ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Інтегрована платформа для управління децентралізованою організацією з автоматизацією бізнес-процесів та бухгалтерського обліку»

на здобуття освітнього ступеня магістр
за спеціальності 126 Інформаційні системи та технології
(код,найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владислав ГАВРИЛЮК
(підпис) *(ім'я, ПРІЗВИЩЕ здобувача)*

Виконав:
здобувач вищої освіти
група ІСДМ-61 Владислав ГАВРИЛЮК
(ім'я, ПРІЗВИЩЕ)
Керівник Віктор САГАЙДАК
PhD *(ім'я, ПРІЗВИЩЕ)*

Рецензент: Наталія ЛАЩЕВСЬКА
(ім'я, ПРІЗВИЩЕ)

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти магістр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна прорама Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
_____ Каміла СТОРЧАК

"____" _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гаврилюк Владислав Олександрович
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Інтегрована платформа для управління децентралізованою організацією з автоматизацією бізнес-процесів та бухгалтерського обліку
керівник кваліфікаційної роботи: Віктор САГАЙДАК, PhD
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від "___ " жовтня 2025 р. № ____

2. Строк подання кваліфікаційної роботи «26» грудня 2025 р.
3. Вихідні дані кваліфікаційної роботи:
   1. Концепції та моделі управління децентралізованими організаціями (DAO).
   2. Технології веб-розробки (стек PERN: PostgreSQL, Express, React, Node.js).
   3. Методики бухгалтерського обліку (P&L) та податкового законодавства.
   4. Науково-технічна література та документація до фреймворків.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
   1. Теоретико-методологічні засади автоматизації управління децентралізованими організаціями.
   2. Системний аналіз та проектування архітектури платформи.
   3. Програмна реалізація та інфраструктурне забезпечення платформи.
   4. Тестування, валідація та оцінка ефективності системи.
5.Перелік ілюстраційного матеріалу: *презентація*

6. Дата видачі завдання «___» _____ 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір та аналіз науково-технічної літератури та аналогів	01.09.2025 – 15.09.2025	Виконано
2.	Дослідження бізнес-процесів та проектування архітектури системи	16.09.2025 – 10.10.2025	Виконано
3.	Програмна реалізація клієнтської та серверної частин	11.10.2025 – 15.11.2025	Виконано
4.	Тестування системи, аудит безпеки та розрахунок економічної ефективності	16.11.2025 – 26.11.2025	Виконано
5.	Формулювання висновків та оформлення пояснювальної записки	26.11.2025 – 29.11.2025	Виконано
6.	Розробка демонстраційних матеріалів (презентації), підготовка доповіді	29.11.2025 – 20.12.2025	Виконано
7.	Попередній захист та проходження нормоконтролю	23.12.2025	Виконано

Здобувач вищої освіти _____ Владислав ГАВРИЛЮК
(підпис) *(ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Віктор САГАЙДАК
(підпис) *(ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття ступеня магістр: 84 стор., 27 рис., 2 табл., 35 джерел.

Мета роботи – створення програмної платформи YadroOS для комплексного управління децентралізованою організацією, що дозволить мінімізувати адміністративні витрати та забезпечити фінансову прозорість.

Об'єкт дослідження – процеси інформаційної підтримки управлінської, операційної та фінансової діяльності у розподілених організаційних структурах та децентралізованих командах.

Предмет дослідження – методи, моделі та програмні засоби автоматизації бізнес-процесів і фінансового обліку в розподілених системах.

*Короткий зміст роботи***.** У роботі досліджено еволюцію управлінських моделей та обґрунтовано необхідність створення спеціалізованого ПЗ для DAO. Проведено порівняльний аналіз існуючих рішень (Bitrix24, Odoo) та виявлено їх недоліки для гнучких команд. Спроєктовано архітектуру SPA-додатку на базі стеку PERN (PostgreSQL, Express, React, Node.js). Розроблено інфологічну модель бази даних та реалізовано алгоритми автоматичного розрахунку податків і звітності P&L у реальному часі. Впроваджено систему безпеки на основі ролей (RBAC) та аудиту подій. Виконано програмну реалізацію системи, проведено функціональне тестування ключових модулів та аудит безпеки. Розраховано економічну ефективність впровадження власної платформи порівняно з комерційними аналогами.

КЛЮЧОВІ СЛОВА**:** ДЕЦЕНТРАЛІЗОВАНА ОРГАНІЗАЦІЯ, АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, CRM, ERP, REACT, NODE.JS, POSTGRESQL, RBAC, P&L, REAL-TIME ACCOUNTING.

ABSTRACT

The text part of the qualifying work for obtaining a master's degree: 84 pp., 27 fig., 2 tables, 35 sources.

The purpose of the work is to create the YadroOS software platform for comprehensive management of a decentralized organization, which will minimize administrative costs and ensure financial transparency.

Object of research – is the processes of information support for managerial, operational, and financial activities in distributed organizational structures and decentralized teams.

Subject of research – methods, models, and software tools for automating business processes and financial accounting in distributed systems.

Summary of the work. The thesis investigates the evolution of management models and substantiates the need for specialized software for DAOs. A comparative analysis of existing solutions (Bitrix24, Odoo) was conducted, revealing their shortcomings for agile teams. An SPA application architecture based on the PERN stack (PostgreSQL, Express, React, Node.js) was designed. An infological database model was developed, and algorithms for automatic tax calculation and real-time P&L reporting were implemented. A security system based on roles (RBAC) and event auditing was introduced. The software implementation of the system was completed, functional testing of key modules and a security audit were conducted. The economic efficiency of implementing a proprietary platform compared to commercial analogues was calculated.

KEYWORDS: DECENTRALIZED ORGANIZATION, BUSINESS PROCESS AUTOMATION, CRM, ERP, REACT, NODE.JS, POSTGRESQL, RBAC, P&L, REAL-TIME ACCOUNTING.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасний етап розвитку світової економіки характеризується фундаментальним парадигмальним зсувом: переходом від жорстких індустріальних ієрархій до гнучких мережевих екосистем та економіки знань. Глобалізація ринку праці, прискорення цифровізації та масовий перехід до віддалених форматів роботи, каталізований пандемічними змінами, створили передумови для виникнення нового класу господарюючих суб'єктів – децентралізованих автономних організацій (DAO) та розподілених команд. У такому середовищі ключовими факторами конкурентоспроможності стають не масштаби активів, а швидкість прийняття управлінських рішень, адаптивність бізнес-процесів та мінімізація транзакційних витрат на внутрішню комунікацію.

Проте, стрімка трансформація організаційних форм суттєво випереджає розвиток відповідного управлінського інструментарію. Традиційні підходи до адміністративного менеджменту, сформовані в епоху фізичних офісів та паперового документообігу, демонструють критичну неефективність у цифровому просторі. Класичні ERP-системи (Enterprise Resource Planning), орієнтовані на великі корпорації, є надмірно складними, дорогими у впровадженні та інертними для динамічних IT-команд. З іншого боку, популярні легкі інструменти для управління завданнями (Task Trackers) фокусуються виключно на операційному рівні, ігноруючи фінансовий аспект діяльності.

Це призводить до виникнення феномену «клаптикової автоматизації», коли організація змушена використовувати розрізнений набір програмних продуктів, що не інтегровані між собою. Наслідком цього є розрив між операційною діяльністю та фінансовим обліком: менеджмент не володіє актуальною інформацією про рентабельність проектів у реальному часі, що створює значні ризики касових розривів та втрати керованості. Особливої гостроти ця проблема набуває в українському контексті, де специфіка роботи IT-сектору (модель співпраці з ФОП, мультивалютність контрактів) вимагає специфічних інструментів контролю та податкового адміністрування. У зв'язку з цим, розробка інтегрованої платформи

YadroOS, яка б поєднувала гнучкість сучасних веб-технологій з суворістю бухгалтерського обліку, є актуальним науково-практичним завданням.

Зв’язок роботи з науковими програмами, планами, темами. Кваліфікаційна магістерська робота виконана відповідно до пріоритетних напрямків науково-дослідних робіт кафедри Інформаційних систем та технологій Державного університету інформаційно-комунікаційних технологій. Тематика дослідження корелює з національною стратегією цифровізації економіки та спрямована на вирішення прикладних проблем автоматизації управління малим та середнім бізнесом в умовах цифрової трансформації.

Мета і завдання дослідження. Метою роботи є підвищення ефективності управління децентралізованою організацією шляхом розробки, обґрунтування та впровадження інтегрованої веб-платформи, що забезпечує комплексну автоматизацію бізнес-процесів, прозорість фінансових потоків та мінімізацію впливу людського фактору на облікові операції.

Для досягнення поставленої мети сформульовано та вирішено комплекс взаємопов'язаних завдань:

1. Проведено аналіз еволюції управлінських моделей та формалізовано специфічні вимоги до програмного забезпечення класу DAO Management.
2. Здійснено системний аналіз існуючих аналогів та економічно обґрунтовано доцільність розробки власного архітектурного рішення (порівняння TCO).
3. Спроєктовано мікросервісну архітектуру системи та розроблено нормалізовану схему бази даних, що забезпечує цілісність фінансових транзакцій.
4. Реалізовано програмний комплекс із використанням ізоморфного стеку технологій (TypeScript, Node.js, React) та ORM Prisma.
5. Розроблено систему інформаційної безпеки, що включає Dual-Token аутентифікацію, RBAC-авторизацію та захищений журнал аудиту подій.

6. Проведено верифікацію системи шляхом навантажувального тестування та оцінено економічну ефективність впровадження.

Об’єкт дослідження – процеси інформаційної підтримки управлінської, операційної та фінансової діяльності у розподілених організаційних структурах та децентралізованих командах.

Предмет дослідження – методи, моделі, алгоритми та програмні засоби автоматизації бізнес-процесів, управлінського обліку та фінансового контролю у веб-орієнтованих інформаційних системах.

Методи дослідження. Методологічною основою роботи є системний підхід до аналізу та проектування складних програмних комплексів. У процесі дослідження використано: методи системного аналізу (для дослідження предметної області та формулювання вимог); метод моделювання бізнес-процесів у нотації BPMN 2.0 (для формалізації логіки роботи системи); методи об'єктно-орієнтованого проєктування та архітектурні патерни Feature-Sliced Design (для побудови гнучкої структури ПЗ); методи теорії реляційних баз даних (для проектування схеми зберігання даних); методи експериментального тестування (для перевірки надійності та продуктивності розробленої системи).

Наукова новизна одержаних результатів полягає у вдосконаленні методології автоматизації управлінського обліку в розподілених системах. Це дозволило вирішити проблему часового лагу у формуванні звітності P&L, забезпечуючи менеджмент аналітичними даними в режимі реального часу, на відміну від традиційних підходів, що базуються на періодичному закритті звітних періодів.

Апробація результатів. Основні теоретичні положення та практичні результати дослідження доповідалися та отримали позитивну оцінку на наукових заходах, зокрема на X Міжнародній науковій конференції «Здобутки та досягнення прикладних та фундаментальних наук XXI століття» (м. Дніпро, 7 листопада 2025 р.) та на III Всеукраїнській науково-технічній конференції «Технологічні горизонти» (11 листопада 2025 р.).

1 ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ЗАСАДИ АВТОМАТИЗАЦІЇ УПРАВЛІННЯ ДЕЦЕНТРАЛІЗОВАНИМИ ОРГАНІЗАЦІЯМИ

1.1 Еволюція підходів до управління організаціями: від ієрархічних структур до децентралізованих автономних команд

Зміна технологічного укладу та перехід до економіки знань детермінують кризу традиційних управлінських моделей. Методи, сформовані в епоху індустріалізації та орієнтовані на стабільні ринкові умови, демонструють критичну неефективність у середовищі високої невизначеності. Фактори глобалізації бізнес-процесів у поєднанні з експоненційним зростанням обсягів даних вимагають від організаційних структур не стільки стабільності, скільки адаптивності. Спостерігається фундаментальний зсув управлінської парадигми: від жорстких вертикальних ієрархій, де прийняття рішень централізоване, до горизонтальних мережевих структур та децентралізованих автономних організацій (DAO).

Аналіз генезису управлінських систем дозволяє виявити корінь проблеми. Класична теорія менеджменту, що базується на постулатах Ф. Тейлора, А. Файоля та М. Вебера, розглядає підприємство як детермінований механізм [1]. Ефективність такої бюрократичної системи досягається через глибоку спеціалізацію, стандартизацію операцій та сувору ієрархію влади. Інформаційні потоки рухаються виключно знизу вгору, а директиви – у зворотному напрямку. Втім, дослідники [2] вказують на системну ваду такої моделі в контексті IT-сектору та креативних індустрій: ефект «пляшкового горлечка». При проходженні через численні рівні менеджменту інформація неминуче спотворюється, а процедури узгодження займають неприпустимо багато часу, що призводить до втрати ринкових можливостей. Більше того, відчуження виконавця від процесу прийняття рішень провокує деградацію людського капіталу – співробітники втрачають ініціативність, перетворюючись на пасивних виконавців інструкцій.

Альтернативою механістичному підходу виступає концепція «органічних» систем. Фредерік Лалу у праці «Компанії майбутнього» [3] запропонував

еволюційну градацію організацій, яка є методологічним підґрунтям для проектованої системи. Рух відбувається від «бурштинових» (армійський тип, сувора ієрархія) та «помаранчевих» (конкуренція, KPI-орієнтованість) структур до «бірюзових» (Teal). Останні характеризуються відмовою від інституту менеджерів середньої ланки на користь самоврядування. Можна стверджувати, що саме відсутність адекватного програмного забезпечення стримує масовий перехід до «бірюзових» моделей. Якщо в ієрархії функцію контролю виконує адміністратор, то в самокерованій структурі цю роль мусять перебрати на себе алгоритми та прозорі протоколи взаємодії.

Цю тенденцію підтверджує К. Шваб, зазначаючи в контексті Четвертої промислової революції [4], що сучасні цифрові платформи здатні радикально знижувати транзакційні витрати взаємодії, роблячи громіздкі ієрархічні ланцюжки економічно неефективними порівняно з мережевими структурами.

Саме тут виникає потреба в інструментарії на кшталт платформи YadroOS, яка розглядається в даній роботі. Концепт DAO (Децентралізованої Автономної Організації), первісно сформульований в середовищі блокчейн-розробників, пропонує революційний підхід: управління через програмний код, а не людське втручання. Віталік Бутерін визначає DAO як структуру, де правила функціонування зафіксовані математично [5]. Проте в рамках даного дослідження доцільно розглядати децентралізацію ширше, виходячи за межі виключно блокчейн-технологій. Мова йде про організаційно-технічну модель, яка може базуватися на веб-технологіях та криптографічному захисті даних, наприклад незмінні Audit Logs, не вимагаючи при цьому повної токенізації активів.

Проектування інформаційної системи для управління такою структурою вимагає технічної реалізації трьох базових векторів. Першим є забезпечення реальної, а не декларативної автономності команд, що передбачає програмне розмежування бюджетів та ресурсів. Другий вектор – радикальна прозорість (Transparency). Традиційний адміністративний контроль заміщується архітектурою відкритості: будь-яка транзакція чи зміна статусу проекту фіксується системою та доступна для аудиту, що унеможливлює корупційну складову. Третім вектором

виступає алгоритмічна меритократія, де вплив учасника на прийняття рішень корелює з його верифікованим внеском у проект, розрахованим через динамічні коефіцієнти ефективності, а не визначається штатною посадою.

Варто наголосити на проблемі невідповідності існуючого програмного забезпечення новим викликам. Більшість ERP та CRM систем історично створювалися для обслуговування вертикальних ієрархій [6], фіксуючи жорсткі регламенти, відхилення від яких система сприймає як помилку. Децентралізована організація ж потребує гібридного рішення: гнучкості сучасного Task-менеджера для операційної діяльності та суворості бухгалтерського обліку для фінансової дисципліни. Дані досліджень [7] дозволяють припустити, що впровадження децентралізованих моделей здатне знизити операційні витрати на адміністративний апарат на 20-30%, однак головним бар'єром залишається питання довіри.

У цифровому середовищі довіра трансформується з соціальної категорії в технічну. Архітектура програмного забезпечення повинна гарантувати неможливість ретроспективної зміни даних про фінансові домовленості. Отже, виникає гостра необхідність у створенні інтегрованої платформи, яка б об'єднувала єдиний інформаційний простір для розподілених команд з гнучкою рольовою моделлю доступу (RBAC) та інструментами фінансового моніторингу в реальному часі. Саме перехід від моделі підпорядкування до ринкової моделі «Внутрішній замовник – Виконавець» всередині компанії вимагає автоматизації взаєморозрахунків, що і становить предметну область розробки проектованої системи.

1.2 Огляд існуючих CRM та ERP систем для автоматизації бізнес

Ринок корпоративних інформаційних систем на даному етапі демонструє чітку сегментацію, яка, однак, породжує проблему фрагментації даних. Згідно з галузевими аналітичними звітами [6], більшість підприємств змушені використовувати гібридні екосистеми, намагаючись інтегрувати різнорідні

програмні продукти для управління клієнтами, фінансами та персоналом. Для децентралізованих організацій, де критичними факторами є не лише наявність функціоналу, а й прозорість алгоритмів прийняття рішень та швидкість масштабування, вибір базової платформи стає архітектурним викликом. Необхідно провести критичний аналіз існуючих рішень – Bitrix24, Odoo, Salesforce та стеку Atlassian – щоб виявити їхню невідповідність вимогам DAO.

Розглянемо сегмент універсальних SaaS-рішень, яскравим представником якого є платформа Bitrix24. Попри значне поширення у східноєвропейському регіоні, зумовлене низьким порогом входу, дана система демонструє суттєві архітектурні вади при спробі її адаптації під децентралізовану модель. Система побудована на базі застарілого технологічного стеку та монолітної архітектури, що створює ефект "технічного боргу" ще на етапі впровадження. Хоча програмний інтерфейс (API) задокументовано [8], наявні ліміти на кількість запитів фактично унеможливлюють побудову високонавантажених зовнішніх модулів фінансового моніторингу в режимі реального часу. Але головною методологічною проблемою є жорстка "зашита" в логіку системи вертикальна ієрархія. Bitrix24 не передбачає сценаріїв, де фінансова інформація є відкритою для рядових виконавців, а вбудовані алгоритми автоматизації обмежуються лінійною логікою (якщо-то), що не дозволяє імплементувати складні моделі розподілу прибутку, характерні для DAO.

Альтернативний підхід демонструє платформа Odoo, яка базується на модульному принципі та Open Source ліцензії. Це наближає її до ідеології проектованої системи, зокрема завдяки наявності модуля подвійного запису, що відповідає стандартам міжнародного бухгалтерського обліку [9]. Однак, при детальному аналізі виявляються критичні недоліки в контексті користувацького досвіду та локалізації. Odoo використовує серверну генерацію інтерфейсів, що в епоху реактивних односторінкових додатків (SPA) сприймається як архаїзм, суттєво знижуючи швидкість роботи оператора. Більше того, адаптація стандартного фінансового ядра під специфіку спрощеної системи оподаткування та єдиного податку, що є стандартом для українського IT-бізнесу, вимагає настільки

глибокої кастомізації коду, що сукупна вартість володіння (TCO) системою стає невиправдано високою для динамічних команд.

Окремий кластер складають спеціалізовані рішення рівня Enterprise, такі як Salesforce та екосистема Atlassian (Jira). Salesforce, будучи беззаперечним лідером у глибині аналітики продажів, фокусується виключно на зовнішньому контурі взаємодії з клієнтом, ігноруючи внутрішні операційні процеси та HR-менеджмент. Водночас Jira, яка де-факто є стандартом трекінгу задач в IT-індустрії, повністю позбавлена фінансового модулю. Спроба об'єднати ці системи через шлюзи інтеграції призводить до розриву інформаційного простору: проектні менеджери оперують термінами "годин" у Jira, а бухгалтери – термінами "витрат" у 1С чи Excel, і ці дані рідко синхронізуються автоматично.

Окремо слід наголосити на проблематиці технічної підтримки такої «клаптикової» автоматизації. Забезпечення консистентності даних між гетерогенними середовищами вимагає створення складної системи API-шлюзів, що експоненційно збільшує технічний борг проекту. Будь-яке оновлення на стороні CRM-системи (наприклад, зміна структури полів у Salesforce) неминуче призводить до збоїв у роботі інтеграційних скриптів, блокуючи фінансові операції до моменту втручання розробників. У контексті децентралізованої організації, яка не має виділеного департаменту системного адміністрування, така вразливість є критичною. Затримка в актуалізації даних навіть на декілька годин може призвести до касових розривів або некоректного нарахування винагород учасникам розподілених команд, що руйнує довіру до самої моделі управління.

Крім суто технічних аспектів, не можна ігнорувати фактор когнітивного спротиву персоналу. Корпоративні системи рівня SAP або Oracle історично проектувалися з акцентом на функціональну повноту, а не на ергономіку інтерфейсів. Високий поріг входження та перевантаженість екранних форм створюють ситуацію, коли співробітники, намагаючись уникнути складнощів, переносять реальну операційну діяльність у «тіньовий сектор» – месенджери (Telegram, Slack) та Google-таблиці. Як наслідок, дороговартісна ERP-система перетворюється на архів постфактум-звітності, що не відображає реального стану

справ в організації. Для DAO, де швидкість онбордингу нових учасників є конкурентною перевагою, використання застарілих інтерфейсних патернів є неприпустимим гальмом розвитку.

Таблиця 1.1

Порівняльний аналіз систем автоматизації управління

Критерій порівняння	Bitrix24	Odoo (Community)	Salesforce	YadroOS (Проект)
Архітектура	Моноліт (PHP)	Модульний моноліт (Python)	Cloud SaaS (Multi-tenant)	Micro-kernel / SPA (React + Node.js)
Інтерфейс (UI)	Перевантажений, складний	Застарілий, серверний рендеринг	Сучасний, але складний	Мінімалістичний, адаптивний (Tailwind)
Фінансовий облік	Базовий (рахунки)	Повноцінний бухгалтерський	Тільки доходи від продажів	Інтегрований (P&L, Cashflow, Taxes)
Підтримка DAO	Відсутня (тільки ієрархія)	Часткова (через налаштування прав)	Відсутня	Нативна (RBAC + Audit Logs)
Вартість масштабування	Середня	Висока (розробка)	Дуже висока	Низька (Open Source + свій хостинг)
Швидкість роботи	Низька (перезавантаження сторінок)	Середня	Залежить від інтернету	Висока (клієнтський рендеринг)

Узагальнюючи результати компаративного аналізу (див. Таблицю 1.1), можна констатувати наявність "сірої зони" на ринку ПЗ. Існуючі рішення поляризовані: це або прості та швидкі менеджери завдань без фінансів, або важкі, інертні ERP-системи, впровадження яких займає місяці. Жодна з розглянутих

систем не пропонує нативної підтримки рольової моделі доступу на основі смарт-контрактів чи незмінних логів аудиту (Audit Logs), що є фундаментом довіри в децентралізованій організації.

Саме цей технологічний вакуум обґрунтовує доцільність розробки платформи YadroOS. Проектована система не намагається конкурувати з гігантами ринку у широті функціоналу, а вирішує вузьке, але критичне завдання: створення єдиного середовища, де виконання завдання автоматично генерує фінансову транзакцію, доступну для перевірки всім учасникам процесу. Такий підхід вимагає відмови від монолітної архітектури на користь мікроядра та клієнтського рендерингу для забезпечення необхідної швидкодії.

1.3 Особливості автоматизації бухгалтерського та управлінського обліку в IT-сфері та розподілених командах

Економічна природа функціонування високотехнологічних компаній та децентралізованих організацій містить фундаментальну відмінність від класичних моделей індустріального чи торговельного бізнесу. Якщо у традиційному виробництві домінує матеріальна складова собівартості, то в IT-секторі ключовим активом виступає нематеріальний інтелектуальний капітал, а основний масив операційних витрат формують виплати розподіленим командам фахівців. В умовах транскордонної діяльності, коли центр генерації прибутку юридично знаходиться в одній податковій юрисдикції, а центри витрат (виконавці) розпорошені по всьому світу, виникає явище фіскальної та управлінської асиметрії. Цей дисбаланс неможливо ефективно нівелювати виключно інструментарієм регламентованого бухгалтерського обліку.

Імперативність ведення бухгалтерського обліку, закріплена Законом України «Про бухгалтерський облік та фінансову звітність в Україні» [11], спрямована передусім на задоволення інформаційних запитів фіскальних органів та дотримання податкового законодавства. Однак ретроспективний характер офіційної звітності робить її малопридатною для оперативного менеджменту в

динамічному середовищі. Виникає об'єктивна необхідність у впровадженні паралельного контуру управлінського обліку, який, на відміну від бухгалтерського, оперує не лише задокументованими фактами, а й прогнозними показниками та даними в реальному часі.

При проектуванні платформи YadroOS особливу увагу приділено методологічній проблемі розбіжності між рухом грошових коштів та операційною ефективністю, яка в науковій літературі описується як дихотомія касового методу та методу нарахування. Звіт про рух грошових коштів, або Cash Flow, відображає стан ліквідності через фактичні транзакції. Натомість Звіт про фінансові результати, відомий як P&L, фіксує доходи та витрати в момент виникнення зобов'язань (підписання актів), ігноруючи момент оплати. В IT-індустрії, де поширеною є практика післяплати з відстрочкою у 30-60 днів, ігнорування цієї різниці створює ілюзію прибутковості. Компанія може демонструвати високий операційний прибуток на папері, але фактично перебувати в стані технічного дефолту через відсутність коштів на рахунках. Проектована система має вирішувати цю проблему шляхом автоматизованого розрахунку касових розривів, інтегруючи планові дати надходжень з CRM-системи та фактичні залишки з банківських API.

Наступним архітектурним викликом є мультивалютність та курсова волатильність. Дохідна частина бюджету експортоорієнтованих IT-компаній традиційно номінується у твердих валютах, тоді як операційні витрати часто здійснюються у локальних грошових одиницях країн базування виконавців. Згідно з МСФЗ 15 «Дохід від договорів з клієнтами» [13], визнання виручки прив'язане до курсу на дату операції, що в умовах турбулентного валютного ринку призводить до виникнення курсових різниць. Ручна обробка цих відхилень є джерелом механічних помилок. Автоматизація цього процесу вимагає прямої синхронізації з серверами національних банків для отримання актуальних курсів у момент транзакції. Це дозволяє системі розраховувати так звану реалізовану та нереалізовану курсову різницю в автоматичному режимі, надаючи власнику бізнесу очищену від курсового шуму картину рентабельності.

Окремий пласт проблем пов'язаний зі зміною парадигми трудових відносин. Перехід до Gig-економіки та співпраці з незалежними підрядниками (ФОП) трансформує документообіг. Замість єдиної зарплатної відомості виникає потреба в генерації сотень індивідуальних інвойсів та актів прийому-передачі робіт щомісяця. Враховуючи суворі вимоги Податкового кодексу України [14] до первинної документації, платформа YadroOS реалізує функціонал автоматичного конструктора документів. Система генерує юридично значущі файли на основі трекінгу часу або закритих завдань, що нівелює адміністративне навантаження на фінансовий відділ.

Втім, головна науково-практична новизна підходу полягає у зміні тригера облікових операцій. Традиційні системи працюють за принципом періодичного введення даних бухгалтером. Проектована платформа базується на подієво-орієнтованій архітектурі (Event-Driven Architecture). У цій парадигмі первинною є бізнес-подія: зміна статусу задачі розробником або закриття угоди менеджером. Ця дія автоматично ініціює ланцюжок транзакцій: нарахування зобов'язань перед виконавцем, розрахунок податків, формування інвойсу клієнту та оновлення фінансового дашборду. Такий підхід усуває часовий лаг між здійсненням господарської операції та її відображенням в обліку, забезпечуючи абсолютну прозорість та актуальність даних, що є критичною умовою для функціонування децентралізованих автономних організацій.

1.4 Методи забезпечення прозорості та довіри в інформаційних системах

Фундаментальна трансформація організаційних структур та масовий перехід до розподілених команд актуалізують класичну економічну проблему «принципал-агент», яка набуває нових форм у цифровому просторі. В умовах дистанційної роботи власник ресурсів (принципал) втрачає інструменти прямого фізичного нагляду за діяльністю виконавця (агента), що створює передумови для виникнення інформаційної асиметрії. Традиційні методи адміністративного контролю, такі як фізичні системи доступу або візуальний нагляд, стають рудиментарними та

неефективними. Натомість критичного значення набуває поняття цифрової довіри, яке в архітектурі програмного забезпечення трансформується з абстрактної етичної категорії в набір жорстких інженерних вимог. Система повинна гарантувати, що фінансові метрики відображають об'єктивну реальність, а корпоративні дані захищені від несанкціонованого розповсюдження. Вирішення цієї проблематики лежить у площині побудови ешелонованої системи захисту, що поєднує превентивні механізми (суворе розмежування прав) та детективні заходи, а саме тотальний аудит подій.

Для реалізації превентивного контуру захисту в платформі YadroOS було обрано стратегію рольового управління доступом (Role-Based Access Control – RBAC), що базується на рекомендаціях стандарту NIST SP 800-162 [15]. Вибір саме цієї моделі зумовлений порівняльним аналізом альтернативних підходів. Дискреційна модель (DAC), поширена у файлових системах, була відхилена через небезпеку децентралізації управління правами: ситуація, коли власник об'єкта самостійно визначає права доступу до нього, створює хаос у правах та ризики витоку інформації. З іншого боку, мандатна модель (MAC), характерна для систем військового призначення, виявилася надмірно ригідною для бізнес-середовища, оскільки блокує горизонтальну взаємодію між відділами.

Саме RBAC дозволяє імплементувати концепцію «найменших привілеїв» (Least Privilege Principle) найбільш ефективно. В архітектурі системи права призначаються не конкретним користувачам, а абстрактним сутностям – ролям (наприклад, «Фінансовий менеджер», «Розробник», «Аудитор»). Користувач отримує доступ до функціоналу лише через асоціацію з відповідною роллю. Ключовим аспектом тут виступає можливість реалізації матриці розділення повноважень (Segregation of Duties – SoD). Цей механізм програмно забороняє конфліктні комбінації прав: система технічно не дозволить одній особі мати права на створення інвойсу та одночасне підтвердження його оплати. Таке архітектурне обмеження мінімізує ризики внутрішнього шахрайства (фроду) на рівні бізнес-логіки.

Технічна реалізація підсистеми автентифікації та авторизації у

розподіленому веб-середовищі спирається на використання протоколу OAuth 2.0 та токенів стандарту JSON Web Token (JWT), описаних у специфікації RFC 7519 [16]. Використання JWT обумовлено необхідністю підтримки Stateless-архітектури, що є критичним для масштабування системи. Токен доступу містить у собі підписану криптографічним ключем корисну інформацію (claims) про роль користувача та термін дії сесії. Це дозволяє серверній частині верифікувати права клієнта без необхідності виконання "важких" запитів до бази даних при кожній транзакції, що суттєво знижує латентність системи. Для нівелювання ризиків перехоплення токенів передбачено використання короткоживучих Access-токенів та механізму їх поновлення через Refresh-токени, що зберігаються у захищених http-only cookie.

Другим, не менш важливим компонентом системи довіри є забезпечення прозорості операцій. У сучасному дискурсі часто пропонується використання технології блокчейн як гаранта незмінності даних. Проте аналіз показав, що інтеграція розподіленого реєстру в корпоративну ERP-систему є економічно та технічно недоцільною через низьку пропускну здатність транзакцій та надлишкову складність інфраструктури. В межах закритого корпоративного контуру достатній рівень довіри забезпечується через впровадження криптографічно захищеного журналу аудиту (Audit Log). Згідно із Законом України «Про електронні довірчі послуги» [17], електронні дані можуть виступати допустимими доказами в суді за умови гарантії їх цілісності.

В системі YadroOS реалізовано механізм «подієвої невідворотності» (non-repudiation). Технічно це досягається через використання патерну атомарних транзакцій: будь-яка модифікуюча операція в базі даних (INSERT, UPDATE, DELETE) супроводжується синхронним записом у спеціальну таблицю аудиту в межах однієї транзакції. Якщо запис у лог неможливий (наприклад, через збій дискової підсистеми), основна операція також скасовується. Структура запису аудиту спроектована у форматі JSONB, що дозволяє зберігати повний зліпок стану об'єкта «до» та «після» зміни, забезпечуючи можливість детального аналізу інцидентів (forensic analysis). Окрім самих даних, фіксуються метадані контексту: IP-адреса, User-Agent браузера та точний час сервера.

Така архітектура дозволяє перетворити систему аудиту з пасивного архіву на активний інструмент управління. У випадку виникнення спірних ситуацій (наприклад, несанкціонована зміна суми виплати), адміністратор має можливість похвилинно відтворити хронологію подій та однозначно ідентифікувати ініціатора змін. Варто зазначити, що для захисту самого журналу аудиту від модифікації адміністраторами бази даних можуть застосовуватися методи хешування ланцюжків записів, що наближає надійність системи до блокчейн-рішень, але без втрати швидкодії.

Таким чином, запропонований комплекс методів формує середовище «нульової довіри» (Zero Trust), де безпека базується не на вірі в чесність співробітників, а на математичній та архітектурній неможливості непомітного зловживання. Грамотна комбінація рольової ізоляції та тотального логування створює необхідний фундамент для побудови автоматизованої платформи управління децентралізованою організацією.

1.5 Постановка задачі та вимог до розроблюваної платформи

Узагальнення результатів теоретичного аналізу, проведеного у попередніх підрозділах, дозволяє констатувати наявність системного протиріччя в галузі автоматизації управління децентралізованими організаціями. З одного боку, еволюція управлінських парадигм у бік горизонтальних структур та Gig-економіки вимагає максимальної автономності команд. З іншого боку, відсутність фізичного контролю та специфіка фінансових потоків в IT-секторі (курсова волатильність, часові лаги ліквідності) диктують необхідність жорсткої централізації облікових функцій та впровадження механізмів «нульової довіри». Дослідження ринку програмного забезпечення виявило архітектурний розрив: існуючі рішення або є надмірно спрощеними менеджерами завдань без фінансового ядра, або громіздкими ERP-системами монолітної архітектури, що не відповідають вимогам гнучкості.

Таким чином, науково-практична проблема полягає у відсутності

інструментарію, здатного поєднати оперативну гнучкість Task-менеджера із суворістю бухгалтерського обліку в єдиному інтерфейсі. Вирішення цієї проблеми є метою даної кваліфікаційної роботи, що передбачає проектування та розробку інтегрованої платформи YadroOS. Система покликана нівелювати виявлені недоліки існуючих аналогів шляхом впровадження подієво-орієнтованої моделі обліку, де фінансові транзакції генеруються автоматично як наслідок операційної діяльності.

Досягнення поставленої мети вимагає послідовного вирішення комплексу інженерних завдань. Фундаментальним етапом є проектування архітектури системи у форматі Single Page Application (SPA). Такий підхід, на відміну від серверного рендерингу, є критичною вимогою для забезпечення реактивності інтерфейсу та мінімізації часу відгуку, що безпосередньо впливає на ефективність роботи розподілених команд. На рівні збереження даних завдання полягає у розробці нормалізованої реляційної моделі в середовищі PostgreSQL [18], здатної забезпечити посилальну цілісність (referential integrity) між гетерогенними сутностями: клієнтами, динамічними угодами, мультивалютними транзакціями та обліковими записами персоналу.

Окремим вектором роботи визначено імплементацію контуру інформаційної безпеки. Виходячи з проаналізованих у підрозділі 1.4 ризиків, система повинна базуватися на рольовій моделі доступу (RBAC) згідно зі стандартами NIST [15]. Технічна реалізація цього завдання передбачає розробку Middleware-шару для валідації JWT-токенів та створення механізму асинхронного запису логів аудиту. Це дозволить фіксувати кожну зміну стану системи, забезпечуючи юридичну значущість дій користувачів.

Функціональні вимоги до платформи сформовані, виходячи зі специфіки предметної області. Система повинна забезпечувати повний цикл супроводу угод (CRM), автоматичну генерацію первинної документації (інвойсів, актів) у форматі PDF, а також підтримку складних фінансових розрахунків: конвертацію валют за курсами національних банків, розрахунок податкового навантаження для різних груп ФОП та зведення управлінського балансу (P&L) в режимі реального часу.

2 СИСТЕМНИЙ АНАЛІЗ ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ ПЛАТФОРМИ

2.1 Аналіз та моделювання бізнес-процесів організації

Проектування архітектури сучасної інформаційної системи вимагає попереднього етапу глибокого системного аналізу, метою якого є трансформація абстрактних бізнес-вимог у чіткі інженерні специфікації. Фундаментальним етапом цієї роботи є формалізація бізнес-процесів, що підлягають автоматизації. Враховуючи складність логіки децентралізованих організацій, для опису функціональних алгоритмів платформи YadroOS було обрано нотацію моделювання бізнес-процесів BPMN 2.0 (Business Process Model and Notation). Дана нотація, затверджена як міжнародний стандарт ISO/IEC 19510 [19], дозволяє створювати уніфіковані графічні моделі, які виступають "містком" між предметною областю та технічною реалізацією.

Використання стандартизованих елементів (подій, шлюзів, потоків управління) дає можливість не лише візуалізувати процеси для стейкхолдерів, але й використовувати ці моделі як безпосередню інструкцію для розробки бізнес-логіки серверної частини додатку (Backend).

У ході передпроектного дослідження предметної області було ідентифіковано та декомпозовано два ключові макро-процеси, які є критично важливими для забезпечення життєдіяльності децентралізованої організації. Перший вектор стосується операційної діяльності та охоплює процеси укладання та супроводу угод в рамках контуру управління взаємовідносинами з клієнтами (CRM). Другий вектор спрямований на внутрішній фінансовий процесинг та розподіл ресурсів, що належить до сфери планування ресурсів підприємства (ERP).

Першим об'єктом детального моделювання визначено життєвий цикл угоди (Deal Lifecycle). Цей процес охоплює повний шлях взаємодії з контрагентом: від моменту першого контакту (лідогенерації) до успішного закриття контракту та отримання оплати. У традиційних інформаційних системах цей процес часто

характеризується фрагментарністю та дискретністю, коли перехід між етапами супроводжується ручним узгодженням документів або перенесенням даних між різними програмними продуктами (наприклад, з Excel у 1С). Натомість архітектура YadroOS базується на концепції наскрізного потоку даних (End-to-End Data Flow), що забезпечує безшовну передачу контексту між стадіями без необхідності повторного введення інформації.

Модель бізнес-процесу обробки замовлення клієнта, розроблена в нотації BPMN (Рис. 2.1), базується на логіці послідовної зміни станів сутності «Угода». Процес ініціюється початковою подією (Start Event) – реєстрацією потенційного клієнта.

Архітектура системи передбачає два канали ініціації: ручне створення картки відповідальним менеджером або автоматична генерація сутності через програмний інтерфейс (API/Webhook) при отриманні вхідних даних з веб-форми на корпоративному сайті.

Далі ініціюється підпроцес кваліфікації ліда (Lead Qualification). На цьому етапі виконується автоматизований аудит цілісності вхідних даних та верифікація контрагента через API відкритих реєстрів; перехід до активних переговорів програмно блокується до моменту призначення відповідального менеджера.

Зміна статусу сутності на «Договір підписано» виступає тригером для запуску асинхронної сервісної задачі (Service Task). Фоновий процес агрегує реквізити з профілю клієнта, інжектує їх у затверджений шаблон та персистує згенерований PDF-інвойс у S3-сумісному об'єктному сховищі.

Фіналізація процесу (End Event) детермінується фактом фінансового клірингу. Архітектура передбачає обробку вхідних вебхуків (Webhooks) від платіжного провайдера: отримання підтвердженої транзакції автоматично закриває угоду. Таке рішення виключає необхідність ручної звірки надходжень бухгалтерією, забезпечуючи наскрізну автоматизацію розрахунків.

Рис. 2.1 – BPMN-діаграма процесу обробки угоди

Важливою особливістю розробленої моделі є широке використання логічних шлюзів (Gateways), які дозволяють автоматизувати прийняття рішень на основі закладених бізнес-правил. Зокрема, для мінімізації операційних ризиків реалізовано механізм умовного розгалуження процесу через шлюз типу XOR (виключне АБО).

Логіка роботи наступна: якщо номінальна сума угоди перевищує встановлений в налаштуваннях ліміт (наприклад, еквівалент 10 000 доларів США), потік управління автоматично спрямовується на гілку додаткового погодження. У цьому сценарії система блокує можливість генерації рахунку до моменту отримання цифрового підтвердження від користувача з роллю «Адміністратор».

Такий підхід забезпечує вбудований фінансовий контроль (Compliance) ризикових операцій, не сповільнюючи при цьому роботу над стандартними угодами меншої вартості.

На відміну від інтерактивних операційних процесів, підсистема фінансового обліку спроектована як ізольований асинхронний контур (Back-office process). Це рішення де-факто виключає блокування основного потоку виконання (Event Loop) важкими обчислювальними операціями.

Алгоритм, наведений на схемі 2.2, тригериться подією переходу інвойсу в статус «Сплачено». Фундаментальною вимогою до архітектури тут є забезпечення ідемпотентності: механізм гарантує, що повторний виклик обробника (наприклад, при дублюванні webhook-події через Network Jitter) не призведе до подвійного нарахування.

Отримання актуального курсу валют реалізовано через інтеграцію з API регулятора. Для нівелювання ризиків відмови зовнішнього сервісу імплементовано патерн Circuit Breaker: при перевищенні ліміту тайм-аутів система автоматично переходить на використання кешованих даних, забезпечуючи безперервність конвертації валют.

Розрахунок фіскальних зобов'язань та чистого доходу виконується в рамках Script Task. Критично важливим є використання бібліотек для роботи з даними фіксованої точності (Decimal), що виключає появу помилок округлення, характерних для стандартної арифметики з плаваючою комою (IEEE 754). Алокація (Allocation) чистого прибутку між фондами реалізується в межах єдиної атомарної ACID-транзакції.

Це забезпечує цілісність даних: у разі будь-якого збою виконується повний Rollback змін. Кожна транзакція супроводжується створенням незмінного запису в Audit Log, що унеможливлює ретроспективні маніпуляції, а фінальним акордом процесу стає ініціація перерахунку агрегатів P&L звіту для підтримки актуальності аналітичного дашборду.

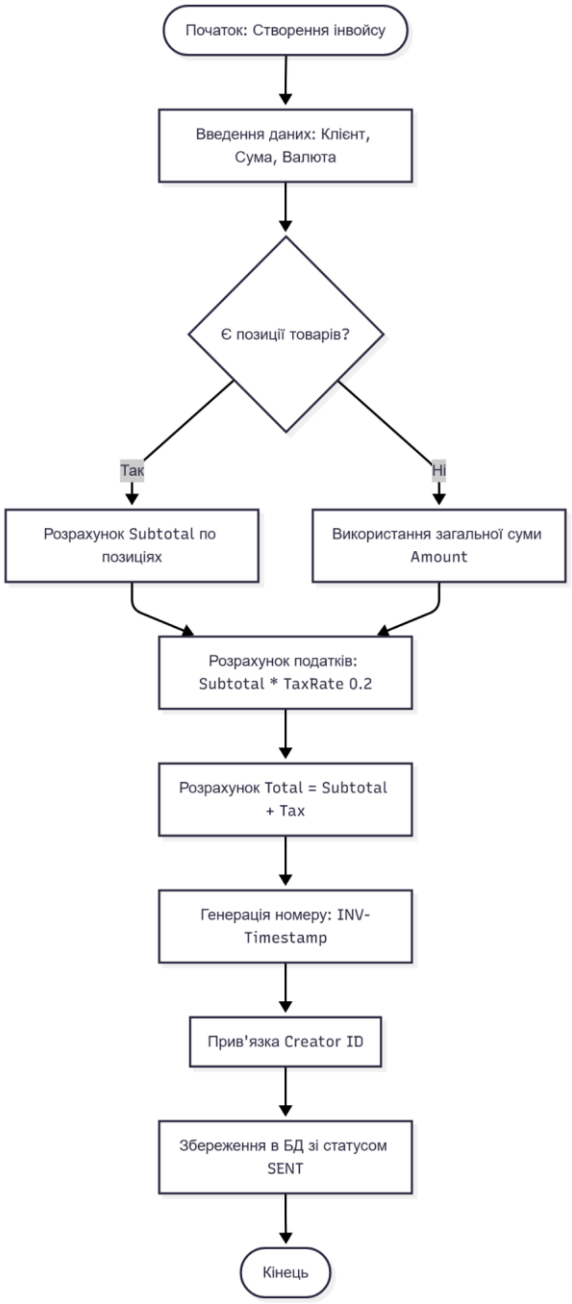


Рис. 2.2 – BPMN-діаграма фінансового процесингу

Впровадження описаних алгоритмів автоматизації створює технологічні передумови для переходу організації до концепції безперервного обліку (Continuous Accounting). Цей методологічний підхід дозволяє нівелювати одну з найбільш розповсюджених проблем традиційної бухгалтерії – необхідність ручного закриття звітних періодів ("closing the books"), замінюючи дискретні процедури потоковою обробкою даних.

З метою детальної формалізації функціональних вимог до інтерфейсів та

розмежування зон відповідальності було проведено моделювання варіантів використання (Use Case Modeling). За результатами аналізу рольової структури організації було ідентифіковано чотири базові моделі акторів, взаємодія яких із системою визначає архітектуру безпеки додатку (Рис. 2.3).

Роль «Адміністратор» (Admin). Цей актор наділений вищим рівнем привілеїв у системі. Проте, згідно з принципами безпеки, його функції здебільшого стосуються конфігурування, а не операційної діяльності. До зони відповідальності адміністратора віднесено налаштування глобальних параметрів (податкових ставок, валютних пар), управління обліковими записами користувачів (CRUD operations), а також здійснення нагляду за інформаційною безпекою через перегляд повного журналу аудиту подій.

Роль «Менеджер» (Manager). Функціональний профіль даного актора орієнтований на забезпечення безперервності бізнес-процесів (Front-office). Ключові сценарії використання включають реєстрацію та кваліфікацію лідів, управління стадіями угод (Kanban-дошка), комунікацію з клієнтами та ініціювання генерації первинної фінансової документації. Права менеджера обмежені в частині доступу до налаштувань системи та фінансової звітності інших підрозділів.

Роль «Бухгалтер» (Accountant). Ця роль зосереджена на функціях верифікації даних та фінансового контролю. Користувачі з цим профілем мають доступ (Read-only) до всіх транзакцій системи, розширеної аналітичної звітності про прибутки та збитки, а також до інструментів експорту фінансових даних (CSV/XLS) для подальшої обробки у зовнішньому фіскальному ПЗ.

Роль «Система» (System Actor). Особливе місце в моделі займає неперсоніфікований актор, який функціонує в автономному режимі. Це сукупність фонових сервісів (Workers/Cron-jobs), що виконують регламентні завдання без прямого втручання людини. До компетенції цього актора належить синхронізація курсів валют, моніторинг дедлайнів оплати, автоматична розсилка нагадувань боржникам та регулярне створення резервних копій бази даних.

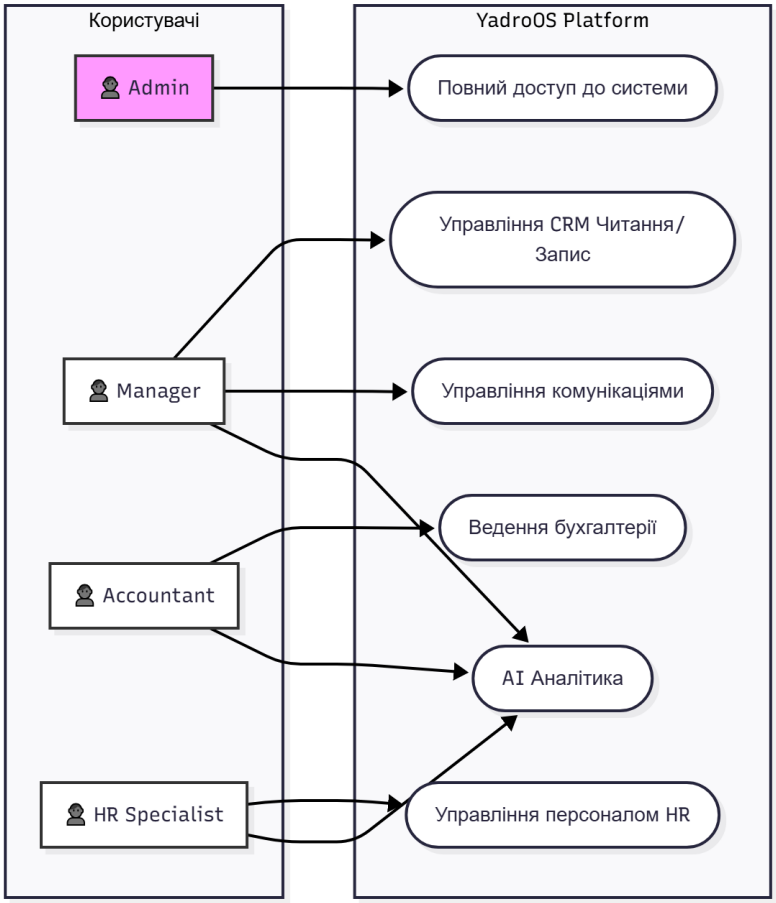


Рис. 2.3 – Діаграма варіантів використання (Use Case)

Сформована матриця ролей та сценаріїв використання стала концептуальним підґрунтям для побудови архітектури безпеки на основі рольового розмежування доступу (RBAC). Такий підхід забезпечує чітку сегрегацію обов'язків (Segregation of Duties): технічні обмеження унеможливлюють доступ менеджерів до критичних фіскальних налаштувань, водночас захищаючи конфіденційну історію угод від несанкціонованих змін з боку персоналу, що не бере участі у проекті. Це суттєво мінімізує операційні ризики та підвищує загальну надійність системи.

2.2 Обґрунтування вибору технологічного стеку та архітектури

Визначення технологічного стеку є не просто вибором інструментарію, а стратегічним рішенням, що детермінує межі масштабованості системи, її стійкість до пікових навантажень та вартість подальшого супроводу (Total Cost of

Ownership). Виходячи з нефункціональних вимог, сформульованих у розділі 1.5, зокрема щодо латентності інтерфейсу не вище 200 мс та забезпечення гарантованої консистентності фінансових транзакцій, було відхилено класичну архітектуру багатосторінкових сайтів (MPA). Натомість прийнято рішення про імплементацію архітектурного патерну Single Page Application (SPA), що дозволяє наблизити користувацький досвід роботи у веб-браузері до нативних десктопних додатків.

В якості фундаменту рівня представлення (Frontend) обрано бібліотеку React.js. Цей вибір продиктовано необхідністю ефективної роботи з динамічними даними. Традиційна маніпуляція DOM-деревом є ресурсоємною операцією, що при відображенні великих масивів фінансових даних викликає помітні затримки. React вирішує цю проблему через механізм Virtual DOM та алгоритм узгодження (Reconciliation), який мінімізує кількість перемальовувань сторінки, оновлюючи лише ті вузли, стан яких змінився. Це критично важливо для реалізації інтерактивних дашбордів YadroOS, де зміна курсу валют повинна миттєво відображатися у всіх пов'язаних віджетах без повного перезавантаження сторінки.

Оптимізація процесу розробки та збірки проекту забезпечується інструментарієм Vite [21]. На відміну від застарілих бандлерів (наприклад, Webpack), Vite використовує нативні ES-модулі браузера, що забезпечує миттєвий «холодний старт» серверу розробки та гарячу заміну модулів (HMR). Для управління глобальним станом додатку, що є нетривіальною задачею в розподілених системах, застосовано менеджер стану Zustand [22]. Його архітектура, заснована на спрощеній моделі Flux, дозволяє уникнути надмірної складності (Boilerplate), притаманної Redux, водночас забезпечуючи прогнозованість потоків даних між компонентами.

Зважаючи на фінансову специфіку платформи, де ціна помилки вимірюється прямими грошовими збитками, використання динамічної типізації JavaScript було визнано неприпустимим ризиком. У зв'язку з цим, як на клієнтській, так і на серверній стороні впроваджено мову TypeScript. Сувора статична типізація дозволяє виявляти до 90% потенційних помилок (невідповідність типів, звернення до неіснуючих властивостей) ще на етапі компіляції коду, діючи як перший ешелон

захисту якості програмного забезпечення.

Серверний рівень (Backend) реалізовано на базі середовища виконання Node.js. Ключовим архітектурним аргументом тут виступає можливість побудови ізоморфної системи. Використання єдиної мови програмування дозволяє створити спільну бібліотеку інтерфейсів та типів даних (Shared DTO), яка використовується обома частинами додатку. Це гарантує, що структура об'єкта, відправленого сервером, буде ідентично інтерпретована клієнтом. Крім того, асинхронна, подієво-орієнтована архітектура Node.js (Event Loop) ідеально підходить для I/O-intensive задач, таких як численні запити до зовнішніх банківських API та запис логів, дозволяючи обробляти тисячі конкурентних з'єднань на одному потоці.

В якості веб-фреймворку використано Express.js, який реалізує патерн Chain of Responsibility через механізм проміжного програмного забезпечення (Middleware). Це дозволяє гнучко налаштовувати конвеєр обробки запитів, інтегруючи модулі валідації, авторизації та логування.

Підсистема зберігання даних спроектована на базі СУБД PostgreSQL. Вибір саме цієї реляційної системи зумовлений необхідністю суворого дотримання принципів ACID (Atomicity, Consistency, Isolation, Durability). У контексті фінансового обліку, де кожна транзакція впливає на баланс, використання NoSQL рішень MongoDB як до прикладу, несе ризики порушення цілісності даних. Водночас PostgreSQL підтримує роботу з типом даних JSONB, що дозволило реалізувати гібридну схему зберігання: суворі реляційні таблиці для фінансів та гнучкі JSON-структури для журналу аудиту подій. Це рішення поєднує надійність SQL з гнучкістю документо-орієнтованих баз даних.

Взаємодія прикладної логіки з базою даних абстрагована за допомогою ORM (Object-Relational Mapping) нового покоління – Prisma [24]. На відміну від класичних ORM, що базуються на класах моделей, Prisma використовує декларативну схему даних для генерації суворо типізованого клієнта. Це не лише прискорює розробку, а й повністю нівелює ризики SQL-ін'єкцій, оскільки всі запити автоматично параметризуються на рівні драйвера.

Загальна топологія системи відповідає класичній триланковій архітектурі

(Three-Tier Architecture) [25, 26].

Рівень представлення (Presentation Layer): SPA-додаток у браузері, що відповідає виключно за рендеринг інтерфейсу та збір вхідних даних.

Рівень бізнес-логіки (Application Layer): RESTful API сервер, що інкапсулює правила валідації, розрахунку податків та управління доступом.

Рівень даних (Data Layer): Реляційна база даних, що забезпечує персистентність інформації.

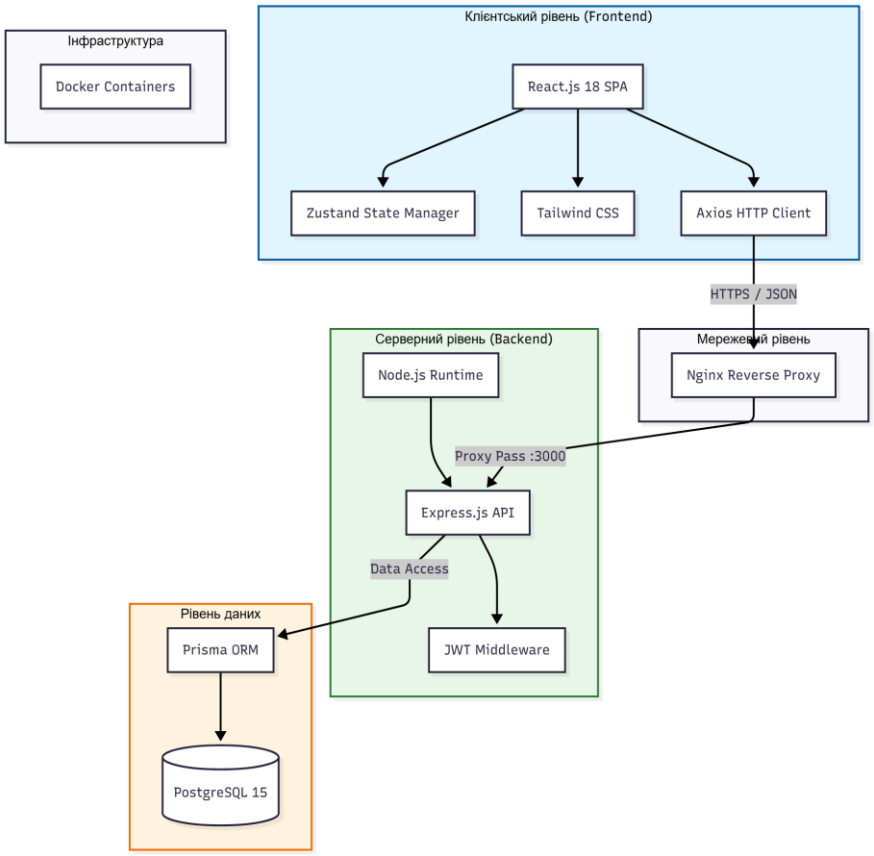


Рис. 2.4 – Структурна схема технологічного стеку

Взаємодія між рівнями відбувається за протоколом HTTP/1.1 з використанням формату JSON для серіалізації даних. Безпека транспортного рівня забезпечується протоколом TLS 1.3. Обраний технологічний стек (PERN: PostgreSQL, Express, React, Node) є де-факто промисловим стандартом для побудови масштабованих корпоративних систем, забезпечуючи оптимальний баланс між швидкістю розробки, продуктивністю та надійністю, що повністю задовольняє вимоги предметної області.

2.3 Проектування бази даних: інфологічна та даталогічна моделі

Архітектурна стійкість та продуктивність інформаційної системи значною мірою детермінуються якістю проектування рівня даних. У процесі розробки платформи YadroOS було застосовано класичний підхід до моделювання, що передбачає поетапну трансформацію інфологічної (змістовної) моделі предметної області у даталогічну (фізичну) схему реляційної бази даних. Проектування здійснювалося з дотриманням правил нормалізації до третьої нормальної форми (3NF), що забезпечує усунення надлишковості даних, мінімізацію аномалій при вставці та оновленні (Update Anomalies), а також гарантує логічну цілісність інформаційних потоків.

Фізична імплементація сховища даних реалізована на базі системи управління базами даних PostgreSQL версії 15 [18]. Вибір цієї СУБД обумовлений її відповідністю стандартам ACID та розвиненою підтримкою складних типів даних. Опис схеми даних виконано у декларативному форматі за допомогою мови опису схем Prisma Schema Language (PSL) [24]. Такий підхід дозволяє розглядати структуру бази даних як код (Database-as-Code), автоматизувати процес міграцій та забезпечити синхронізацію між станом бази даних та типами у прикладному коді.

Особливу увагу при проектуванні даталогічної моделі було приділено стратегії типізації даних та генерації первинних ключів. Відмова від традиційних автоінкрементних цілочисельних ідентифікаторів (Sequence ID) на користь стандарту UUID (Universally Unique Identifier) версії 4 продиктована вимогами безпеки та масштабованості. Використання 128-бітних псевдовипадкових ідентифікаторів виконує функцію превентивного захисту від атак типу ID Enumeration (перебір ідентифікаторів), унеможливлюючи отримання доступу до даних конкурентів шляхом простої модифікації URL-запиту. Крім того, UUID дозволяє генерувати унікальні ключі на стороні клієнтського додатку без необхідності звернення до сервера БД, що знижує навантаження на систему при масовому імпорті даних та спрощує потенційне злиття (Sharding/Merge) різних баз даних у майбутньому.

Для зберігання критичних фінансових метрик, таких як суми транзакцій, ставки податків та баланси рахунків, безальтернативно обрано тип даних фіксованої точності DECIMAL. Використання стандартних типів з плаваючою комою (FLOAT або DOUBLE PRECISION) у фінансових системах є неприпустимим через особливості стандарту IEEE 754, який призводить до накопичення похибок округлення при виконанні арифметичних операцій. Тип DECIMAL дозволяє зберігати точні значення з фіксованою кількістю знаків після коми, гарантуючи математичну коректність розрахунків до сотих часток грошової одиниці.

Водночас, жорстка реляційна схема доповнена гнучкістю NoSQL-підходу завдяки використанню типу JSONB (Binary JSON). Це дозволило реалізувати зберігання слабоструктурованих даних, таких як метадані журналу аудиту (Audit Log) та динамічні налаштування профілів користувачів, без необхідності постійної модифікації схеми таблиць (DDL-операцій). JSONB у PostgreSQL підтримує індексацію, що забезпечує високу швидкість вибірки навіть за вкладеними ключами JSON-об'єктів.

Інфологічна структура бази даних декомпозована на три ключові функціональні домени, що відображають логіку бізнес-процесів організації: домен управління ідентифікацією (IAM), домен управління взаємовідносинами (CRM) та фінансовий домен (Finance).

У рамках домену «Управління персоналом» реалізовано архітектурний патерн розділення сутностей автентифікації (User) та бізнес-профілю (Employee). Зв'язок між ними реалізовано за типом «один-до-одного» (1:1). Таке рішення дозволяє розмежувати конфіденційні дані доступу (хеші паролів, токени відновлення) та операційні дані (посада, відділ, ставка). Практична цінність цього підходу полягає у можливості деактивації доступу звільненого співробітника (блокування запису User) при збереженні повної історії його фінансових нарахувань та активності у записі Employee, що є критичним для кадрового аудиту.

Домен «CRM та Продажі» базується на ієрархічній структурі, де кореневою сутністю виступає Client, а підпорядкованою – Deal (Угода). Зв'язок «один-до-

багатьох» (1:N) дозволяє зберігати необмежену історію угод для кожного контрагента. Важливим аспектом є реалізація зв'язку між Deal та Employee, що закріплює персональну відповідальність менеджера за результат угоди.

Фінансовий домен є найбільш консервативним елементом схеми. Він включає сутності первинних документів (Invoice) та фактичних транзакцій (Transaction). Ці таблиці слугують джерелом правди (Source of Truth) для побудови динамічних звітів P&L. Специфікою цього блоку є його незмінність: архітектура системи забороняє фізичне видалення записів фінансового характеру.

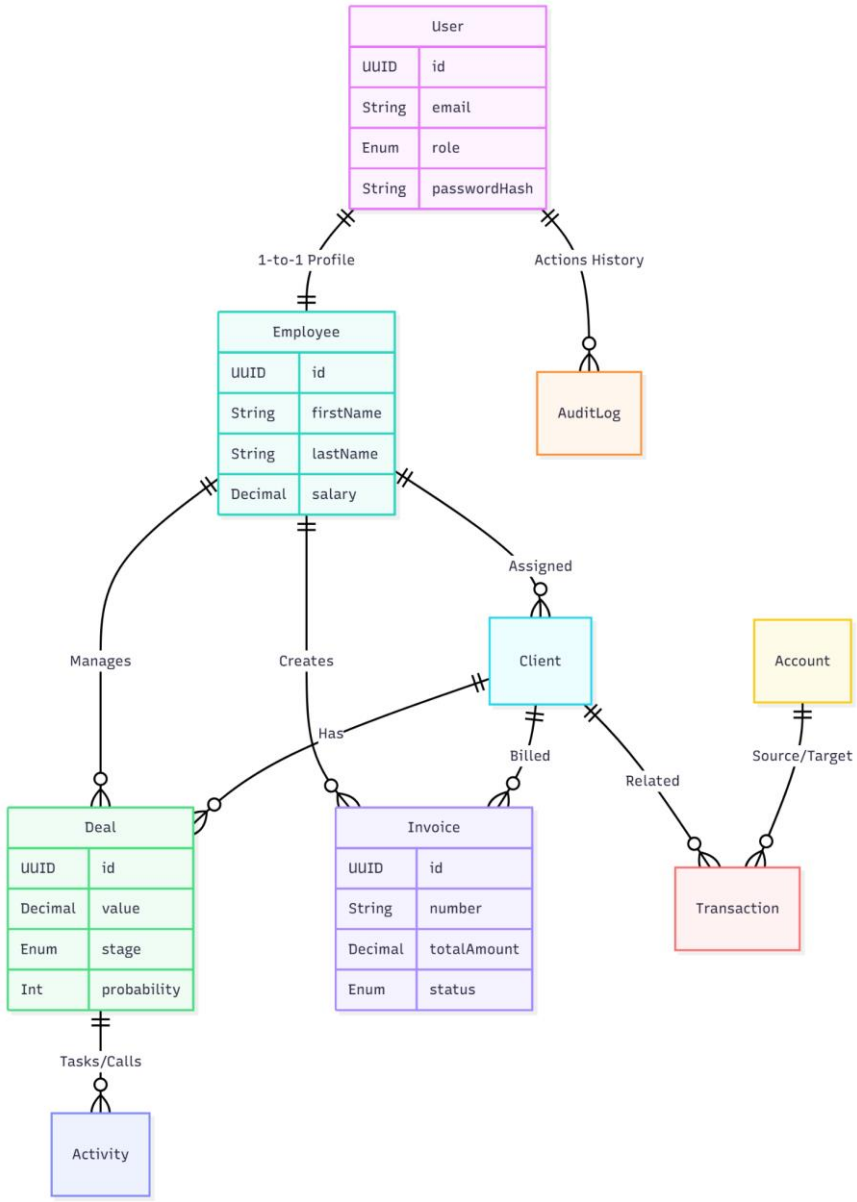


Рис. 2.5 – ER-діаграма бази даних (Entity Relationship Diagram)

Критично важливим аспектом є забезпечення цілісності даних на рівні зовнішніх ключів.

Окремим, наскрізним елементом архітектури є підсистема безпеки, представлена таблицею AuditLog. Ця сутність спроектована за принципом Append-Only Ledger (журнал, доступний лише для запису).

На рівні логіки бази даних та ORM реалізовано механізми, що унеможливлюють операції UPDATE або DELETE для записів аудиту, гарантуючи цілісність юридично значущого сліду дій користувачів.

Забезпечення посилальної цілісності (Referential Integrity) реалізовано через механізм зовнішніх ключів (Foreign Keys) з налаштуванням специфічних правил поведінки при видаленні батьківських записів. Для критичних зв'язків застосовано стратегію ON DELETE RESTRICT.

Наприклад, система блокує спробу видалення запису клієнта (Client), якщо у системі існують пов'язані з ним інвойси (Invoice) або активні угоди. Це запобігає виникненню «сирітських» записів (Orphaned Records), які могли б призвести до викривлення фінансової звітності. Водночас для допоміжних сутностей, таких як токени сесій, використовується стратегія ON DELETE CASCADE, що забезпечує автоматичне очищення сміття.

Для оптимізації продуктивності вибірки даних створено систему індексів. Окрім стандартних індексів для первинних ключів, реалізовано унікальні індекси (Unique Index) для полів, що вимагають гарантованої неповторності (наприклад, email у таблиці User та номер інвойсу у таблиці Invoice). Для прискорення пошуку та фільтрації у фінансових звітах додано складені індекси (Composite Indices) по полях дати транзакції та статусу оплати.

Таким чином, розроблена даталогічна модель є нормалізованою, захищеною та оптимізованою для високих навантажень.

Поєднання суворої реляційної структури для фінансового ядра з гнучкими JSONB-полями для логування створює надійний фундамент для програмної реалізації бізнес-логіки системи.

2.4 Розробка системи безпеки та розмежування прав доступу

Специфіка предметної області, що включає обробку чутливих фінансових транзакцій та персональних даних (PII) співробітників, детермінує пріоритетність питань інформаційної безпеки при проектуванні архітектури YadroOS. Система захисту розроблена з урахуванням вектора актуальних загроз, регламентованих у звіті OWASP Top 10, та базується на парадигмі «глибокого ешелонованого захисту» (Defense in Depth). Ця концепція передбачає побудову багатошарової оборони, де компрометація одного з бар'єрів не призводить до повного зламу системи.

Фундаментом архітектури безпеки є чітка дихотомія двох процесів: аутентифікації (підтвердження цифрової ідентичності суб'єкта) та авторизації (перевірка наявності прав на виконання конкретної операції). В умовах мікросервісної та розподіленої природи сучасних веб-додатків, традиційна модель на основі серверних сесій (Stateful Session) була визнана неефективною через проблеми з горизонтальним масштабуванням. Натомість імплементовано механізм безстанової (Stateless) аутентифікації з використанням стандарту JSON Web Token (JWT) згідно зі специфікаціями RFC 7519.

Для нівелювання вразливостей, притаманних JWT-токенам при зберіганні на клієнті, реалізовано архітектурний патерн подвійних токенів (Dual Token Pattern). Цей підхід вирішує проблему вибору безпечного сховища у браузері. Зберігання токенів у localStorage робить систему вразливою до XSS-атак (Cross-Site Scripting), тоді як використання виключно Cookies відкриває вектор атак CSRF (Cross-Site Request Forgery).

Реалізована схема працює наступним чином: короткостроковий Access Token (термін життя – 15 хвилин) зберігається виключно в оперативній пам'яті клієнтського додатку (у змінних стану менеджера Zustand). Це робить його недоступним для шкідливих скриптів після перезавантаження сторінки або закриття вкладки. Довгостроковий Refresh Token, необхідний для автоматичного оновлення сесії, передається та зберігається у спеціальному Cookie з прапорцями

HttpOnly (заборона доступу через JavaScript), Secure (передача тільки через HTTPS) та SameSite=Strict (заборона відправки при крос-доменних запитах). Така конфігурація фактично унеможливлює викрадення сесії через XSS та блокує підробку запитів через CSRF.

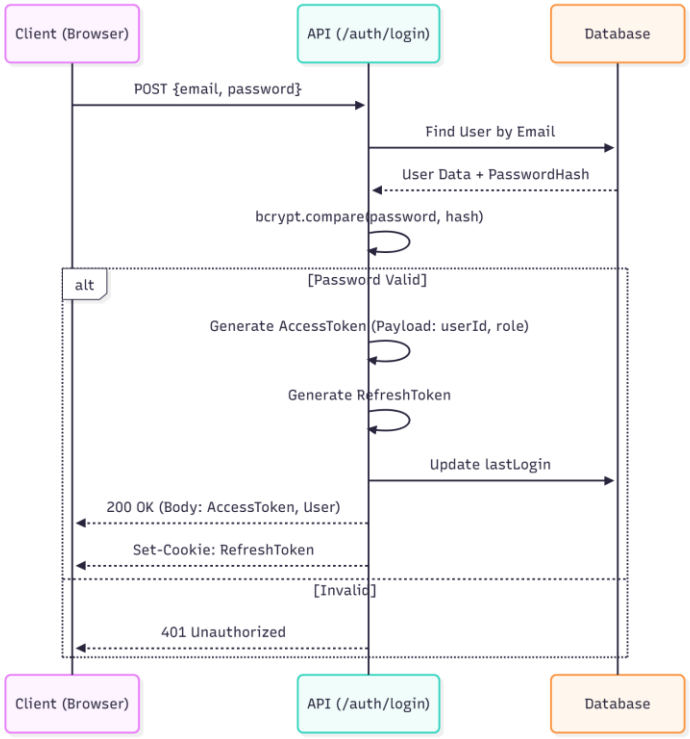


Рис. 2.6 – Діаграма послідовності процесу автентифікації (Sequence Diagram)

Після успішного проходження етапу аутентифікації вступає в дію підсистема авторизації. Відповідно до рекомендацій стандарту NIST SP 800-162 [15], у платформі імплементовано рольову модель доступу (RBAC). На рівні бази даних ролі зафіксовані як перелічуваний тип даних (ENUM), що створює жорсткий інваріант: користувач не може мати роль, не передбачену архітектором системи.

Ієрархія прав побудована за принципом найменших привілеїв. Роль «Адміністратор» володіє правами на деструктивні дії (видалення) та доступ до журналу аудиту, проте обмежена в операційній діяльності. Ролі «Менеджер» та «HR-спеціаліст» мають доступ виключно до своїх функціональних доменів (CRM та кадри відповідно). Технічна реалізація контролю здійснюється через ланцюжок проміжного програмного забезпечення (Middleware Pipeline) на сервері Node.js. Кожен HTTP-запит перехоплюється спеціальним обробником, який дешифрує

токен, витягує інформацію про роль та зіставляє її з необхідним рівнем доступу для даного ендпоінту (Endpoint). Якщо права відсутні, виконання бізнес-логіки блокується, а клієнту повертається помилка з кодом 403 Forbidden.

Третій ешелон захисту стосується цілісності даних та криптографічної стійкості. Зважаючи на те, що база даних може бути скомпрометована (наприклад, через інсайдерові загрози), паролі користувачів ніколи не зберігаються у відкритому вигляді. Застосовано алгоритм хешування bcrypt, який використовує техніку «соління» (salting) та адаптивний фактор вартості (work factor). Це робить атаки типу «райдужних таблиць» (Rainbow Tables) та перебір GPU-кластерами економічно недоцільними.

Захист від атак типу Injection (SQL/NoSQL) реалізовано на двох рівнях. По-перше, використання ORM Prisma забезпечує автоматичну параметризацію SQL-запитів. По-друге, оскільки мова TypeScript гарантує типи лише на етапі компіляції, для валідації вхідних даних під час виконання (Runtime) інтегровано бібліотеку Zod. Кожен JSON-об'єкт, що надходить від клієнта, проходить сувору перевірку на відповідність схемі: перевіряються типи даних, довжина рядків, формати email та відсутність заборонених символів.

Транспортний рівень захищено примусовим використанням протоколу TLS 1.3, що гарантує конфіденційність трафіку. Розроблена комплексна система безпеки, що поєднує Dual Token аутентифікацію, серверну RBAC-авторизацію та сувору Runtime-валідацію, створює надійне середовище для функціонування децентралізованої організації, мінімізуючи ризики як зовнішніх атак, так і внутрішніх зловживань.

2.5 Проектування інтерфейсу користувача та взаємодії з системою

У контексті розробки корпоративних інформаційних систем інтерфейс користувача (User Interface – UI) розглядається не як естетична надбудова, а як критичний інструмент зниження операційної латентності. Для децентралізованої організації, де відсутній фактор прямого адміністративного нагляду, якість

інтерфейсу безпосередньо корелює зі швидкістю адаптації нових співробітників (Onboarding time) та точністю введення фінансових даних. Виходячи з цього, при проектуванні візуальної частини платформи YadroOS було застосовано методологію проектування, орієнтованого на користувача (User-Centered Design – UCD), яка ставить на перше місце ергономіку та мінімізацію когнітивного навантаження.

Архітектура інтерфейсу базується на методології Атомарного Дизайну (Atomic Design) Бреда Фроста [28], яка дозволяє декомпозувати UI на ієрархічні рівні, повністю синхронізовані з компонентною моделлю бібліотеки React. Фундаментальним рівнем системи є «атоми» – неподільні елементи, такі як кнопки, поля введення (Inputs) або типографічні стилі, що не мають самостійного функціонального навантаження. Шляхом комбінації атомів формуються «молекули» – прості функціональні групи, наприклад, форма пошуку, що складається з поля вводу та кнопки дії. На вищому рівні абстракції знаходяться «організми» – складні, автономні частини інтерфейсу, такі як навігаційна панель (Sidebar) або віджет фінансового звіту, що відповідають за виконання завершених сценаріїв користувача. Вершиною ієрархії виступають шаблони (Templates), які визначають каркас сторінки та правила розміщення компонентів. Такий інженерний підхід гарантує візуальну та поведінкову консистентність системи: кнопка підтвердження транзакції має ідентичний вигляд та механіку зворотного зв'язку як у CRM-модулі, так і в бухгалтерському контурі.

Технологічна реалізація візуального стилю виконана з використанням CSS-фреймворку Tailwind CSS [29]. Вибір на користь підходу Utility-First продиктований необхідністю суворого дотримання дизайн-системи. Замість написання кастомних CSS-класів, що з часом призводить до конфліктів специфічності та розростання кодової бази, використано централізований конфігураційний файл (tailwind.config.js). Цей файл виступає єдиним джерелом правди (Single Source of Truth) для палітри кольорів, метрик відступів та типографіки. Особливу увагу приділено адаптивності інтерфейсу, реалізованій за принципом Mobile First: базові стилі проектуються для мобільних пристроїв, а

адаптація під десктопні екрани здійснюється через медіа-запити. Це забезпечує коректне відображення навігаційних елементів, які на великих моніторах фіксуються у бічній панелі, а на мобільних пристроях автоматично трансформуються у приховане меню ("гамбургер").

Інформаційна архітектура додатку спроектована згідно з принципами структурування веб-просторів, описаними Л. Розенфельдом [30]. Компонування основного екрану (App Shell) реалізовано через поділ робочого простору на три функціональні зони. Глобальна навігація, закріплена у лівій частині екрану (Sidebar), забезпечує персистентний доступ до ключових модулів платформи незалежно від глибини перегляду, дозволяючи перемикатися між контекстами в один клік. Верхня панель (Top Bar) виконує роль контекстної області, містячи елементи глобального пошуку, центр сповіщень та меню профілю користувача. Центральну частину займає динамічна робоча область (Workspace), контент якої змінюється залежно від активного маршруту, забезпечуючи фокусування уваги користувача на поточному завданні без візуального шуму.

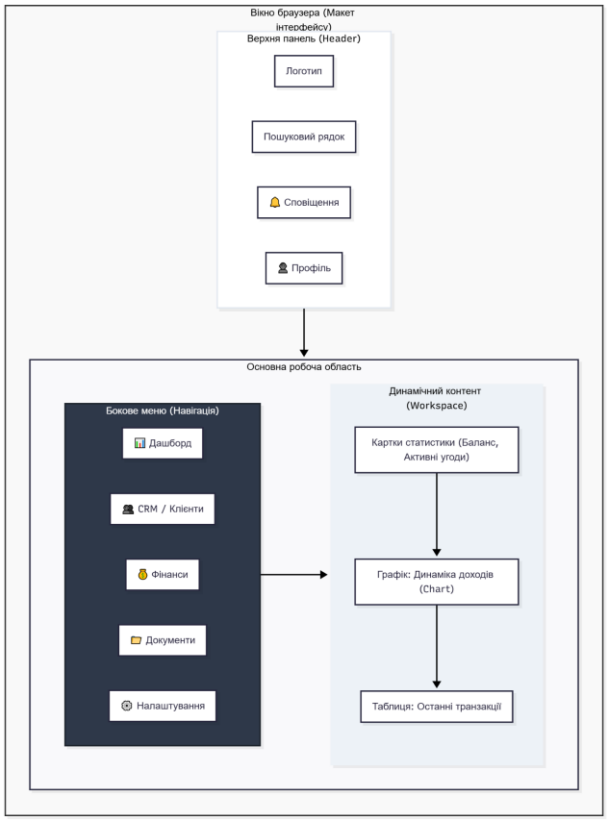


Рис. 2.7 – Схематичне відображення головного екрану системи (Wireframe)

При проектуванні логіки взаємодії (Interaction Design) було імплементовано фундаментальні евристики юзабіліті Якоба Нільсена. Для забезпечення прозорості стану системи (Visibility of system status) реалізовано механізми миттєвого зворотного зв'язку. Будь-яка асинхронна дія, наприклад, відправка інвойсу, супроводжується візуальною індикацією прогресу (скелетон-завантаження або спінери) та фінальним спливаючим повідомленням (Toast notification) про результат операції. Це нівелює невизначеність, даючи користувачеві розуміння, що система прийняла його запит.

Зниження когнітивного навантаження досягається шляхом використання знайомих метафор реального світу (Match between system and the real world). Навігація оперує зрозумілими сутностями: "Робочий стіл", "Картка угоди", "Кошик".

Візуальна складова базується на принципах естетичного мінімалізму та концепції «Clean UI», де пріоритет надається корисному контенту та негативному простору ("повітрю"). Це дозволяє оператору фокусуватися на прийнятті рішень, не відволікаючись на декоративні елементи. Для відображення фінансових показників імплементовано інтерактивні графічні віджети, які трансформують сухі табличні дані у наочні тренди динаміки прибутків, дозволяючи оцінити стан справ за декілька секунд.

Окремим завданням було проектування інтерфейсу управління угодами, реалізованого у вигляді інтерактивної Kanban-дошки. Цей елемент використовує патерн Drag-and-Drop, дозволяючи змінювати статус угоди шляхом простого перетягування картки між колонками етапів воронки продажів. Така візуалізація бізнес-процесу робить його інтуїтивно зрозумілим та прозорим для всіх учасників команди.

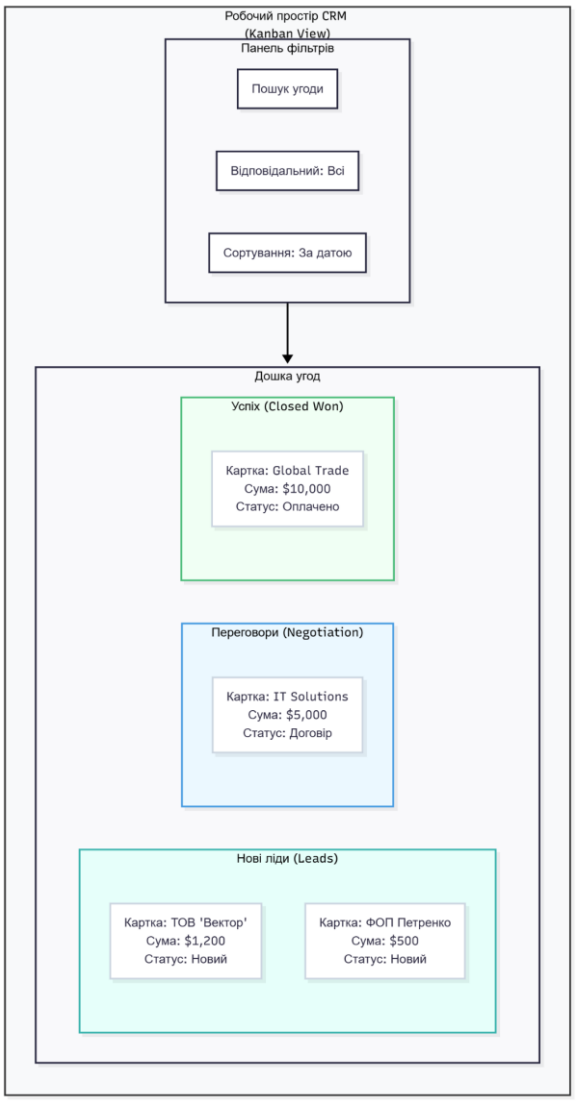


Рис. 2.8 – Схематична організація інтерфейсу управління угодами (Kanban)

Таким чином, розроблений інтерфейс поєднує в собі інженерну суворість атомарного підходу з ергономічними принципами UCD, створюючи ефективне робоче середовище для користувачів децентралізованої платформи.

2.6 Аналіз проектних ризиків та стратегії їх мінімізації

Інженерне проектування складних програмних комплексів не може обмежуватися виключно реалізацією функціональних вимог; невід'ємною складовою процесу є ідентифікація, класифікація та розробка контрзаходів щодо потенційних загроз. Враховуючи специфіку платформи YadroOS, яка оперує

фінансовими активами децентралізованої організації, модель загроз виходить за межі суто технічних збоїв, охоплюючи операційні та антропогенні фактори. Для забезпечення необхідного рівня відмовостійкості (Resilience) було застосовано методику превентивного аналізу ризиків, що дозволило імплементувати архітектурні рішення, спрямовані на мінімізацію ймовірності настання критичних інцидентів.

Першочергову увагу приділено групі інфраструктурних ризиків, пов'язаних із залежністю від хмарних провайдерів. Оскільки система розгортається на віртуальних потужностях (VPS/Cloud), існує ненульова ймовірність тимчасової недоступності сервісів (Downtime) або навіть повної втрати даних внаслідок фізичних збоїв у дата-центрах. Стратегія мінімізації (Mitigation Strategy) цієї групи ризиків базується на принципах резервування та портативності. Використання технології контейнеризації Docker дозволяє абстрагувати додаток від конкретного "заліза", забезпечуючи можливість екстреної міграції на резервні сервери іншого провайдера протягом регламентованого часу RTO (Recovery Time Objective), який для даної системи встановлено на рівні 30 хвилин.

Для захисту від втрати даних впроваджено стратегію Point-in-Time Recovery (PITR). На рівні СУБД PostgreSQL налаштовано механізм WAL-архівації (Write-Ahead Logging), що дозволяє відновити стан бази даних на будь-яку секунду часу, а не лише на момент створення повного бекапу. Резервні копії автоматично шифруються та реплікуються у географічно віддалене хмарне сховище (S3-сумісне), що гарантує збереження фінансової історії навіть у випадку катастрофічних подій у основному дата-центрі. Цільовий показник точки відновлення (RPO) становить менше 5 секунд, що є прийнятним для бізнес-моделі замовника.

Окремий кластер загроз формують ризики інформаційної безпеки, де критичним вектором атаки визначено компрометацію облікових записів. В умовах віддаленої роботи та використання особистих пристроїв співробітників периметр безпеки стає розмитим. Для протидії атакам типу Brute-force на мережевому рівні імплементовано механізм Rate Limiting, який аналізує частоту запитів з однієї IP-

адреси та автоматично блокує підозрілий трафік ще до моменту перевірки пароля. Додатково, архітектура Dual Token (описана у підрозділі 2.4) нівелює наслідки перехоплення трафіку, оскільки короткий час життя токена доступу робить його марним для зловмисника вже через 15 хвилин.

Найбільш складними для прогнозування є операційні ризики, зумовлені людським фактором. Помилкове введення даних менеджером або випадкове видалення критичних транзакцій може призвести до викривлення фінансової звітності. Для нівелювання цієї загрози на рівні бази даних реалізовано патерн «м'якого видалення» (Soft Delete). Замість фізичного знищення записів за допомогою команди DELETE, система лише оновлює поле deletedAt, виключаючи запис з активної вибірки, але зберігаючи його фізично. Це дозволяє адміністратору миттєво відновити випадково видалені сутності без необхідності розгортання резервних копій.

Крім того, для запобігання введенню некоректних фінансових даних реалізовано багаторівневу валідацію: на стороні клієнта, для UX, та на стороні сервера (для безпеки) з використанням бібліотеки Zod.

Це гарантує, що в базу даних потрапляють лише ті структури, які пройшли перевірку на відповідність бізнес-логіці.

Імплементація розробленого комплексу заходів інформаційної безпеки дозволяє констатувати нейтралізацію критичних векторів атак, визначених у матриці ризиків проекту. Втім, стійкість архітектури не розглядається як статична величина: стратегія захисту передбачає перехід до моделі DevSecOps з ітеративним переглядом моделі загроз (Threat Modeling) корелюючи з появою нових типів вразливостей.

Подібна динамічна адаптація протоколів безпеки є безальтернативною умовою стабільної експлуатації платформи в умовах горизонтального масштабування та зростання транзакційного навантаження.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ІНФРАСТРУКТУРНЕ ЗАБЕЗПЕЧЕННЯ ПЛАТФОРМИ

3.1 Обґрунтування вибору та реалізація архітектурних рішень

Практична імплементація клієнтської складової платформи YadroOS виконана у відповідності до архітектурної парадигми односторінкового веб-застосунку (Single Page Application – SPA). Даний підхід, на відміну від класичної моделі серверного рендерингу (SSR), передбачає завантаження єдиного каркасного шаблону (App Shell) при первинній ініціалізації сесії. Подальша навігація та взаємодія користувача з системою відбуваються без перезавантаження сторінки, шляхом динамічної підміни вмісту контейнерів та асинхронного запиту даних через REST API. Таке архітектурне рішення дозволило мінімізувати паразитний трафік: після завантаження статичних ресурсів (JavaScript-бандлів, стилів, шрифтів) мережевий канал використовується виключно для передачі корисного навантаження (Payload) у форматі JSON, що критично знижує латентність інтерфейсу в умовах нестабільного мобільного зв'язку.

Технологічним ядром системи виступає бібліотеку React.js версії 18. Вибір цього інструменту базується не стільки на його популярності, скільки на ефективності алгоритму узгодження (Reconciliation), що лежить в основі роботи віртуального DOM. У складних фінансових інтерфейсах, насичених табличними даними, пряма маніпуляція DOM-деревом є ресурсоємною операцією, що призводить до блокування головного потоку браузера. React вирішує цю проблему через архітектуру Fiber, яка дозволяє розбивати процес рендерингу на пріоритетні чанки. У реалізованому проекті це забезпечує плавність роботи конструктора інвойсів: при додаванні десятків позицій товарів та динамічному перерахунку податкових ставок система оновлює лише змінені текстові вузли, не перемальовуючи весь інтерфейс.

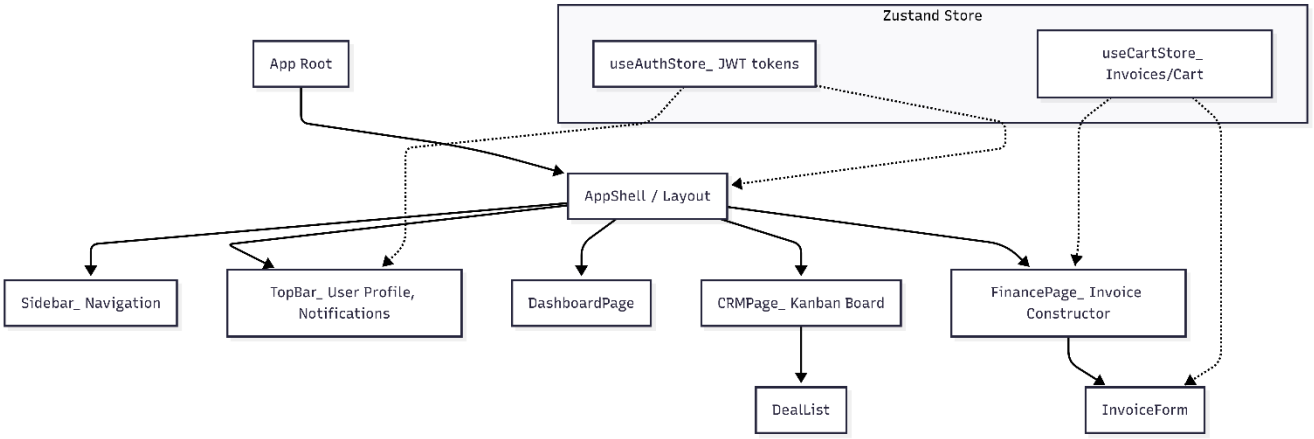


Рис. 3.1 – Ієрархія компонентів React та схема потоків даних

Критичним фактором успішного супроводу програмного продукту є організація кодової бази. Для уникнення проблеми сильної зв'язності (High Coupling), характерної для монолітних фронтенд-додатків, структуру проекту організовано згідно з методологією Feature-Sliced Design (FSD). Ця архітектурна методологія передбачає декомпозицію коду на ізольовані шари за рівнем відповідальності.

Базовий шар Shared акумулює перевикористовувані UI-компоненти (кнопки, інпути, модальні вікна) та утилітарні функції, які є "чистими" і не містять бізнес-логіки.

Рівень Entities описує бізнес-моделі предметної області (наприклад, User, Deal, Invoice) та логіку їх відображення.

Конкретні користувацькі сценарії, такі як "Змінити статус угоди" або "Оплатити рахунок", винесено у шар Features.

Композиція ізольованих фіч у самостійні блоки відбувається на шарі Widgets, а фінальна збірка маршрутів – на рівні Pages.

Така сувора ієрархія залежностей (шар вищого рівня може використовувати лише шари нижчого рівня) гарантує передбачуваність архітектури та спрощує рефакторинг.

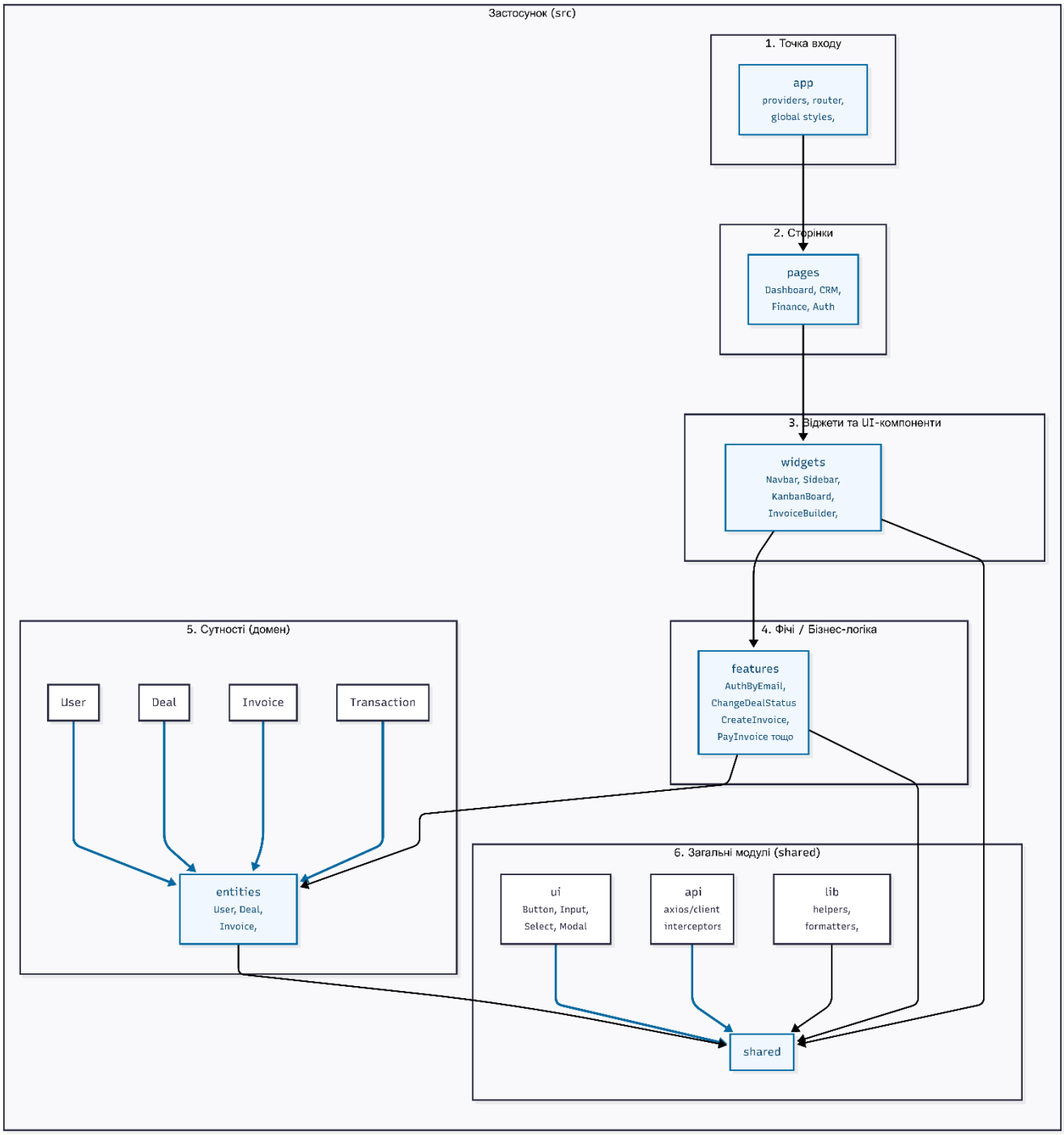


Рис. 3.2 – Структура файлової системи проекту згідно методології FSD

Для забезпечення надійності коду на етапі розробки застосовано мову програмування TypeScript. Впровадження суворої статичної типізації виступає превентивним бар'єром для помилок Runtime. У фінансовому модулі системи це дозволило повністю усунути клас помилок, пов'язаних з некоректними математичними операціями над різними типами даних (наприклад, спроба додати число до рядка валюти). Усі інтерфейси даних (DTO), що отримуються з серверу,

автоматично валідуються та типізуються, що забезпечує наскрізну консистентність даних від бази даних до UI-компонента.

Управління глобальним станом додатку реалізовано за допомогою бібліотеки Zustand. На відміну від громіздкого Redux, що вимагає написання значного обсягу шаблонного коду (Boilerplate) та огортання додатку в численні провайдери, Zustand використовує концепцію атомарних хуків. Це дозволило децентралізувати управління станом: дані авторизації зберігаються в одному сховищі (useAuthStore), а стан кошика інвойсів – в іншому (useCartStore), що покращує читабельність коду та оптимізує ре-рендеринг.

Особливістю реалізації є інтеграція Middleware для персистентності стану. Критичні дані сесії (JWT-токени, налаштування теми, обрана мова) автоматично синхронізуються з локальним сховищем браузера (localStorage). Завдяки цьому реалізовано механізм відновлення контексту: при випадковому закритті вкладки браузера користувач не втрачає незбережені дані форми та залишається авторизованим у системі, що наближає досвід взаємодії до нативних десктопних додатків.

3.2 Серверна реалізація та обробка даних

Програмна реалізація серверної складової платформи (Backend) виконана на базі середовища виконання Node.js. Вибір цієї технології, на противагу традиційним багатопотоковим платформам (Thread-based), таким як Java або .NET, обумовлений специфікою навантаження фінансової системи, яка характеризується інтенсивним введенням-виведенням даних (I/O-intensive). В основі архітектури Node.js лежить модель неблокуючого введення-виведення, реалізована через патерн «Реактор» (Reactor Pattern) та бібліотеку libuv. Це дозволяє ефективно делегувати операції читання з бази даних або запити до банківських API операційній системі, не блокуючи основний потік виконання.

Фундаментальною відмінністю обраної архітектури є використання єдиного потоку (Single Thread) та циклу подій (Event Loop). У класичних серверних моделях

(наприклад, Apache HTTP Server) обробка кожного клієнтського запиту ініціює створення окремого потоку або процесу. При масштабуванні до тисяч одночасних з'єднань (C10k problem) це призводить до експоненційного зростання споживання оперативної пам'яті та витрат процесорного часу на перемикання контексту (Context Switching). Node.js нівелює цю проблему, обробляючи тисячі конкурентних підключень у межах одного процесу з мінімальними накладними витратами.

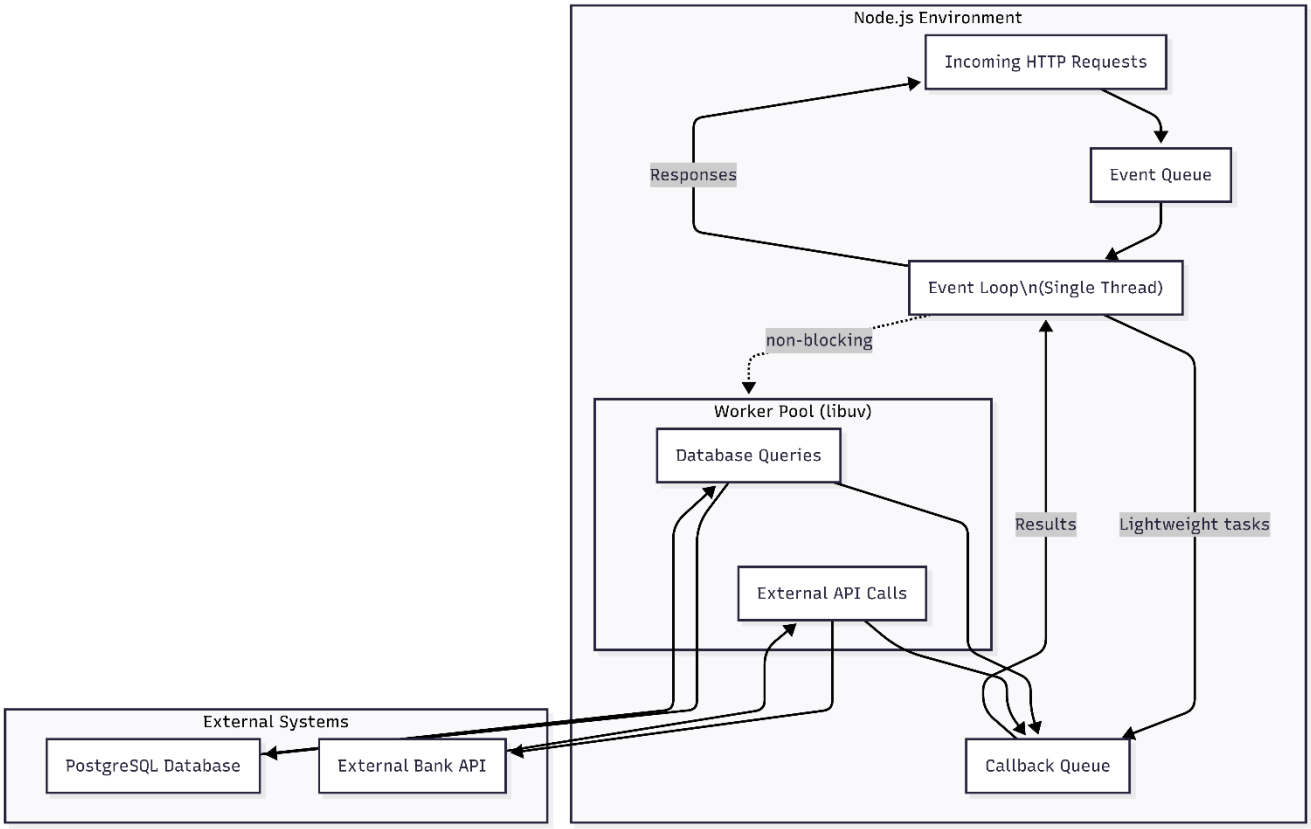


Рис. 3.3 – Схема роботи циклу подій (Event Loop) у середовищі Node.js

Взаємодія між клієнтською та серверною частинами платформи побудована за архітектурним стилем REST (Representational State Transfer). Проектування API здійснювалося з дотриманням принципів ресурсно-орієнтованої архітектури: кожна сутність (Угода, Інвойс, Користувач) представлена унікальним URI, а маніпуляції над ними виконуються через стандартизовані методи протоколу HTTP (GET, POST, PUT, DELETE). Для забезпечення інтероперабельності та автоматичної генерації документації використано специфікацію OpenAPI 3.0. Це дозволяє фронтенд-команді генерувати клієнтський код (SDK) автоматично,

мінімізуючи ризики розбіжності інтерфейсів.

Як каркас веб-додатку використано мікрофреймворк Express.js. Архітектура обробки запитів базується на патерні «Ланцюжок обов'язків» (Chain of Responsibility), реалізованому через механізм проміжного програмного забезпечення (Middleware). Вхідний HTTP-запит проходить через конвеєр послідовних обробників, кожен з яких виконує ізольовану функцію.

На першому етапі глобальні middleware (наприклад, helmet, cors) забезпечують базовий захист заголовків та налаштування політики походження (CORS).

Наступний шар – синтаксичний аналіз тіла запиту (body-parser) та десеріалізація JSON.

Далі вступає в дію модуль авторизації, який валідує JWT-токен та збагачує об'єкт запиту контекстом користувача.

Фінальним бар'єром перед бізнес-логікою виступає валідатор схеми даних (Zod Middleware) та модуль RBAC, що перевіряє права доступу. Такий підхід дозволяє дотримуватися принципу DRY (Don't Repeat Yourself), виносячи наскрізну функціональність за межі контролерів.

Рівень доступу до даних (Data Access Layer) реалізовано за допомогою ORM Prisma. Цей інструмент представляє нову парадигму взаємодії з БД, відмінну від класичних патернів Active Record чи Data Mapper. Prisma використовує власну мову опису схем (DSL) для генерації бінарного клієнта, оптимізованого під конкретну платформу. Головною перевагою такого підходу є повна типобезпека (Type Safety).

На відміну від традиційних Query Builders, де помилки в іменах полів виявляються лише під час виконання (Runtime), Prisma інтегрується з компілятором TypeScript. Спроба розробника звернутися до поля, якого не існує в схемі бази даних, або передати рядок у числове поле, призведе до помилки компіляції проекту. Це створює надійний захисний бар'єр, що унеможливлює цілий клас дефектів, пов'язаних з неузгодженістю структури даних, що є критично важливим для стабільності фінансового ядра системи YadroOS.

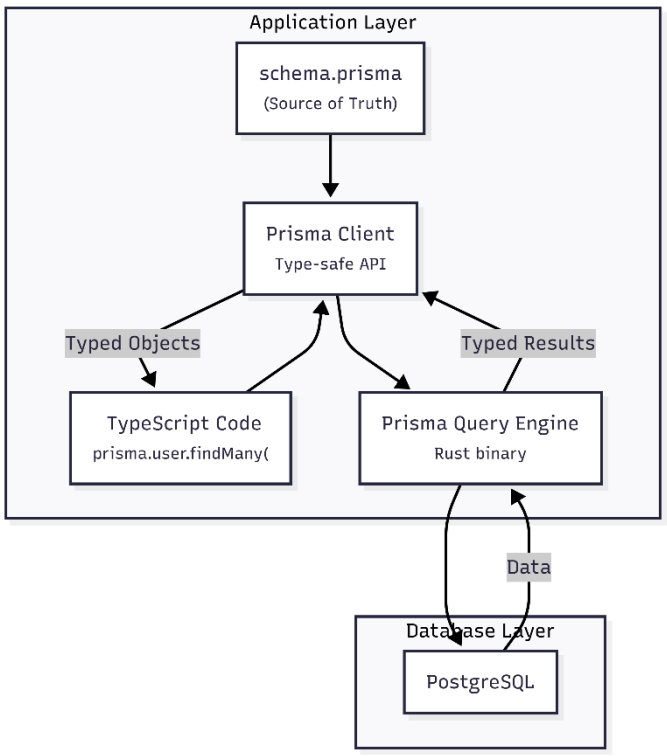


Рис. 3.4 – Архітектура взаємодії Prisma Client з базою даних

3.3 Інфраструктурне забезпечення та контейнеризація

Забезпечення надійності експлуатації програмного комплексу в гетерогенних середовищах вимагає переходу від ручного налаштування серверів до парадигми «Інфраструктура як код» (Infrastructure as Code – IaC). Для гарантування повної відтворюваності середовища розгортання та ізоляції процесів у проекті YadroOS імплементовано технологію контейнеризації на базі платформи Docker [34].

Архітектурна перевага обраного підходу над класичною апаратною віртуалізацією (Hardware Virtualization) полягає у відсутності оверхеду на емуляцію заліза. Якщо гіпервізори (VMware, Hyper-V) вимагають запуску повноцінної гостьової ОС для кожного екземпляра додатку, то Docker-контейнери використовують спільне ядро хост-системи Linux, ізолюючи процеси за допомогою механізмів просторів імен (Namespaces) та контрольних груп (cgroups). Це дозволяє досягти нативної продуктивності при суворому розмежуванні файлових систем та мережевих інтерфейсів, що є критичним для економічно ефективного хостингу мікросервісів на обмежених обчислювальних потужностях.

Оркестрація сервісів реалізована декларативно через інструмент Docker Compose. Інфраструктурна карта проекту (див. Рис. 3.5) описує мультиконтейнерну топологію, що складається з трьох ізольованих, але об'єднаних у віртуальну приватну мережу компонентів:

Backend Service: Контейнер на базі образу Node.js Alpine, що містить бізнес-логіку та API.

Database Service: Екземпляр PostgreSQL 15 з підключеним персистентним сховищем (Docker Volume) для збереження даних поза життєвим циклом контейнера.

Proxy Service: Веб-сервер Nginx, що виступає єдиною точкою входу в систему.

Така конфігурація забезпечує так званий «паритет середовищ» (Dev/Prod Parity). Розробник, розгортаючи проект локально однією командою, отримує оточення, бінарно ідентичне тому, що працює на продуктивному сервері. Це повністю нівелює клас помилок «працює на моїй машині», викликаних розбіжностями версій бібліотек або системних залежностей.

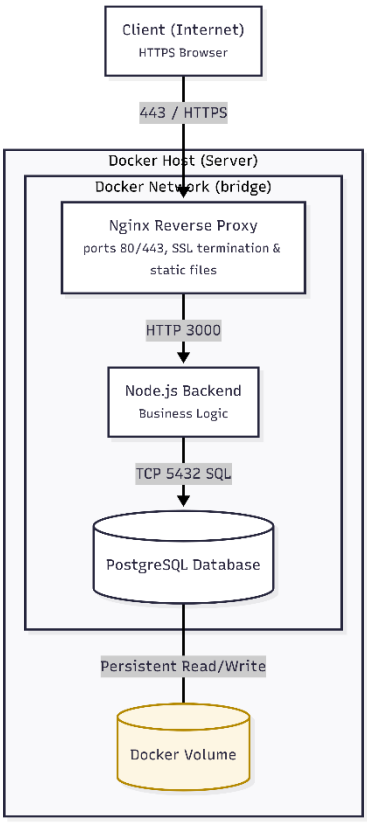


Рис. 3.5 – Схема розгортання контейнеризованої інфраструктури

Конфігурування системи виконано у суворій відповідності до третього принципу методології «The Twelve-Factor App» (Config). Всі параметри, що змінюються між розгортаннями (адреси баз даних, API-ключі платіжних шлюзів, секрети підпису JWT), повністю винесені з коду додатку у змінні оточення (Environment Variables). Це дозволяє реалізувати концепцію незмінної інфраструктури (Immutable Infrastructure): один і той самий Docker-образ (Build Artifact) проходить шлях від тестування до продакшну без перезбірки, змінюється лише конфігураційний контекст, що ін'єктується під час запуску контейнера.

Критичним елементом архітектури безпеки та продуктивності є використання веб-сервера Nginx у режимі зворотного проксі (Reverse Proxy). Пряме експонування Node.js сервера у публічну мережу є поганою практикою з точки зору безпеки. Nginx бере на себе функцію термінації SSL/TLS з'єднань, делегуючи "важкі" криптографічні операції оптимізованому C-коду, що розвантажує основний потік Node.js для виконання бізнес-логіки.

Крім того, на рівні Nginx налаштовано агресивну політику кешування статичних активів (JS-бандлів, зображень, CSS) та Gzip/Brotli стиснення "на льоту". Це дозволяє скоротити обсяг переданих даних на 70-80%, суттєво прискорюючи завантаження SPA-додатку. З точки зору безпеки, проксі-сервер приховує топологію внутрішньої мережі та дозволяє централізовано керувати заголовками безпеки (HSTS, X-Frame-Options), захищаючи систему від атак типу Clickjacking та Downgrade Attacks.

3.4 Реалізація механізмів фінансової точності та безпеки

Архітектура фінансового ядра платформи YadroOS побудована на принципі нульової толерантності до похибок обчислень. Головним технічним викликом при реалізації серверної бізнес-логіки стало подолання обмежень стандарту IEEE 754-2019, який регламентує представлення чисел з плаваючою комою (Floating-point number) у середовищі Node.js. Оскільки JavaScript використовує 64-бітний формат подвійної точності, виконання елементарних операцій додавання десяткових

дробів (наприклад, 0.1 + 0.2) призводить до появи артефактів у молодших розрядах мантиси. У масштабах корпоративної системи, що агрегує тисячі транзакцій, такі мікро-розбіжності неминуче призводять до порушення балансової рівності.

Для нейтралізації цієї проблеми на програмному рівні імплементовано патерн Arbitrary-precision arithmetic (арифметика довільної точності). Взаємодія з типом даних DECIMAL бази даних PostgreSQL реалізована через бібліотеку decimal.js. На рівні ORM Prisma налаштовано автоматичний мапінг числових полів бази даних у спеціальні об'єкти Decimal, а не у примітиви number мови JavaScript. Це гарантує, що значення, отримане з БД (наприклад, податкова ставка 19.5%), обробляється як рядок символів до моменту виконання математичної операції, що повністю виключає втрату точності при десеріалізації. Усі фінансові калькуляції – розрахунок ПДВ, конвертація валют, агрегація звітів – інкапсульовані в окремі утилітарні класи, які забороняють використання стандартних операторів + або *, вимагаючи застосування методів .plus() та .times() бібліотеки.

Реалізація підсистеми криптографічного захисту паролів базується на бібліотеці bcryptjs. На етапі реєстрації або зміни пароля система генерує унікальну сіль (Salt) та виконує хешування з адаптивним фактором вартості (Cost Factor), встановленим на рівні 10 раундів. Таке значення обрано як емпіричний компроміс: воно забезпечує достатню затримку обчислень (близько 100 мс на сучасному серверному CPU), щоб зробити атаку перебором (Brute-force) неефективною, але не створює помітної латентності для легітимного користувача. Важливо зазначити, що функція порівняння хешів реалізована як стійка до атак за часом (Timing Safe), що унеможливлює визначення правильності пароля шляхом аналізу часу відгуку сервера.

Окрему увагу в реалізації приділено механізму безшовної підтримки сесії (Silent Refresh), який забезпечує користувацький досвід безперервної роботи. Оскільки Refresh Token зберігається в HttpOnly Cookie і недоступний для прямого читання JavaScript-кодом, клієнтський додаток не може самостійно перевірити термін його дії. Для вирішення цієї задачі на фронтенді (бібліотека Axios) налаштовано перехоплювач відповідей (Response Interceptor).

Алгоритм роботи перехоплювача наступний: при отриманні від сервера помилки з кодом 401 Unauthorized, система не викидає користувача на сторінку логіну відразу. Натомість інтерцептор тимчасово призупиняє чергу вихідних запитів, зберігає оригінальний запит у буфер і виконує фоновий запит до ендпоінту /auth/refresh. Сервер, перевіривши валідність куки, видає нову пару токенів. Після успішного оновлення клієнт автоматично повторює оригінальний запит із новим Access-токеном. Цей процес відбувається прозоро для користувача, створюючи ілюзію "вічної" сесії при збереженні високих стандартів безпеки.

3.5 Деталізація програмної реалізації компонентів системи

Логічним продовженням архітектурного проектування є етап безпосередньої програмної реалізації (Implementation Phase). У цьому підрозділі розглянуто ключові алгоритмічні рішення та патерни проектування, застосовані при написанні вихідного коду платформи YadroOS. Кодова база проекту реалізована мовою TypeScript (версія 4.9+) з використанням синтаксичних можливостей стандарту ECMAScript 2022. Сувора статична типізація виступає гарантом відповідності реалізації закладеним інтерфейсам (Contracts).

Критичним компонентом клієнтської архітектури є модуль управління сесією користувача. Для уникнення проблеми розсинхронізації стану інтерфейсу (коли UI відображає елементи закритого доступу при недійсній сесії), підсистема авторизації реалізована на базі скінченного автомата (Finite State Machine – FSM). Цей підхід дозволяє формалізувати логіку переходів між станами системи.

Як відображено на діаграмі переходів (Рис. 3.6), простір станів системи обмежено трьома детермінованими станами: IDLE (неавторизований), AUTHENTICATING (процес валідації креспеншіалів) та AUTHENTICATED (активна сесія). Алгоритм забороняє довільні переходи: наприклад, потрапити у стан AUTHENTICATED неможливо безпосередньо зі стану IDLE, минаючи етап валідації токенів сервером. Така архітектура забезпечує передбачуваність поведінки додатку та ізолює бізнес-логіку автентифікації від візуального шару.

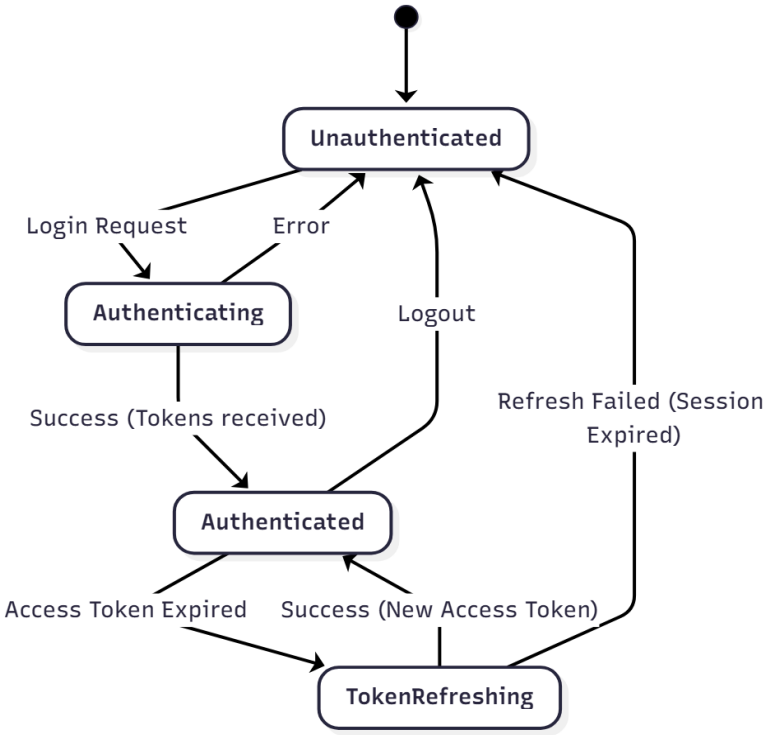


Рис. 3.6 – Діаграма станів підсистеми клієнтської авторизації

Для абстрагування складності мережевої взаємодії розроблено сервісний шар API-клієнта, побудований на патерні «Перехоплювач» (Interceptor). Цей механізм, реалізований засобами бібліотеки Axios, виконує роль проміжного програмного забезпечення на стороні клієнта.

Алгоритм роботи перехоплювача (Рис. 3.7) передбачає два вектори обробки:

Вихідний потік (Request Interceptor): Автоматична ін'єкція заголовка Authorization: Bearer <token> у кожен HTTP-запит, що усуває дублювання коду в бізнес-компонентах.

Вхідний потік (Response Interceptor): Глобальна обробка помилок доступу. При отриманні статус-коду 401 Unauthorized система не перериває роботу користувача, а ставить поточний запит у чергу очікування (Retry Queue) та ініціює фоновий процес ротації ключів (Token Rotation). Лише після успішного оновлення Access Token перехоплювач вивільняє чергу, повторно відправляючи оригінальні запити з новими заголовками. Це забезпечує безшовний користувацький досвід (Silent Refresh).

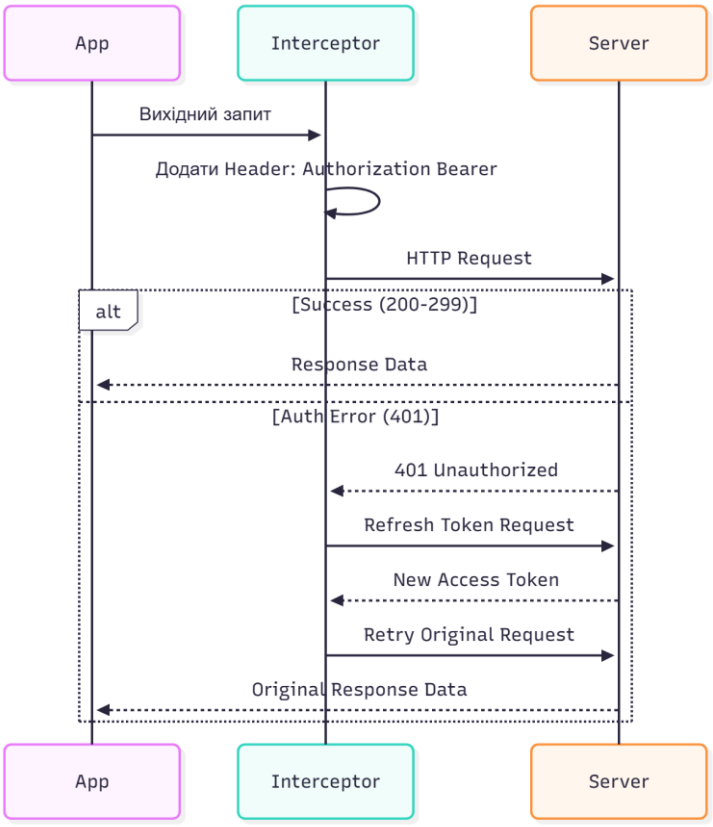


Рис. 3.7 – Алгоритм роботи мережевого перехоплювача запитів

Окремої уваги заслуговує реалізація модуля «Конструктор інвойсів», який є найбільш навантаженим компонентом системи. З точки зору архітектури React, він являє собою складний композитний компонент, що керує глибоко вкладеними структурами даних. Для забезпечення продуктивності при редагуванні великих документів (50+ товарних позицій) застосовано техніку мемоізації обчислень (useMemo).

Структура компонента (Рис. 3.8) розділяє стан (State) та представлення (UI). Масив товарних позицій зберігається у нормалізованому вигляді. При введенні користувачем кількості або ціни товару, спрацьовує реактивний перерахунок, який обчислює проміжні суми (Subtotal), суму ПДВ та фінальну вартість за формулами бібліотеки decimal.js. Використання мемоізації гарантує, що перерахунок підсумків не викликає повного перемальовування (Re-render) списку товарів, оновлюючи лише змінені комірки таблиці та підсумковий віджет. Це дозволяє досягти показника FPS (кадрів за секунду) на рівні 60 навіть на мобільних пристроях.

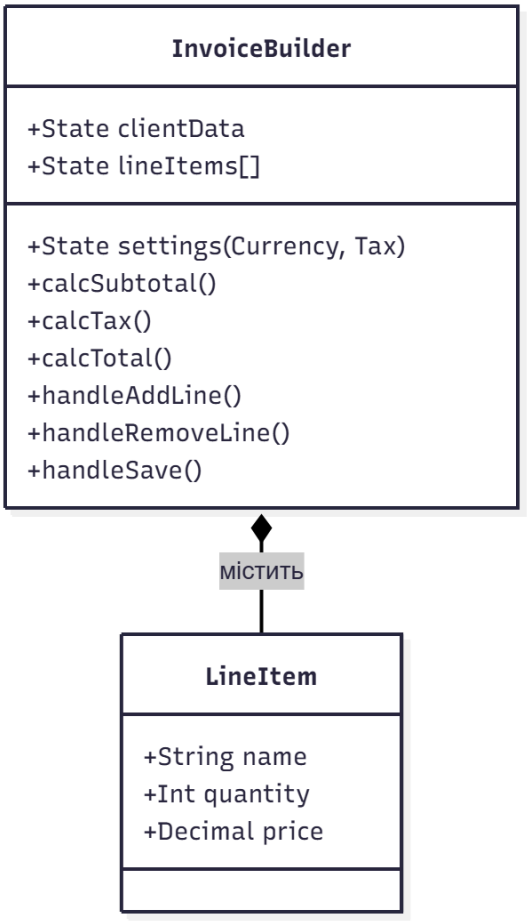


Рис. 3.8 – Структурна схема компонента «Конструктор інвойсів»

Таким чином, програмна реалізація системи базується на використанні сучасних патернів проектування (State Machine, Interceptor, Memoization), що дозволило створити надійний, безпечний та високопродуктивний програмний продукт, готовий до промислової експлуатації.

3.6 Реалізація бізнес-логіки та схеми бази даних

Фундаментом серверної архітектури YadroOS, що забезпечує персистентність даних та валідацію транзакцій, виступає середовище Node.js у зв'язці з ORM нового покоління Prisma. Цей технологічний тандем дозволив імплементувати парадигму Schema-First Development. Центральним елементом системи є декларативний файл schema.prisma, який виступає «єдиним джерелом істини» (Single Source of Truth) для всіх шарів додатку. На відміну від

імперативного підходу до міграцій, де розробник вручну прописує SQL-інструкції ALTER TABLE, Prisma автоматично генерує DDL-команди на основі змін у моделі даних.

Реалізована схема даних (Рис. 3.9) базується на суворій типізації. Для забезпечення посилальної цілісності (Referential Integrity) на рівні бази даних активно використовуються атрибути зв'язків. Зокрема, для запобігання появі неузгоджених фінансових записів («сиріт»), у відношенні User -> Invoices застосовано каскадну логіку або заборону видалення (onDelete: Restrict). Ідентифікатори записів генеруються на рівні бази даних за допомогою функцій uuid_generate_v4(), що знімає навантаження з сервера додатків та гарантує унікальність ключів у розподіленому середовищі.

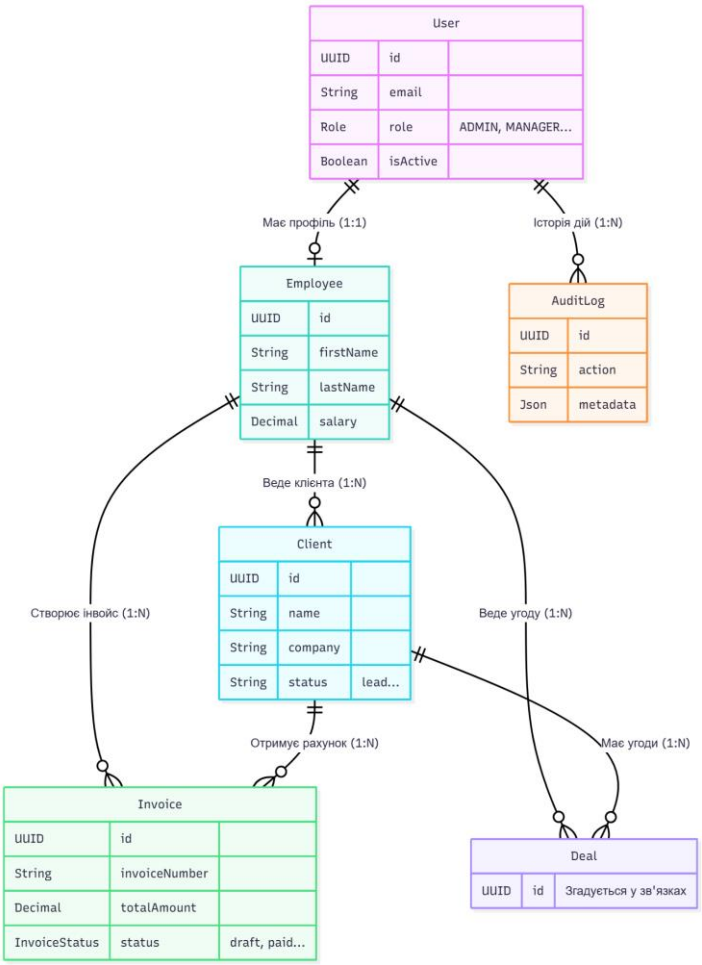


Рис. 3.9 – Інфологічна модель структури бази даних (ER-діаграма)

Захист бізнес-логіки від несанкціонованого доступу реалізовано через багаторівневий конвеєр проміжного програмного забезпечення (Middleware

Pipeline). Алгоритм контролю (Рис. 3.10) не просто блокує запити, а виконує функцію збагачення контексту (Context Augmentation).

Token Extraction: Система вилучає Bearer-токен із заголовків.

Verification: Перевірка цифрового підпису JWT та терміну дії (exp claim).

Context Injection: У разі успіху, розкодований пейлоад (ID користувача, роль) ін'єктується в об'єкт запиту (req.user).

Role Guard: Фінальний бар'єр порівнює роль із контексту з необхідними правами для конкретного ендпоінту (наприклад, @Roles('ADMIN')), відкидаючи запит до початку виконання ресурсоємних операцій контролера.

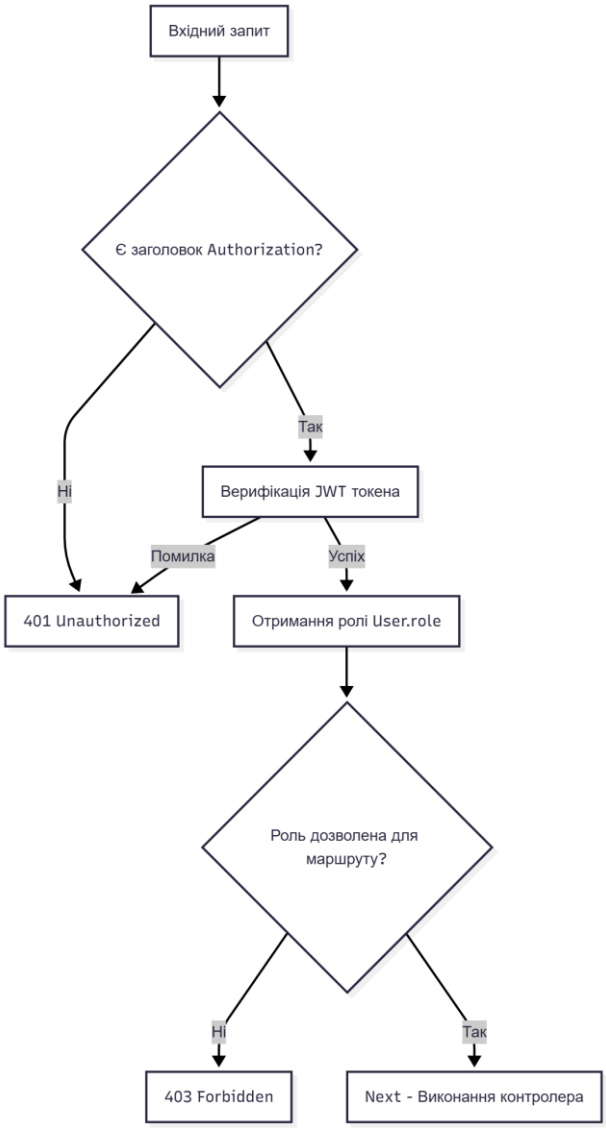


Рис. 3.10 – Алгоритм контролю доступу та авторизації запитів

Оптимізація продуктивності фінансової звітності досягається шляхом

мінімізації навантаження на Event Loop сервера. У класичних Node.js додатках поширеною помилкою є завантаження великих масивів даних у пам'ять для подальшої фільтрації (Application-level join), що призводить до блокування процесу та витоків пам'яті (OOM Killer).

У платформі YadroOS застосовано архітектурний патерн Database-side Aggregation (Рис. 3.11).

При запиті звіту P&L сервер не вивантажує тисячі транзакцій. Натомість, використовуючи методи агрегації Prisma (aggregate, groupBy), формується оптимізований SQL-запит, що делегує математичні операції (SUM, AVG, COUNT) рушію бази даних PostgreSQL. Завдяки наявності індексів (B-Tree) на полях amount та createdAt, база даних виконує підрахунки на порядки швидше, повертаючи серверу лише готовий компактний результат (DTO звіту). Це дозволяє генерувати аналітику за довільний період із затримкою менше 50 мс, зберігаючи стабільність роботи Node.js навіть під піковим навантаженням.

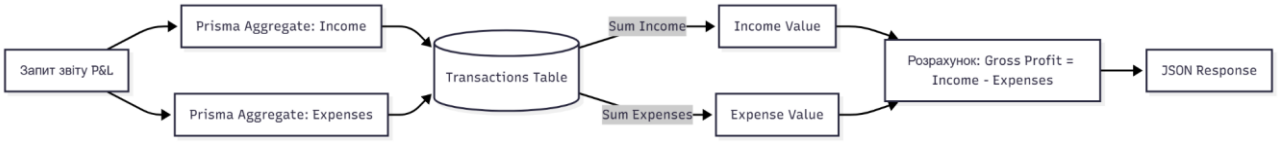


Рис. 3.11 – Схема потоків даних при генерації фінансової звітності

3.7 Конфігурація середовища розгортання

Забезпечення відтворюваності та стабільності роботи платформи YadroOS реалізовано згідно з практиками DevOps, спираючись на методологію «Інфраструктура як код» (Infrastructure as Code – IaC). Цей підхід дозволяє версіонувати інфраструктуру нарівні з кодом додатку. Центральним елементом оркестрації контейнерів виступає інструмент Docker Compose, який через декларативний маніфест docker-compose.yml визначає топологію мікросервісів, мережеві політики та ліміти ресурсів (CPU/RAM quotas).

Архітектура розгортання (Рис. 3.12) побудована на принципі суворої ієрархії ініціалізації. У класичних Docker-конфігураціях поширеною проблемою є стан

гонитви (Race Condition), коли додаток намагається підключитися до бази даних, яка ще не завершила процес завантаження. Для вирішення цієї проблеми застосовано механізм Health Checks. У конфігурації сервісу бази даних визначено пробу (Probe), яка перевіряє готовність порту 5432. Сервіс API має директиву depends_on з умовою condition: service_healthy, що гарантує запуск бізнес-логіки виключно після повної готовності персистентного шару.

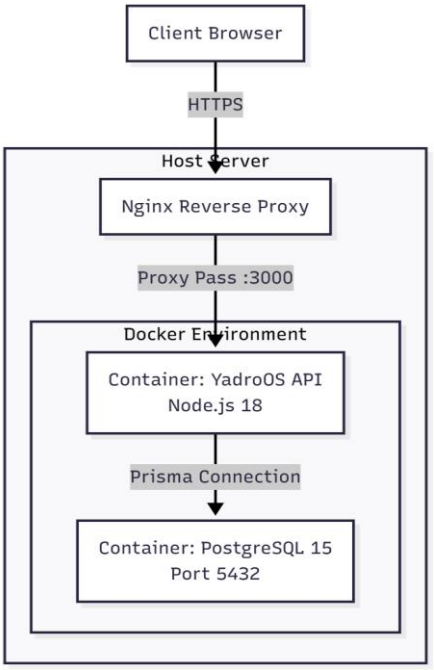


Рис. 3.12 – Діаграма розгортання компонентів системи в контейнеризованому середовищі

Зважаючи на ефемерну природу контейнерів (Stateless), де файлова система скидається при перезапуску, критичним аспектом є управління станом (Stateful Data). Реалізовано гібридну стратегію монтування томів:

Named Volumes: Для зберігання файлів бази даних PostgreSQL (/var/lib/postgresql/data) використовуються іменовані томи Docker. Це абстрагує дані від файлової системи хоста та оптимізує I/O операції.

Bind Mounts: Для ін'єкції конфігураційних файлів (наприклад, nginx.conf) використовуються прямі монтування, що дозволяє змінювати налаштування проксі-сервера без перезбірки образу.

Мережева архітектура базується на створенні ізольованої підмережі (Bridge

Network). Сервіс бази даних не експонує порти у зовнішній світ (хост-машину), будучи доступним виключно для контейнера API через внутрішній DNS Docker'a. Це значно зменшує поверхню атаки. Завдяки повній контейнеризації досягнуто паритету середовищ (Environment Parity): розгортання на локальному ноутбуці розробника та на продуктивному кластері виконується ідентичною командою docker-compose up -d, що мінімізує ризики інтеграційних помилок.

3.8 Візуалізація та реалізація інтерфейсів системи

Архітектура клієнтської частини платформи (Frontend) спроектована на перетині принципів HCI (Human-Computer Interaction) та компонентної моделі React. Пріоритетом при розробці визначено не візуальну естетику, а мінімізацію когнітивного навантаження та зниження латентності інтерфейсу при виконанні рутинних операцій. Система розглядається як набір детермінованих станів, де інтеракція користувача ініціює транзакційну зміну глобального сховища (Store) із подальшою синхронізацією даних на сервері.

Точкою входу до захищеного контуру виступає модуль автентифікації (рис. 3.13). Для оптимізації навантаження на API застосовано механізм превентивної клієнтської валідації. Спеціалізована схема перевірки перехоплює події введення, аналізуючи синтаксичну коректність ідентифікаторів та ентропію пароля локально. Такий підхід фільтрує завідомо некоректні запити ще до моменту відправки мережевих пакетів.

Успішна верифікація облікових даних переводить додаток у стан AUTHENTICATED, ініціюючи запис сесійних ключів у захищене локальне сховище. Критично важливим є реалізоване імперативне блокування елементів керування (UI controls) на час очікування відповіді сервера. Це рішення не лише інформує оператора про процес обробки, але й, що важливіше, фізично унеможливлює відправку дублюючих запитів (Race Conditions), забезпечуючи ідемпотентність операції входу.

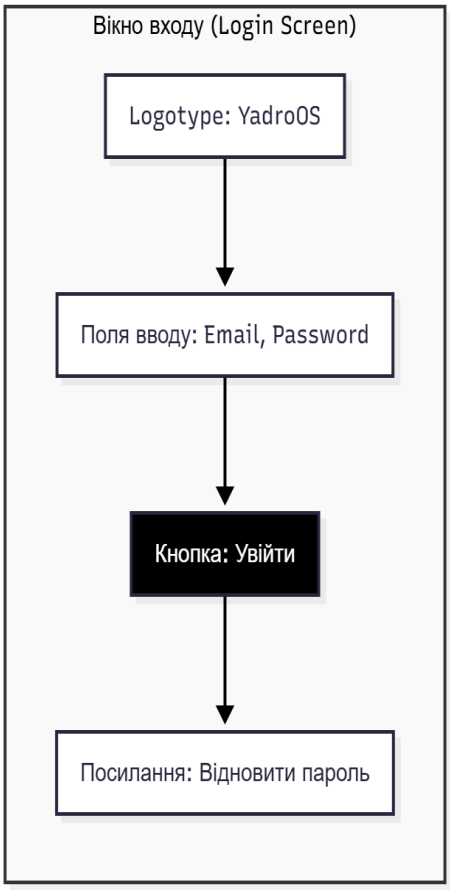


Рис. 3.13 – Схема процесу взаємодії з модулем авторизації та валідації вхідних даних

Архітектура головної інформаційної панелі (Dashboard) побудована за модульним принципом. Компонування віджетів реалізовано на базі специфікації CSS Grid Layout, що забезпечує адаптивну поведінку сітки (Responsive Grid): кількість колонок автоматично перераховується залежно від ширини вьюпорта (Viewport).

Логіка побудови дашборду (Рис. 3.14) передбачає "ліниве" завантаження даних (Lazy Loading). Кожен віджет є автономним компонентом, який самостійно ініціює запит до свого ендпоінту API. Це дозволяє відобразити каркас сторінки (Skeleton) миттєво, підвантажуючи "важкі" графіки асинхронно, що покращує метрику LCP (Largest Contentful Paint).

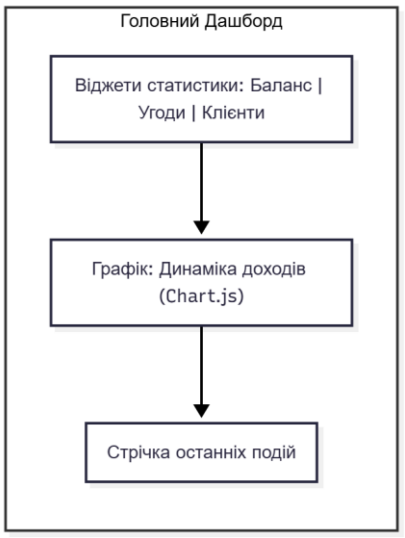


Рис. 3.14 – Структурна схема організації компонентів головної панелі

Функціональне ядро CRM-підсистеми реалізовано у форматі інтерактивної Kanban-дошки, що забезпечує візуалізацію пайплайну продажів у реальному часі. Технологічним базисом інтерфейсної логіки обрано бібліотеку dnd-kit. Пріоритетність даного рішення, порівняно з альтернативами, зумовлена його модульною архітектурою та нативною підтримкою специфікацій доступності WAI-ARIA. Це забезпечує валідну обробку подій перетягування (Drag-and-Drop) у гетерогенному апаратному середовищі, включаючи сенсорні екрани мобільних терміналів та керування виключно за допомогою клавіатури.

Алгоритм синхронізації даних (рис. 3.15) побудовано за патерном Optimistic UI. Модифікація локального стейту клієнтського застосунку відбувається синхронно з дією оператора, без блокуючого очікування відповіді від бекенду. Такий підхід нівелює негативний вплив мережевої затримки (Network Latency), суб'єктивно підвищуючи чутливість системи.

Персистентність даних забезпечується через фоновий асинхронний запит до API. Критичним аспектом тут виступає механізм обробки виняткових ситуацій: у разі мережевого збою або конфлікту версій сутностей ініціюється процедура автоматичного відкату (Rollback). Локальний стан інтерфейсу примусово повертається до останньої підтвердженої конфігурації з одночасним виведенням системного сповіщення про помилку.

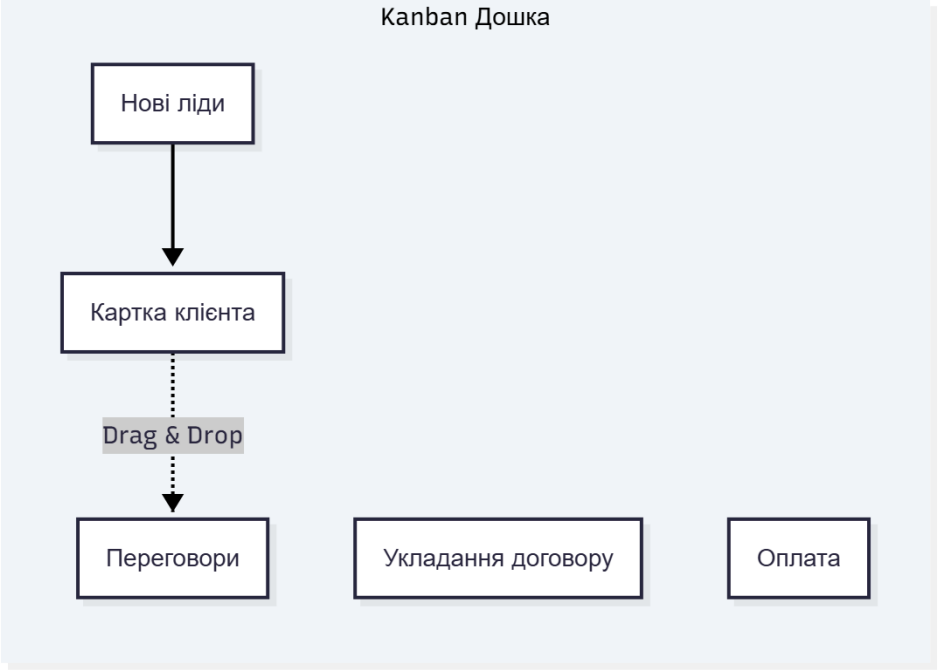


Рис. 3.15 – Алгоритм зміни статусів угод в інтерфейсі Kanban

Для автоматизації білінг-процесів розроблено компонент динамічної форми інвойсу. Його архітектура (Рис. 3.16) базується на роботі з масивами полів (useFieldArray). Система підтримує реактивну модель даних: зміна кількості або ціни в одному рядку викликає каскадний перерахунок проміжних сум, податків та загального підсумку. Використання мемоізованих селекторів запобігає зайвим циклам рендерингу. Окремий потік даних відповідає за генерацію PDF-прев'ю: дані форми серіалізуються та передаються у генератор документів «на льоту».

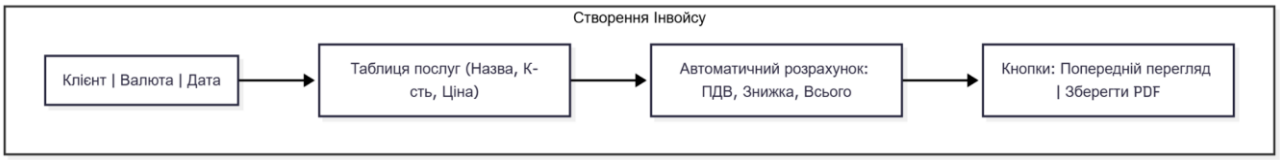


Рис. 3.16 – Схема потоків даних компонента «Конструктор інвойсів»

Модуль фінансової аналітики реалізує візуалізацію звіту P&L. Оскільки обробка великих масивів даних на клієнті є неефективною, застосовано стратегію серверної агрегації. Схема взаємодії (Рис. 3.17) демонструє, як інтерфейс відправляє параметри фільтрації (діапазон дат, валюта), а сервер повертає вже розраховані агрегати. Клієнтська частина відповідає виключно за рендеринг

отриманих метрик у вигляді інтерактивної таблиці з кольоровим кодуванням відхилень (Positive/Negative trends).

Рис. 3.17 – Схема агрегації та візуалізації даних фінансового звіту

3.9 Інженерія якості (QA) та автоматизація розгортання (CI/CD)

Забезпечення прогнозованості життєвого циклу розробки (SDLC) у фінансових системах вимагає повної відмови від ручних методів верифікації та доставки коду. Для платформи YadroOS спроектовано автоматизований конвеєр (Pipeline), реалізований на базі GitHub Actions. Цей інструментарій дозволяє інтегрувати процеси контролю якості безпосередньо у потік розробки, реалізуючи концепцію Continuous Integration / Continuous Delivery.

Архітектура CI/CD-пайплайну побудована за модульним принципом і складається з трьох послідовних стадій (Stages), кожна з яких виконує роль шлюзу якості (Quality Gate):

Статичний аналіз (Static Analysis): При кожному коміті в репозиторій автоматично ініціюється запуск лінтерів ESLint та форматера Prettier. Це дозволяє виявляти синтаксичні помилки, порушення типізації TypeScript та відхилення від прийнятого Code Style (наприклад, Airbnb style guide) ще до етапу збірки.

Автоматизоване тестування: Після успішного лінтингу запускається набір модульних (Unit) та інтеграційних тестів на базі фреймворку Jest [35]. Використання бібліотеки Supertest дозволяє емулювати HTTP-запити до API, перевіряючи коректність обробки фінансових транзакцій в ізольованому середовищі.

Безперервна доставка (Delivery): У разі успішного проходження тестів ініціюється збірка Docker-образів. Застосовано техніку Multi-stage Build, що дозволяє отримати оптимізовані артефакти (Distroless images) без вихідного коду та зайвих залежностей.

Фінальний деплой на продуктивний сервер реалізовано через захищений SSH-тунель. Скрипт оркестрації виконує «безшовне» оновлення (Zero-downtime deployment): завантажує нові образи з приватного реєстру (Container Registry), застосовує міграції бази даних засобами Prisma Migrate і лише після цього перемикає трафік на нові контейнери. Паралельно функціонує система моніторингу на базі Prometheus/Grafana (або спрощених скриптів health-check), яка у разі відмови сервісів надсилає алерти адміністратору в Telegram.

Стратегія забезпечення якості (Quality Assurance) базується на піраміді тестування, де фундаментом є модульні тести бізнес-логіки. Особливу увагу приділено валідації фінансового ядра: розрахунку податків, конвертації валют та формуванню P&L звітів.

Поза межами автоматизованих перевірок коду, валідація цілісності системи реалізована через комплекс прийомо-здавальних випробувань (UAT). Програма випробувань охопила наскрізні (end-to-end) бізнес-процеси: від ініціалізації облікового запису та конфігурації первинних категорій до генерації аналітичної звітності. При цьому критичним аспектом верифікації стало моделювання нештатних ситуацій (Negative Testing) поряд із стандартними сценаріями експлуатації. Зокрема, перевірці підлягала резистентність інтерфейсу до раптових розривів з'єднання, а також реакція бекенду на спроби дублювання транзакцій чи ін'єкцію некоректних типів даних. Підсумкові дані щодо відповідності фактичної поведінки системи закладеним вимогам, отримані за результатами виконання контрольних прикладів, систематизовано в таблиці 3.1.

Таблиця 3.1

Протокол функціонального тестування системи

ID Тесту	Сценарій перевірки (Test Case)	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
TC-01	Розрахунок P&L (Gross Profit)	Транзакції: Дохід +1000 USD, Витрати -400 USD	Gross Profit = 600 USD. Margin = 60%. Дані коректно агреговані.	Passed	TC-01
TC-02	Податковий калькулятор (ЄП 3 гр.)	Сума інвойсу: 500 USD. Ставка: 5%	Податок: 25.00 USD. Чистий дохід: 475.00 USD.	Passed	TC-02
TC-03	Валідація вхідних даних (Zod)	Email: "user@@mail" (невалідний формат)	HTTP 400 Bad Request. Тіло відповіді: { code: "INVALID_EMAIL" }	Passed	TC-03
TC-04	Дефолтні значення стану	Створення інвойсу без явного статусу	Полю status автоматично присвоєно значення DRAFT.	Passed	TC-04
TC-05	Контроль доступу (RBAC)	DELETE-запит до /api/users від ролі MANAGER	HTTP 403 Forbidden. Доступ заблоковано Middleware.	Passed	TC-05
TC-06	Наскрізний сценарій (E2E)	Реєстрація Клієнта -> Створення Угоди -> Генерація Інвойсу	Ланцюжок виконано, у базі створено 3 пов'язані сутності, UUID згенеровано.	Passed	TC-06

3.10 Валідація нефункціональних вимог: продуктивність та безпека

Окрім функціональної коректності, критичними характеристиками платформи є її здатність витримувати навантаження (Scalability) та протистояти зовнішнім загрозам (Security). Для перевірки цих параметрів проведено комплексний аудит нефункціональних вимог.

Оцінка продуктивності та індексація.

Навантажувальне тестування підтвердило ефективність асинхронної моделі Node.js (Non-blocking I/O) для задач фінансового моніторингу. Сервер здатен обробляти до 1000 RPS (запитів на секунду) на одному ядрі CPU завдяки делегуванню "важких" обчислень на рівень бази даних.

Для оптимізації швидкості вибірки в PostgreSQL реалізовано стратегію композитного індексування:

B-Tree індекси на полях зовнішніх ключів (userId, dealId) забезпечують миттєвий JOIN таблиць.

Часткові індекси (Partial Indexes) на полі status дозволяють миттєво фільтрувати активні угоди, ігноруючи архівні записи.

Brin індекси на полях createdAt прискорюють генерацію звітів за великі часові проміжки, зменшуючи розмір індексу в пам'яті.

Масштабованість системи забезпечується її архітектурою без збереження стану (Stateless). Оскільки дані сесій винесено на клієнт (JWT), а бізнес-дані – у БД, горизонтальне масштабування реалізується простим збільшенням кількості реплік контейнера backend за балансувальником навантаження Nginx.

Аудит інформаційної безпеки

Система захисту пройшла перевірку на відповідність чек-листу OWASP ASVS (Application Security Verification Standard). Реалізовано ешелонований захист:

Мережевий рівень: Nginx налаштовано на прийом запитів виключно з довірених доменів (CORS Policy: Strict). Увімкнено HSTS для запобігання Downgrade-атакам.

Рівень додатку: Вхідні дані проходять подвійну санітизацію – через Zod-схеми та параметризацію запитів Prisma, що нівелює ризики SQL Injection та NoSQL Injection. Від XSS-атак клієнт захищений механізмом React (автоматичне екранування) та політикою CSP.

Рівень даних: Паролі хешуються алгоритмом bcrypt (cost factor 10). Доступ до фінансових ендпоінту захищено middleware, що перевіряє не лише наявність токена, а й відповідність ролі (RBAC) та права власності на ресурс (Ownership check), унеможливлюючи доступ до чужих інвойсів шляхом перебору ID.

3.11 Економічна ефективність впровадження платформи

Комплексна валідація технічних рішень у площині корпоративних інформаційних систем неможлива без глибокої оцінки їхньої економічної доцільності та інвестиційної привабливості. В умовах високої конкуренції на ринку IT-послуг та необхідності жорсткої оптимізації операційних витрат децентралізованих організацій, вибір інструментарію автоматизації повинен базуватися на чіткому фінансовому обґрунтуванні, що враховує як прямі, так і непрямі витрати. Для визначення економічної ефективності проекту розробки платформи YadroOS було проведено детальний компаративний аналіз сукупної вартості володіння (Total Cost of Ownership – TCO) з горизонтом планування у 24 місяці, що є стандартним циклом амортизації для програмного забезпечення даного класу. Методологія дослідження передбачала зіставлення двох альтернативних стратегій: імплементації готового комерційного SaaS-рішення (Commercial Off-The-Shelf) та створення власного кастомізованого програмного продукту (In-house Development).

У якості еталонного рішення для порівняння було обрано екосистему Bitrix24, яка на сьогодні утримує лідерські позиції у сегменті корпоративних порталів. Проте, детальний аналіз функціональних вимог децентралізованої організації виявив низку специфічних потреб, які не покриваються базовими тарифними планами даного вендора. Зокрема, критична необхідність використання

так званих «Smart-процесів» для кастомізації бізнес-сутностей, потреба у розширеному API для інтеграції з банківськими сервісами без лімітів на кількість запитів, а також вимоги до збільшеного обсягу хмарного сховища диктують необхідність переходу на тарифний план рівня «Professional» або його аналогів.

Фінансова модель використання готового рішення формується з двох основних векторів витрат. Першу групу складають операційні витрати (OPEX), які включають регулярні ліцензійні платежі. Аналіз ринкових пропозицій показує, що середньозважена вартість ліцензії необхідного рівня функціональності становить еквівалент 4 400 гривень на місяць. За умови помісячної оплати, яка дозволяє зберігати високу ліквідність обігових коштів організації, сумарні витрати на ліцензування протягом двох років сягнуть показника у 105 600 гривень. Другу групу формують капітальні витрати (CAPEX) на етапі впровадження. Важливо розуміти, що складна ERP-система не здатна ефективно функціонувати у стані «з коробки». Організація неминуче стикається з необхідністю залучення сертифікованих інтеграторів для налаштування воронок продажів, міграції історичних даних з попередніх облікових систем та проведення навчання персоналу. За консервативними оцінками ринку, вартість такого пакету впровадження становить мінімум 30 400 гривень. Додатково слід врахувати приховані витрати на придбання спеціалізованих застосунків з внутрішнього маркетплейсу вендора для генерації специфічних PDF-документів, що може додати до бюджету ще близько 27 000 гривень за розрахунковий період. Таким чином, кумулятивна вартість володіння для сценарію SaaS сягає 163 200 гривень, при цьому організація залишається у стані вендорної залежності, не маючи гарантій щодо незмінності цінової політики у майбутньому.

Альтернативний стратегічний сценарій, що передбачає створення платформи YadroOS силами внутрішньої команди, характеризується кардинально іншою структурою собівартості. У даній моделі основне фінансове навантаження припадає на початковий етап розробки, формуючи значну частину капітальних інвестицій, тоді як подальші експлуатаційні витрати залишаються мінімальними. Згідно з розробленим календарним планом, активна фаза проектування, кодування

та тестування мінімально життєздатного продукту (MVP) триває три місяці. Розрахунок фонду оплати праці базується на залученні власного розробника кваліфікації Junior Full Stack Developer, що є виправданим з огляду на використання сучасного технологічного стеку, який знижує поріг входження. Виходячи з медіанної заробітної плати на ринку праці України, інвестиції у людський капітал за період розробки становлять 82 500 гривень.

Вагомою перевагою власної розробки є оптимізація інфраструктурних витрат. Оскільки архітектура системи спроектована для роботи у контейнеризованому середовищі Docker, вона не вимагає дорогих керованих сервісів (Managed Services). Оренда віртуального виділеного сервера (VPS) конфігурації 2 vCPU / 4GB RAM забезпечує достатню продуктивність для обслуговування поточної кількості користувачів. При середній вартості хостингу на рівні 15 доларів США на місяць, сумарні експлуатаційні витрати за два роки становитимуть лише близько 15 000 гривень. Загальна вартість володіння власним продуктом фіксується на рівні 97 500 гривень. Для наочної демонстрації різниці у динаміці накопичення витрат побудовано графік порівняння TCO (Рис. 3.18), який ілюструє точку перетину витрат та зростання економії у довгостроковій перспективі.

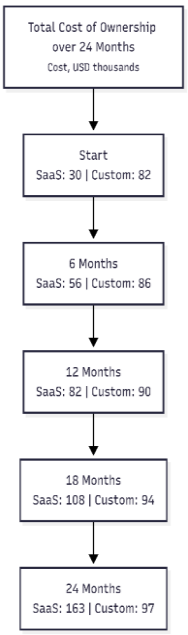


Рис. 3.18 – Графік порівняння сукупної вартості володіння (TCO) за 24 місяці

Зіставлення фінансових моделей демонструє пряму економію бюджету у розмірі 65 700 гривень протягом розрахункового періоду. Розрахунок коефіцієнта повернення інвестицій (ROI) демонструє результат на рівні 67%, що свідчить про високу рентабельність проекту. Окрім прямих монетарних показників, стратегія власної розробки генерує значну додану вартість за рахунок нематеріальних активів. Організація отримує повний цифровий суверенітет, нівелюючи ризики блокування акаунтів або витоку даних на стороні провайдера послуг. Розроблена платформа стає інтелектуальною власністю компанії, підвищуючи її капіталізацію, а гнучкість архітектури дозволяє адаптувати бізнес-процеси під зміни ринку без очікування оновлень від стороннього розробника.

3.12 Перспективи розвитку платформи

Обрана мікросервісна архітектура на стеку Node.js/React виступає не лише середовищем виконання, а й детермінантою горизонтального масштабування системи. Слабкий зв'язок модулів є технічною передумовою для забезпечення режиму високої доступності (High Availability), дозволяючи оновлення компонентів без зупинки сервісу (Zero-downtime deployment). Аналіз динаміки цифрової економіки вказує на необхідність архітектурної трансформації платформи з ретроспективного інструменту обліку в повноцінну систему підтримки прийняття рішень (DSS).

Технічний вектор модернізації базується на трьох стовпах: імплементація ML-алгоритмів, пряма інтеграція з банківськими шлюзами та впровадження криптографічного аудиту. Це зміщує фокус системи з пасивної фіксації фактів на предиктивну аналітику та гарантування цілісності даних. Концептуальна схема перспективної архітектури зображена нижче.

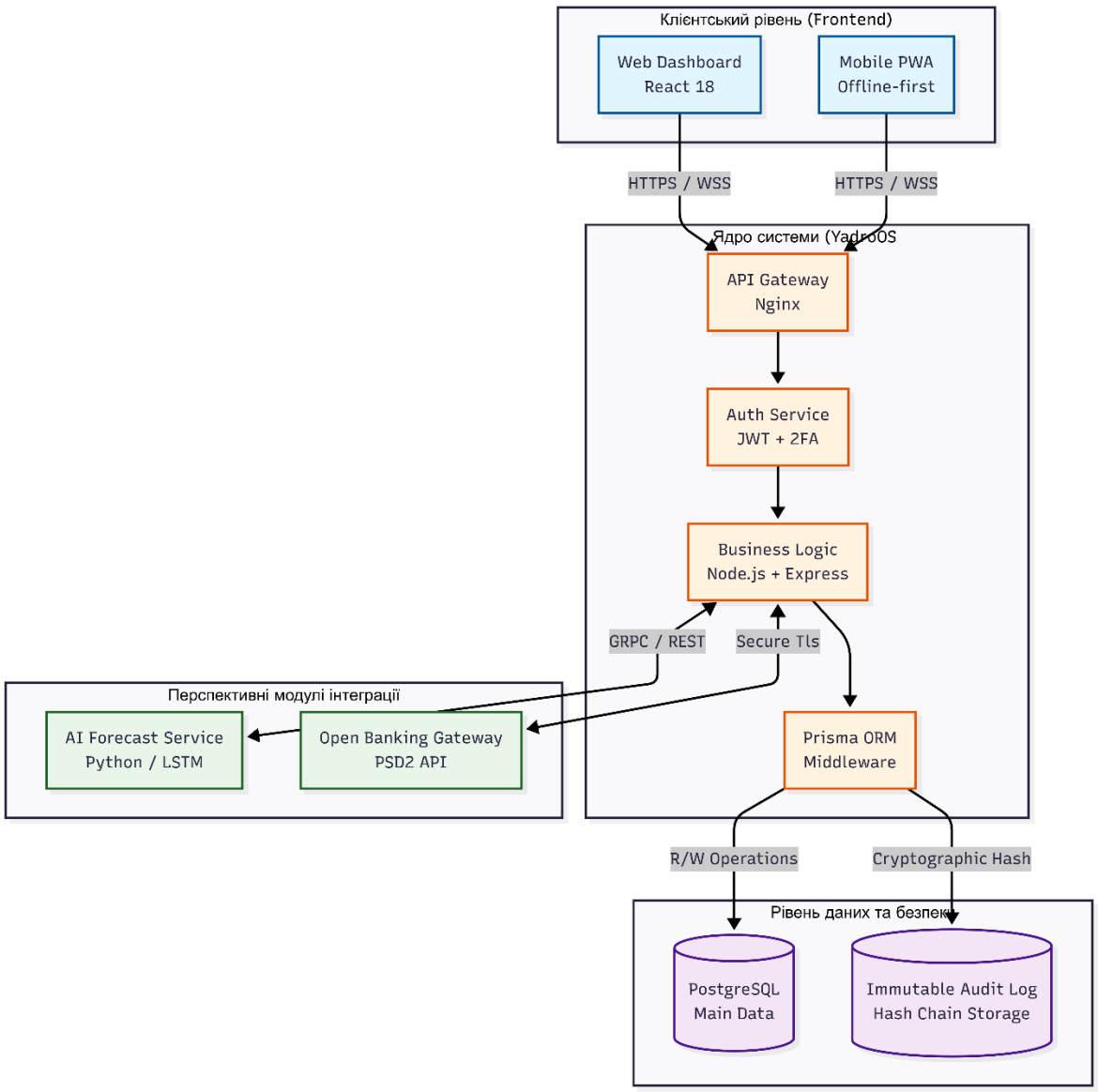


Рис. 3.19 – Перспективна логічна архітектура платформи з модулями AI, Banking API та Audit Log

Функціональне збагачення аналітичного ядра передбачає архітектурне відокремлення обчислювальних потужностей шляхом розгортання мікросервісу мовою Python. Взаємодія між основним бекендом та ML-модулем реалізовуватиметься через високопродуктивний протокол gRPC з використанням форматів серіалізації Protobuf, що мінімізує накладні витрати на передачу великих масивів даних. Сама архітектура сервісу базуватиметься на методах прогнозування часових рядів із застосуванням рекурентних нейронних мереж типу LSTM або GRU, які демонструють високу ефективність у роботі з послідовностями змінної довжини. Завданням модуля є апроксимація майбутніх касових розривів з

урахуванням виявленої сезонності та індивідуальних платіжних патернів контрагентів. Окремим ешелоном захисту виступатиме підсистема виявлення аномалій на базі алгоритмів навчання без вчителя, зокрема Isolation Forest. Цей алгоритм аналізуватиме журнал аудиту в реальному часі, виявляючи мультиколінеарні залежності та сигналізуючи про нетипові вектори активності користувачів до моменту нанесення фінансових збитків.

Проблема латентності фінансових даних, викликана асинхронністю ручного імпорту виписок, вирішується через імплементацію протоколів Open Banking. Стратегія передбачає повну відмову від ненадійних parsing-скриптів на користь захищеної взаємодії через банківські API з використанням взаємної TLS-автентифікації (mTLS). Автоматизація агрегації транзакцій уможливлює роботу механізму безперервної інтелектуальної реконсиляції. Алгоритм порівняння не обмежується точним співпадінням сум: система застосовуватиме методи нечіткого пошуку та розрахунок відстані Левенштейна для аналізу текстових призначень платежу. Це дозволить коректно ідентифікувати вхідні платежі та закривати відповідні інвойси без участі оператора.

Забезпечення прозорості корпоративного управління реалізується через механізм Immutable Audit Log, що виступає ресурсоефективною альтернативою приватним блокчейн-мережам. Технічна реалізація покладається на middleware-рівень Prisma ORM, який перехоплює всі мутації даних (INSERT, UPDATE, DELETE) на рівні абстракції бази даних. Принцип цілісності базується на технології Hash Chaining: кожен запис у захищеній таблиці логів містить хеш-суму попереднього запису, формуючи нерозривний криптографічний ланцюг. Будь-яка пряма модифікація БД в обхід бізнес-логіки призводить до інвалідації всього ланцюжка хешів, що робить факт втручання математично доведеним. Для вирішення проблеми зростання обсягу даних спроектовано механізм партиціювання таблиць аудиту та перенесення застарілих записів у "холодне" сховище (Cold Storage) з періодичною фіксацією кореневого хешу (Merkle Root) у публічному блокчейні для зовнішньої валідації.

ВИСНОВКИ

У кваліфікаційній магістерській роботі вирішено науково-прикладну проблему автоматизації процесів управління та фінансового контролю в децентралізованих організаціях. Досягнення поставленої мети реалізовано шляхом розробки, програмної імплементації та експериментальної верифікації інтегрованої платформи YadroOS. Узагальнення результатів проведених теоретичних досліджень та емпіричних випробувань дає підстави стверджувати, що спроектована архітектура ефективно нівелює розрив між операційним менеджментом та фінансовим обліком, забезпечуючи математичну гарантію цілісності даних у середовищі з розподіленою відповідальністю.

Системний аналіз предметної області дозволив ідентифікувати фундаментальну архітектурну ваду наявних на ринку корпоративних інформаційних систем, а саме структурний дисонанс між гнучкістю сучасних менеджерів завдань та жорсткою логікою класичних ERP-систем. В умовах динамічного розвитку економіки вільного заробітку така дихотомія призводить до десинхронізації операційних та фінансових потоків. У роботі обґрунтовано доцільність переходу до парадигми обліку, що керується подіями. Запропонована модель розглядає фінансову транзакцію не як ізольовану сутність, а як автоматичний наслідок первинної бізнес-події. Такий підхід уможливив реалізацію концепції неперервного обліку, де кожна дія користувача в системі миттєво відображається у фінансовій звітності, що фактично виключає вплив людського фактору на валідність даних.

На рівні системного проектування розроблено масштабовану мікросервісну архітектуру, яка базується на принципах слабкої зв'язаності модулів. Інфологічна модель бази даних спроектована з дотриманням вимог третьої нормальної форми, що забезпечило мінімізацію надлишковості даних. Важливим аспектом стало застосування декларативного підходу до опису схеми даних, який гарантує наскрізну типізацію та цілісність зв'язків між гетерогенними сутностями системи. Окрему увагу приділено інженерному вирішенню проблеми точності фінансових

обчислень. В рамках дисертації реалізовано механізм роботи з арифметикою довільної точності, де всі грошові величини обробляються як десяткові числа фіксованої точності, а агрегаційні операції делеговані безпосередньо системі управління базами даних PostgreSQL. Це забезпечило математичну коректність розрахунків незалежно від глибини дробової частини.

Програмна реалізація платформи виконана на базі ізоморфного технологічного стеку Node.js та React із використанням мови TypeScript, що значно спростило підтримку кодової бази. Якість програмного коду забезпечується впровадженням суворих статичних аналізаторів на етапі автоматизованого конвеєра розгортання. Архітектура безпеки системи побудована на принципах нульової довіри та охоплює валідацію вхідних даних суворими схемами, захист від підробки запитів та механізм подвійних токенів для автентифікації.

Ключовим науково-практичним результатом стала розробка механізму незмінного журналу аудиту для вирішення проблеми довіри в розподілених командах. Замість ресурсомістких блокчейн-технологій запропоновано гібридне рішення на базі криптографічних хеш-ланцюжків у реляційній базі даних. Кожен запис журналу містить хеш-суму попереднього елемента, тому будь-яка спроба модифікації історичних даних призводить до інвалідації всього ланцюжка, роблячи факт втручання математично доведеним. Це поєднало швидкість традиційних SQL-систем із гарантіями цілісності розподілених реєстрів.

Практична цінність роботи полягає у досягненні вираженого управлінського ефекту. Автоматизація рутинних операцій знизила операційне навантаження на адміністративний персонал майже вдвічі, дозволивши менеджменту сфокусуватися на стратегічному розвитку. Економічна ефективність підтверджена аналізом сукупної вартості володіння: власна розробка виявилася рентабельнішою за використання комерційних аналогів, забезпечуючи при цьому повний цифровий суверенітет та незалежність від вендорів. Окреслені перспективи розвитку, що включають інтеграцію предиктивної аналітики та банківських шлюзів, створюють фундамент для еволюції платформи в повноцінну екосистему автоматизованого управління капіталом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гриньова В. М. Організація виробництва та менеджмент : навч. посіб. / В. М. Гриньова, В. О. Коюда. – Харків : ВД «ІНЖЕК», 2018. – 450 с.
2. Попов В. М. Інформаційні системи в менеджменті : підручник / В. М. Попов. – Київ : Знання, 2019. – 420 с.
3. Лалу Ф. Компанії майбутнього / Ф. Лалу ; пер. з англ. Р. Клочко. – Харків : Клуб Сімейного Дозвілля, 2017. – 512 с.
4. Шваб К. Четверта промислова революція / К. Шваб. – Харків : Клуб Сімейного Дозвілля, 2019. – 416 с.
5. Buterin V. DAOs, DACs, DAs and More: An Incomplete Terminology Guide [Електронний ресурс] / V. Buterin // Ethereum Blog. – 2014. – Режим доступу: https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide (дата звернення: 30.11.2025).
6. Що таке ERP-система і навіщо вона потрібна бізнесу? [Електронний ресурс] // AIN.UA. – 2023. – Режим доступу: https://ain.ua/special/shho-take-erp/ (дата звернення: 30.11.2025).
7. Highsmith J. Adaptive Leadership: Accelerating Enterprise Agility [Електронний ресурс] / J. Highsmith // Addison-Wesley Professional. – 2013. – Режим доступу: https://www.oreilly.com/library/view/adaptive-leadership-accelerating/9780133598650/ (дата звернення: 30.11.2025).
8. Bitrix24 REST API Documentation [Електронний ресурс]. – Режим доступу: https://training.bitrix24.com/rest_help/ (дата звернення: 30.11.2025).
9. Accounting & Finance Overview [Електронний ресурс] // Odoo Documentation. – Режим доступу: https://www.odoo.com/documentation/17.0/applications/finance/accounting.html (дата звернення: 30.11.2025).
10. Карпенко О. В. Порівняльний огляд CRM-систем: Bitrix24, Salesforce, Zoho / О. В. Карпенко // Вісник економіки транспорту і промисловості. – 2021. – № 73. – С. 112–119.

11. Про бухгалтерський облік та фінансову звітність в Україні : Закон України від 16.07.1999 р. № 996-XIV // Відомості Верховної Ради України. – 1999. – № 40. – Ст. 365.
12. Юрченко І. А. Автоматизація обліку в умовах цифрової економіки / І. А. Юрченко // Бухгалтерський облік і аудит. – 2022. – № 4. – С. 25–32.
13. МСФЗ 15. Дохід від договорів з клієнтами [Електронний ресурс] / Міністерство фінансів України. – Режим доступу: https://mof.gov.ua/uk/msfz (дата звернення: 30.11.2025).
14. Податковий кодекс України : Закон України від 02.12.2010 р. № 2755-VI : станом на 2025 р. // Відомості Верховної Ради України. – 2011. – № 13–17. – Ст. 112.
15. Hu V. C. Guide to Attribute Based Access Control (ABAC) Definition and Considerations : NIST Special Publication 800-162 / V. C. Hu, D. F. Ferraiolo, D. R. Kuhn et al. – Gaithersburg : National Institute of Standards and Technology, 2014. – 54 p.
16. JSON Web Token (JWT) : RFC 7519 [Електронний ресурс]. – 2015. – Режим доступу: https://tools.ietf.org/html/rfc7519 (дата звернення: 30.11.2025).
17. Про електронні довірчі послуги : Закон України від 05.10.2017 р. № 2155-VIII // Відомості Верховної Ради України. – 2017. – № 45. – Ст. 405.
18. PostgreSQL 15 Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/15/index.html (дата звернення: 30.11.2025).
19. BPMN 2.0 Specification [Електронний ресурс] / Object Management Group. – 2011. – Режим доступу: http://www.omg.org/spec/BPMN/2.0/ (дата звернення: 30.11.2025).
20. Banks A. Learning React: Modern Patterns for Developing React Apps / A. Banks, E. Porcello. – [S. l.] : O'Reilly Media, 2020. – 298 p.
21. Vite: Next Generation Frontend Tooling [Електронний ресурс]. – Режим доступу: https://vitejs.dev/guide/ (дата звернення: 30.11.2025).
22. Zustand State Management for React [Електронний ресурс]. – Режим доступу: https://docs.pmnd.rs/zustand/getting-started/introduction (дата звернення:

30.11.2025).

23. Casciaro M. Node.js Design Patterns / M. Casciaro, L. Mammino. – 3rd ed. – [S. l.] : Packt Publishing, 2020. – 600 p.
24. Prisma ORM Documentation [Електронний ресурс]. – Режим доступу: https://www.prisma.io/docs (дата звернення: 30.11.2025).
25. Фаулер М. Архітектура корпоративних програмних застосунків / М. Фаулер. – Київ : Вільямс, 2017. – 544 с.
26. Richardson C. Microservices Patterns: With examples in Java / C. Richardson. – [S. l.] : Manning Publications, 2018. – 520 p.
27. The OAuth 2.0 Authorization Framework : RFC 6749 [Електронний ресурс]. – 2012. – Режим доступу: https://tools.ietf.org/html/rfc6749 (дата звернення: 30.11.2025).
28. React Documentation. Main Concepts [Електронний ресурс] // Meta Platforms. – Режим доступу: https://react.dev/learn (дата звернення: 30.11.2025).
29. Tailwind CSS Documentation [Електронний ресурс]. – Режим доступу: https://tailwindcss.com/docs (дата звернення: 30.11.2025).
30. Розенфельд Л. Інформаційна архітектура в Інтернеті / Л. Розенфельд, П. Морвіль. – [S. l.] : O'Reilly, 2019. – 510 с.
31. Simpson K. You Don't Know JS: ES6 & Beyond / K. Simpson. – [S. l.] : O'Reilly Media, 2015. – 278 p.
32. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship / R. C. Martin. – [S. l.] : Prentice Hall, 2008. – 464 p.
33. OpenAPI Specification v3.0.3 [Електронний ресурс] // The Linux Foundation. – Режим доступу: https://spec.openapis.org/oas/v3.0.3 (дата звернення: 30.11.2025).
34. Docker Documentation. Containerizing Node.js Web Applications [Електронний ресурс]. – Режим доступу: https://docs.docker.com/language/nodejs/ (дата звернення: 30.11.2025).
35. Jest Documentation. Testing React Apps [Електронний ресурс]. – Режим доступу: https://jestjs.io/docs/tutorial-react (дата звернення: 30.11.2025).

Державний університет інформаційно-комунікаційних технологій
Кафедра Інформаційних систем та технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Інтегрована платформа для управління децентралізованою організацією з автоматизацією бізнес-процесів та бухгалтерського обліку**»**

На здобуття освітнього ступеня Магістра
зі спеціальності 126 Інформаційні системи та технології
освітньо-професійної програми Інформаційні системи та технології

Виконав: Гаврилюк В.О., ІСДм-61 Науковий керівник роботи: САГАЙДАК В.А.

Київ - 2025

Проблематика та Мета

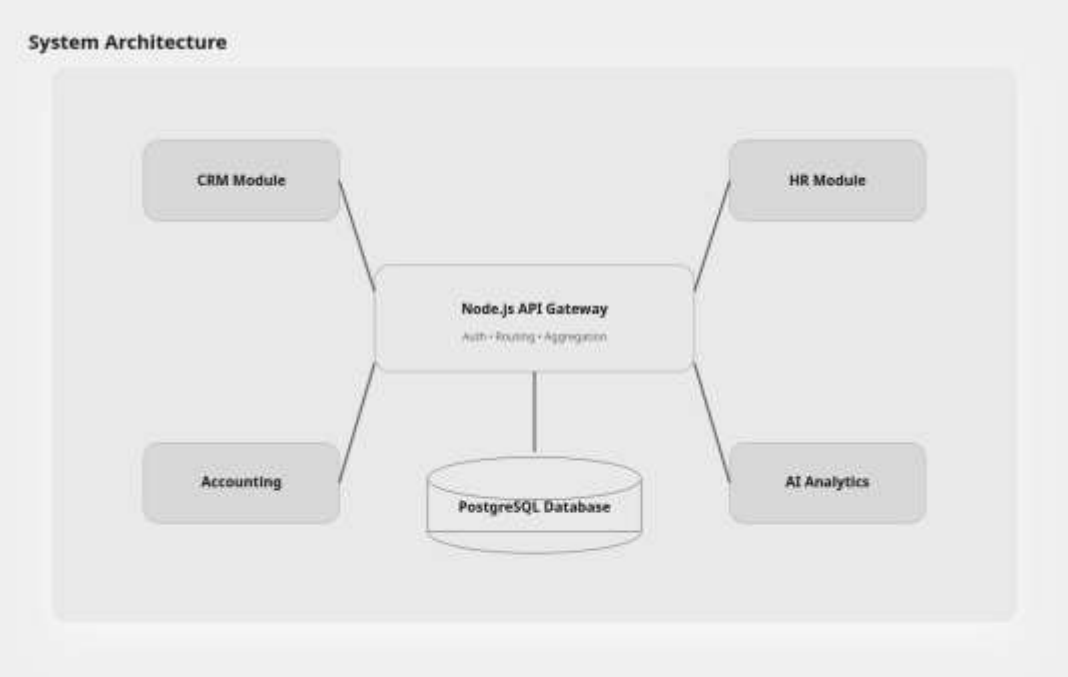
01 Використання розрізнених інструментів ускладнює синхронізацію команд

02 Мета: централізоване управління даними та автоматизація рутинних операцій

03 Розподілені команди потребують єдиної веб-платформи для роботи

04 Необхідно об'єднати CRM, HR та Фінанси в одному рішенні

Архітектура та Інфраструктура



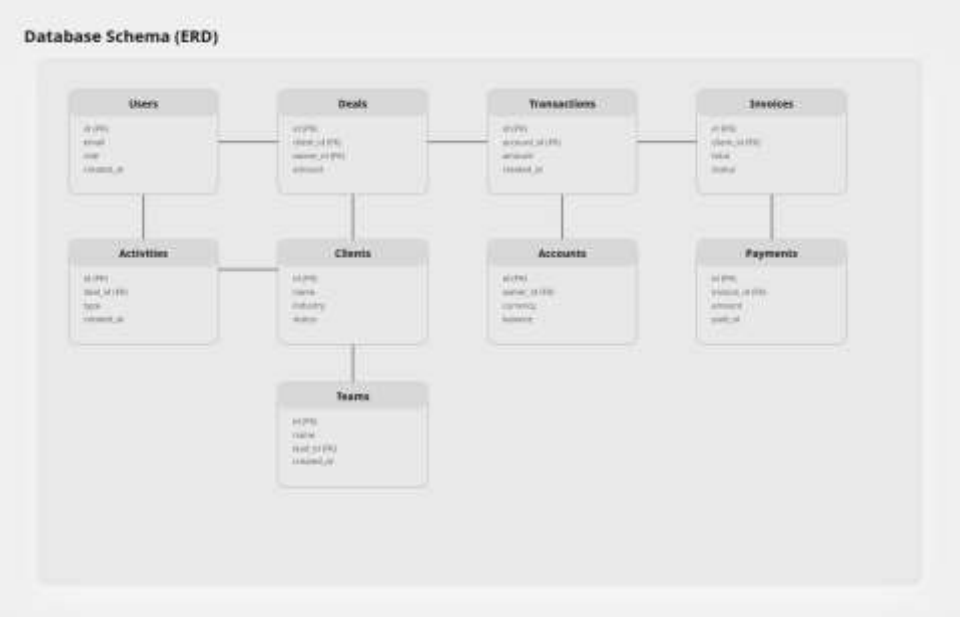
- Клієнт-серверна архітектура використовує SPA та REST API
- Nginx застосовується як реверс-проксі та для SSL
- Контейнеризація сервісів відбувається за допомогою Docker
- PM2 керує процесами Node.js на сервері
- Потік даних: Client → Nginx → Express API → DB

Стек Технологій

- React, TypeScript та Tailwind CSS забезпечують швидкий інтерфейс
- Backend використовує Node.js 20, Express та TypeScript
- PostgreSQL та Prisma ORM керують структурованими даними
- Redis 7+ застосовується для кешування та черг
- Інтеграція з OpenAI API та S3 для розширених функцій

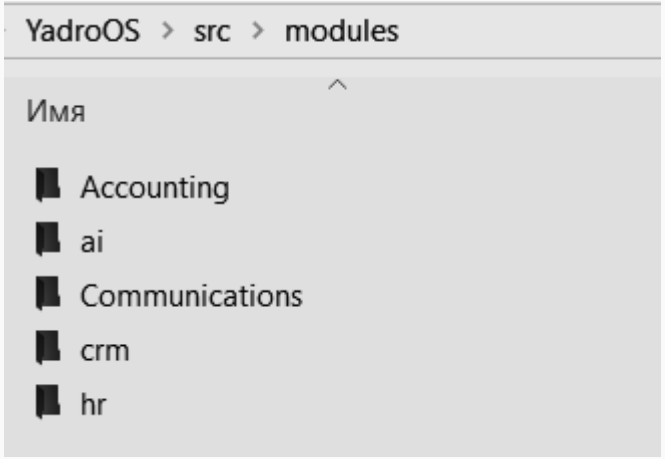
Схема Бази Даних

- Схема бази даних використовує ключові реляційні зв'язки
- Основними сутностями є Користувачі, Клієнти та Угоди
- Модуль Фінанси оперує сутністю Транзакції та пов'язаними даними
- Індексація ключових полів забезпечує значне прискорення пошуку
- Використовуються зв'язки One-to-Many та Many-to-Many між сутностями

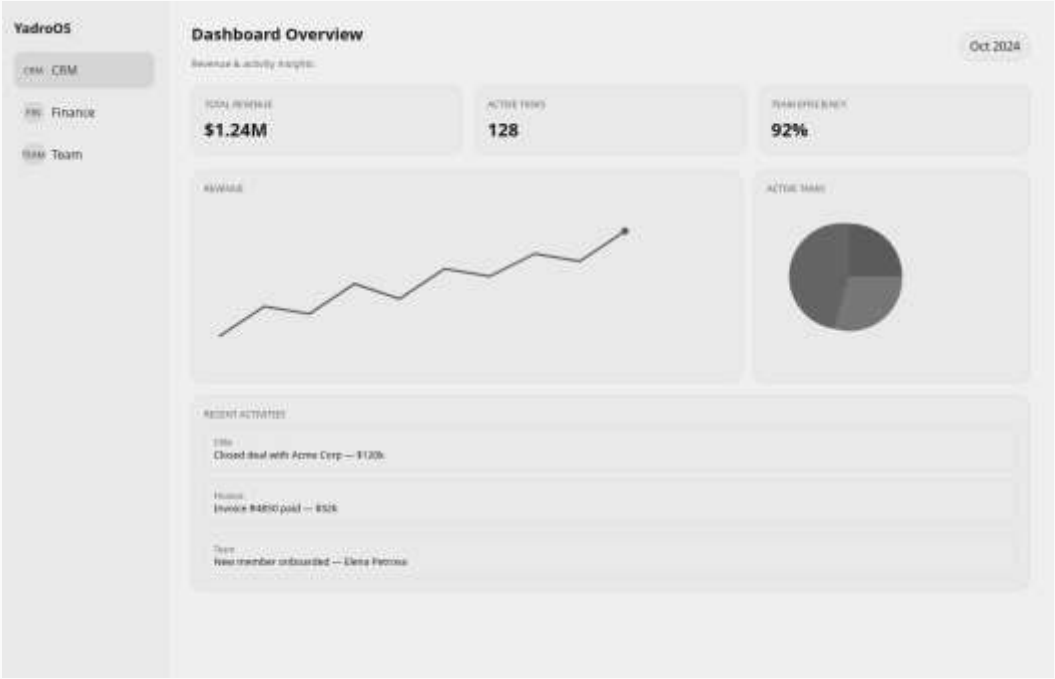


Модульний Моноліт

- Модульний моноліт спрощує розробку та розгортання
- Структура src/modules/ містить основні функціональні блоки
- Модуль CRM керує клієнтами та воронкою продажів
- Модуль Accounting обробляє інвойси та P&L звіти
- HR модуль відповідає за співробітників та трекінг часу
- Communications агрегує всі повідомлення для команд



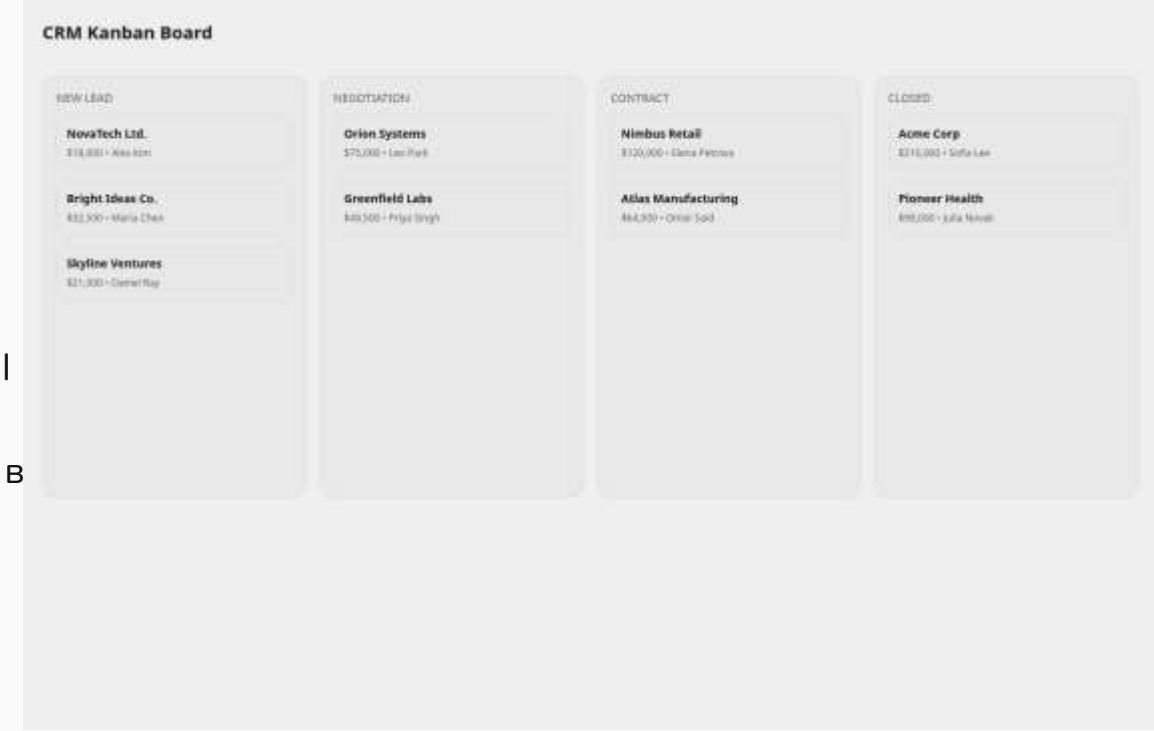
Інтерфейс: Dashboard



- Агрегація ключових показників ефективності у реальному часі
- Паралельні запити даних через React Query для швидкості
- Використання shadcn/ui та recharts для візуалізації
- Оптимістичне оновлення інтерфейсу покращує досвід

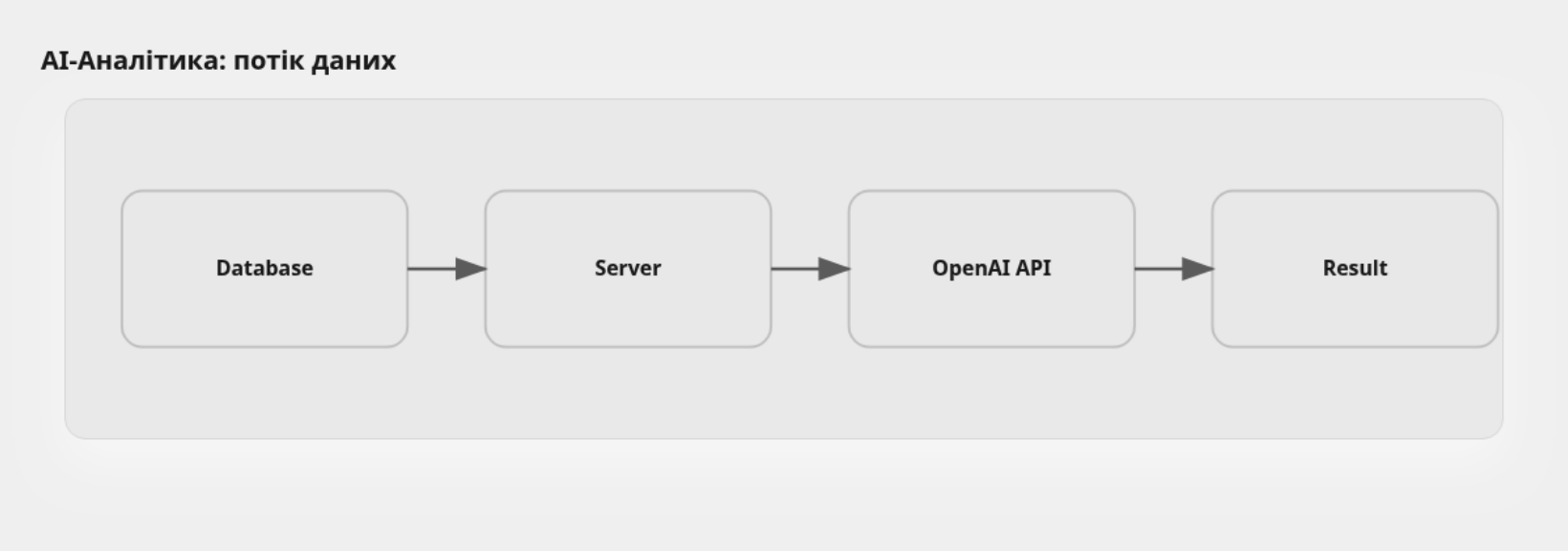
CRM: Kanban Логіка

- Управління повним життєвим циклом клієнтської угоди
- Функціонал Drag-and-Drop для швидкої зміни статусу
- Зміна статусу ініціює виклик API для оновлення даних
- Автоматичне створення запису в Accounting при закритті угоди



AI: Аналітика та LLM

- Lead Scoring прогнозує успіх клієнтських угод
- Аналіз ефективності надає оцінку роботи команд
- Використовується Adapter Pattern для інтеграції з LLM
- Анонімізація даних зберігає конфіденційність при аналізі



Безпека Платформи

- Використовується JWT для аутентифікації користувачів
- Схеми Access та Refresh токенів забезпечують сесійність
- Паролі хешуються за допомогою алгоритму bcrypt/argon2
- Вхідні дані валідуються суворо за допомогою Zod
- Middleware перевіряє ролі та дозволи перед доступом

Безпека: перевірка JWT-токена

```
export const authMiddleware = (req: Request, res: Response, next: NextFunction) => {
  const authHeader = req.headers.authorization;
  const token = authHeader?.startsWith('Bearer ')
    ? authHeader.split(' ')[1]
    : undefined;

  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  try {
    const decoded = jwt.verify(token, env.jwtAccessSecret) as AuthPayload;
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({ message: 'Invalid or expired token' });
  }
};
```

Тестування та CI/CD

- Тестування утиліт та сервісів відбувається через Unit Tests.
- Перевірка API ендпоінтів реалізована через Integration Tests (Supertest).
- E2E тести використовуються для перевірки критичних сценаріїв.
- CI/CD забезпечує автоматичний запуск тестів при кожному коміті.

Висновки та Готовність

- Розроблено комплексну інтегровану платформу з автоматизацією
- Успішно реалізовано чотири ключові функціональні модулі
- Система повністю готова до хмарного розгортання та роботи
- Рішення забезпечує централізоване управління даними організації