігається деградація швидкодії черг, достатньо інспектувати трафік і затримки лише в межах мережі Redis, не залучаючи компоненти БД або веб-

фронт. Крім того, мережна ізоляція полегшує поступовий перехід на розподілені інфраструктури: сервіси можна виносити на інші вузли або переналаштовувати таргети Prometheus без рефакторингу застосунку.

У середовищі Windows важливе значення мають правила монтування томів і налаштування WSL2-бекенда Docker Desktop. Пряме примаплювання робочої теки проєкту у контейнер забезпечує розробницьку зручність (гаряча перезбірка фронтенду, миттєве оновлення PHP-коду), але потребує уважного ставлення до прав доступу та кінців рядків. Для коректної роботи Artisan-команд і запису в storage та bootstrap/cache у Dockerfile або entrypoint передбачено ініціалізаційні команди, які встановлюють володіння від імені www-data та створюють відсутні підкаталоги. Питання CRLF/LF розв’язується через налаштування Git на рівні репозиторію; це унеможливлює поломку shell-скриптів усередині контейнерів. Для налагодження Xdebug у Windows застосовується host.docker.internal як адреса хоста, а зіставлення шляху забезпечується змінною PHP_IDE_CONFIG, що вказує на серверне ім’я, узгоджене з конфігурацією IDE; це дозволяє встановлювати точкові зупинки в коді, який фактично виконується у контейнері.

Порти сервісів підібрані таким чином, щоб не перетинатися зі стандартними портами ОС: 8081 для HTTP-доступу до сайту, 3307 для MariaDB, 8082 для phpMyAdmin, 6380 для Redis, 8001 для RedisInsight, 9090 для Prometheus і 3030 для Grafana. Ці значення фіксуються у файлах .env та docker-compose, що унеможливлює випадкові конфлікти при запуску інших локальних сервісів. Дані сервісів зберігаються у виділених іменованих томах — зокрема, для MariaDB і Grafana — що полегшує резервне копіювання та повернення до попередніх станів середовища без впливу на файлову систему хоста. Додатково конфігурація Supervisor, що примаплюється до контейнера планувальника, дозволяє централізовано керувати підпроцесами, включно з політиками рестарту та лог-ротацією, не покладаючись на системні демони хоста.

Безпековий контур у контейнеризованому середовищі складається з кількох шарів. Доступ до бази даних і Redis дозволений лише з внутрішніх

мереж; зовнішні порти відкриті лише для nginx та окремих інструментів розробника. Облікові записи на ВД та паролі визначаються у .env і не потрапляють до образів або контрольованих Git'ом файлів; зокрема, локальні значення паролів і ключів JWT ізольовані у .env.docker. У production-сценаріях ці секрети надаються через зовнішні секрети Docker або змінні середовища CI/CD, що усуває ризик їх витоку у публічні репозиторії. Таблиці черг і журналів невдалих завдань зберігають мінімально необхідний обсяг інформації; вміст чутливих полів, якщо він присутній у payload'ах, маскується під час логування. Prometheus збирає лише агрегати, придатні для спостереження продуктивності та бізнес-показників, уникаючи персональних даних.

Операційні практики передбачають використання профілів Compose для розмежування локальної та повної (наближеної до продукції) збірок. У першій конфігурації допускається гаряча перезбірка фронтенду та відкриті панелі керування на нестандартних портах; у другій — акцент на стабільність і мінімальний відкритий периметр. Такий поділ дозволяє зберігати один набір конфігурацій і водночас забезпечувати коректну поведінку під час автоматизованих деплоїв. Взаємодія з GitLab CI/CD або іншим оркестратором зводиться до команд compose build і compose up -d з передачею потрібних змінних середовища; база даних і артефакти моніторингу при цьому не втрачають стан завдяки іменованим томам.

У підсумку розроблене Docker-середовище надає відокремлені, але погоджені між собою контури виконання для веб-фронту, застосунку, БД, кешів, асинхронних процесів і моніторингу. Така схема мінімізує взаємні впливи між сервісами, забезпечує відтворюваність і прогнозованість поведінки під навантаженням, а також значно спрощує онбординг нових розробників: для початку роботи достатньо мати Docker Desktop з WSL2, клонувати репозиторій і підняти середовище однією командою. Архітектурно це відповідає вимогам сучасних корпоративних систем: чітке розділення обов'язків, контроль меж взаємодії через мережеві політики, стандартизована спостережуваність і можливість масштабувати кожен компонент незалежно

від решти.

3.6 Висновки до розділу

Даний розділ присвячений комплексному опису архітектури, структури та принципів взаємодії основних компонентів розробленої системи. У процесі дослідження було створено повноцінне середовище програмного забезпечення, яке складається з клієнтської, серверної та інфраструктурної частин, побудованих на сучасних технологіях веб-розробки й організованих за принципами модульності, масштабованості та безпеки. Система реалізує класичну клієнт-серверну модель, де кожен рівень виконує чітко визначені функції, а взаємодія між ними відбувається через стандартизовані інтерфейси.

Серверна частина спроєктована з використанням фреймворку Laravel 11, що ґрунтується на архітектурному шаблоні MVC. Такий підхід забезпечив логічне розмежування коду за рівнями даних, бізнес-логіки та подання, зменшив зв’язність компонентів і підвищив керованість системи. На рівні моделі реалізовано сутності користувачів, платежів і допоміжних елементів із суворими правилами цілісності та валідації; контролери координують обробку HTTP-запитів, а ресурсні класи формують уніфіковані JSON-контракти, які використовуються клієнтським застосунком. Бізнес-логіка винесена у сервіси, що ізолюють складні операції — авторизацію, роботу з платіжними системами, збір метрик — від контролерів і моделей. Для виконання асинхронних задач впроваджено черги Redis та диспетчер Horizon, що дозволяє розподіляти навантаження та забезпечує відмовостійкість обробки. Підсистема моніторингу реалізована на основі Prometheus і Grafana, що дає змогу контролювати стан черг, продуктивність і ключові бізнес-показники.

Клієнтська частина системи реалізована як односторінковий застосунок на Vue 3 з TypeScript, інтегрований у структуру Laravel. Архітектура SPA побудована навколо Pinia для керування станом, Vue Router для навігації та vue-i18n для багатомовності. Вона забезпечує реактивну взаємодію

користувача із сервером через REST-інтерфейси, використовуючи JWT-токени для автентифікації та ролі для контролю доступу. Усі мережеві запити централізовано реалізовано через сервіс Axios із уніфікованим обробленням помилок, що гарантує стабільну комунікацію між клієнтом і бекендом. Використання Tailwind CSS та адаптивних компонентів надало інтерфейсу цілісності, узгодженості та можливості швидкої модифікації без порушення єдиного стилю.

Взаємодія між клієнтською й серверною частинами організована за принципом REST API із суворим дотриманням контрактів. Сервер виступає єдиним джерелом істини: усі зміни даних, перевірки й бізнес-рішення відбуваються на його стороні, тоді як клієнт лише відображає результати. Авторизація реалізована за допомогою JWT, що дозволяє розмежовувати доступ до приватних та адміністративних ресурсів. Критичні операції виконуються атомарно в межах транзакцій, а допоміжні процеси — асинхронно у чергах, що забезпечує високу продуктивність і стабільність системи навіть під значним навантаженням. Уся аналітика та метрики збираються на сервері й передаються в систему моніторингу без залучення клієнтської частини, що підвищує безпеку та узгодженість даних.

Інфраструктурна частина реалізована на основі Docker і складається з набору контейнерів, кожен із яких відповідає за окрему підсистему: nginx як фронтовий веб-сервер, php-fpm із Laravel-застосунком, MariaDB як основна база даних, Redis для кешування й черг, Horizon та Scheduler для фонових процесів, Prometheus і Grafana для моніторингу. Розділення сервісів за мережами — додаткова, базова, Redis-та моніторинг-мережа — ізолює компоненти, підвищує безпеку та спрощує адміністрування. Така контейнеризація забезпечує відтворюваність середовища, легке розгортання в різних системах і можливість незалежного масштабування кожного сервісу без втручання у структуру застосунку.

У результаті створено повноцінне, відтворюване та масштабоване середовище, у якому кожен компонент виконує чітко визначену функцію. Серверна частина відповідає за бізнес-логіку та стабільність даних, клієнтська

— за інтерактивність і зручність взаємодії користувача, інфраструктурна — за надійність, контроль і спостережуваність. Система демонструє коректне застосування принципів розділення обов’язків, модульності та багаторівневої взаємодії, що відповідає вимогам до сучасних інформаційних систем і створює основу для подальшого розвитку та впровадження у виробниче середовище.

4 ЕКОНОМІЧНИЙ РОЗДІЛ

4.1 Актуальність соціальних мереж та перспективи їх використання

Як було зазначено у попередніх розділах цієї роботи, сьогодні питання автоматизації фінансових процесів, контролю витрат і забезпечення достовірності даних стають ключовими для ефективної діяльності підприємств і організацій. Значна кількість бізнес-процесів переходить у цифрове середовище, що потребує створення гнучких та надійних клієнт–серверних систем, здатних забезпечувати високу швидкість обробки інформації, стабільність і безпеку фінансових операцій.

У межах виконання магістерської кваліфікаційної роботи було розроблено автоматизовану клієнт–серверну систему фінансового обліку та моніторингу, яка поєднує серверну частину, створену на базі фреймворку Laravel 11, і клієнтський інтерфейс на Vue 3. Система функціонує у контейнеризованому середовищі Docker та використовує Redis для кешування, Horizon для контролю асинхронних процесів і Prometheus / Grafana для моніторингу продуктивності. Така архітектура дозволяє досягти високої надійності, масштабованості й простоти супроводу.

Головною метою розробки стало підвищення ефективності фінансового контролю, прозорості звітності та швидкості взаємодії користувача з даними. Система реалізує механізми автоматичного обліку платежів, формування звітів, контролю регулярних транзакцій та аналітики у реальному часі.

Для визначення рівня комерційного потенціалу розробленої системи було проведено її технологічний аудит, спрямований на оцінку технічної досконалості, інноваційності та практичної придатності. До експертного аналізу було залучено трьох фахівців у галузі інформаційних технологій: д.т.н., професора Паламарчука Є.А., к.т.н., доцентів Маслія Р.В. та Сторчака В.Г.

Оцінювання комерційного потенціалу створеної клієнт–серверної системи автоматизованого фінансового обліку здійснювалося за критеріями,

наведеними у таблиці 4.1.

Таблиця 4.1 – Критерії оцінювання рівня комерційного потенціалу будь-якої розробки та їх бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Критерій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздат-ність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
Ринкові перспективи					
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатацій-них витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів

Продовження таблиці 4.1

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Критерій	0	1	2	3	4
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років

Продовження таблиці 4.1

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Критерій	0	1	2	3	4
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Запрошені експерти оцінили розробку так, як це наведено в таблиці 4.2.

Таблиця 4.2 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали експерта		
	Паламарчук Є.А.	Маслій Р.В.	Сторчак В.Г.
	Бали, виставлені експертами:		
1	3	3	3
2	3	3	4
3	3	3	3
4	3	3	3
5	3	3	4
6	3	4	3
7	4	3	3
8	3	4	3
9	3	3	4
10	4	4	3
11	3	4	3
12	3	3	3
Сума балів	$СБ_1 = 38$	40	39

Середньоарифметична сума балів становила:

$$\overline{СБ} = \frac{\sum_{1}^{3} Б_i}{3} = \frac{38+40+39}{3} = \frac{117}{3} = 39$$

Загальний рівень комерційного потенціалу будь-якої розробки можна встановити, керуючись таблицею 4.3.

Таблиця 4.3 – Рівні технічного та комерційного потенціалу розробки

Середньоарифметична сума балів, розрахована на основі висновків експертів	Рівень технічного та комерційного потенціалу розробки
0 – 10	Низький
11 – 20	Нижче середнього
21 – 30	Середній
31 – 40	Вище середнього
41 – 48	Високий

Керуючись рекомендаціями таблиці 4.3, можна зробити висновок, що розроблена клієнт-серверна система автоматизованого фінансового обліку, оцінена запрошеними експертами у 39 балів, має технічний рівень та комерційний потенціал, які слід вважати суттєво «вище середнього».

Такий результат зумовлений поєднанням зрілої архітектури та практичної цінності функціоналу: система забезпечує повний цикл роботи з фінансовими операціями — від реєстрації та підтвердження платежів до формування звітів і контролю регулярних транзакцій — із чітким розмежуванням доступу за ролями та токен-аутентифікацією для веб-клієнта. Реалізація на базі Laravel і Vue дає змогу гарантувати стабільність серверної логіки та реактивність інтерфейсу, а контейнеризація в Docker спрощує розгортання і масштабування, забезпечуючи відтворюваність середовища. Вбудовані механізми асинхронної обробки через черги з моніторингом у Horizon, а також спостереження за ключовими показниками продуктивності та бізнес-метриками завдяки зв’язці Prometheus/Grafana підсилюють операційну

надійність рішення.

З практичного боку, система дозволяє оперативно фіксувати та узгоджувати платежі, відслідковувати їх статуси у реальному часі, отримувати сповіщення про події, що впливають на фінансові показники користувачів, і відразу відображати консолідовані дані у вигляді аналітичних панелей. Сукупність цих властивостей підтверджує доцільність подальшої апробації продукту в реальних умовах експлуатації та свідчить про його перспективність для комерційного використання.

4.2 Розрахунок витрат на розроблення сервісу автоматизованої системи обліку фінансів

Під час виконання даної роботи було зроблено такі витрати:

1. Основна заробітна плата виконавців $З_о$:

$$З_о = \frac{М}{Т_р} \cdot t \text{ грн}, \qquad (4.1)$$

де М – місячний посадовий оклад конкретного виконавця, грн;

У 2025 році величини посадових окладів науковців знаходиться в межах (15000…25000) грн/місяць;

$Т_р$ – число робочих днів в місяці; приймемо $Т_р$ = 25 днів.

Розрахунки основної заробітної плати виконавців зведемо до таблиці 4.4.

Таблиця 4.4 – Розрахунок основної заробітної плати виконавців (розробників)

Найменування посади виконавця	Місячний посадовий оклад, грн	Оплата за робочий день (або за годину), грн	Число днів роботи	Витрати на оплату праці, грн	Примітка
1. Науковий керівник магістерської кваліфікаційної роботи	22 000 грн	880 грн	2 дні	≈ 4 400 грн	Консультації, перевірка структури, рецензування
2. Студент-розробник-магістрант	18 000 грн	720 грн	90 днів	64 800 грн	Розробка серверної та клієнтської частини
3. Консультант з економічної частини	17 000 грн	810 грн	1 день	1 360 грн	Допомога у фінансових розрахунках
4. Консультант із програмної частини (DevOps / бекенд)	20 000 грн	800 грн	6 днів	4 800 грн	Консультації щодо CI/CD, Docker, Prometheus, Grafana
5. Консультант з дизайну / UI-UX	16 000 грн	640 грн	4 дні	2 560 грн	Рекомендації щодо інтерфейсу, UX-навігації, адаптивності
6. Інші консультанти	15000 грн				
Всього				77 920 грн грн	

2. Додаткова заробітна плата виконавців Зд розраховується як (10…12)% від величини основної заробітної плати виконавців, тобто:

$$З_д = (0,1 ... 0,12) \cdot З_о \quad . \qquad (4.2)$$

Для даного випадку отримаємо:

$$З_д = 0,1075 \cdot 77\,920 = 8376,4 \approx 8376 \text{ грн.}$$

3. Нарахування на заробітну плату $Н_{зп}$ розраховуються за формулою:

$$Н_{ЗП} = (З_о + З_д) \cdot \frac{\beta}{100}, \quad (4.3)$$

де $З_о$ – основна заробітна плата виконавців, грн;

$З_д$ – додаткова заробітна плата виконавців, грн;

β – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування; β = 22%.

Тоді отримаємо:

$$Н_{зп} = (77\ 920+8376) \cdot 0{,}22 = 18985.12 \approx 18985 \text{ грн.}$$

4. Витрати на матеріали М розраховуються по кожному виду матеріалів:

$$М = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i - \sum_{1}^{n} В_i \cdot Ц_в \text{ грн}, \quad (4.4)$$

де $Н_i$ – витрати матеріалу *i*-го найменування, кг; $Ц_i$ – вартість матеріалу *i*-го найменування, грн/кг; $К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); $В_i$ – маса відходів матеріалу *i*-го найменування, кг; $Ц_в$ – ціна відходів матеріалу *i*-го найменування, грн/кг; n – кількість видів матеріалів.

5. Витрати на комплектуючі К розраховуються за формулою:

$$К = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i \text{ грн}, \quad (4.5)$$

де $Н_i$ – кількість комплектуючих *i*-го виду, шт.; $Ц_i$ – ціна комплектуючих *i*-го виду, грн; $К_i$ – коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15); n – кількість видів комплектуючих.

За аналогією з іншими розробками вартість всіх використаних матеріальних ресурсів становить приблизно 1475 грн.

6. Амортизацію А обладнання, комп'ютерів та приміщень А можна розрахувати за формулою:

$$А = \frac{Ц \cdot Н_а}{100} \cdot \frac{Т}{12} \text{ грн}, \quad (4.6)$$

де Ц – загальна балансова вартість основних засобів, грн;

$Н_а$ – річна норма амортизаційних відрахувань: $Н_а$ = (2...25)%;

Т – термін, використання обладнання, приміщень тощо, місяці.

Зроблені розрахунки зведено до таблиці 4.5.

Таблиця 4.5 – Розрахунок амортизаційних відрахувань

Найменування обладнання, приміщень тощо	Балансова вартість, грн	Норма амортиза-ції, %	Термін використання, міс.	Величина амортизаційних відрахувань, грн
1. Персональні комп'ютери, принтери тощо	60 000 грн	25 %	4 (≈ 16.6 %)	1 650,00 грн
2. Монітор 27″ (2 шт.)	24 000 грн	25 %	4 (≈ 16.6 %)	660,00 грн
3. Мережеве обладнання (роутер, світч, UPS)	8 000 грн	20 %	4 (≈ 16.6 %)	528,00 грн
4. Приміщення кафедри та факультету	25 000 грн	2,5 %	4 (≈ 16.6 %)	26,04 грн
5. Ліцензійне програмне забезпечення	12 000 грн	30 %	4 (≈ 16.6 %)	1 188,00 грн
6. Допоміжне офісне обладнання	6 000 грн	30 %	4 (≈ 16.6 %)	594,00 грн
Всього				А = 4 646,04 ≈ 4 646

7. Витрати на силову електроенергію $В_е$ розраховуються за формулою:

$$В_е = \frac{В \cdot П \cdot Ф \cdot К_п}{К_д}, \quad (4.7)$$

де В – вартість 1 кВт-год. електроенергії, в 2025 р. В ≈ 4,5 грн/кВт; П – установлена потужність обладнання, кВт; П = 0,85 кВт; Ф – фактична кількість годин роботи обладнання, годин.

Приймемо, що Ф = 275 годин;

$К_п$ – коефіцієнт використання потужності; $К_п < 1$ =

0,91. $К_д$ – коефіцієнт корисної дії, $К_д$ = 0,83.

Тоді витрати на електроенергію складуть:

$$В_е = \frac{В \cdot П \cdot Ф \cdot К_п}{К_д} = \frac{4{,}5 \cdot 0{,}85 \cdot 275 \cdot 0{,}91}{0{,}83} = 1153{,}26 \approx 1154 \text{ грн.}$$

8. Інші витрати $В_{ін}$ можна прийняти як (50…300)% від основної заробітної плати виконавців, тобто:

$$В_{ін} = К_{ін} \cdot З_о = (0{,}5..3{,}0) \cdot З_о. \quad (4.8)$$

Для нашого випадку домовимося, що $К_{ін}$ = 1,00. Тоді:

$В_{ін}$ = (1,0 · 77 920) = 77 920 грн.

9. Сума всіх попередніх статей витрат дає нам витрати на виконання етапу роботи безпосередньо розробником-магістрантом – В.

Для нашого випадку:

В = 77920 + 8376 + 18985 + 1475 + 4646 + 1154 + 77 920 = 190 476 грн.

10. Розрахунок загальних витрат на розробку та остаточне доопрацювання виконаної роботи здійснюється за формулою:

$$ЗВ = \frac{В}{\beta}, \quad (4.9)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Оскільки дана розробка ще потребує деякого дуже незначного доопрацювання, то можна прийняти, що β ≈ 0,8.

Тоді: ЗВ = 190 476 / 0.8 = 238 095 грн або приблизно 238 тисячі грн.

Тобто прогнозовані витрати на розроблення та остаточне доопрацювання сервісу автоматизованої системи обліку фінансів, становлять приблизно 238 тисячі грн.

4.3 Розрахунок економічного ефекту від можливої комерціалізації даної розробки

Аналіз сучасного ринку програмних рішень для фінансового обліку,

контролю та автоматизації бізнес-процесів показує, що розроблена клієнт-серверна система на базі Laravel та Vue має значні перспективи розвитку й комерційного успіху. Незважаючи на високу конкуренцію у сфері бухгалтерських та фінансових SaaS-продуктів (таких як Odoo, QuickBooks, Finmap тощо), дана розробка відрізняється гнучкістю архітектури, розширеними можливостями моніторингу, модульністю та адаптивністю під потреби користувача.

Система орієнтована на середній та малий бізнес, державні та громадські організації, які потребують зручного інструменту для ведення фінансового обліку, аналітики платежів, управління підписками та моніторингу витрат у реальному часі. Завдяки інтеграції інструментів візуалізації (Grafana), моніторингу (Prometheus), розмежуванню прав доступу (Spatie Permissions) та підтримці багатомовності (Spatie Translatable), продукт має потенціал вийти на міжнародний рівень.

За попередніми оцінками, очікувана кількість користувачів системи може сягнути 100 тисяч осіб протягом перших трьох років експлуатації. При середній вартості підписки $5 на місяць (або $60 на рік) орієнтовний річний дохід складе $10000 \cdot \$60 = \$600000$

Додаткові прибутки можуть формуватися через розміщення партнерської реклами, інтеграції з фінансовими сервісами та розширені аналітичні модулі.

З урахуванням прогнозованого зростання кількості користувачів на 10% щороку та зростання вартості підписки на 5% щороку, можна очікувати, що протягом трирічного циклу використання (2025–2027 рр.) система генеруватиме стабільне зростання доходу.

Тоді очікуваний дохід Д від функціонування розробленої мобільної клієнт-серверної системи соціальної мережі становитиме:

2026-й рік: $Д_{2026}= 2400 \cdot 1{,}05 \cdot 10000 \cdot 1{,}1=27\ 720\ 000$ грн$\approx$ 28 млн грн;
а зростання доходу $\Delta Д_{2026} = 28\text{-}24 = 4$ млн грн.

2027-й рік: $Д_{2027} = 2520 \cdot 1{,}05 \cdot 11000 \cdot 1{,}1 = 32\ 016\ 600 \approx 32$ млн грн;

а зростання доходу $\Delta Д_{2027} = 32 - 28 = 4$ млн грн.

2028-й рік: $Д_{2028} = 2646 \cdot 1,05 \cdot 12100 \cdot 1,1 = 36\,979\,173 \approx 37$ млн грн;

а зростання доходу $\Delta Д_{2028} = 37\text{-}32 = 5$ млн грн.

Можливе збільшення чистого прибутку $\Delta П_i$, що його може отримати потенційний інвестор від комерціалізації даної розробки становитиме:

$$\Delta П_i = \sum_{1}^{n} \Delta Д_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\upsilon}{100}), \quad (4.10)$$

де $\Delta Д_i$ – збільшення величини доходу у кожному році;

λ – коефіцієнт, який враховує сплату податку на додану вартість; $\lambda = 0,8333$;

ρ – коефіцієнт, який враховує рентабельність продукту. Рекомендується приймати $\rho = (0,2...0,5)$; візьмемо $\rho = 0,5$;

υ – ставка податку на прибуток. У 2026-2028 роках $\upsilon = 18\%$.

Тоді зростання чистого прибутку $\Delta П_1$ для потенційного інвестора протягом першого (2024) року може скласти:

$$\Delta П_1 = 28 \cdot 0,8333 \cdot 0,5 \cdot (1 - \frac{18}{100}) \approx 9,5 \quad \text{млн грн.}$$

Зростання чистого прибутку $\Delta П_2$ для потенційного інвестора протягом другого (2025) року може скласти:

$$\Delta П_2 = 32 \cdot 0,8333 \cdot 0,5 \cdot (1 - \frac{18}{100}) \approx 11,9 \quad \text{млн грн.}$$

Зростання чистого прибутку $\Delta П_3$ для потенційного інвестора протягом третього (2026) року може скласти:

$$\Delta П_3 = 37 \cdot 0,8333 \cdot 0,5 \cdot (1 - \frac{18}{100}) \approx 12,6 \quad \text{млн грн.}$$

За висновками експертів, теперішню вартість інвестицій PV, які можуть бути вкладені в придбання розробленого сервісу автоматизованої системи обліку фінансів становлять 12,6 млн грн, тобто: PV = 12,6 млн грн.

Далі розраховуємо абсолютний ефект від можливо вкладених в придбання розробленої мобільної клієнт-серверної системи соціальної мережі інвестицій $E_{абс}$:

$$E_{абс} = ПП - PV, \quad (4.11)$$

де ПП – приведена вартість зростання всіх можливих чистих прибутків потенційного інвестора, грн;

PV – теперішня вартість інвестицій, PV ≈ 12,6 млн грн.

У свою чергу, приведена вартість зростання всіх чистих прибутків ПП розраховується за формулою:

$$ПП = \sum_{1}^{T} \frac{\Delta П_i}{(1+\tau)^t} \quad (4.12)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких

виявляються результати виконаної та впровадженої роботи, грн;

T – період часу, протягом якого виявляються результати впровадженої розробки, роки; T = 3 роки;

τ – ставка дисконтування, за яку можна взяти щорічний рівень інфляції в країні. Для України в 2023 році приймемо ставку = 0,10 (10%);

t – період часу (в роках) від моменту початку розробки до моменту отримання потенційним інвестором чистих прибутків.

Тоді приведена вартість зростання всіх можливих чистих прибутків ПП, що їх може отримати потенційний інвестор від можливої комерціалізації даної розробки, складе:

$$ПП = \frac{9,5}{(1+0,10)^2} + \frac{11,9}{(1+0,10)^3} + \frac{12,6}{(1+0,10)^4} = 7,85 + 8,94 + 8,60 = 25,39 \text{ млн грн.}$$

Абсолютний ефект від можливої комерціалізації даної розробки складе:

$E_{абс}$ = 25,4 – 12,6 = 12,8 млн грн.

Далі розрахуємо відносну ефективність $E_в$ вкладених у дану розробку потенційних інвестицій:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \quad (4.13)$$

де $E_{абс}$ – абсолютний ефект вкладених інвестицій, $E_{абс}$ = 12,8 млн грн;

PV – теперішня вартість початкових інвестицій, PV = 12,6 млн грн; $T_ж$ – життєвий цикл використання даної розробки, роки. $T_ж$ = 4. Для даного випадку:

$$E_в = \sqrt[4]{1 + \frac{12,8}{12,6}} - 1 = \sqrt[4]{1 + 1,0159} - 1 = \sqrt[4]{2,0159} - 1 = 1,4 - 1 = 0,4 = 40\%.$$

Далі визначимо ту мінімальну дохідність, нижче за яку потенційний інвестор не буде зацікавлений вкладати кошти у комерціалізацію даної розробки. Мінімальна дохідність або мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau_{мін} = d + f, \quad (4.14)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках;

в 2026 році в Україні d = (0,10...0,12);

f – показник, що характеризує ризикованість вкладень;

зазвичай, величина f = (0,05...0,30), але може бути і значно більше.

Для даного випадку отримаємо:

$$\tau_{мін} = 0,10 + 0,20 = 0,30 \text{ або } \tau_{мін} = 30\%.$$

Оскільки величина $E_в$ = 40% > $\tau_{мін}$ = 30%, то потенційний інвестор (після проведення додаткових розрахунків) у принципі може бути зацікавлений у комерціалізації даної розробки.

Далі розраховуємо термін окупності коштів, які можуть бути вкладені у придбання та комерціалізацію розробленої мобільної клієнт-серверної системи соціальної мережі:

$$T_{ОК} = \frac{1}{E_В}. \quad (4.15)$$

Для даного випадку термін окупності можливих інвестицій $T_{ок}$ складе:

$$T_{ок} = \frac{1}{0,4} \approx 2,5 \text{ роки.}$$

Оскільки $T_{ок}$ < (3…5) років, то комерціалізація розробленої мобільної клієнт-серверної системи соціальної мережі в принципі є можливою.

Якщо рівень інфляції в країні зросте до 20%, то отримаємо:

$$ПП = \frac{9,5}{(1+0,20)^2} + \frac{11,9}{(1+0,20)^3} + \frac{12,6}{(1+0,20)^4} = 6,60 + 6,88 + 6,05 = 19,53 \text{ млн грн.}$$

Абсолютний ефект від можливої комерціалізації даної розробки складе:

$$E_{абс} = 19,5 – 12,6 = 6,9 \text{ млн грн.}$$

Далі розрахуємо відносну ефективність $E_в$ вкладених у дану розробку потенційних інвестицій:

$$E_В = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \quad (4.16)$$

де $E_{абс}$ – абсолютний ефект вкладених інвестицій, $E_{абс}$ = 6,9 млн грн;

PV – теперішня вартість початкових інвестицій, PV = 12,6 млн грн; $T_ж$ – життєвий цикл використання даної розробки, роки. $T_ж$ = 4. Для даного випадку:

$$E_в = \sqrt[4]{1 + \frac{6,9}{12,6}} - 1 = \sqrt[4]{1 + 0,5476} - 1 = \sqrt[4]{1,5476} - 1 = 1,114 - 1 = 0,114 \approx 11,4\%.$$

Оскільки величина $E_в$ = 11,4% < $\tau_{мін}$ = 30%, то потенційний інвестор може бути не зацікавлений у комерціалізації даної розробки.

Зроблені розрахунки наведено на рисунку 4.1.

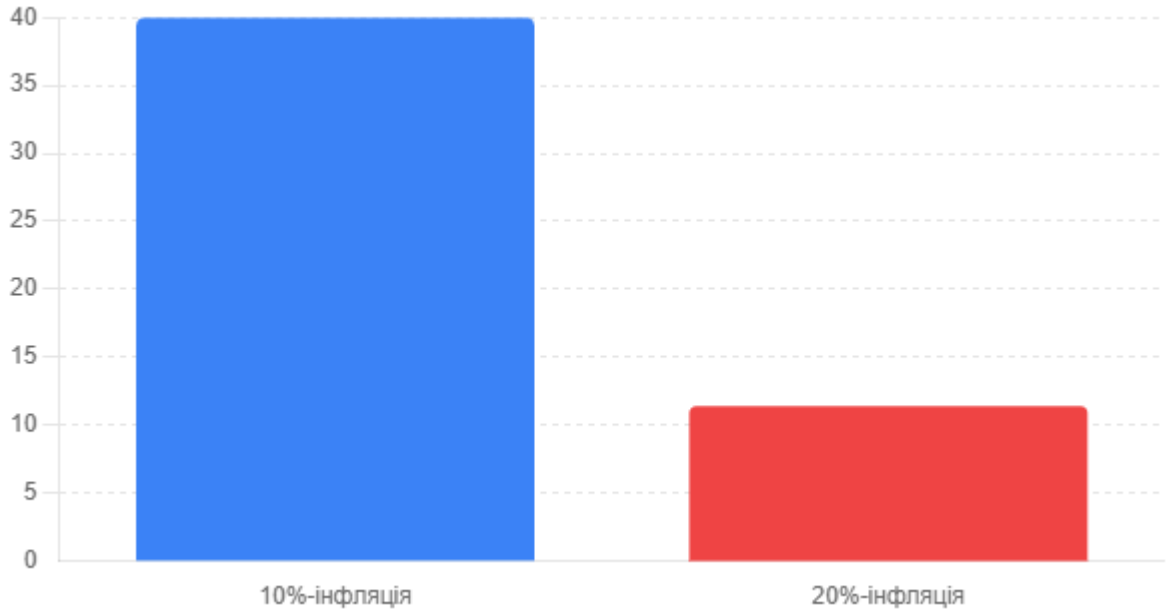


Рисунок 4.1 – Моделювання залежності величини внутрішньої дохідності потенційних інвестицій від рівня інфляції в країні

Аналіз діаграм на рисунку 4.1 показує, що при рівні інфляції в 10% величина внутрішньої дохідності інвестицій становить $E_в$ = 40%, що більше порогового значення $\tau_{мін}$ = 30% і тому комерціалізація даної розробки може бути доцільною. При рівні інфляції в 20% величина внутрішньої дохідності інвестицій, вкладених в комерціалізацію даної розробки, становить $E_в$ = 11.4%, що нижче порогового значення $\tau_{мін}$ = 30%, і тому комерціалізація даної розробки потенційним інвестором може бути проблематичною.

Остаточне рішення з цього питання потребує проведення додаткових розрахунків (можливо – зниження рівня ризикованості вкладень тощо).

Результати виконаної економічної частини магістерської кваліфікаційної роботи наведено у таблиці 4.6.

Таблиця 4.6 – Результати виконаної економічної частини магістерської кваліфікаційної роботи

Показники	Задані у ТЗ	Досягнуті у магістерській роботі	Висновок
1. Витрати на розроблення мобільної клієнт-серверної системи соціальної мережі	Не більше 200 тис. грн	238 тисяч грн	Практично досягнуто
2. Абсолютний економічний ефект від впровадження розробки, млн грн	не менше 10 млн грн	12,8 млн грн	Виконано
3. Внутрішня дохідність потенційних інвестицій, %	не менше 30%	40 %	Досягнуто
4. Термін окупності потенційних інвестицій, роки	до 3-х років	2,5 роки	Досягнуто

Таким чином, заплановані у технічному завданні основні техніко-економічні показники розробленої мобільної клієнт-серверної системи соціальної мережі, що базується на основі геолокації, практично виконані.

ВИСНОВКИ

У результаті виконання магістерської кваліфікаційної роботи досягнуто поставленої мети — створено автоматизовану клієнт–серверну систему фінансового обліку, яка забезпечує комплексну підтримку процесів зберігання, обробки, аналітики та контролю фінансових операцій у реальному часі.

Робота містить повний цикл розробки програмного забезпечення: від аналітичного обґрунтування архітектури та вибору технологічного стеку до впровадження контейнеризованого середовища й проведення технологічного аудиту готової системи.

На основі аналізу сучасних тенденцій розвитку інформаційних технологій було визначено, що ефективність облікових систем напряму залежить від рівня автоматизації, стабільності серверної інфраструктури та зручності користувацького інтерфейсу.

Враховуючи ці фактори, архітектура системи побудована за класичною клієнт–серверною моделлю з використанням фреймворку Laravel 11 для серверної частини та Vue 3 із TypeScript для клієнтського інтерфейсу. Така побудова забезпечує чітке розмежування логіки, даних і подання, а також високий рівень масштабованості та повторного використання коду.

На етапі проектування серверної частини реалізовано модулі аутентифікації, авторизації, керування ролями, системи обліку платежів і генерації звітів. Застосовано бібліотеки Spatie Laravel Permission, Laravel Sanctum, Tymon JWT Auth, що забезпечили надійний контроль доступу до даних і захист від несанкціонованих дій. Для асинхронної обробки подій використано механізм черг із моніторингом у Laravel Horizon, а для спостереження за станом системи — Prometheus та Grafana, які дозволяють отримувати аналітичні метрики продуктивності.

Клієнтська частина розроблена у вигляді односторінкового застосунку (SPA) з використанням Pinia для управління станом, Vue Router для навігації,

Axios для взаємодії з API та Tailwind CSS для уніфікованого стилю інтерфейсу. Завдяки реактивній архітектурі досягнуто високу швидкодію, узгодженість між компонентами та зручність користувацької взаємодії. Система підтримує багатомовність через vue-i18n та забезпечує повноцінну інтеграцію з серверною частиною через REST API.

Особлива увага приділена розробці інфраструктури розгортання. Побудоване Docker-середовище об’єднує контейнери з окремими службами: nginx, php-fpm, MariaDB, Redis, Horizon, Scheduler, Prometheus і Grafana. Такий підхід гарантує повну ізоляцію процесів, спрощує налаштування CI/CD і підвищує відтворюваність результатів у різних середовищах. Контейнеризація забезпечує зручне масштабування, спрощує підтримку й оновлення компонентів, а також зменшує ризики конфігураційних помилок.

Проведений технологічний аудит підтвердив відповідність розробленої системи сучасним вимогам до продуктивності, безпеки, масштабованості та стабільності. За результатами експертної оцінки система отримала 39 балів із можливої максимальної кількості, що відповідає рівню «вище середнього». Це свідчить про високий ступінь готовності продукту до практичного використання та подальшої комерціалізації.

Розроблена система має значний потенціал розвитку. Подальше вдосконалення може бути спрямоване на впровадження модулів аналітики на основі машинного навчання, інтеграцію з платіжними шлюзами, розширення інструментів візуалізації фінансових показників, а також розробку мобільної версії інтерфейсу. Отримані результати можуть бути використані як у реальних комерційних рішеннях, так і як навчальний приклад побудови комплексних веб-систем із високим рівнем автоматизації.

Таким чином, у магістерській роботі вирішено важливу прикладну задачу розроблення сучасної клієнт–серверної системи автоматизованого фінансового обліку, що поєднує гнучкість архітектури, надійність програмної реалізації та простоту масштабування. Система відповідає сучасним вимогам до інформаційних продуктів та демонструє високий рівень

технологічної зрілості, практичної цінності та потенціалу для подальшого розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гаврилюк С. М. Розроблення клієнт–серверних веб-орієнтованих інформаційних систем: методи, архітектура та інструментальні засоби // *Вісник Національного університету "Львівська політехніка"*. 2020. № 3. С. 45–53.

2. Мамалига Н. Є., Квєтний Р. Н. РОЗРОБКА АРХІТЕКТУРИ ТА АЛГОРИТМІВ СЕРВІСУ АВТОМАТИЗОВАНОЇ СИСТЕМИ ОБЛІКУ ФІНАНСІВ: Молодь в науці: дослідження, проблеми, перспективи, м. Вінниця, 22-23 червня 2025 р. Вінниця, 2025.

3. Коваленко О. В., Дорошенко А. М. *Інформаційні системи в управлінні*. Навчальний посібник. Київ: КНЕУ, 2019. 312 с.

4. Laravel Framework. Офіційна документація. URL: https://laravel.com/docs (дата звернення 14.10.2025).

5. Laravel Horizon. Моніторинг та керування чергами. URL: https://laravel.com/docs/11.x/horizon (дата звернення 14.10.2025).

6. Laravel Sanctum. SPA-аутентифікація. URL: https://laravel.com/docs/11.x/sanctum (дата звернення 14.10.2025).

7. Laravel Telescope. Налагодження та інспекція запитів. URL: https://laravel.com/docs/11.x/telescope (дата звернення 14.10.2025).

8. Laravel Pint. Статичний аналіз і стиль коду. URL: https://laravel.com/docs/11.x/pint (дата звернення 14.10.2025).

9. Laravel Sail. Локальне Docker-середовище. URL: https://laravel.com/docs/11.x/sail (дата звернення 14.10.2025).

10. Spatie Laravel Permission. Ролі та дозволи. URL: https://spatie.be/docs/laravel-permission (дата звернення 14.10.2025).

11. Spatie Laravel Translatable. Багатомовні поля. URL: https://spatie.be/docs/laravel-translatable (дата звернення 14.10.2025).

12. Tymon JWT Auth. Аутентифікація за JWT у Laravel. URL: https://jwt-auth.readthedocs.io/ (дата звернення 14.10.2025).

13. Barryvdh Laravel IDE Helper. Генерація підказок IDE. URL: https://github.com/barryvdh/laravel-ide-helper (дата звернення 14.10.2025).

14. PHPUnit. Документація фреймворку тестування PHP. URL: https://phpunit.de/documentation.html (дата звернення 14.10.2025).

15. Vue.js 3. Офіційна документація. URL: https://vuejs.org/ (дата звернення 14.10.2025).

16. Vue Router. Маршрутизація для Vue 3. URL: https://router.vuejs.org/ (дата звернення 14.10.2025).

17. Pinia. Управління станом у Vue. URL: https://pinia.vuejs.org/ (дата звернення 14.10.2025).

18. Vue I18n. Інтернаціоналізація у Vue. URL: https://vue-i18n.intlify.dev/ (дата звернення 14.10.2025).

19. Vite. Інструмент збірки фронтенду. URL: https://vitejs.dev/ (дата звернення 14.10.2025).

20. Tailwind CSS. Офіційна документація. URL: https://tailwindcss.com/docs (дата звернення 14.10.2025).

21. Axios. HTTP-клієнт для браузера і Node.js. URL: https://axios-http.com/docs/intro (дата звернення 14.10.2025).

22. SweetAlert2. Модальні діалоги. URL: https://sweetalert2.github.io/ (дата звернення 14.10.2025).

23. @vueuse/head. Керування <head> у Vue 3. URL: https://github.com/vueuse/head (дата звернення 14.10.2025).

24. bhPlugin Vue3 Datatable. Компонент таблиці для Vue 3. URL: https://www.bhplugin.com/vue3-datatable (дата звернення 14.10.2025).

25. TypeScript. Документація мови. URL: https://www.typescriptlang.org/docs/ (дата звернення 14.10.2025).

26. Docker. Офіційна документація. URL: https://docs.docker.com/ (дата звернення 14.10.2025).

27. Docker Compose. Документація. URL: https://docs.docker.com/compose/ (дата звернення 14.10.2025).

28. Nginx. Офіційна документація. URL: https://nginx.org/en/docs/ (дата звернення 14.10.2025).

29. Supervisor. Контроль процесів. URL: http://supervisord.org/ (дата звернення 14.10.2025).

30. Redis. Документація системи кешування та черг. URL: https://redis.io/docs/latest/ (дата звернення 14.10.2025).

31. RedisInsight. Інструмент для роботи з Redis. URL: https://docs.redis.com/latest/ri/ (дата звернення 14.10.2025).

32. MariaDB. Офіційна документація. URL: https://mariadb.org/documentation/ (дата звернення 14.10.2025).

33. phpMyAdmin. Документація. URL: https://www.phpmyadmin.net/docs/ (дата звернення 14.10.2025).

34. Prometheus. Система моніторингу. URL: https://prometheus.io/docs/ (дата звернення 14.10.2025).

35. Grafana. Документація. URL: https://grafana.com/docs/ (дата звернення 14.10.2025).

36. PromPHP Prometheus Client PHP. Бібліотека для PHP. URL: https://github.com/PromPHP/prometheus_client_php (дата звернення 14.10.2025).

37. Laravel Tinker : інтерактивне REPL-середовище для Laravel [Електронний ресурс]. URL: https://laravel.com/docs/11.x/tinker (дата звернення 14.10.2025).

38. FakerPHP/Faker : бібліотека для генерації тестових даних у PHP [Електронний ресурс]. URL: https://fakerphp.github.io/ (дата звернення 14.10.2025).

39. Laravel Pail : консольний інструмент для роботи з логами Laravel [Електронний ресурс]. URL: https://laravel.com/docs/11.x/pail (дата звернення 14.10.2025).

40. Mockery/Mockery : бібліотека для створення mock-об’єктів у PHP-тестуванні [Електронний ресурс]. URL: https://github.com/mockery/mockery

(дата звернення 14.10.2025).

41. Nuno Maduro Collision : інструмент для покращеного відображення помилок у CLI Laravel [Електронний ресурс]. URL: https://github.com/nunomaduro/collision (дата звернення 14.10.2025).

42. Moment.js : бібліотека для роботи з датами та часом у JavaScript [Електронний ресурс]. URL: https://momentjs.com/ (дата звернення 14.10.2025).

43. SweetAlert2 : бібліотека для створення модальних вікон та інтерактивних сповіщень [Електронний ресурс]. URL: https://sweetalert2.github.io/ (дата звернення 14.10.2025).

44. Vue Height Collapsible : компонент Vue 3 для створення анімованих прихованих блоків і секцій, що розгортаються [Електронний ресурс].
URL: https://www.npmjs.com/package/vue-height-collapsible (дата звернення 14.10.2025).

45. Vue3 Perfect Scrollbar : кастомний скролбар для Vue 3 з покращеною візуалізацією [Електронний ресурс]. URL: https://github.com/mercs600/vue3-perfect-scrollbar (дата звернення 14.10.2025).

46. Vue3 Popper : бібліотека для створення адаптивних спливаючих підказок і меню у Vue 3 [Електронний ресурс]. URL: https://github.com/Tyriar/vue3-popper (дата звернення 14.10.2025).

47. @intlify/unplugin-vue-i18n : плагін оптимізації локалізації Vue, забезпечує статичний імпорт і мінімізацію бандла [Електронний ресурс]. URL: https://github.com/intlify/unplugin-vue-i18n (дата звернення 14.10.2025).

48. @rollup/plugin-alias : плагін для створення коротких шляхів імпорту в проєктах на Vite та Rollup [Електронний ресурс]. URL: https://github.com/rollup/plugins/tree/master/packages/alias (дата звернення 14.10.2025).

49. @tailwindcss/forms : офіційний Tailwind CSS плагін для стилізації

формових елементів [Електронний ресурс]. URL: https://github.com/tailwindlabs/tailwindcss-forms (дата звернення 14.10.2025).

50. @tailwindcss/typography : плагін Tailwind CSS для оформлення текстових блоків і статей [Електронний ресурс]. URL: https://github.com/tailwindlabs/tailwindcss-typography (дата звернення 14.10.2025).

51. @types/node : типові визначення для середовища Node.js, необхідні при роботі з TypeScript [Електронний ресурс]. URL: https://www.npmjs.com/package/@types/node (дата звернення 14.10.2025).

52. @vitejs/plugin-vue : офіційний плагін для інтеграції Vue 3 у збирач Vite [Електронний ресурс]. URL: https://github.com/vitejs/vite/tree/main/packages/plugin-vue (дата звернення 14.10.2025).

53. Autoprefixer : PostCSS-плагін для автоматичного додавання вендорних префіксів і забезпечення кросбраузерності CSS [Електронний ресурс].
URL: https://github.com/postcss/autoprefixer (дата звернення 14.10.2025).

54. Concurrently : інструмент для паралельного запуску декількох npm-скриптів [Електронний ресурс]. URL: https://github.com/open-cli-tools/concurrently (дата звернення 14.10.2025).

55. Laravel Vite Plugin : офіційний плагін для інтеграції Vite у Laravel [Електронний ресурс].
URL: https://laravel.com/docs/11.x/vite (дата звернення 14.10.2025).

56. PostCSS : інструмент для трансформації CSS за допомогою JavaScript-плагінів [Електронний ресурс].
URL: https://postcss.org/ (дата звернення 14.10.2025).

57. Vite : високопродуктивний збирач фронтенду, орієнтований на

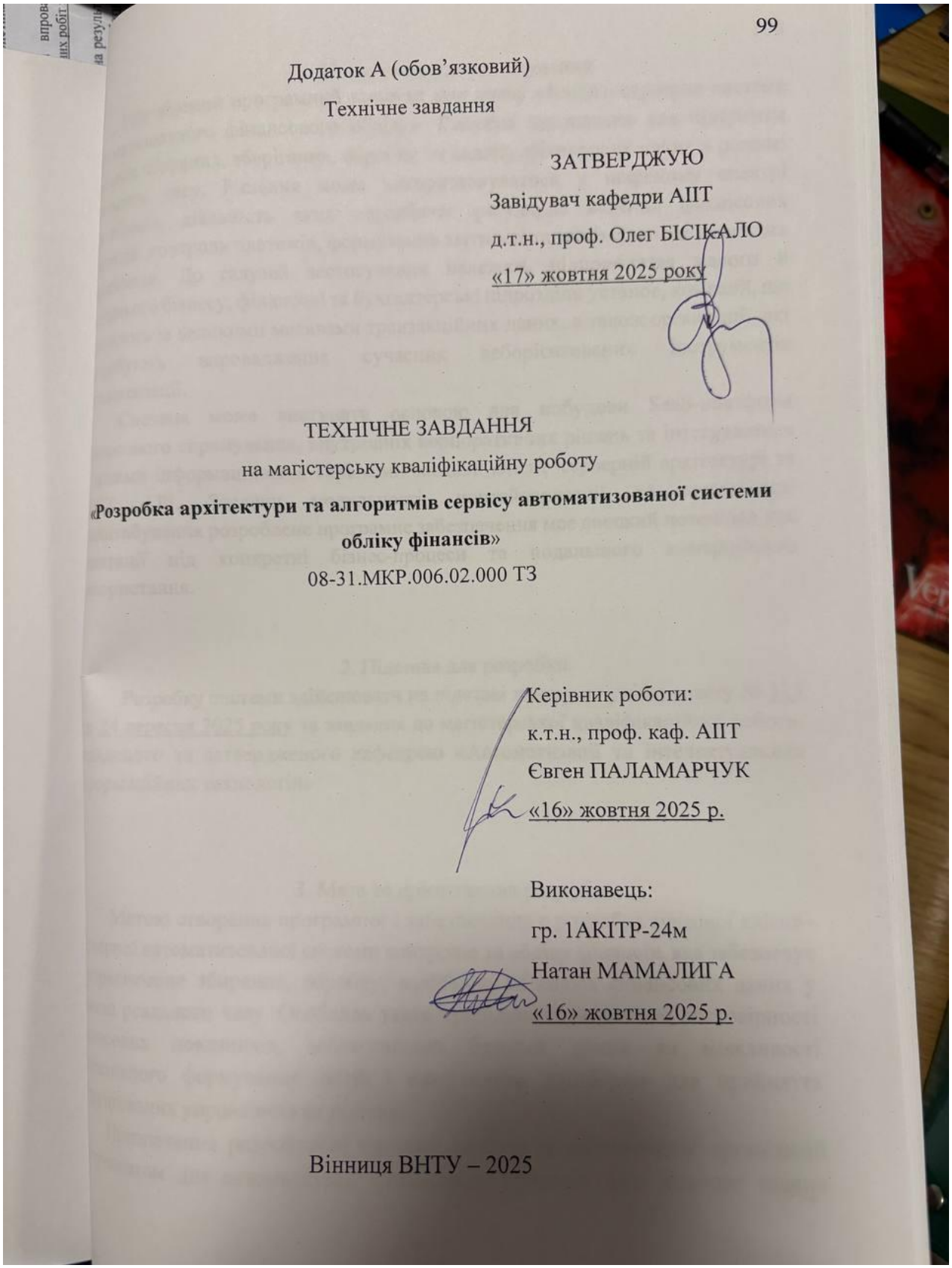
сучасні SPA та Vue 3 [Електронний ресурс]. URL: https://vitejs.dev/ (дата звернення 14.10.2025).

58. Vue TSC : TypeScript-інструмент для перевірки типів у Vue 3-компонентах [Електронний ресурс]. URL: https://github.com/vuejs/language-tools/tree/master/packages/vue-tsc (дата звернення 14.10.2025).

ДОДАТКИ

Додаток А (обов’язковий)

Технічне завдання



99

Додаток А (обов’язковий)

Технічне завдання

ЗАТВЕРДЖУЮ
Завідувач кафедри АІІТ
д.т.н., проф. Олег БІСІКАЛО
«17» жовтня 2025 року

ТЕХНІЧНЕ ЗАВДАННЯ
на магістерську кваліфікаційну роботу
«Розробка архітектури та алгоритмів сервісу автоматизованої системи обліку фінансів»
08-31.МКР.006.02.000 ТЗ

Керівник роботи:
к.т.н., проф. каф. АІІТ
Євген ПАЛАМАРЧУК
«16» жовтня 2025 р.

Виконавець:
гр. 1АКІТР-24м
Натан МАМАЛИГА
«16» жовтня 2025 р.

Вінниця ВНТУ – 2025

1. Назва та галузь застосування

Розроблений програмний продукт має назву «Клієнт–серверна система автоматизованого фінансового обліку». Система призначена для підтримки процесів збирання, зберігання, обробки та аналізу фінансових даних у режимі реального часу. Рішення може використовуватися у широкому спектрі організацій, діяльність яких передбачає регулярне ведення фінансових операцій, контроль платежів, формування звітності та моніторинг економічних показників. До галузей застосування належать підприємства малого й середнього бізнесу, фінансові та бухгалтерські підрозділи установ, компанії, що працюють із великими масивами транзакційних даних, а також організації, які потребують впровадження сучасних веборієнтованих інструментів автоматизації.

Система може виступати основою для побудови SaaS-платформ фінансового спрямування, внутрішніх корпоративних рішень та інтегруватися з іншими інформаційними сервісами завдяки клієнт–серверній архітектурі та REST API. Завдяки модульності, контейнеризації та можливості масштабування розроблене програмне забезпечення має високий потенціал для адаптації під конкретні бізнес-процеси та подальшого комерційного використання.

2. Підстава для розробки

Розробку системи здійснювати на підставі наказу по університету № 313 від 24 вересня 2025 року та завдання до магістерської кваліфікаційної роботи, складеного та затвердженого кафедрою «Автоматизації та інтелектуальних інформаційних технологій»

3. Мета та призначення розробки

Метою створення програмного забезпечення є розробка сучасної клієнт–серверної автоматизованої системи контролю та обліку фінансів, яка забезпечує централізоване збирання, обробку, валідацію та аналіз фінансових даних у режимі реального часу. Особлива увага приділена підвищенню достовірності фінансових показників, забезпеченню безпеки даних та можливості оперативного формування звітів і аналітичних дашбордів для прийняття обґрунтованих управлінських рішень.

Призначення розробленої системи полягає у забезпеченні організацій інструментом для автоматизації фінансових процесів, що дозволяє значно

скоротити час на ручну підготовку звітів і мінімізувати помилки, пов’язані з дублюванням чи некоректністю даних. Система підтримує багатомовність, гнучку рольову модель доступу, модуль автоматичного імпорту та перевірки даних, що робить її придатною для застосування у підприємствах із різною організаційною структурою.

Архітектура програмного забезпечення розроблена з урахуванням можливості подальшого масштабування та розширення, включно з поступовим переходом до мікросервісної моделі. Реалізовані механізми аудиту й підвищеної безпеки даних сприяють дотриманню нормативних вимог і забезпечують прозорість обробки фінансової інформації. Завдяки інтерактивним інструментам аналітики система є ефективним засобом підтримки управлінських рішень у фінансовій сфері.

4. Джерела розробки

1. Гаврилюк С. М. Розроблення клієнт–серверних веб-орієнтованих інформаційних систем: методи, архітектура та інструментальні засоби // *Вісник Національного університету “Львівська політехніка”*. 2020. № 3. С. 45–53.

2. Мамалига Н. Є., Квєтний Р. Н. РОЗРОБКА АРХІТЕКТУРИ ТА АЛГОРИТМІВ СЕРВІСУ АВТОМАТИЗОВАНОЇ СИСТЕМИ ОБЛІКУ ФІНАНСІВ: Молодь в науці: дослідження, проблеми, перспективи, м. Вінниця, 22-23 червня 2025 р. Вінниця, 2025.

3. Коваленко О. В., Дорошенко А. М. *Інформаційні системи в управлінні*. Навчальний посібник. Київ: КНЕУ, 2019. 312 с.

4. Laravel Framework. Офіційна документація. URL: https://laravel.com/docs (дата звернення 14.10.2025).

5. Laravel Horizon. Моніторинг та керування чергами. URL: https://laravel.com/docs/11.x/horizon (дата звернення 14.10.2025).

5. Показники призначення

5.1. Основні технічні характеристики системи

Функціональні можливості:

- Клієнт-серверна архітектура з використанням REST API.
- Серверна частина: Laravel 11, PHP 8.2, Horizon, Redis, MariaDB.
- Клієнтська частина: Vue.js 3, TypeScript, Pinia, Vue Router, Axios.
- Контейнеризація всіх компонентів у Docker (nginx, php-fpm, MariaDB, Redis, Horizon, Prometheus, Grafana).
- Підтримка ролей і прав доступу (Spatie Permission, Sanctum/JWT).
- Вбудована система кешування, черги для асинхронних операцій, автоматичний імпорт і валідація даних.
- Підсистема моніторингу продуктивності на базі Prometheus та Grafana.
- Модульність та можливість масштабування серверної інфраструктури.

5.2. Мінімальні системні вимоги

Для серверної частини (Docker-середовище):

- Процесор: 2 ядра або більше
- Оперативна пам’ять: 4 ГБ (рекомендовано 8 ГБ)
- Пам’ять на диску: від 10 ГБ для контейнерів, образів та БД
- Операційна система: Windows / Linux / macOS з підтримкою Docker
- Docker Engine 20+
- Docker Compose 2.0+

Для клієнтської частини (користувача):

- Сучасний браузер: Chrome, Firefox, Edge, Safari
- Підтримка JavaScript ES6+
- Роздільна здатність екрану від 1366×768

5.3. Вхідні дані

- Дані про фінансові операції (платежі, суми, дати, категорії).
- Дані користувачів (ролі, права доступу, персональна інформація).
- Дані з імпортованих файлів або зовнішніх джерел.
- Налаштування аналітичних параметрів (періоди, фільтри, групування).

- Інформація для формування звітів і дашбордів.

5.4 Результати роботи програми

- базова інформацію про користувача;
- роль, активність та статус доступу;
- отримання повного переліку транзакцій;
- перегляд детальної інформації про кожний платіж.
- переглядати власну історію платежів;
- визначати статус кожної операції;
- здійснювати повторні дії (наприклад, оплату або перегляд деталей).
- формування запиту на оплату;
- валідацію введених даних;
- взаємодію з платіжним сервісом;
- запис результату транзакції у базу даних;
- прив’язку платежу до конкретного користувача.
- структуровані записи в базі даних для користувачів та платежів;
- історію транзакцій;
- логування поточних подій та змін у системі.
- використання ролей і прав доступу;
- авторизації через токени;
- валідації всіх запитів.

6. Економічні показники

До економічних показників входять:

- витрати на розробку – до 250 тис. грн.
- узагальнений коефіцієнт якості розробки – більше 2-х
- термін окупності – до 3х років

7. Стадії розробки:

1. Розділ 1 «Загальні відомості» має бути виконаний до 02.10.2025 р.

2. Розділ 2 «Вибір технологій для реалізації системи» має бути виконаний до 15.10.2025 р.

3. Розділ 3 «Розробка програмного забезпечення» має бути виконаний до 07.11.2025 р.

4. Розділ 4 «Економічний розділ» має бути виконаний до 17.11.2025 р.

8. Порядок контролю та приймання

1. Рубіжний контроль провести до 14.11.2025.

2. Попередній захист магістерської кваліфікаційної роботи провести до 02.12.2025.

3. Захист магістерської кваліфікаційної роботи провести в період з 15.12.2025 р. до 19.12.2025 р.

Додаток Б (обов’язковий)

Ілюстративна частина

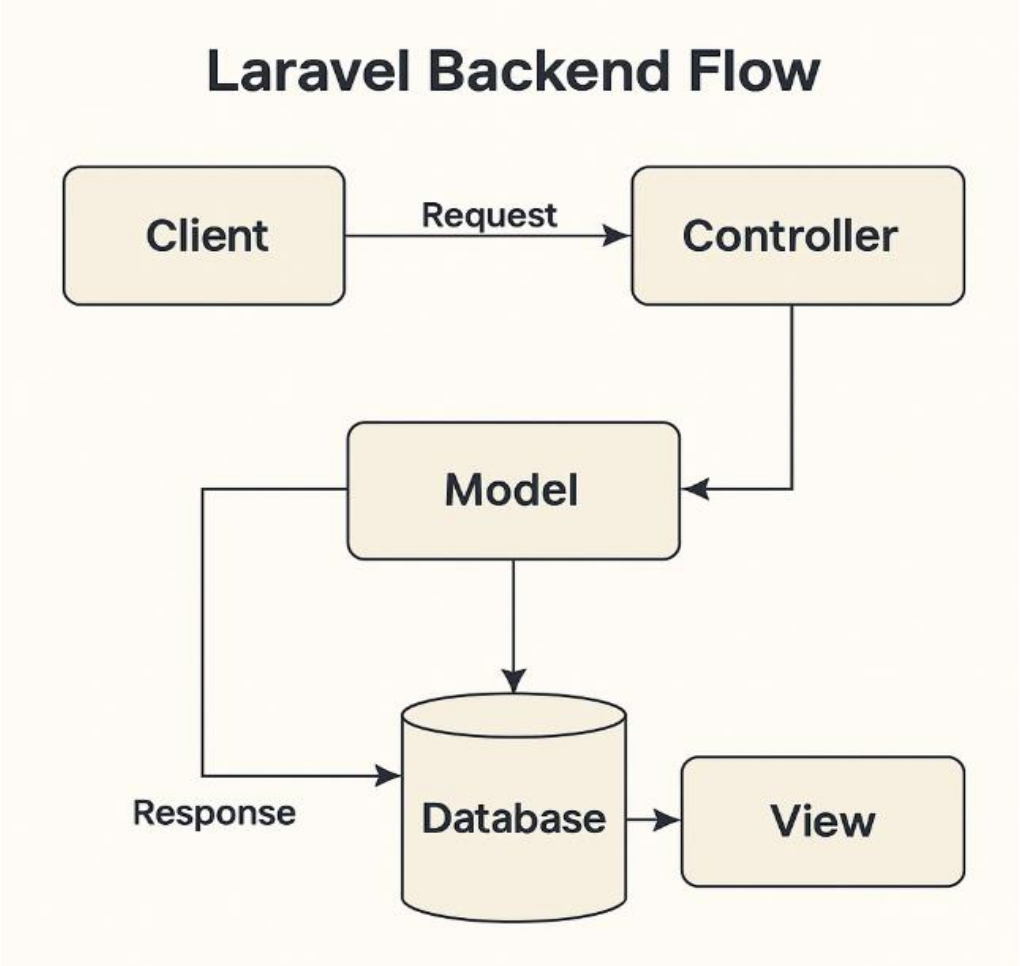


Рисунок Б.1 – Схема роботи серверної частини

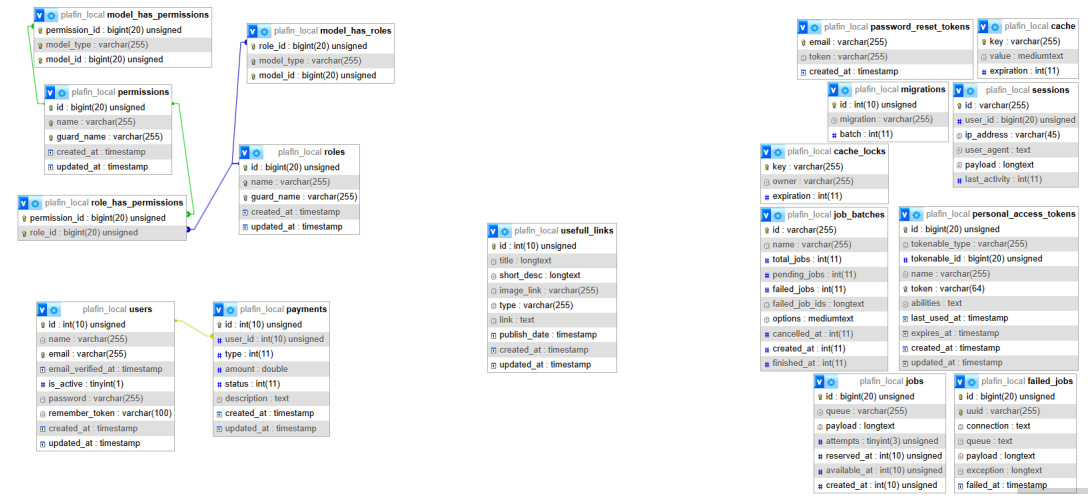


Рисунок Б.2 – ER-модель схеми бази даних

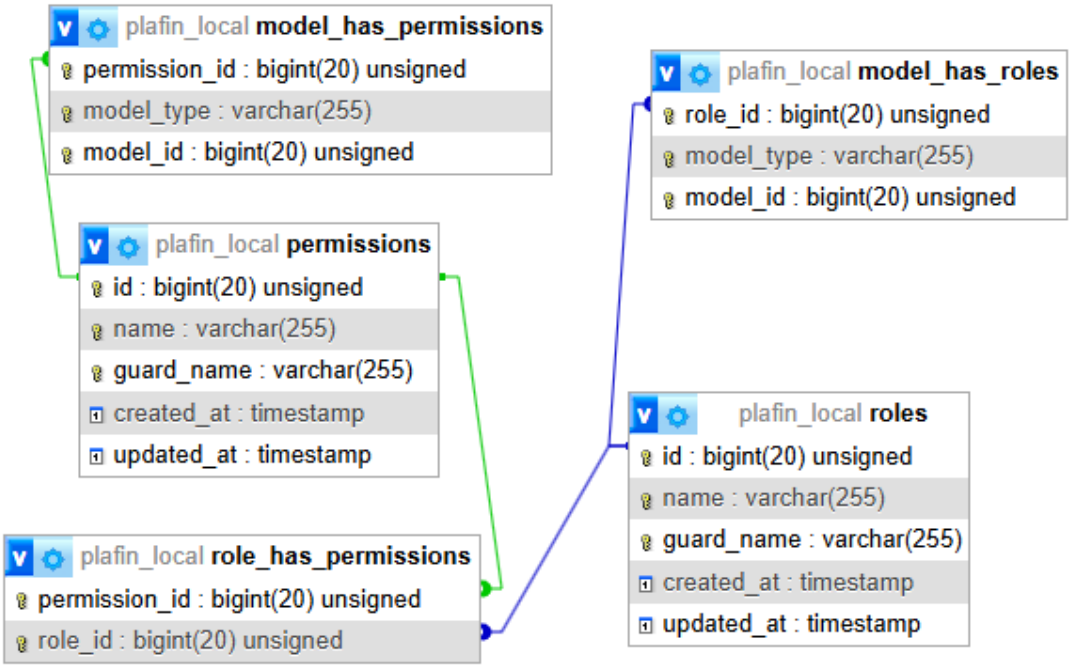


Рисунок Б.3 – ER-модель схеми рольово-дозвільної моделі.

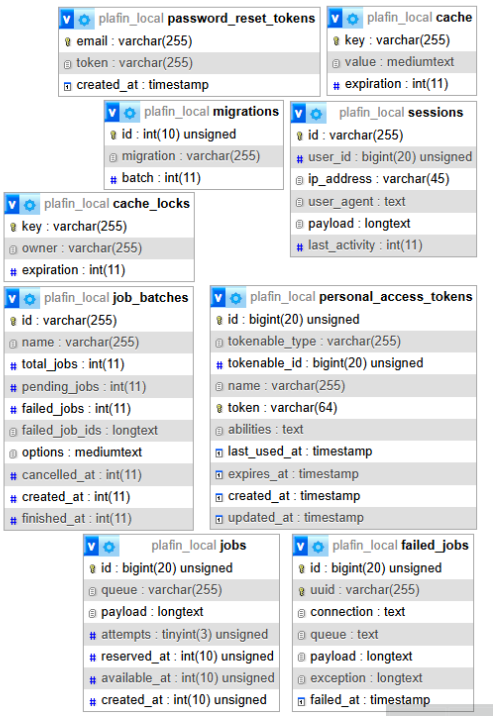


Рисунок Б.4 – ER-модель схеми технічних таблиць.

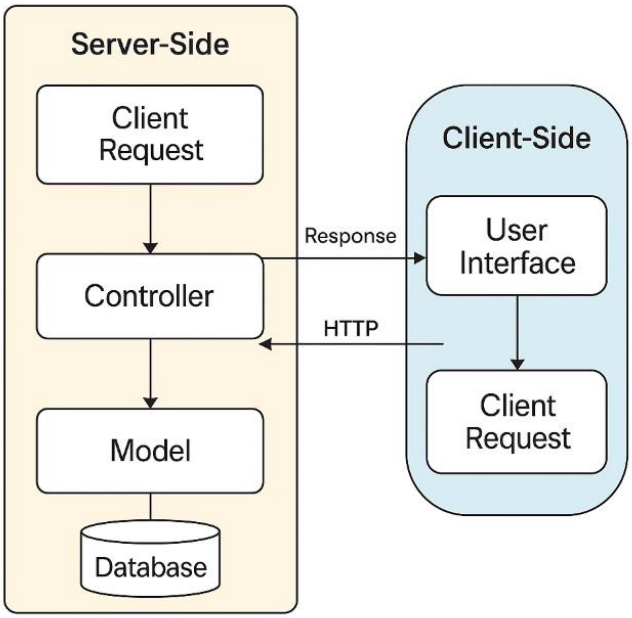


Рисунок Б.5 – Схема взаємодії систем.

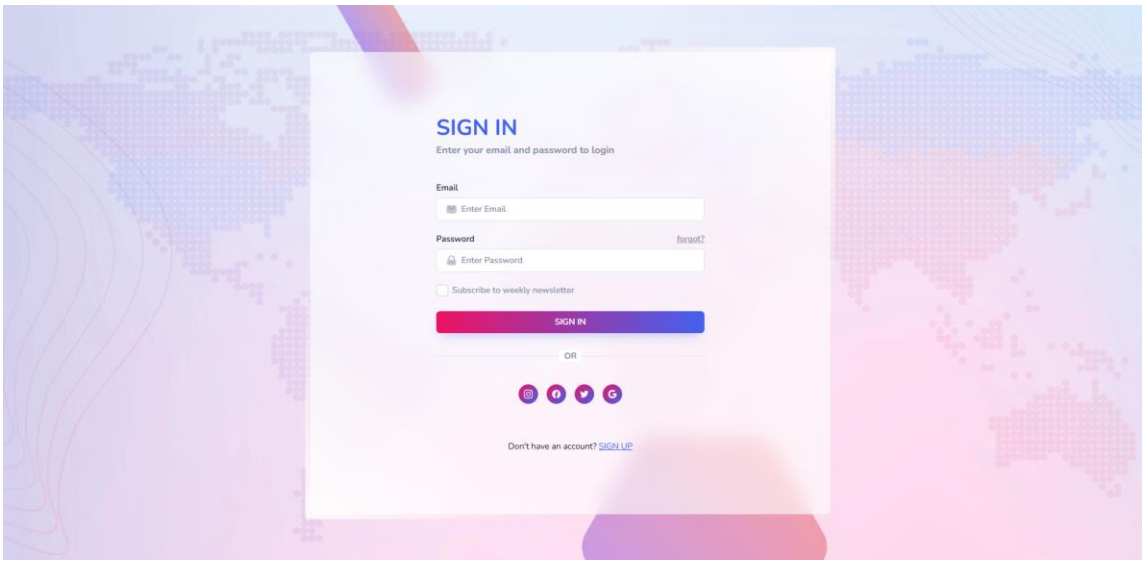


Рисунок Б.6 – Сторінка авторизації

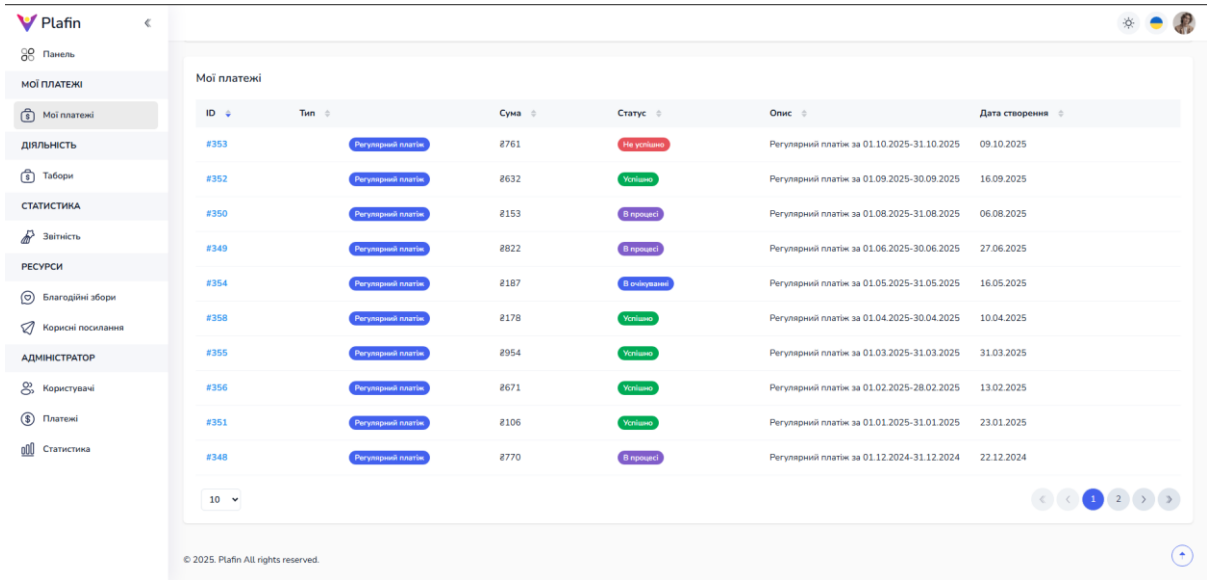


Рисунок Б.7 – Сторінка платежів

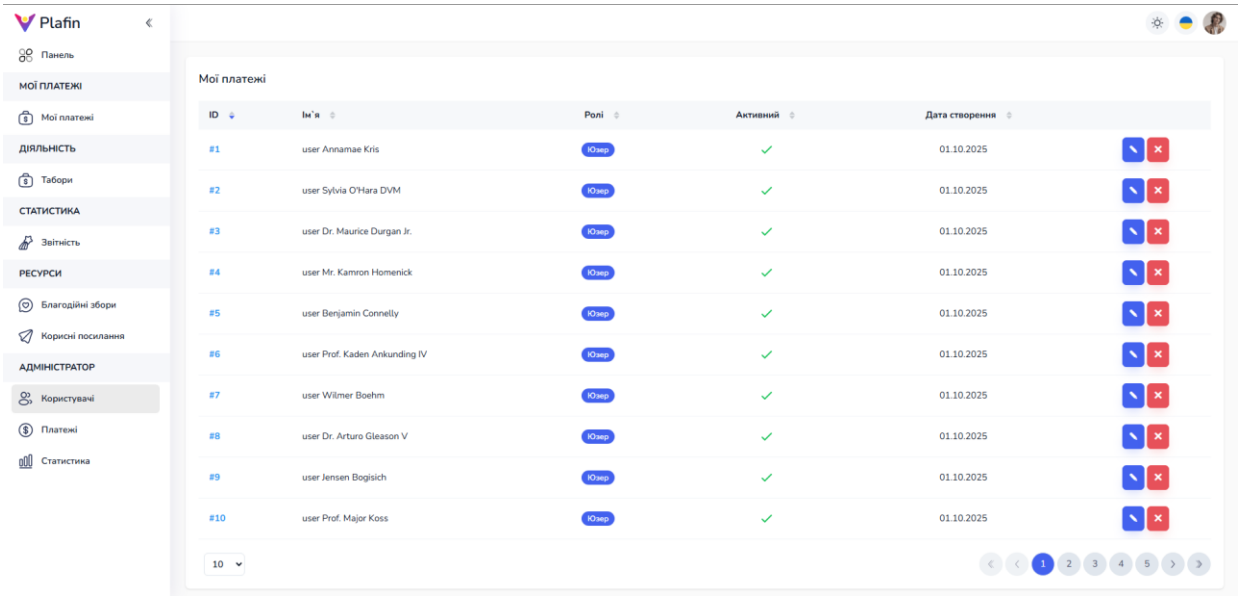


Рисунок Б.8 – Сторінка користувачів для адміністратора

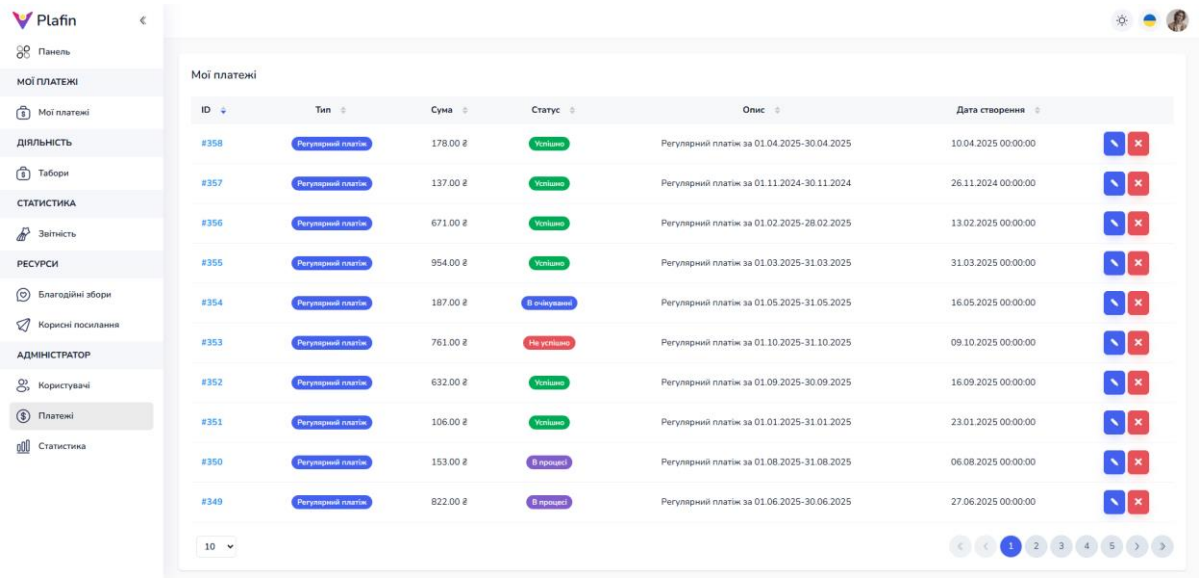


Рисунок Б.9 – Сторінка користувачів для адміністратора

Додаток В (обов'язковий)

Лістинг моделей

Payment Model

```
class Payment extends Model
{
   /**
    * @var string[]
    */
   protected $fillable = [ // user_id, type, amount, status, description

   protected $casts = [  // type => PaymentTypeEnum, status => PaymentStatusEnum];

   public function user(): BelongsTo {  // return $this->belongsTo(User::class); }

   // SCOPES
   public function scopeWithUser(Builder $query, int|null $userId): Builder
   {
      ... // фільтрація за user_id через ->when()
   }

   public function scopeWithWaitingStatus(Builder $query, bool $waiting = false): Builder
   {
      ... // фільтрація за статусом PaymentStatusEnum::WAIT
   }
}
```

UsefullLink Model

```
class UsefullLink extends Model
{
   use HasFactory, HasTranslations;

   protected $table = 'usefull_links';

   public array $translatable = [/* title, short_desc */];

   protected $fillable = [/* title, short_desc, image_link, link, publish_date, type */];

   protected $casts = [/* publish_date => datetime */];

   protected array $dates = [/* publish_date */];
}
```

UsefullLink Model

```
class User extends Authenticatable implements MustVerifyEmail, JWTSubject
{
   use HasFactory, Notifiable, HasRoles;

   protected $fillable = [/* name, email, is_active, password */];
   protected $hidden = [/* password, remember_token */];

   protected function casts(): array { return [/* email_verified_at => datetime, password => hashed, is_active => boolean */]; }

   public function getJWTIdentifier(): mixed { return /* primary key */; }
```

```
    public function getJWTCustomClaims(): array { return [/* empty */]; }

    public function payments(): HasMany { return /* $this->hasMany(Payment::class) */; }

    public function scopeActive(Builder $query) { return /* whereIsActive(true) */; }

    public function scopeWithRole(Builder $query, RoleEnum|int|array|null $role = null): Builder { return /* роль оброблена,
фільтр через ->role() */; }
}
```

Додаток Г (обов’язковий)

Лістинг контролерів

Admin Payment Controller

```
class PaymentController extends Controller
{
   public function index(Request $request): JsonResponse|AnonymousResourceCollection { /* $limit, Service::getAdminList(), return Resource::collection() */ }

   public function show() { /* ... */ }

   public function update() { /* ... */ }

   public function destroy() { /* ... */ }
}
```

Admin User Controller

```
class UserController extends Controller
{
   public function index(Request $request): JsonResponse|AnonymousResourceCollection { /* limit, UserService::getList(), ResponseService::formResourceRequest(), return Resource::collection() */ }

   public function show() { /* ... */ }

   public function update() { /* ... */ }

   public function destroy() { /* ... */ }
}
```

Private User Controller

```
class AuthController extends Controller
{
   public function login(LoginRequest $request): JsonResponse { /* validate creds, check email, verify, Auth::attempt(), return user + token */ }

   public function register(RegisterRequest $request): JsonResponse { /* validate data, check existing email, create user, assign role, send email job, login, return verification */ }

   public function loginSocials() { /* ... */ }

   public function logout(): JsonResponse { /* JWT::getToken(), logout, invalidate, return Logged out */ }

   public function me(Request $request): JsonResponse { /* read token, set token, refresh if exp < 300, return user + new token, catch blacklist/errors */ }
}
```

Private Verify Email Controller

```
class VerifyEmailController extends Controller
{
   public const HOME = '/dashboard';
   public const NOT_FOUND = '/404';
```

```
    public function __invoke(Request $request): RedirectResponse { /* find user, login, if verified redirect HOME+token, else verify & redirect, else redirect 404 */ }
}
```

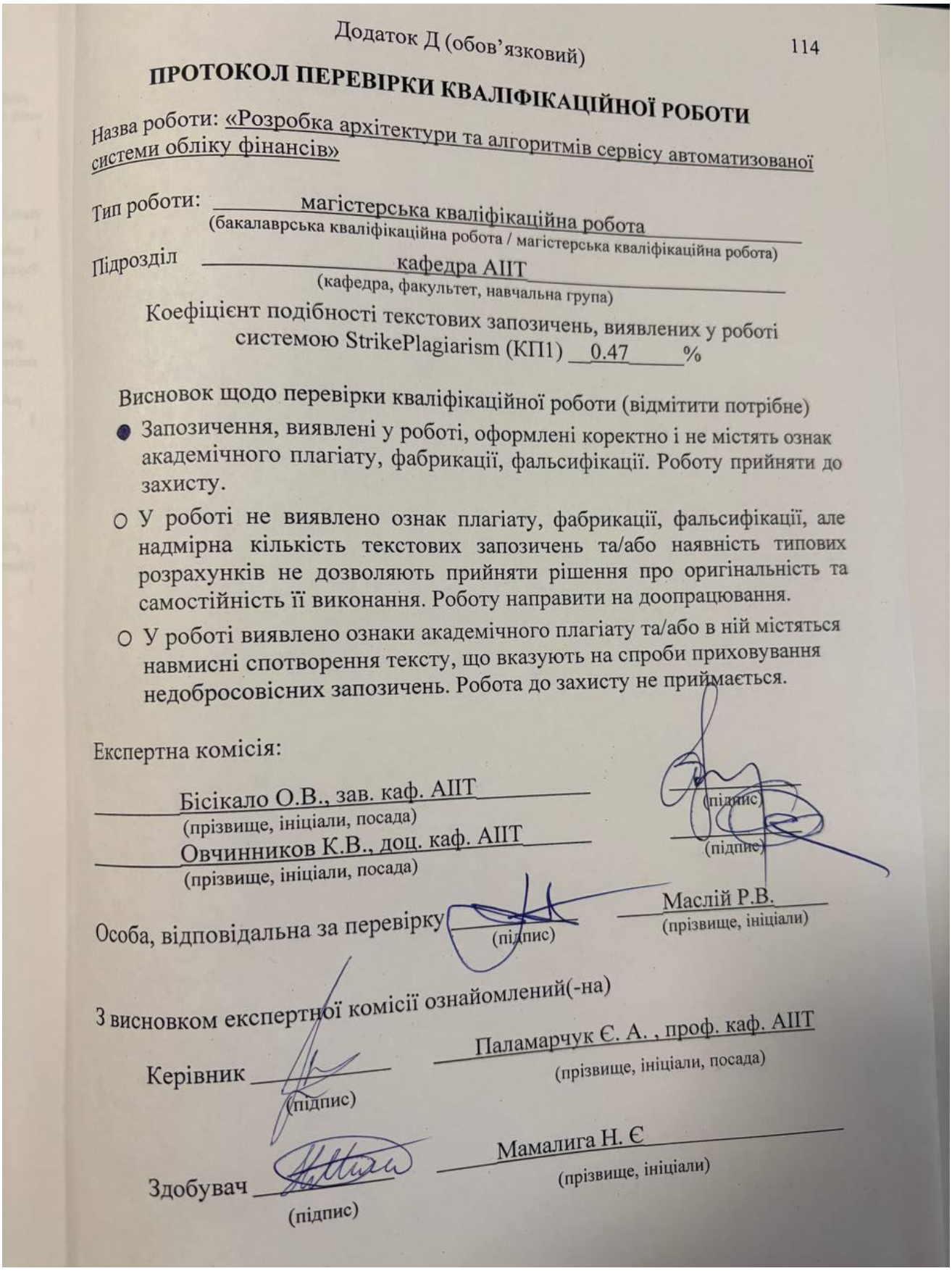
Private Payment Controller

```
class PaymentController extends Controller
{
    public function index(IndexRequest $request): JsonResponse|AnonymousResourceCollection { /* auth id, limit, waiting, PaymentService::getList(), return Resource::collection() */ }

    public function checkout(Request $request): JsonResponse { /* PaymentService::checkout(), return link */ }

    public function success(Request $request, $paysystem_id): JsonResponse|RedirectResponse { /* PaymentService::success(), redirect /my-payments?success=true */ }

    public function webhook(Request $request) { /* debug webhook payload */ }
}
```

Private UsefullLink Controller

```
class UsefullLinkController extends Controller
{
    public function index(Request $request): JsonResponse|AnonymousResourceCollection { /* UsefullLinkService::getList(), return Resource::collection() */ }
}
```

Додаток Д (обов'язковий)



Додаток Д (обов'язковий) 114

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: «Розробка архітектури та алгоритмів сервісу автоматизованої системи обліку фінансів»

Тип роботи: магістерська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра АІІТ
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 0.47 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ● Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.
- ○ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- ○ У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

Бісікало О.В., зав. каф. АІІТ (підпис)
(прізвище, ініціали, посада)

Овчинников К.В., доц. каф. АІІТ (підпис)
(прізвище, ініціали, посада)

Особа, відповідальна за перевірку (підпис) Маслій Р.В.
(прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник (підпис) Паламарчук Є. А. , проф. каф. АІІТ
(прізвище, ініціали, посада)

Здобувач (підпис) Мамалига Н. Є
(прізвище, ініціали)