Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Бакалавр
(назва освітнього ступеня)

на тему: Розробка масштабованої B2B-платформи для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту.

Виконав(ла): студент(ка) 4 курсу, групи СП-43
спеціальності 121 «Інженерія програмного Забезпечення»
(шифр і назва спеціальності)

_____ Смик В. В.
(підпис) (прізвище та ініціали)

Керівник _____ Мудрик І. Я.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ _____
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ _____
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет *комп'ютерно-інформаційних систем і програмної інженерії*
(повна назва факультету)

Кафедра *програмної інженерії*
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М.Р.
(підпис) (прізвище та ініціали)
« 6 » квітня 2026 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня *бакалавр*
(назва освітнього ступеня)

за спеціальністю *121 «Інженерія програмного забезпечення»*
(шифр і назва спеціальності)

студенту Смик Володимир Володимирович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка масштабованої B2B-платформи для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту.

Керівник роботи Мудрик Іван Ярославович, доктор філософії
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*6*» *квітня* 2026 року № 4/9-171

2. Термін подання студентом завершеної роботи *22.06.2026*

3. Вихідні дані до роботи *Технічне завдання*

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Аналіз вимог та предметної області
2. Проєктування
3. Конструювання та тестування
4. Безпека життєдіяльності, основи охорони праці.
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Слайди презентації, діаграма варіантів використання системи, схема бази даних, схема архітектури застосунку, схема розгортання системи у безсерверному середовищі.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці			

7. Дата видачі завдання *06.04.2026р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	
1	*Розробка технічного завдання*	*6.04 – 12.04*	*Виконано*
2	*Робота над першим розділом «Аналіз вимог до програмної системи»*	*13.04 – 26.04*	*Виконано*
3	*Робота над другим розділом «Проєктування та розробка програмної системи»*	*27.04 – 03.05*	*Виконано*
4	*Робота над третім розділом «Тестування, впровадження та аналіз результатів»*	*04.05 – 17.05*	*Виконано*
5	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*18.05 – 24.05*	*Виконано*
6	*Оформлення пояснювальної записки і графічного матеріалу*	*25.05 – 7.06*	*Виконано*
7	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*	
8	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*	
9	*Захист кваліфікаційної роботи бакалавра*		

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД— база даних — організована сукупність структурованих даних, що зберігаються та опрацьовуються інформаційною системою

ВДТ— відеодисплейний термінал — пристрій відображення інформації на основі електронного дисплея (монітор)

ОС— операційна система — комплекс програм, що керує ресурсами обчислювальної системи

ПЗ— програмне забезпечення — сукупність програм та супровідної документації для роботи обчислювальної системи

ПК— персональний комп’ютер

СКВ— система контролю версій — програмний засіб для реєстрації та керування змінами в кодовій базі

ШІ— штучний інтелект — здатність технічної системи виконувати завдання, що традиційно потребують людського інтелекту

Агент— програмний компонент на основі мовної моделі, що автономно опрацьовує звернення користувача та формує відповідь

Віджет— вбудовуваний інтерфейсний компонент, що розміщується на вебсторінці організації для взаємодії з користувачем

Ескалація— передавання звернення від інтелектуального агента до оператора у разі неможливості його автономного вирішення

Лістинг— оформлений фрагмент програмного коду, наведений у тексті роботи

Мультиорендарність— архітектурний підхід, за якого один екземпляр системи обслуговує декілька організацій з ізоляцією їхніх даних

Орендар— організація, що використовує платформу та дані якої ізольовано від даних інших організацій

Рефакторинг— зміна внутрішньої структури коду без зміни його зовнішньої поведінки з метою покращення якості

Хук— функція бібліотеки React, що надає доступ до стану та можливостей життєвого циклу функційного компонента

Терміни та скорочення англійською мовою

API— Application Programming Interface — прикладний програмний інтерфейс для взаємодії програмних компонентів

B2B— Business-to-Business — модель, за якої продукт або послуга призначені для використання іншими організаціями

CSS— Cascading Style Sheets — мова опису зовнішнього вигляду вебсторінок

DOI— Digital Object Identifier — цифровий ідентифікатор наукової публікації

FK— Foreign Key — зовнішній ключ, що реалізує зв’язок між записами у базі даних

HTML— HyperText Markup Language — мова розмітки гіпертекстових документів

HTTP— HyperText Transfer Protocol — протокол передавання даних у мережі

JSON— JavaScript Object Notation — текстовий формат обміну структурованими даними

LLM— Large Language Model — велика мовна модель, навчена на значних обсягах текстових даних

MCP— Model Context Protocol — протокол взаємодії мовних моделей із зовнішніми інструментами та джерелами даних

RAG— Retrieval-Augmented Generation — генерація, доповнена пошуком: підхід, що поєднує семантичний пошук із формуванням відповіді мовною моделлю

SDK— Software Development Kit — набір інструментів для розроблення програмного забезпечення

SSR— Server-Side Rendering — формування вмісту вебсторінки на боці

АНОТАЦІЯ

Кваліфікаційна робота бакалавра. Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, спеціальність 121 «Інженерія програмного забезпечення». ТНТУ, 2026, сторінок [ ], таблиць [ ], рисунків [ ], презентація.

Тема: Розробка масштабованої B2B-платформи для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту.

У даній кваліфікаційній роботі розглядається процес розробки масштабованої B2B-платформи для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту. Метою роботи є створення ефективної програмної системи, що поєднує серверну та клієнтську складові з підтримкою текстового й голосового каналів взаємодії та забезпечує автономне опрацювання звернень із ескалацією складних випадків операторові.

У роботі представлено аналіз предметної області та наявних рішень, сформульовано вимоги до системи, описано етапи розробки, зокрема проєктування моделі даних, реалізацію серверного програмного інтерфейсу на основі безсерверної реактивної платформи, побудову клієнтської частини засобами фреймворку Next.js за компонентним підходом та інтеграцію інтелектуальної підсистеми опрацювання звернень. Особливу увагу приділено питанням ізоляції даних організацій за принципом багатоорендарності, наскрізної типової безпеки та масштабованості системи.

Результатом роботи є працездатна платформа автоматизації клієнтської підтримки, що дозволяє автономно опрацьовувати типові звернення агентом штучного інтелекту, обслуговувати декілька організацій єдиним екземпляром системи з ізоляцією їхніх даних та забезпечувати оператора зручним інтерфейсом керування зверненнями в режимі, близькому до реального часу.

Ключові слова: автоматизація клієнтської підтримки, штучний інтелект, B2B-платформа, багатоорендарність, Next.js, React, TypeScript, монорепозиторій, реактивна архітектура.

ABSTRACT

Bachelor's qualification work. Ternopil Ivan Puluj National Technical University, Department of Software Engineering, specialty 121 "Software Engineering". TNTU, 2026, pages [ ], tables [ ], figures [ ], presentation.

Topic: Development of a scalable B2B platform for customer support automation based on multimodal artificial intelligence agents.

This qualification work explores the development of a scalable B2B platform for customer support automation based on multimodal artificial intelligence agents. The aim of the work is to create an efficient software system that combines server-side and client-side parts with support for text and voice interaction channels and provides autonomous handling of customer inquiries with escalation of complex cases to a human operator.

The work presents an analysis of the subject area and existing solutions, formulates the system requirements, and describes the development stages, including data model design, implementation of the server-side application programming interface based on a serverless reactive platform, construction of the client side using the Next.js framework following a component-based approach, and integration of the intelligent inquiry-handling subsystem. Particular attention is given to the issues of tenant data isolation based on the multi-tenancy principle, end-to-end type safety, and system scalability.

The result of the work is a functional customer support automation platform that enables autonomous handling of typical inquiries by an artificial intelligence agent, serves multiple organizations from a single instance of the system with isolation of their data, and provides operators with a convenient interface for managing inquiries in near real time.

Keywords: customer support automation, artificial intelligence, B2B platform, multi-tenancy, Next.js, React, TypeScript, monorepository, reactive architecture.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології відіграють визначальну роль у забезпеченні конкурентоспроможності бізнесу та ефективності взаємодії з клієнтами. Зі зростанням кількості онлайн-сервісів і обсягу клієнтських звернень організації дедалі частіше потребують інструментів, здатних опрацьовувати комунікації з користувачами централізовано, без розгортання та супроводу власної серверної інфраструктури. Саме тому широкого поширення набула модель «програмне забезпечення як послуга» (Software as a Service, SaaS), орієнтована на корпоративного споживача (Business-to-Business, B2B), яка надає готову платформу підтримки за підпискою.

Традиційний підхід, за якого всі звернення опрацьовуються операторами вручну, погано масштабується: витрати на підтримку зростають пропорційно до масштабу бізнесу, а час реагування на запити збільшується. Це робить актуальною автоматизацію клієнтської підтримки засобами програмних агентів штучного інтелекту, здатних автономно опрацьовувати значну частину типових звернень, а складніші випадки передавати операторові. Сучасні агенти на основі великих мовних моделей дозволяють не лише формувати відповіді природною мовою, а й виконувати прикладні дії через інструментальні виклики та спиратися на доменні знання організації за допомогою генерації, доповненої пошуком (Retrieval-Augmented Generation, RAG).

Метою цієї кваліфікаційної роботи є розробка масштабованої B2B-платформи для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту. Платформа поєднує текстовий і голосовий канали взаємодії з користувачем, що визначає її мультимодальний характер: окрім текстового діалогу через вбудовуваний чат-віджет, передбачено голосове спілкування засобами зовнішнього сервісу телефонії. Підтримка декількох каналів підвищує доступність сервісу для різних категорій користувачів та розширює сценарії його застосування. Для побудови серверної частини використано безсерверну реактивну платформу з документоорієнтованим сховищем, що

поєднує зберігання даних, виконання прикладної логіки та автоматичну синхронізацію стану з клієнтами в єдиній обчислювальній моделі.

Оскільки платформа надається корпоративним клієнтам, важливою її властивістю є багатоорендарність (multi-tenancy) — здатність обслуговувати декілька організацій єдиним екземпляром системи з надійною ізоляцією їхніх даних. Клієнтську частину реалізовано засобами фреймворку Next.js на основі бібліотеки React із застосуванням компонентного підходу, а кодову базу організовано у форматі монорепозиторію на основі менеджера залежностей pnpm, що забезпечує повторне використання коду, узгоджені конфігурації та наскрізну типову безпеку від сховища даних до інтерфейсу користувача.

У роботі розглянуто процес створення платформи з позиції аналізу предметної області, проєктування архітектури й моделі даних, реалізації серверної та клієнтської частин, а також інтеграції інтелектуальної підсистеми опрацювання звернень. Окрему увагу приділено питанням ізоляції даних орендарів, безпечного керування обліковими даними зовнішніх інтеграцій та забезпечення масштабованості системи. Детально описано, як застосування сучасних технологій веброзробки та реактивної архітектури дозволяє створювати масштабовані, надійні та зручні в обслуговуванні системи.

Сучасні тенденції в галузі програмного забезпечення, зокрема поширення великих мовних моделей та агентного підходу до автоматизації, підкреслюють доцільність інтеграції засобів штучного інтелекту в бізнес-процеси підтримки користувачів. У цьому контексті розроблена платформа сприятиме скороченню навантаження на службу підтримки, зменшенню часу реагування на звернення та підвищенню якості обслуговування клієнтів. У роботі також окреслено перспективи подальшого розвитку системи, зокрема додавання нових каналів взаємодії та інтеграції з іншими корпоративними системами.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

Розроблення будь-якої програмної системи розпочинається з аналізу предметної області, що дозволяє визначити актуальність завдання, дослідити наявні підходи до його розв'язання та сформувати обґрунтовані вимоги до майбутнього продукту. У цьому розділі розглянуто предметну область автоматизації клієнтської підтримки на основі штучного інтелекту. Здійснено огляд наявних на ринку рішень-конкурентів та проведено їх порівняльний аналіз з метою виявлення спільних переваг і обмежень. На підставі результатів аналізу обґрунтовано вибір напрямку дослідження та технологічного стека, визначено методологію розроблення, а також сформульовано функціональні та нефункціональні вимоги до платформи, що розробляється. Отримані в цьому розділі результати становлять підґрунтя для проєктування архітектури системи, розглянутого в наступному розділі

1.1 Огляд конкурентів

Ринок програмних рішень для автоматизації клієнтської підтримки на основі штучного інтелекту є сформованим і конкурентним. Серед платформ, що реалізують модель автономного опрацювання звернень із подальшою ескалацією складних випадків операторові, доцільно виокремити кількох провідних представників, аналіз яких дозволяє визначити галузеві стандарти функціональності та сформувати вимоги до власної системи.

Першим аналогом є платформа Intercom із вбудованим агентом штучного інтелекту Fin.

Intercom є хмарною платформою клієнтської підтримки, що поєднує живий чат, довідковий центр та систему опрацювання звернень в єдиному продукті. Платформа підтримує канали чату, електронної пошти, SMS та соціальних мереж, інтерпретуючи запити користувачів засобами штучного інтелекту з подальшим автоматичним вирішенням або маршрутизацією до відповідного оператора.

Центральним компонентом платформи є агент Fin, побудований на основі великих мовних моделей. [27]

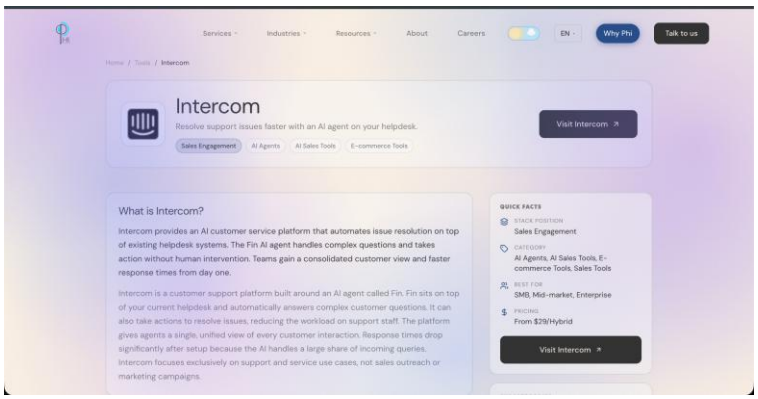

Рис. 1.1 – Вигляд вбудовуваного віджета та агента Fin платформи Intercom

Агент Fin опрацьовує звернення автономно, формуючи відповіді на основі бази знань організації та виконуючи прикладні дії без втручання оператора. Агент зберігає контекст упродовж усього діалогу, що дозволяє коректно опрацьовувати уточнювальні запити користувача як продовження попередніх, а не як ізольовані звернення. Платформа надає засоби налаштування інструкцій для агента, що забезпечують дотримання політик підтримки та стилю комунікації; визначені одноразово ім'я, тон та поведінка під час ескалації зберігаються незмінними впродовж тисяч діалогів, що усуває характерну для роботи операторів варіативність стилю. Через конектори даних до зовнішніх систем агент отримує можливість персоналізувати відповіді й виконувати складні завдання від імені користувача, а багатокрокові сценарії реалізуються механізмом процедур — інструкціями природною мовою, що поєднують інструментальні виклики та оновлення стану звернення. Понад 450 інтеграцій дозволяють вбудувати платформу в наявні робочі процеси підтримки. Окремо реалізовано механізм самовдосконалення: платформа виявляє прогалини в базі знань, навчається на історичних зверненнях та надає засоби пакетного тестування й інспекції відповідей, що дозволяє систематично підвищувати результативність агента. Builts AI + 3

Тарифікація побудована за моделлю оплати за результат: кошти стягуються лише за успішне вирішення звернення або виконання сценарію з передаванням оператору, тоді як ескалації за замовчуванням не тарифікуються. Вартість одного вирішення становить близько 0,99 долара США, що за значних обсягів звернень призводить до швидкого зростання витрат. Заявлені та реальні показники результативності суттєво різняться: розробник декларує частку автономного вирішення на рівні 67 % за понад 40 мільйонами діалогів, проте за опублікованими прикладами впровадження реальна частка вирішення в умовах конкретних організацій перебуває в межах 42–50 %. Крім того, пересурсна модель приховує додаткові складники витрат: використання Fin неможливе без передплаченого тарифного плану Intercom, а функція підтримки операторів вимагає окремого модуля, що тарифікується за кожного оператора, унаслідок чого платформні та додаткові платежі швидко накопичуються.

Другим аналогом є платформа Zendesk із підсистемою Zendesk AI. Zendesk є однією з найпоширеніших у світі платформ клієнтської підтримки, побудованою навколо системи опрацювання звернень (англ. ticketing). Агенти штучного інтелекту Zendesk позиціонуються як автономні працівники, що опрацьовують звернення в каналах чату, електронної пошти та вебінтерфейсу самостійно й передають їх операторові лише за потреби; на відміну від скриптових чат-ботів попереднього покоління.

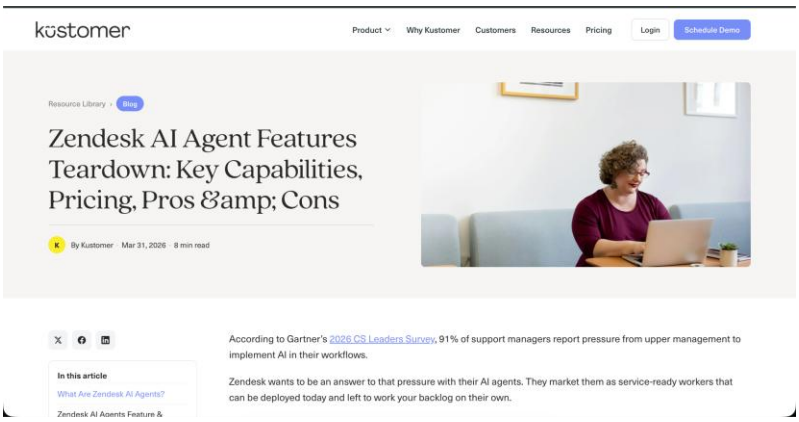

Рис. 1.2 – Вигляд інтерфейсу платформи Zendesk

Базові можливості штучного інтелекту входять до тарифних планів Suite, тоді як розширені корпоративні функції доступні через додатковий модуль Advanced AI вартістю 50 доларів США за оператора щомісяця. Цей модуль є обов'язковим для повноцінної автоматизації: без нього функціональність обмежується конструктором сценаріїв на основі правил, тоді як саме модуль Advanced AI розблоковує автоматичне вирішення звернень, інтелектуальну маршрутизацію та розпізнавання намірів. Інтелектуальна маршрутизація передбачає аналіз кожного вхідного звернення з визначенням наміру, тональності та мови повідомлення з подальшим застосуванням цих ознак для спрямування звернення до відповідної групи операторів. Результативність агента безпосередньо залежить від якості бази знань організації: застарілі або неповні довідкові матеріали призводять до ескалацій, а підвищення частки автономно вирішених звернень потребує систематичного доопрацювання контенту, уточнення сценаріїв та ітеративного налаштування на основі аналітики.

Платформа реалізує тарифікацію за результат, проте її структура є складною та багатокомпонентною. Окрім фіксованої плати за оператора, стягується змінна плата за кожне автономно вирішене звернення, унаслідок чого сукупні витрати команди швидко зростають зі збільшенням обсягу автоматизованих звернень. Додаткові модулі контролю якості та управління персоналом ще більше підвищують вартість володіння. Інтерфейс платформи критикують за громіздкість та залежність від численних застосунків з маркетплейсу, що зумовлює додаткові витрати й ускладнює користувацький досвід. Суттєвим архітектурним обмеженням є те, що платформа прив'язує користувача до власних мовних моделей без можливості вибору альтернативної моделі, а голосовий канал реалізовано через додатковий модуль контактного центру, а не як інтегровану першокласну складову. Унаслідок цього організації з високим обсягом звернень або потребою у гнучкому виборі мовної моделі нерідко змушені виносити рівень агента штучного інтелекту за межі платформи, інтегруючи його через програмний інтерфейс, що ускладнює архітектуру рішення.

Підсумовуючи аналіз наявних рішень, можна виокремити спільні для них обмеження, що залишають простір для альтернативного підходу. По-перше, домінівна цінова модель з оплатою за кожне автономно вирішене звернення робить вартість володіння важко прогнозованою та такою, що зростає пропорційно до успішності автоматизації, фактично штрафуючи організацію за ефективне використання системи. По-друге, прив'язка до власних мовних моделей постачальника позбавляє організацію контролю над ключовим компонентом системи та можливості обрати модель відповідно до власних вимог щодо якості, вартості чи конфіденційності. По-третє, голосовий канал у розглянутих рішеннях реалізовано як другорядну надбудову, а не як рівноправну складову, що обмежує його інтеграцію із загальним потоком опрацювання звернень. Виявлені обмеження обґрунтовують доцільність розроблення платформи, яка передбачала б прозору модель володіння, гнучкість у виборі мовної моделі через абстрактний програмний інтерфейс та від початку інтегровану підтримку як текстового, так і голосового каналів взаємодії, що й визначає напрям подальшого дослідження.

1.2 Порівняльний аналіз конкурентів

Для визначення обґрунтованого напрямку розробки необхідно проаналізувати переваги та недоліки наявних рішень і на основі цього аналізу сформувати вимоги, що дозволять створити конкурентоспроможну систему. Зіставлення розглянутих платформ доцільно проводити за сукупністю критеріїв, що охоплюють як функціональні можливості, так і нефункціональні характеристики: ступінь автономності опрацювання звернень, набір каналів взаємодії, гнучкість вибору мовної моделі, модель тарифікації та сукупну вартість володіння, а також складність інтеграції та ступінь залежності від постачальника. Саме ці критерії визначають придатність платформи для застосування в умовах конкретної організації та формують підставу для проєктування власного рішення.

Для платформи Intercom можна виділити такі переваги:

- високий ступінь автономності: агент Fin опрацьовує звернення самостійно, зберігаючи контекст діалогу та виконуючи багатокрокові сценарії через механізм процедур;
- широкий набір інтеграцій із зовнішніми системами, що спрощує вбудовування платформи в наявні робочі процеси підтримки;
- сучасний вбудовуваний віджет та підтримка багатьох каналів взаємодії, зокрема чату, електронної пошти та повідомлень;
- механізм самовдосконалення, що виявляє прогалини в базі знань та навчається на історичних зверненнях;
- прозора в декларованій частині модель тарифікації за результат, за якої оплата формально стягується лише за фактично вирішені звернення.

Недоліки платформи Intercom:

- висока сукупна вартість за значних обсягів звернень унаслідок пересурсної тарифікації;
- приховані складники витрат: обов'язковість передплаченого тарифного плану та окрема плата за модуль підтримки операторів;
- суттєвий розрив між заявленою (близько 67 %) та реальною (42–50 %) часткою автономного вирішення звернень в умовах конкретних організацій;
- залежність від постачальника, характерна для пропрієтарного хмарного рішення із закритим стеком.

Для платформи Zendesk можна виділити такі переваги

- зрілість та поширеність платформи, розвинена екосистема інтеграцій та сторонніх застосунків;
- розвинена система опрацювання звернень із інтелектуальною маршрутизацією за наміром, тональністю та мовою повідомлення;
- масштабованість, достатня для обслуговування корпоративних навантажень;

- розширені засоби аналітики результативності та виявлення прогалин у базі знань.

Недоліки платформи Zendesk:

- громіздкий та застарілий інтерфейс, що ускладнює освоєння та потребує значного часу на впровадження;
- висока сукупна вартість володіння через стягування плати за оператора, обов'язковий модуль Advanced AI, пересурсні платежі за вирішення та додаткові модулі контролю якості й управління персоналом;
- прив'язка до власних мовних моделей платформи без можливості вибору альтернативної моделі;
- реалізація голосового каналу як додаткового модуля контактного центру, а не як інтегрованої першокласної складової;
- автоматичне стягування плати за перевищення обсягу вирішень без попереднього погодження, що ускладнює прогнозування витрат.

Для наочного зіставлення розглянуті платформи доцільно порівняти за ключовими критеріями. За ступенем автономності опрацювання звернень обидва наявні рішення та проєктована платформа є рівноцінними: усі вони забезпечують автономне опрацювання звернень агентом штучного інтелекту з підтримкою текстового каналу у вигляді вбудовуваного чат-віджета та підсистемою генерації, доповненої пошуком [22]. Спільною є й багатоорендарність, що дозволяє обслуговувати декілька організацій єдиним екземпляром системи.

Принципові відмінності виявляються в інших критеріях. Голосовий канал у платформах Intercom та Zendesk реалізовано як додатковий модуль, тоді як у проєктованому рішенні його передбачено як інтегровану складову платформи. Гнучкість вибору мовної моделі в Intercom є обмеженою, у Zendesk — відсутньою через прив'язку до власних моделей, натомість архітектура проєктованої платформи передбачає можливість зміни мовної моделі як конфігурованого параметра. Істотно різняться й моделі тарифікації: Intercom застосовує оплату за результат поверх передплаченого плану, Zendesk поєднує плату за оператора з

пересурсними платежами за вирішення, унаслідок чого прогнозованість витрат в обох випадках є низькою; проєктоване рішення є самостійно розгортуваним, що забезпечує вищу прогнозованість сукупної вартості володіння. Нарешті, обидві наявні платформи є пропрієтарними рішеннями із закритим стеком та високим ступенем залежності від постачальника, тоді як проєктована платформа побудована на основі відкритого стека технологій, що знижує таку залежність.

Отже, проведений аналіз засвідчує, що чинні платформи надають широкий спектр функцій та забезпечують високий ступінь автономності опрацювання звернень, проте характеризуються спільними істотними обмеженнями: високою та складно прогнозованою вартістю володіння, прив'язкою до власних мовних моделей, залежністю від постачальника та реалізацією голосового каналу як другорядної надбудови. Зазначені обмеження зумовлюють доцільність розробки власної платформи на основі відкритого стека технологій із інтегрованою підтримкою текстового й голосового каналів, гнучким вибором мовної моделі та реактивною архітектурою, що забезпечує прогнозовану вартість володіння та незалежність від постачальника. Виявлені переваги наявних рішень водночас формують орієнтир для визначення функціональних вимог до проєктованої системи, що розглядаються в наступних підрозділах.

1.3 Обґрунтування вибору напрямку дослідження

Вибір напрямку дослідження у сфері веброзробки зумовлений низкою чинників: попитом на ринку, доступністю технологій, ефективністю інструментів та можливостями масштабування. Вебплатформи є основним каналом взаємодії організацій із клієнтами, що визначає актуальність розробки програмних рішень саме у вебсередовищі. Сучасна веброзробка охоплює клієнтську частину (фронтенд), що формує інтерфейс взаємодії з користувачем, серверну частину (бекенд), що реалізує прикладну логіку та опрацювання даних, а також сховище даних. Якість програмного продукту значною мірою визначається обґрунтованістю вибору технологій на кожному з цих рівнів.

Для реалізації платформи обрано стек технологій на основі мови програмування TypeScript [3]. TypeScript є надбудовою над мовою JavaScript, що доповнює її системою статичних типів. Вибір цієї мови зумовлений тим, що статична типізація дозволяє виявляти значну частину помилок на етапі компіляції, а не під час виконання, та забезпечує наскрізну типову узгодженість від сховища даних до інтерфейсу користувача. Застосування єдиної мови як на клієнтському, так і на серверному рівні спрощує розробку та супровід, оскільки усуває потребу в перемиканні між різними мовами та дозволяє повторно використовувати спільний код і типи.

Клієнтську частину реалізовано засобами фреймворку Next.js [5] на основі бібліотеки React [4]. React є однією з найпоширеніших бібліотек для побудови інтерфейсів користувача, що ґрунтується на компонентному підході: інтерфейс конструюється з ізольованих повторно використовуваних компонентів, кожен з яких відповідає за окрему частину представлення. Next.js є фреймворком, що надбудовується над React і розширює його можливостями серверного рендерингу та маршрутизації. Цей фреймворк обрано з таких причин. Архітектурна парадигма App Router у поєднанні із серверними компонент [7]ами React дозволяє розмежувати код, що виконується на сервері, та код, що виконується у браузері, скорочуючи обсяг коду, який завантажується на клієнт, і підвищуючи швидкість

початкового відображення сторінок. Маршрутизація відображається на структуру файлової системи, що спрощує організацію сторінок застосунку. Компонентний підхід забезпечує побудову інтерфейсу з повторно використовуваних одиниць, що підвищує супроводжуваність кодової бази та дозволяє незалежно розробляти й тестувати окремі компоненти.

Кодову базу організовано у форматі монорепозиторію — єдиного репозиторію, у межах якого співіснують декілька застосунків та бібліотек спільного використання. Засобом організації монорепозиторію обрано механізм робочих просторів менеджера залежностей pnpm [8]. Такий підхід обрано з кількох причин. По-перше, pnpm використовує контентно-адресоване сховище пакунків, що усуває дублювання ідентичних версій залежностей на файловій системі та забезпечує детерміновану структуру залежностей. По-друге, на відміну від класичних менеджерів залежностей, pnpm унеможливлює неявний доступ до недекларованих залежностей, чим підвищує достовірність графа залежностей та відтворюваність складання. По-третє, монорепозиторій дозволяє застосункам платформи безпосередньо споживати спільні бібліотеки — дизайн-систему, серверний шар та централізовані конфігурації, — що забезпечує повторне використання коду та узгодженість конфігурацій між усіма складовими платформи. Для оркестрації завдань збирання, перевірки коду та перевірки типів застосовано систему інкрементального складання Turborepo [9], що кешує результати виконання завдань на основі хешування вхідних даних та перевиконує лише ті з них, вхідні дані яких змінилися, чим суттєво скорочує час циклів розробки.

Сукупність застосованих у проєкті технологій та їхнє призначення в межах архітектури платформи узагальнено в таблиці 2.1. Технології згруповано за рівнями системи — інструментарій монорепозиторію, клієнтський рівень, серверний рівень та зовнішні сервіси, — що відображає розподіл відповідальності між складовими платформи.

Таблиця 2.1 — Технологічний стек платформи

Рівень	Технологія	Призначення
Монорепозиторій	pnpm	Менеджер залежностей та організація робочих просторів
Монорепозиторій	Turborepo	Інкрементальне складання та кешування завдань
Монорепозиторій	TypeScript	Статична типізація та наскрізна типова безпека
Клієнтський рівень	Next.js	Фреймворк із серверним рендерингом та маршрутизацією
Клієнтський рівень	React	Побудова інтерфейсу за компонентним підходом
Клієнтський рівень	Tailwind CSS	Утилітарна стилізація компонентів
Клієнтський рівень	@workspace/ui	Спільна дизайн-система платформи
Серверний рівень	Convex	Реактивне сховище даних та виконання серверних функцій
Серверний рівень	Convex Agent, RAG	Інтелектуальна підсистема опрацювання звернень
Зовнішні сервіси	Clerk	Автентифікація та керування обліковими записами
Зовнішні сервіси	Vapi	Голосовий канал взаємодії
Зовнішні сервіси	AWS Secrets Manager	Захищене зберігання облікових даних інтеграцій

Як видно з таблиці, кожен рівень системи реалізовано засобами спеціалізованих технологій із чітко визначеним призначенням, а їхня взаємодія

підпорядкована загальній архітектурній схемі (рисунок 2.1). Інструментарій монорепозиторію забезпечує організацію та складання кодової бази, клієнтський і серверний рівні реалізують прикладну функціональність, а зовнішні сервіси під'єднано до серверного рівня для делегування автентифікації, голосової взаємодії та зберігання секретів.

Серверну частину реалізовано на основі безсерверної (англ. serverless) реактивної платформи Convex [12], що поєднує транзакційне документоорієнтоване сховище даних, середовище виконання прикладних функцій та механізм реактивної синхронізації стану з клієнтами в єдиній обчислювальній моделі. Convex обрано з таких причин. Реактивна модель синхронізації забезпечує автоматичне поширення змін даних на підписані клієнти без потреби в ручній організації повторних запитів до сервера та інвалідації кешу, що спрощує реалізацію інтерфейсів, які мають оновлюватися в режимі, близькому до реального часу. Схему даних визначають декларативно в коді, причому це визначення є водночас джерелом істини для перевірки коректності вхідних даних під час виконання та основою для автоматичної генерації статичних типів TypeScript, чим досягається типова узгодженість між сховищем даних та клієнтським кодом. Крім того, платформа надає вбудовані засоби інтеграції з компонентами штучного інтелекту, зокрема для керування діалоговими потоками та реалізації генерації, доповненої пошуком.

Інтелектуальну підсистему платформи побудовано на основі великої мовної моделі з підключенням інструментальних викликів, що дозволяють агенту виконувати прикладні дії, та підсистеми генерації, доповненої пошуком, що забезпечує формування відповідей на основі доменних знань організації. Голосовий канал взаємодії реалізовано через інтеграцію із зовнішнім сервісом телефонії Vapi [17]. Автентифікацію операторів делеговано зовнішньому провайдеру Clerk [16], що звільняє від потреби самостійно реалізовувати механізми керування обліковими записами, а безпечне зберігання облікових даних зовнішніх інтеграцій винесено до спеціалізованого захищеного сховища секретів.

Отже, вибір наведеного стека технологій зумовлений його сучасністю, наскрізною типовою безпекою, реактивною моделлю взаємодії клієнта із сервером та вбудованою підтримкою компонентів штучного інтелекту.

1.4 Вибір методології розробки

Методології розробки програмного забезпечення є системами принципів, практик і процедур, що керують процесом створення програмних продуктів. Вони визначають структуру роботи, послідовність етапів розробки, розподіл відповідальності та методи контролю якості. Основною метою застосування методологій є підвищення ефективності розробки, зниження ризиків та забезпечення відповідності кінцевого продукту вимогам. Вибір методології залежить від низки чинників, зокрема від розміру та складності проєкту, стабільності вимог, чисельності команди та наявних ресурсів.

Існує багато різних методологій, кожна з яких має свої переваги та недоліки. Класичні методології, такі як водоспадна (каскадна) модель, передбачають послідовне виконання етапів розробки, починаючи з аналізу вимог і завершуючи тестуванням та супроводом, причому перехід до наступного етапу відбувається лише після повного завершення попереднього. Такий підхід забезпечує чітку структуру процесу та зрозумілість планування, проте є недостатньо гнучким за умови зміни вимог: внесення коректив на пізніх етапах потребує повернення до попередніх стадій і пов'язане зі значними витратами. Це робить каскадну модель малопридатною для проєктів, у яких вимоги остаточно не сформовані на початку розробки та можуть уточнюватися в процесі.

Альтернативою класичним методологіям є гнучкі (англ. Agile) методології, орієнтовані на ітеративну розробку та постійну адаптацію до змін. Agile передбачає розбиття проєкту на короткі цикли — ітерації (спринти), — кожен з яких триває зазвичай від одного до чотирьох тижнів і завершується працездатним інкрементом продукту. Такий підхід дозволяє регулярно отримувати зворотний зв'язок щодо проміжних результатів та вносити корективи на ранніх етапах, що сприяє

швидшому реагуванню на зміни вимог і підвищенню якості кінцевого продукту. Agile ґрунтується на засадах, що віддають пріоритет працездатному програмному забезпеченню перед вичерпною документацією, співпраці та готовності до змін перед суворим дотриманням початкового плану. Серед основних практик гнучких методологій — ітеративне постачання функціональності, безперервне тестування та інтеграція, а також регулярний аналіз результатів кожної ітерації з метою вдосконалення процесу розробки.

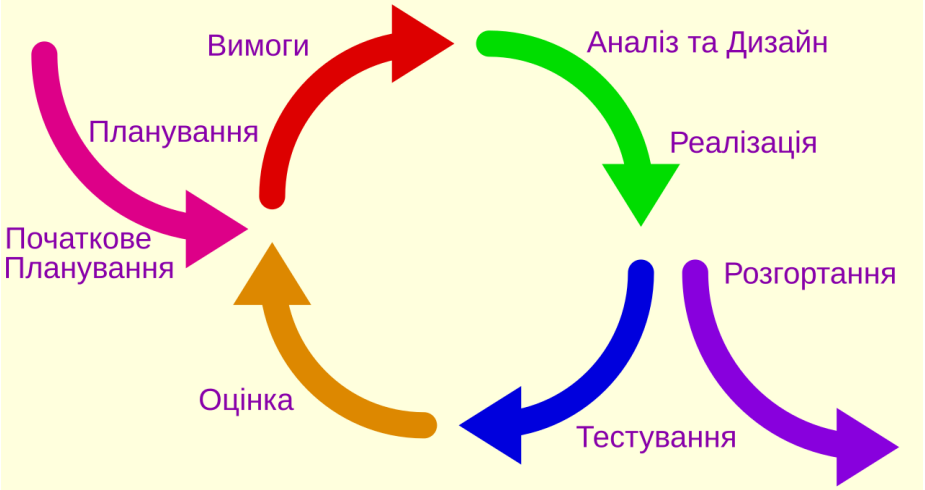

Рис. 1.3 – Цикл ітеративно-інкрементальної розробки

Для реалізації досліджуваної платформи обрано ітеративно-інкрементальний підхід на засадах Agile. Вибір зумовлений кількома чинниками, що випливають із характеру проєкту. По-перше, архітектура платформи передбачає поступове нарощування функціональності окремими відносно незалежними модулями — опрацювання звернень, операторська панель, налаштування віджета, зовнішні інтеграції, інтелектуальна підсистема, — що природно узгоджується з ітеративною моделлю, за якої кожен модуль може розроблятися та вводитися в дію як окремий інкремент. По-друге, вимоги до системи, зокрема в частині поведінки інтелектуального агента та сценаріїв опрацювання звернень, уточнюються в процесі розробки на основі результатів проміжного тестування, що робить гнучкий підхід доцільнішим за каскадний. По-третє, модульно-доменна організація кодової бази у форматі монорепозиторію на технічному рівні підтримує ітеративну

розробку: відносна незалежність функціональних доменів дозволяє вносити зміни локально, не порушуючи роботу суміжних частин системи.

1.5 Формування вимог

Вимоги до системи є фундаментальним аспектом процесу розробки програмного забезпечення, що визначає, як система має функціонувати та взаємодіяти з користувачами й зовнішніми системами. Чітке формулювання вимог на ранньому етапі дозволяє узгодити очікування щодо кінцевого продукту, слугує основою для проєктування архітектури та критерієм оцінювання готової системи. Вимоги прийнято поділяти на дві основні категорії: функціональні та нефункціональні. Функціональні вимоги описують конкретні функції та завдання, які система повинна виконувати, тобто визначають її поведінку — операції, які вона має підтримувати, та реакції на дії користувачів і зовнішні події. Нефункціональні вимоги, своєю чергою, визначають атрибути якості системи, що не стосуються окремих функцій, але є визначальними для її загальної придатності: продуктивність, безпеку, масштабованість, надійність, зручність використання та супроводжуваність. Збалансоване врахування обох категорій забезпечує створення ефективного, надійного та зручного програмного забезпечення, що відповідає потребам користувачів і бізнес-цілям. Функціональні вимоги до платформи доцільно згрупувати відповідно до основних категорій користувачів та підсистем. У системі виокремлено двох основних акторів: оператора підтримки (автентифікований співробітник організації-орендаря) та кінцевого користувача (анонімний відвідувач, що взаємодіє з платформою через вбудовуваний віджет), а також зовнішні системи, що інтегруються з платформою.

Функціональні вимоги, пов'язані з опрацюванням звернень:

- Опрацювання звернень користувачів агентом штучного інтелекту з автономним формуванням відповідей на основі бази знань організації.

- Виконання агентом прикладних дій через інструментальні виклики, зокрема ескалація звернення до оператора та позначення звернення як розв'язаного.
- Керування станом діалогу зі скінченною множиною станів (нерозв'язаний, ескальований, розв'язаний).
- Збереження історії повідомлень діалогу та контактних даних відвідувача.

Функціональні вимоги до каналів взаємодії:

- вбудовуваний чат-віджет для текстової взаємодії з кінцевим користувачем на сторонніх вебсайтах;
- голосовий канал взаємодії через інтеграцію із зовнішнім сервісом телефонії;
- налаштування параметрів віджета на рівні організації, зокрема вітального повідомлення та типових підказок.

Функціональні вимоги до операторської панелі:

- автентифікація операторів та розмежування доступу в межах організації;
- відображення списку звернень із можливістю фільтрування за станом та поступовим підвантаженням записів;
- перегляд історії повідомлень окремого звернення та участь оператора в діалозі;
- керування під'єднаними зовнішніми інтеграціями та їхніми обліковими даними;
- завантаження документів до бази знань для подальшого використання підсистемою генерації, доповненої пошуком.

Нефункціональні вимоги до платформи:

- багато орендарність: система має обслуговувати декілька організацій єдиним екземпляром із надійною ізоляцією даних кожного орендаря на всіх рівнях, що унеможливлює доступ однієї організації до даних іншої;

- масштабованість: архітектура має витримувати зростання кількості організацій, звернень та обсягу даних без потреби в суттєвій перебудові, зокрема за рахунок індексованого доступу до даних та безсерверної моделі виконання;
- продуктивність: інтерфейс операторської панелі має оновлюватися в режимі, близькому до реального часу, без ручного повторного опитування сервера, що забезпечується реактивною моделлю синхронізації стану;
- типова безпека: контракти взаємодії клієнта із сервером мають верифікуватися на етапі компіляції, що унеможливлює неузгодженість між клієнтським та серверним кодом;
- безпека даних: чутливі облікові дані зовнішніх інтеграцій мають зберігатися поза основним сховищем у спеціалізованому захищеному сервісі, а доступ до них — обмежуватися відповідно до принципу найменших привілеїв;
- зручність використання: інтерфейс має бути інтуїтивно зрозумілим, візуально узгодженим та адаптивним до різних класів пристроїв;
- супроводжуваність: кодова база має бути модульною, із повторним використанням компонентів та локальністю змін, за якої модифікація окремого функціонального домену не потребує втручання в суміжні

На основі сформульованих вимог та визначених у попередньому викладі акторів побудовано діаграму варіантів використання, що формалізує функціональні можливості системи з позиції її користувачів. Діаграму подано на рисунку 1.4.

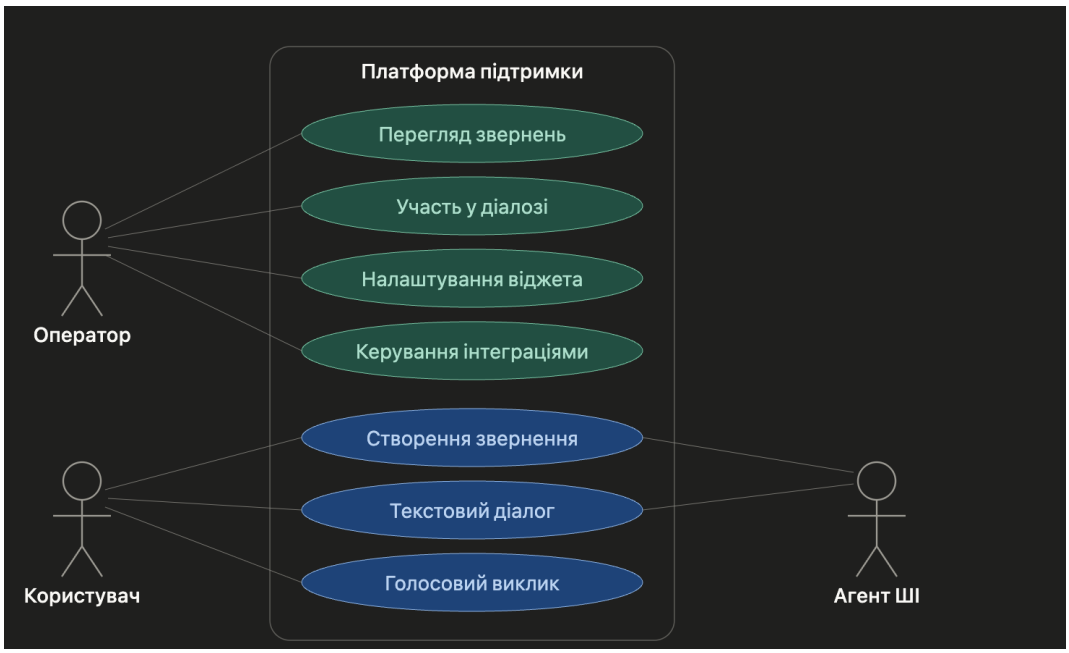

Рис. 1.4 – Діаграма варіантів використання платформи

Як видно з діаграми, оператор підтримки взаємодіє з прецедентами перегляду звернень, участі в діалозі, налаштування віджета та керування інтеграціями, що відповідають його ролі з адміністрування платформи та опрацювання звернень. Кінцевий користувач взаємодіє з прецедентами створення звернення, ведення текстового діалогу та голосового виклику. Зовнішній актор — агент штучного інтелекту — бере участь у прецедентах опрацювання звернень, автономно формуючи відповіді та керуючи станом діалогу.

Розмежування прецедентів між акторами відображає принципову архітектурну особливість платформи — розподіл відповідальності між автентифікованими та анонімними учасниками взаємодії. Прецеденти, доступні операторові, потребують попередньої автентифікації та розмежування доступу в межах організації, тоді як прецеденти кінцевого користувача виконуються в анонімному контексті контактної сесії без обов'язкової реєстрації. Участь агента штучного інтелекту як окремого актора підкреслює, що значну частину опрацювання звернень автоматизовано й винесено з-під безпосереднього контролю оператора, який залучається лише за потреби ескалації. Побудована діаграма таким чином не лише унаочнює функціональні можливості системи, а й попередньо

окреслює межі контурів безпеки та ступінь автоматизації, що деталізуються на етапі проєктування.

Таким чином, у цьому розділі проаналізовано предметну область автоматизації клієнтської підтримки, розглянуто наявні рішення та виявлено їхні обмеження, обґрунтовано вибір технологічного стека й методології розробки, а також сформульовано функціональні та нефункціональні вимоги до платформи, формалізовані у вигляді діаграми варіантів використання. Отримані результати визначають цільову поведінку та атрибути якості системи й слугують підґрунтям для проєктування її архітектури, моделі даних та програмних компонентів, що розглядаються в наступному розділі.

Окремої уваги в контексті аналізу предметної області заслуговує застосування великих мовних моделей як ядра інтелектуального опрацювання звернень. Сучасні мовні моделі, побудовані на архітектурі трансформер [23]ів, здатні сприймати запит користувача природною мовою та формувати зв'язну відповідь з урахуванням наданого контексту. Проте безпосереднє застосування мовної моделі без доступу до актуальних знань конкретної організації є недостатнім, оскільки модель оперує лише тими відомостями, на яких її було навчено, і не володіє специфічною інформацією про продукти чи послуги окремої компанії. Для подолання цього обмеження застосовують підхід генерації, доповненої пошуком, за якого модель отримує доступ до зовнішньої бази знань: релевантні фрагменти знаходяться шляхом семантичного пошуку та передаються моделі як контекст для формування відповіді. Саме цей підхід покладено в основу інтелектуальної підсистеми платформи, що розробляється.

Слід також зважати на те, що автоматизація клієнтської підтримки не передбачає цілковитого усунення людини з процесу. Частина звернень за своєю природою потребує участі оператора — через складність, нетиповість або потребу в ухваленні рішень, що виходять за межі компетенції автоматизованої системи. Тому ключовою вимогою до платформи є не лише якість автономного опрацювання звернень, а й наявність механізму плавної передачі складних випадків від інтелектуального агента до живого оператора.

2 РОЗРОБКА СИСТЕМНОЇ АРХІТЕКТУРИ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка системної архітектури та програмного забезпечення є одним із ключових етапів створення досліджуваної платформи автоматизації клієнтської підтримки. Цей процес охоплює визначення структурних компонентів системи, характеру їхньої взаємодії та функціональних можливостей, а також безпосередню реалізацію програмного коду, що втілює ці можливості. Архітектура програмної системи визначає її загальну структуру, спосіб організації компонентів, межі відповідальності між ними та принципи розподілу даних і обчислень, а отже, фактично зумовлює властивості, яких система набуває в процесі експлуатації. Обґрунтоване проєктування архітектури є визначальним для досягнення сформульованих у попередньому розділі вимог щодо масштабованості, надійності, продуктивності та супроводжуваності системи, яка поєднує декілька застосунків, спільні бібліотеки та серверну частину з інтегрованими компонентами штучного інтелекту. Складність такої системи, зумовлена множинністю застосунків, наявністю спільних бібліотек та інтеграцією зовнішніх сервісів, висуває підвищені вимоги до впорядкованості кодової бази та чіткості розмежування її складових.

На етапі проєктування архітектури визначено основні рівні системи, їхні складові компоненти й характер взаємодії між ними, а також обрано технології та інструменти реалізації: монорепозиторій на основі менеджера залежностей pnpm, фреймворк Next.js для клієнтської частини та безсерверну реактивну платформу Convex для серверної. Це дозволило сформувати чітку структуру кодової бази з розмежуванням відповідальності між компонентами, що слугує основою для подальшої розробки. Належне проєктування архітектури на ранньому етапі знижує ймовірність проблем на стадіях реалізації, тестування та впровадження, забезпечуючи гнучкість системи й здатність до адаптації за зміни вимог, зокрема в частині поведінки інтелектуального агента та сценаріїв опрацювання звернень. Зміни, локалізовані в межах окремого функціонального домену або рівня системи, не поширюються на суміжні складові, що спрощує супровід і розвиток платформи.

Розробка програмного забезпечення платформи охоплює проєктування моделі даних, реалізацію серверного програмного інтерфейсу з розмежуванням функцій за рівнем довіри, побудову інтерфейсу користувача за компонентним підходом та налаштування клієнт-серверної взаємодії на основі реактивної синхронізації стану. Кожен із цих аспектів підпорядкований сформульованим вимогам: модель даних проєктується з урахуванням ізоляції орендарів та шаблонів доступу до записів, серверний інтерфейс — з розмежуванням контурів безпеки за джерелом виклику, а клієнтська частина — за принципом композиції з повторно використовуваних компонентів. Окрему увагу приділено забезпеченню наскрізної типової безпеки — узгодженості контрактів між клієнтом і сервером на етапі компіляції, — а також ізоляції даних організацій-орендарів у багатоорендарній моделі, що унеможливлює доступ однієї організації до даних іншої. Застосування статичної типізації, системи інкрементального збирання та засобів статичного аналізу коду дозволяє підвищити якість програмного забезпечення, зменшити кількість помилок, виявляючи їх на етапі компіляції, та скоротити цикли розробки за рахунок повторного складання лише змінених частин кодової бази.

Реалізація платформи відповідного масштабу потребує узгодженого застосування обраних технологій, дотримання технічних стандартів та методологій розробки з урахуванням потреб кінцевих користувачів і операторів підтримки. Окрім суто технічних аспектів, важливим є врахування зручності використання інтерфейсу, узгодженості візуальної ідентичності та адаптивності системи до різних класів пристроїв, оскільки саме ці характеристики визначають придатність платформи до практичного застосування.

2.1 Вибір архітектури проєкту

Архітектура програмної системи визначає її загальну структуру, спосіб організації компонентів, характер взаємодії між ними та принципи розподілу відповідальності. Обґрунтований вибір архітектури є визначальним для досягнення сформульованих у попередньому розділі нефункціональних вимог — масштабованості, супроводжуваності та надійності. З огляду на складність платформи, що поєднує декілька застосунків, спільні бібліотеки та серверну частину з інтегрованими компонентами штучного інтелекту, до організації її кодової бази й розподілу компонентів висуваються підвищені вимоги.

Однією з основних архітектурних альтернатив є вибір між монолітною та модульною організацією кодової бази. У монолітній архітектурі всі компоненти системи об'єднані в єдину нероздільну кодову базу. Такий підхід спрощує початкову розробку та розгортання, проте зі зростанням системи ускладнює її супровід: кодова база стає важкою для розуміння, будь-яка зміна потенційно впливає на всю систему, а паралельна робота над різними частинами ускладнюється. Протилежним підходом є поділ системи на множину незалежних розгортуваних сервісів (мікросервісів), що забезпечує гнучкість масштабування окремих компонентів, проте суттєво підвищує складність розгортання, потребує організації міжсервісної комунікації та керування розподіленими станами, що є невиправданим для проєкту відповідного масштабу.

Для реалізації платформи обрано проміжний підхід — організацію кодової бази у форматі монорепозиторію (англ. monorepository). Монорепозиторій є єдиним репозиторієм, у межах якого співіснують декілька логічно відокремлених, проте технологічно взаємопов'язаних застосунків та бібліотек спільного використання. Такий підхід поєднує переваги єдиної кодової бази — спільне версіонування, атомарність змін, що зачіпають декілька компонентів, узгодженість конфігурацій — із чітким логічним розмежуванням складових, характерним для модульних архітектур.

Вибір монорепозиторію для досліджуваної платформи зумовлений безпосередньо її структурою та сформульованими в попередньому розділі вимогами. По-перше, платформа складається з декількох застосунків — адміністративної панелі, віджета чату та модуля його вбудовування, — які поділяють спільний код: елементи дизайн-системи, типи даних і клієнт серверного інтерфейсу. За роздільного зберігання цих застосунків у окремих репозиторіях спільний код довелося б публікувати як зовнішні пакунки та синхронізувати їхні версії вручну, що ускладнило б розробку та спричинило б ризик розходження версій. Монорепозиторій усуває цю проблему, дозволяючи застосункам безпосередньо споживати спільні бібліотеки в межах єдиного репозиторію. По-друге, вимога наскрізної типової безпеки потребує, щоб зміна типів на серверному рівні негайно відображалася в усіх застосунках-споживачах; спільне версіонування в межах монорепозиторію забезпечує атомарність таких змін, за якої модифікація контракту та її використання потрапляють до системи контролю версій єдиною узгодженою зміною. По-третє, з огляду на одноосібну розробку проєкту, монорепозиторій знижує організаційні накладні витрати, оскільки усуває потребу в керуванні декількома репозиторіями, окремими процесами публікації пакунків та узгодженні їхніх версій. Натомість мікросервісна організація, що передбачала б поділ на незалежно розгортувані сервіси з міжсервісною комунікацією, була б невиправданою: вона підвищила б складність розгортання й експлуатації без відповідного виграшу, оскільки масштаб проєкту не потребує незалежного масштабування окремих компонентів.

В основу платформи покладено клієнт-серверну архітектурну модель, за якої систему розподілено на клієнтську частину, що відповідає за взаємодію з користувачем, та серверну частину, що реалізує зберігання даних і прикладну логіку. На відміну від класичної клієнт-серверної моделі із запитово-відповідальним характером взаємодії, у досліджуваній платформі застосовано її реактивний різновид: серверна частина не лише відповідає на запити клієнта, а й автоматично сповіщає підписані клієнти про зміну стану, що забезпечує синхронізацію інтерфейсу в режимі, близькому до реального часу. Клієнтський

рівень реалізовано засобами фреймворку Next.js у межах монорепозиторію, а серверний — на основі безсерверної реактивної платформи Convex. Загальну структуру модулів платформи та їхній розподіл за рівнями архітектури подано на рисунку 2.1.

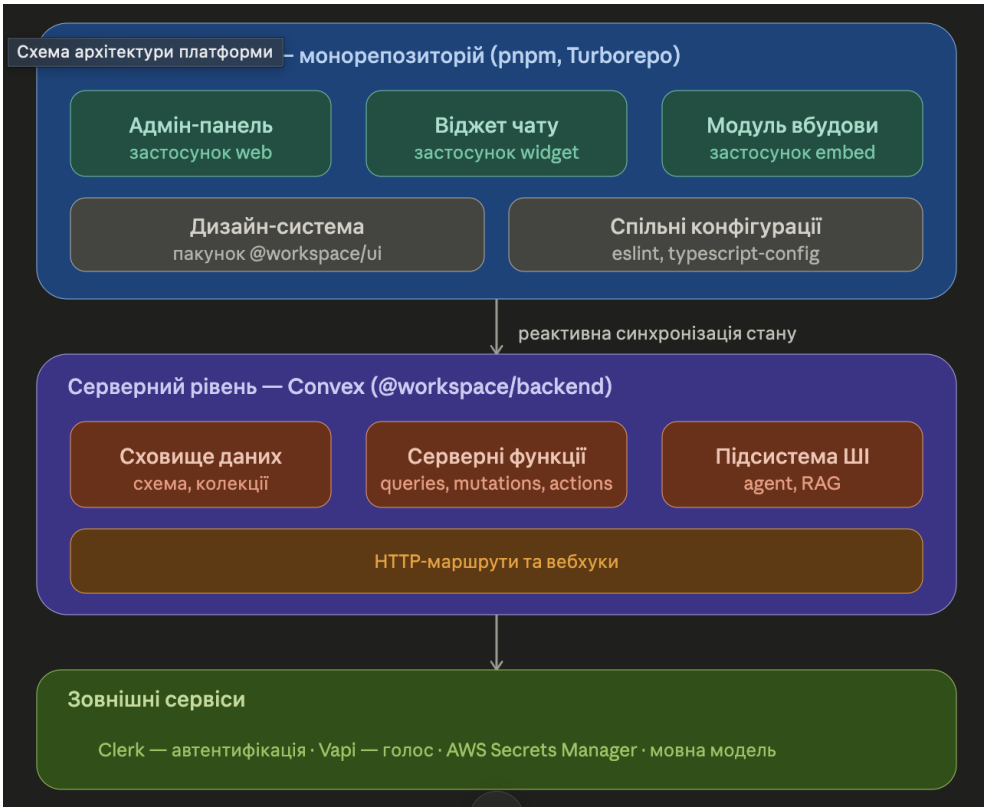

Рис. 2.1 – структурна схема архітектури платформи

Архітектуру платформи організовано у вигляді трьох логічних рівнів, розмежованих за характером відповідальності: клієнтського рівня, серверного рівня та рівня зовнішніх сервісів. Розглянемо кожен із них детально.

Клієнтський рівень відповідає за формування інтерфейсу взаємодії з користувачем та виконується у вебоглядачі. Його реалізовано у межах монорепозиторію засобами фреймворку Next.js і поділено на три самостійні застосунки. Адміністративна панель є основним робочим середовищем оператора підтримки, що надає засоби перегляду звернень, участі в діалогах та налаштування платформи. Віджет чату є вбудовуваним інтерфейсом текстової взаємодії, призначеним для розміщення на сторонніх вебсайтах. Модуль вбудовування

забезпечує інтеграцію віджета у сторінку-носій. Спільний для застосунків код — елементи дизайн-системи та централізовані конфігурації — винесено в окремі пакунки спільного використання, що усуває його дублювання між застосунками.

Серверний рівень реалізує зберігання даних, прикладну логіку, авторизацію та оркестрацію інтелектуального опрацювання звернень. Його побудовано на основі безсерверної реактивної платформи Convex та оформлено як окремий пакунок монорепозиторію, що споживається клієнтськими застосунками. Цей рівень охоплює документоорієнтоване сховище даних із декларативно визначеною схемою, серверні функції, стратифіковані за рівнем довіри, інтелектуальну підсистему опрацювання звернень на основі агента штучного інтелекту з генерацією, доповненою пошуком, а також HTTP-маршрути та вебхуки для приймання подій від зовнішніх систем. Реактивна природа рівня забезпечує автоматичне поширення змін стану на підписані клієнти.

Рівень зовнішніх сервісів об'єднує спеціалізовані сторонні системи, інтегровані із серверним рівнем для делегування функцій, що недоцільно реалізовувати власноруч. Автентифікацію операторів та керування обліковими записами делеговано сервісу Clerk. Голосовий канал взаємодії реалізовано через інтеграцію із сервісом телефонії Vapi. Безпечне зберігання облікових даних зовнішніх інтеграцій покладено на спеціалізований сервіс керування секретами AWS Secrets Manager [18]. Генерацію відповідей агентом штучного інтелекту забезпечує зовнішня велика мовна модель. Винесення цих функцій до зовнішніх сервісів дозволяє зосередити розробку на прикладній логіці платформи та підвищує надійність за рахунок використання спеціалізованих рішень.

Запропонований поділ на рівні з чітко визначеними межами відповідальності забезпечує модульність системи та слабке зчеплення між її складовими: кожен рівень взаємодіє із суміжними через визначені інтерфейси, а внутрішні зміни в межах рівня не поширюються на інші. Це безпосередньо задовольняє сформульовані в попередньому розділі вимоги масштабованості та супроводжуваності.

Засобом організації монорепозиторію обрано механізм робочих просторів менеджера залежностей pnpm, що декларується в кореневому конфігураційному файлі. Простір пакунків поділено двома шаблонами включення на дві категорії: каталог застосунків та каталог пакунків спільного використання. Такий поділ формалізує розмежування між кінцевими застосунками, що споживаються користувачами, та повторно використовуваними модулями.

Категорія застосунків об'єднує три самостійні застосунки: основну адміністративну панель оператора, вбудовуваний віджет чату та модуль вбудовування віджета на сторонні вебсайти. Категорія пакунків спільного використання концентрує наскрізні артефакти: дизайн-систему, серверний шар та централізовані конфігурації статичного аналізу й компілятора. Внутрішньопроєктні залежності розв'язуються через спеціальний протокол робочих просторів, який зв'язує пакунки локального простору безпосередньо, оминаючи зовнішній реєстр пакунків. Завдяки цьому адміністративна панель декларативно споживає спільні бібліотеки, утворюючи орієнтований ациклічний граф залежностей між компонентами платформи.

Вибір pnpm як інструментального ядра монорепозиторію зумовлений його моделлю адресації залежностей на основі контентно-адресованого сховища, що усуває дублювання ідентичних версій пакунків на файловій системі та забезпечує детерміновану, строго ізольовану структуру залежностей. На відміну від класичних менеджерів залежностей, pnpm унеможливлює неявний доступ до недекларованих транзитивних залежностей, чим підвищує відтворюваність складання та достовірність графа залежностей.

Для оркестрації завдань складання та перевірки коду застосовано систему інкрементального збирання Turborepo, що дозволяє виконувати завдання в межах монорепозиторію узгоджено та з кешуванням проміжних результатів. Детальний розгляд цього інструментарію наведено в підрозділі 2.3.

Загальна архітектура системи, подана на рисунку 2.1, поділяється на два основні рівні: клієнтський, реалізований засобами фреймворку Next.js, та серверний, реалізований на основі безсерверної реактивної платформи Convex.

Взаємодію між ними побудовано за реактивною моделлю синхронізації стану, що детально розглядається в подальших підрозділах.

2.2 Проєктування системи з використанням UML

Уніфікована мова моделювання (англ. Unified Modeling Language, UML) є стандартизованим засобом графічного подання структури та поведінки програмних систем. Застосування UML-діаграм на етапі проєктування дозволяє формалізувати вимоги, унаочнити взаємодію компонентів та узгодити проєктні рішення до початку реалізації. Для опису досліджуваної платформи побудовано діаграму класів клієнтської частини, що подає структурну організацію компонентів, та діаграму послідовності, що ілюструє динаміку опрацювання звернення.

Розглянемо структурну організацію клієнтської частини, подану у вигляді діаграми класів. Оскільки клієнтську частину реалізовано за компонентним підходом на основі функційних компонентів React, діаграма класів відображає компоненти, їхні контракти вхідних параметрів та відношення композиції між ними. Діаграму класів презентаційного модуля подано на рисунку 2.3.

На діаграмі відображено кореневий компонент-контейнер, що компонує підпорядковані презентаційні компоненти, та джерело даних, з якого компоненти отримують вміст для відображення. Відношення композиції вказують на те, що дочірні компоненти існують лише в межах контейнера, а напрям залежностей відображає однобічний потік даних від джерела до компонентів представлення.

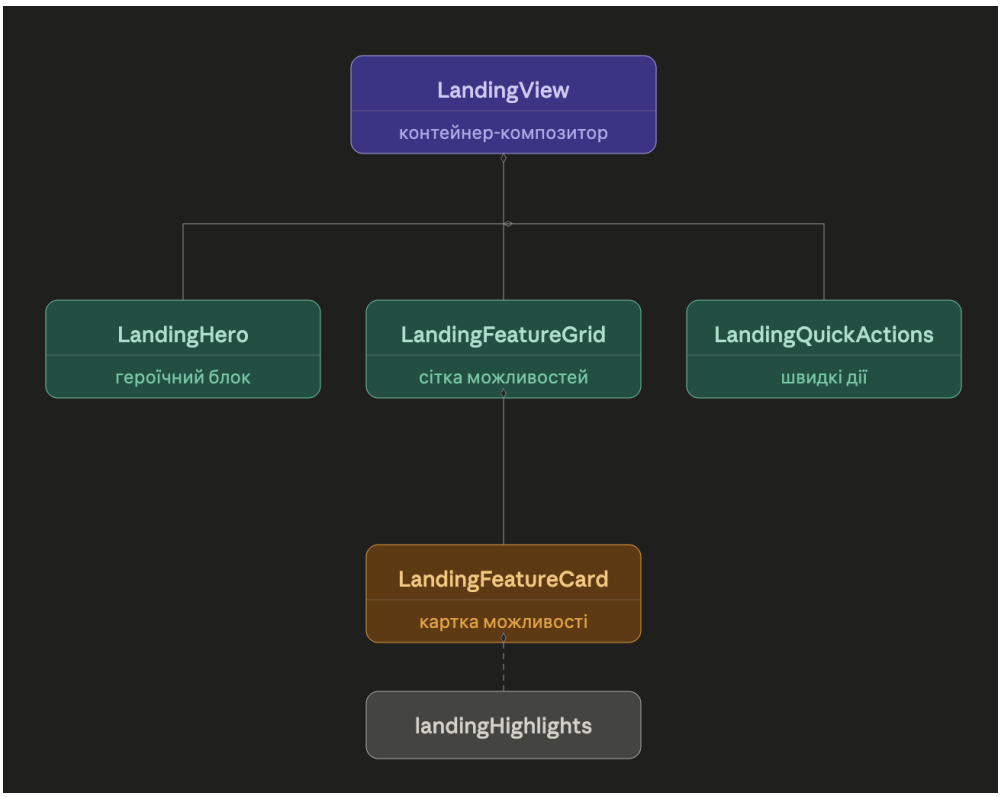

Рис. 2.3 – Діаграма класів презентаційного модуля

На діаграмі класів відображено структурну організацію презентаційного модуля landing. Контейнер-композитор LandingView перебуває у відношенні композиції з трьома підпорядкованими компонентами — LandingHero, LandingFeatureGrid та LandingQuickActions, — компонуючи їх у детермінованому порядку. Компонент LandingFeatureGrid, своєю чергою, перебуває у відношенні композиції з компонентом LandingFeatureCard, делегуючи йому рендеринг кожного окремого запису, та споживає декларативне джерело даних landingHighlights. Кожен компонент є функційним компонентом React, а його контракт вхідних параметрів формалізовано окремим типом із суфіксом Attributes (наприклад, LandingFeatureCardAttributes), що верифікується компілятором TypeScript. Така структура відображає архітектурну засаду декомпозиції за принципом композиції, за якої складна сторінка зводиться до ієрархії дрібнозернистих автономних компонентів.

Динаміку взаємодії компонентів системи під час опрацювання звернення подано на діаграмі послідовності (рисунок 2.4). Діаграма ілюструє ключовий

сценарій: надходження повідомлення від кінцевого користувача через віджет та його автономне опрацювання інтелектуальним агентом.

Процес опрацювання звернення охоплює таку послідовність взаємодій. Кінцевий користувач надсилає повідомлення через віджет чату, що ініціює виклик відповідної серверної дії. Серверна дія виконує валідацію контактної сесії та перевірку чинності підписки організації, після чого, за умови успішної перевірки, формує запит на генерацію відповіді до інтелектуального агента. Агент виконує семантичний пошук релевантних фрагментів у базі знань організації засобами підсистеми генерації, доповненої пошуком, та на основі знайденого контексту формує відповідь природною мовою. Сформована відповідь повертається серверній дії, яка зберігає її у сховищі та завдяки реактивній моделі синхронізації автоматично доставляє оновлення віджету користувача. Така послідовність відображає основний потік автономного опрацювання звернення, за якого відповідь формується без участі оператора.

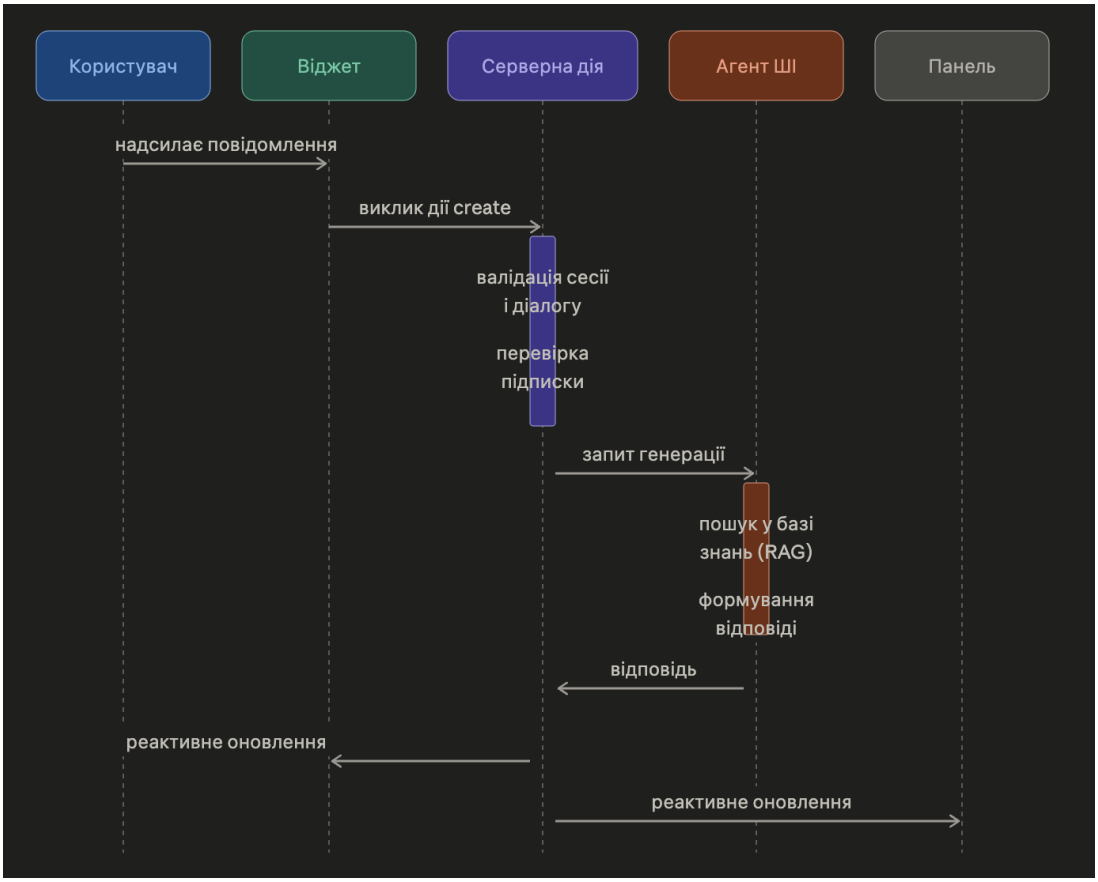

Рис. 2.4 – Діаграма послідовності опрацювання звернення

Відповідно до діаграми послідовності, кінцевий користувач надсилає повідомлення через віджет чату, що ініціює виклик серверної дії опрацювання повідомлення. Серверна дія виконує валідацію контактної сесії та діалогу, перевіряє стан підписки організації і, за умови активного діалогу та чинної підписки, ініціює генерацію відповіді мовною моделлю з підключеним інструментарієм. Агент штучного інтелекту формує відповідь на основі бази знань організації, після чого результат зберігається та реактивно доставляється як клієнту віджета, так і операторській панелі. У разі потреби агент виконує інструментальний виклик ескалації, що переводить діалог у стан очікування втручання оператора.

Побудовані UML-діаграми у сукупності формалізують функціональні вимоги до системи, її структурну організацію та динаміку взаємодії компонентів, що слугує підґрунтям для безпосередньої реалізації, розглянутої в подальших підрозділах.

2.3 Конструювання та інжиніринг проєкту

Конструювання платформи здійснюється в межах монорепозиторію із застосуванням сукупності інженерних практик, що забезпечують відтворюваність та керованість процесу: керування залежностями, інкрементального складання, статичного аналізу коду й контролю версій. Цей підрозділ розглядає інженерний інструментарій, застосований під час конструювання платформи.Керування залежностями та зв'язування пакунків монорепозиторію покладено на менеджер pnpm, який через протокол робочих просторів дозволяє застосункам безпосередньо споживати спільні бібліотеки. Версію менеджера зафіксовано директивою packageManager у кореневому маніфесті, що гарантує відтворюваність складання в усіх середовищах розробки та неперервної інтеграції.Оркестрацію інженерних завдань — складання, перевірки типів та статичного аналізу — делеговано системі інкрементального збирання Turborepo. Її конфігурація описує граф залежностей між завданнями, задаючи топологічно впорядковану послідовність їх виконання, та активує кешування результатів на основі хешування вхідних даних. Унаслідок

цього під час повторного складання перевиконується лише мінімально необхідна підмножина завдань, вхідні дані яких змінилися, що суттєво скорочує час циклів конструювання.Забезпечення якості коду інтегровано безпосередньо в етап конструювання. Статичний аналіз засобами ESLint [10] та перевірку типів засобами компілятора TypeScript налаштовано через окремі пакунки спільного використання (@workspace/eslint-config та @workspace/typescript-config), що забезпечує єдиний набір правил та параметрів компіляції для всіх складових монорепозиторію та дозволяє виявляти значну частину дефектів до запуску застосунку. Детальний розгляд процедур тестування та верифікації наведено в третьому розділі.

Для контролю версій платформи застосовано систему Git із розміщенням віддаленого репозиторію на сервісі GitHub. Зміни групувалися в коміти за принципом логічної завершеності: кожен коміт відповідає окремій смисловій зміні — додаванню функціональної можливості, виправленню помилки або рефакторингу, — що забезпечує зрозумілість історії та можливість точкового повернення до конкретного стану. Фрагмент історії комітів репозиторію наведено на рисунку 2.5.

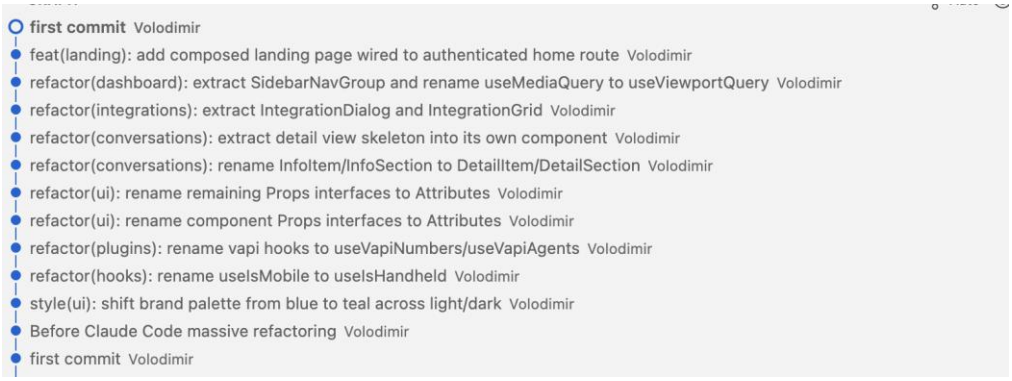

Рис. 2.5 – Фрагмент дерева комітів

З огляду на одноосібну розробку застосовано полегшену стратегію гілкування: основну розробку зосереджено в головній гілці, що завжди відповідає працездатному стану, а для відносно відокремлених функціональних можливостей за потреби створювалися окремі тематичні гілки з подальшим злиттям після завершення.

Окремої уваги потребувало налаштування складу версіонованих файлів. З-під контролю версій свідомо виключено каталоги залежностей, артефакти складання, кеш системи інкрементального збирання та файли змінних середовища. Виключення файлів середовища має безпосереднє значення для безпеки, оскільки саме вони містять ключі доступу до зовнішніх сервісів автентифікації, телефонії та мовної моделі; ці дані відокремлено від кодової бази та передають застосунку через механізм змінних середовища під час розгортання. Аналогічно з версіонування виключено згенеровані платформою Convex типи, оскільки вони відтворюються автоматично з первинних джерел, і їхнє зберігання спричиняло б надлишковість та конфлікти під час злиття змін.

Застосування системи Git у поєднанні з GitHub та сукупності інженерних практик — фіксації версії менеджера залежностей, інкрементального складання з кешуванням, централізованого статичного аналізу й перевірки типів та свідомого керування складом версіонованих файлів — забезпечило керованість, відтворюваність і безпеку процесу конструювання платформи

2.4 Реалізація проєкту

Реалізація проєкту полягає у втіленні спроєктованої архітектури в програмний код, що реалізує сформульовані вимоги. Цей підрозділ розглядає ключові аспекти реалізації: оглядово — серверну частину на основі реактивної платформи Convex, детально — структурну організацію клієнтського застосунку засобами фреймворку Next.js та реалізацію презентаційного модуля landing за компонентним підходом.

Серверні функції стратифіковано за рівнем довіри на три категорії. Публічні функції доступні для виклику з клієнтських застосунків і становлять зовнішню межу програмного інтерфейсу. Внутрішні функції доступні лише для виклику з інших серверних функцій та не експонуються клієнтові, що дозволяє інкапсулювати чутливу логіку. Системні функції призначені для виконання

фонових та службових операцій. Така стратифікація формує чіткі контури безпеки, унеможливлюючи безпосередній виклик привілейованої логіки з боку клієнта.

За характером операцій функції поділяються на запити, мутації та дії. Запити є реактивними проєкціями стану сховища: вони не змінюють даних, а їхній результат автоматично переобчислюється та доставляється клієнтові за будь-якої зміни залежних даних. Мутації виконують транзакційну зміну стану сховища. Дії призначені для взаємодії із зовнішніми сервісами та виконання операцій із побічними ефектами, зокрема звернень до мовної моделі.

Інтелектуальну підсистему реалізовано засобами спеціалізованого компонента @convex-dev/agent, що інкапсулює взаємодію з мовною моделлю та керування діалоговими потоками, у поєднанні з компонентом @convex-dev/rag для генерації, доповненої пошуком. Доступ до моделей здійснюється через пакет @ai-sdk/openai. Текстовий агент підтримки використовує модель OpenAI gpt-4o-mini, обрану як компроміс між якістю генерації та вартістю опрацювання звернень. Підсистему генерації, доповненої пошуком, побудовано на основі моделі векторних вкладень text-embedding-3-small з розмірністю простору 1536, що перетворює фрагменти бази знань організації на числові вектори для подальшого семантичного пошуку. Опрацювання звернення передбачає семантичний пошук релевантних фрагментів за векторною подібністю з подальшим формуванням відповіді мовною моделлю на основі знайденого контексту. Агентові надано інструментальні виклики, зокрема пошук у базі знань та ескалацію звернення до оператора.

Окремо реалізовано підсистему видобування тексту із завантажених документів, що використовує мультимодальні можливості моделей: зображення опрацьовуються моделлю gpt-4o-mini, а документи у форматах PDF та HTML — потужнішою моделлю gpt-4o. Саме поєднання текстового та голосового каналів взаємодії з опрацюванням різнотипного контенту (текст, зображення, документи, голос) визначає мультимодальний характер платформи.

Інтеграцію із зовнішніми системами реалізовано через HTTP-маршрути та механізм вебхуків, що дозволяють зовнішнім сервісам — провайдеру автентифікації, сервісу телефонії — повідомляти платформу про відповідні події.

Центральний застосунок web реалізовано засобами фреймворку Next.js версії 15 на основі архітектурної парадигми App Router із застосуванням React 19 та серверних компонентів. Принциповою рисою організації коду є модульно-доменний поділ: уся прикладна логіка згрупована в каталозі modules/, де кожен підкаталог інкапсулює окремий функціональний домен предметної області — auth, billing, conversations, dashboard, files, integrations, landing, plugins, widget-customization. Усередині кожного домену застосовано уніфіковану стратифікацію за схемою ui/views (компоненти-контейнери, що компонують підпорядковані елементи) та ui/components (атомарні презентаційні компоненти), а доменоспецифічні константи й типи виносяться в окремі файли модуля. Така стратифікація реалізує принцип єдиної відповідальності на рівні архітектури каталогів та забезпечує локальність змін.

Маршрутизація відображається на файлову систему каталогу app/, де групи маршрутів, відмежовані синтаксисом дужок, дозволяють об'єднувати сторінки під спільним макетом без впливу на структуру URL-адрес. Файли сторінок зведено до тонких точок входу, що делегують усю композицію відповідним представленням домену. Розв'язання шляхів імпорту впорядковано через псевдоніми компілятора TypeScript (@/*, @workspace/ui/*, @workspace/backend/*), задекларовані у tsconfig.json, чим усунено крихкі відносні шляхи.

Презентаційний модуль landing реалізовано як композовану цільову сторінку автентифікованого користувача за принципом декомпозиції на ієрархію дрібнозернистих функційно автономних компонентів. Модуль репрезентовано кореневим представленням landing-view.tsx, набором презентаційних компонентів та декларативним джерелом даних constants.ts. Жоден компонент модуля не містить вбудованого стану чи побічних ефектів — усі вони є чистими функціями відображення, що уможливлює їх рендеринг на серверному боці.

Контейнер-композитор LandingView виконує функцію представлення верхнього рівня, формуючи обмежувальний макетний контекст та компонуючи три підпорядковані секції в детермінованому порядку (рисунок 2.6).

```
export const LandingView = () => {
  return (
    <div className='mx-auto flex w-full max-w-screen-lg flex-col gap-8 px-4 py-6'>
      <LandingHero />
      <LandingFeatureGrid />
      <LandingQuickActions />
    </div>
  );
};
```

Рис. 2.6 – Код контейнера-композитора LandingView

Делегування макетної відповідальності єдиному контейнеру дозволяє підлеглим компонентам залишатися індиферентними щодо власного позиціонування, що є виявом інверсії компонування.

Компонент героїчного блоку LandingHero реалізує первинний акцентний блок сторінки, що формує ціннісну пропозицію продукту. Кнопки заклику до дії реалізовано через патерн поліморфного рендерингу за допомогою властивості asChild, (рисунок 2.7).

```
<div className='mt-6 flex flex-wrap items-center gap-3'>
  <Button asChild>
    <Link href='/conversations'>
      Open Conversations
      <ArrowRightIcon className='size-4' />
    </Link>
  </Button>
  <Button asChild variant='outline'>
    <Link href='/customization'>Customize widget</Link>
  </Button>
</div>
```

Рис. 2.7 – Реалізація патерну поліморфного рендерингу в компоненті LandingHero

Пара компонентів LandingFeatureGrid та LandingFeatureCard втілює патерн «список — елемент», що розмежовує логіку ітерування від логіки відображення окремого запису. Компонент LandingFeatureGrid виступає координатором адаптивної сітки (від одноколонкового компонування на мобільних пристроях до чотириколонкового на широких екранах через утилітарні класи sm:grid-cols-2 lg:grid-cols-4) та декларативно відображає колекцію landingHighlights на множину дочірніх карток (рисунок 2.8).

```
export const LandingFeatureGrid = () => {
  return (
    <section className='grid gap-4 sm:grid-cols-2 lg:grid-cols-4'>
      {landingHighlights.map((highlight) => (
        <LandingFeatureCard key={highlight.title} highlight={highlight} />
      ))}
    </section>
  );
};
```

Рис. 2.8 – Код компонента адаптивної сітки LandingFeatureGrid

Компонент LandingFeatureCard є чисто презентаційною одиницею, що приймає єдиний типізований об'єкт highlight та деструктурує з нього піктограму, заголовок і опис. Контракт вхідних параметрів формалізовано окремим типом LandingFeatureCardAttributes згідно з прийнятою конвенцією іменування, що верифікується компілятором TypeScript (рисунок 2.9).

```tsx
type LandingFeatureCardAttributes = {
  highlight: LandingHighlight;
};

export const LandingFeatureCard = ({
  highlight
}: LandingFeatureCardAttributes) => {
  const { icon: Icon, title, description } = highlight;

  return (
    <Card className='h-full'>
      <CardHeader>
        <span className='bg-primary/10 text-primary flex size-9 items-center justify
          <Icon className='size-5' />
        </span>
        <CardTitle className='mt-3 text-base'>{title}</CardTitle>
        <CardDescription>{description}</CardDescription>
      </CardHeader>
    </Card>
  );
};
```

Рис. 2.9 – Код презентаційного компонента LandingFeatureCard

Дані для сітки централізовано винесено в модуль constants.ts у вигляді строго типізованого масиву landingHighlights, кожен запис якого описує ціннісну тезу платформи. Відокремлення даних від рендерингу реалізує принцип конфігурованості представлення та дозволяє модифікувати наповнення сторінки без втручання в код компонентів (рисунок 2.10).

```tsx
export type LandingHighlight = {
  title: string;
  description: string;
  icon: LucideIcon;
};

export const landingHighlights: LandingHighlight[] = [
  {
    title: 'AI-assisted replies',
    description:
      'Draft, refine and enhance operator responses with a single click so no custome
    icon: BotIcon
  },
```

Рис. 2.10 – Фрагмент декларативного джерела даних constants.ts

Компонент швидких дій LandingQuickActions забезпечує навігаційний рівень сторінки, відображаючи колекцію landingShortcuts як множину карток-ярликів, кожна з яких є гіперпосиланням на функціональний маршрут панелі керування. Інтерактивний зворотний зв'язок реалізовано через групові стани наведення (group та group-hover), що синхронно змінюють колір рамки картки й піктограми-стрілки (рисунок 2.11)

```
{landingShortcuts.map((shortcut) => {
  const { icon: Icon, title, description, href } = shortcut;

  return (
    <Link key={href} href={href} className='group'>
      <Card className='hover:border-primary/50 h-full transition-colors'>
        <CardHeader>
          <div className='flex items-center justify-between'>
            <span className='bg-muted text-foreground flex size-9 items-center just
              <Icon className='size-5' />
            </span>
            <ChevronRightIcon className='text-muted-foreground group-hover:text-pri
          </div>
          <CardTitle className='mt-3 text-base'>{title}</CardTitle>
          <CardDescription>{description}</CardDescription>
        </CardHeader>
      </Card>
    </Link>
  );
})}
```

Рис. 2.11 – Реалізація групових станів наведення в компоненті LandingQuickActions

Таким чином, презентаційний модуль landing реалізовано як ієрархію автономних компонентів із чітким розмежуванням відповідальностей: контейнер відповідає за композицію, презентаційні компоненти — за відображення, а декларативні джерела даних — за конфігурацію наповнення. Така організація забезпечує повторне використання компонентів, локальність змін та можливість незалежного тестування, що відповідає сформульованим вимогам до супроводжуваності клієнтської частини.

Описаний на прикладі презентаційного модуля підхід є наскрізним для всієї клієнтської частини платформи й послідовно застосовується в усіх функціональних доменах застосунку. Кожен домен — від керування зверненнями до налаштування інтеграцій — побудовано за єдиною структурною схемою, у якій представлення

верхнього рівня компонує підпорядковані презентаційні компоненти, а доменоспецифічні дані, типи й константи виносяться в окремі модулі. Складніші домени, на відміну від презентаційного, доповнюються інтерактивними компонентами, що взаємодіють із серверною частиною через реактивні запити та мутації, проте навіть у них збережено принципове розмежування між компонентами, відповідальними за відображення, та логікою отримання й зміни даних. Уніфікованість цієї структури в межах усього застосунку має практичну цінність: розробник, ознайомившись з організацією одного домену, легко орієнтується в будь-якому іншому, що знижує когнітивне навантаження та пришвидшує внесення змін. Типобезпека, забезпечена наскрізним застосуванням TypeScript та автоматично згенерованими типами серверного шару, поширюється на всі рівні компонентної ієрархії, унеможливлюючи передавання некоректних даних між компонентами на етапі компіляції. Водночас у процесі розвитку кодової бази окремі компоненти неминуче набували надмірної складності або містили дубльовану логіку, що потребувало цілеспрямованого вдосконалення їхньої структури. Відповідні заходи з оптимізації та рефакторингу, спрямовані на усунення таких недоліків та підвищення якості кодової бази, розглянуто в наступному підрозділі.

2.5 Оптимізація та рефакторинг кодової бази

Поряд із реалізацією нової функціональності у процесі розробки проведено оптимізацію та рефакторинг наявної кодової бази, спрямовані на підвищення її супроводжуваності, усунення дублювання коду та забезпечення термінологічної узгодженості. Рефакторинг охопив чотири основні напрями: уніфікацію логіки реактивних запитів до області перегляду, впровадження єдиної конвенції типізації інтерфейсів компонентів, декомпозицію монолітних компонентів та уніфікацію візуальної ідентичності дизайн-системи.

Складовою рефакторингу стало впорядкування логіки реагування на зміни геометрії області перегляду (англ. viewport). Розпорошену в компонентах логіку

визначення медіа-умов консолідовано у спеціалізовані користувацькі хуки (англ. custom hooks) із семантично виразною номенклатурою: хук визначення медіа-умови отримав назву useViewportQuery, а хук визначення класу пристрою — useIsHandheld, що точніше відображає їхню семантику в термінах класів пристроїв, а не лише ширини екрана.

Хук useViewportQuery інкапсулює підписку на об'єкт MediaQueryList, отримуваний через браузерний інтерфейс window.matchMedia, та керує життєвим циклом слухача подій засобами ефекту useEffect. Принципово важливим є коректне прибирання підписки у функції очищення ефекту, що унеможливлює витік пам'яті при демонтажі компонента (рисунок 2.12).

Хук useIsHandheld побудовано як надбудову над механізмом медіа-запитів, що реалізує реактивний предикат належності пристрою до класу портативних. Граничне значення ширини екрана, за яким пристрій класифікується як портативний, винесено в окрему іменовану константу MOBILE_BREAKPOINT.

```
export function useViewportQuery() {
  const [isOpen, setIsOpen] = useState(false);

  useEffect(() => {
    const mediaQuery = window.matchMedia('(max-width: 768px)');
    setIsOpen(mediaQuery.matches);

    const handler = (e: MediaQueryListEvent) => {
      setIsOpen(e.matches);
    };

    mediaQuery.addEventListener('change', handler);
    return () => mediaQuery.removeEventListener('change', handler);
  }, []);

  return { isOpen };
}
```

Рис. 2.12 – Код користувацького хука useViewportQuery

Хук useIsHandheld реалізує реактивний предикат належності пристрою до класу портативних на основі граничного значення MOBILE_BREAKPOINT,

винесеного в окрему константу, що централізує точку модифікації порогового значення адаптивності. Винесення цієї логіки в окремі повторно використовувані абстракції усуває її дублювання в численних компонентах.

З метою забезпечення термінологічної узгодженості впроваджено єдину конвенцію іменування контрактів вхідних параметрів компонентів. Типи, що описують властивості компонентів, послідовно іменуються із застосуванням суфікса Attributes (наприклад, LandingFeatureCardAttributes, SidebarNavGroupAttributes). Така конвенція встановлює однозначну, наскрізно узгоджену таксономію типів у межах усієї кодової бази, що покращує читабельність та полегшує статичний пошук контрактів компонентів. Патерн реалізовано через локальні типові аліаси на основі літералів об'єктних типів, що деструктуруються безпосередньо в сигнатурі компонента, забезпечуючи строгу статичну верифікацію переданих параметрів засобами компілятора TypeScript.

Окремим напрямом оптимізації стала структурна декомпозиція надмірно об'ємних компонентів за принципом виокремлення відповідальностей. Зокрема, з компонента бічної навігаційної панелі AppSidebar виокремлено повторно використовуваний компонент SidebarNavGroup, що усунув трикратне дублювання ідентичних блоків навігації та інкапсулював логіку відображення групи навігаційних записів із підсвічуванням активного пункту. Внаслідок цього компонент AppSidebar замість трьох розгорнутих однотипних блоків навігації містить три декларативні виклики компонента SidebarNavGroup із відповідними параметрами (рисунок 2.13).

```
<SidebarContent className='overflow-x-hidden'>
  <SidebarNavGroup
    heading='Customer Support'
    entries={customerSupportNavItems}
    currentPath={pathname}
  />
  <SidebarNavGroup
    heading='Configuration'
    entries={configurationNavItems}
    currentPath={pathname}
  />
  <SidebarNavGroup
    heading='Account'
    entries={accountNavItems}
    currentPath={pathname}
  />
</SidebarContent>
```

Рис. 2.13 – Використання виокремленого компонента SidebarNavGroup

Сам виокремлений компонент є чистою презентаційною одиницею, що приймає типізований контракт SidebarNavGroupAttributes та відображає групу навігаційних записів, підсвічуючи активний пункт на основі зіставлення поточного шляху з адресою запису (рисунок 2.14).

```
type SidebarNavGroupAttributes = {
  heading: string;
  entries: NavItem[];
  currentPath: string;
};

export const SidebarNavGroup = ({
  heading,
  entries,
  currentPath
}: SidebarNavGroupAttributes) => {
  return (
    <SidebarGroup>
      <SidebarGroupLabel>{heading}</SidebarGroupLabel>
      <SidebarMenu>
        {entries.map((entry) => {
          const selected = currentPath.startsWith(entry.url);

          return (
            <SidebarMenuItem key={entry.url}>
              <SidebarMenuButton
                asChild
```

Рис. 2.14 – Код виокремленого компонента SidebarNavGroup

Аналогічну декомпозицію застосовано до інших об'ємних представлень: з представлення інтеграцій виокремлено самостійні компоненти діалогу та сітки інтеграцій, а з представлення детального перегляду діалогу — окремий компонент стану завантаження. Така декомпозиція редукувала складність вихідних компонентів, підвищила ступінь повторного використання коду та локалізувала зони можливих змін.

Завершальним напрямом рефакторингу стала уніфікація візуальної ідентичності дизайн-системи шляхом перегляду централізованих кольорових токенів у файлі глобальних стилів. Брендову палітру системно змінено з одночасним узгодженим оновленням семантичних змінних — основного акцентного кольору, кольору фокусного обведення, токенів бічної панелі, діаграм та акцентів — для світлої та темної тем. Централізація кольорових змінних на рівні дизайн-системи забезпечила атомарність зміни візуальної ідентичності без необхідності модифікації окремих компонентів, що є прямим наслідком обраної архітектури з єдиним джерелом істини для стильових констант.

3 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДТРИМКА

Розроблена платформа становить B2B-сервіс автоматизації клієнтської підтримки, призначений для організацій, що надають підтримку власним клієнтам через вебсайт. Система складається з двох взаємодоповнюваних інтерфейсів: вбудовуваного віджета, через який кінцеві користувачі (відвідувачі сайту організації) звертаються по підтримку, та операторської панелі, через яку співробітники організації керують зверненнями. Цей підрозділ описує інтерфейс системи з позиції користувача та основний сценарій її застосування.

3.1 Інтерфейс вбудовуваного віджета

Віджет інтегрується у вебсайт організації через єдиний фрагмент коду — тег сценарію з ідентифікатором організації, який розміщують на сторінці-носії. Після вбудовування на сторінці з’являється віджет підтримки. На рисунку 3.1 наведено демонстраційну сторінку-носій із фрагментом коду вбудовування та відкритим віджетом у режимі текстового діалогу: відображаються вітальне повідомлення агента, повідомлення користувача та поле введення.

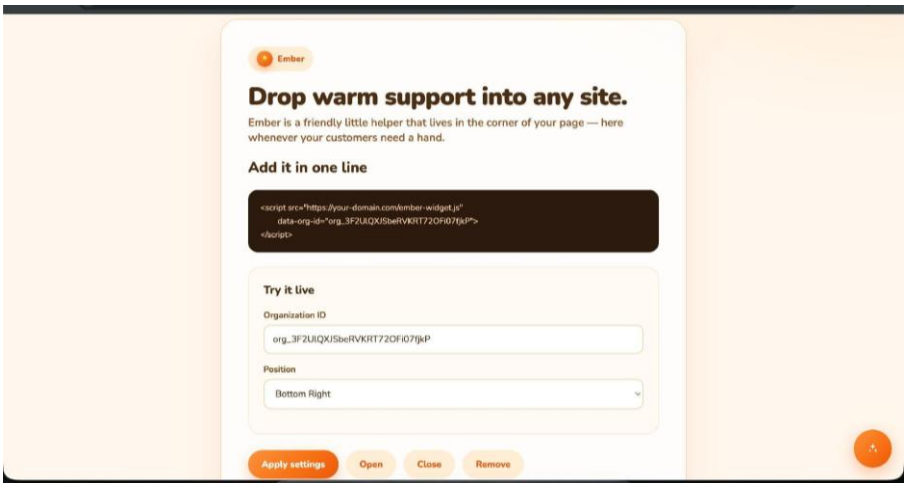

Рис. 3.1 – Вбудований віджет у режимі текстового діалогу на сторінці-носії

На екрані вибору користувачеві пропонується розпочати текстовий чат, а за умови налаштованої голосової інтеграції — також голосовий виклик або

замовлення телефонного дзвінка. Доступність голосових каналів визначається наявністю відповідних налаштувань організації, що ілюструє адаптивність інтерфейсу до конфігурації конкретного орендаря.

Операторська панель є робочим середовищем співробітника організації. Вона надає засоби перегляду списку звернень, ведення діалогу з користувачем, налаштування параметрів віджета, керування базою знань та зовнішніми інтеграціями. Доступ до панелі обмежено автентифікованими операторами в межах їхньої організації. Головний робочий екран панелі — список звернень із обраним діалогом — подано на рисунку 3.2.

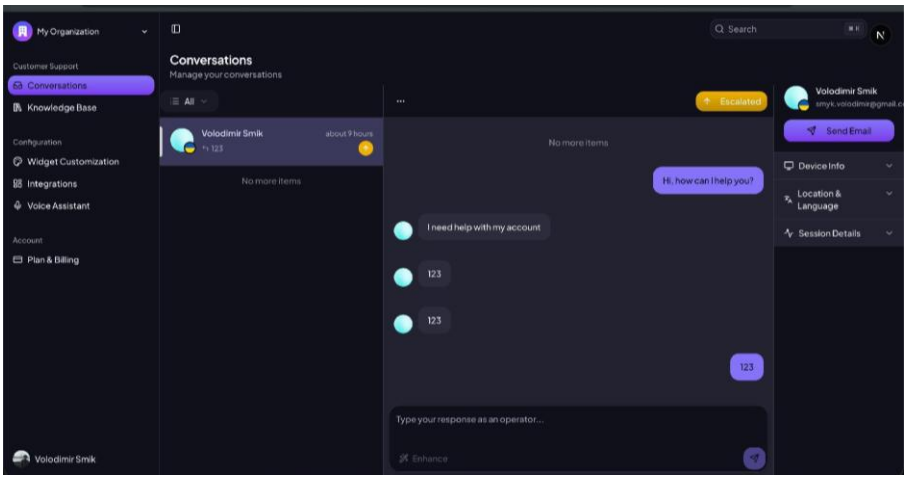

Рисунок 3.2 — Операторська панель: список звернень та діалог

Ліворуч розташовано навігаційне меню з розділами підтримки, конфігурації та облікового запису. У центральній частині відображається список звернень та обраний діалог з історією повідомлень; поле введення дозволяє оператору відповісти користувачеві, а функція покращення відповіді (Enhance) застосовує мовну модель для редагування тексту оператора. Праворуч подано контекстну інформацію про сесію відвідувача — дані пристрою, місцеперебування та мову. Статус звернення (на рисунку — «Escalated») відображає його стан у скінченному автоматі станів: за неможливості автоматичного вирішення звернення передається операторові.

Налаштування параметрів віджета здійснюється на окремій сторінці конфігурації (рисунок 3.3): оператор задає вітальне повідомлення, що

відображається під час відкриття віджета, та типові підказки для швидкого початку діалогу.

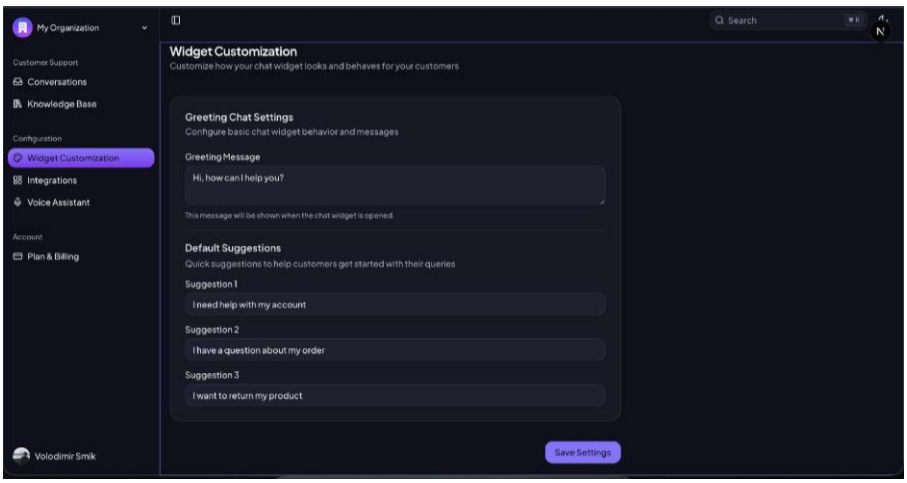

Рисунок 3.3 — Сторінка налаштування параметрів віджета

Доменні знання організації завантажуються до бази знань, що використовується підсистемою генерації, доповненої пошуком. Сторінку керування базою знань подано на рисунку 3.4.

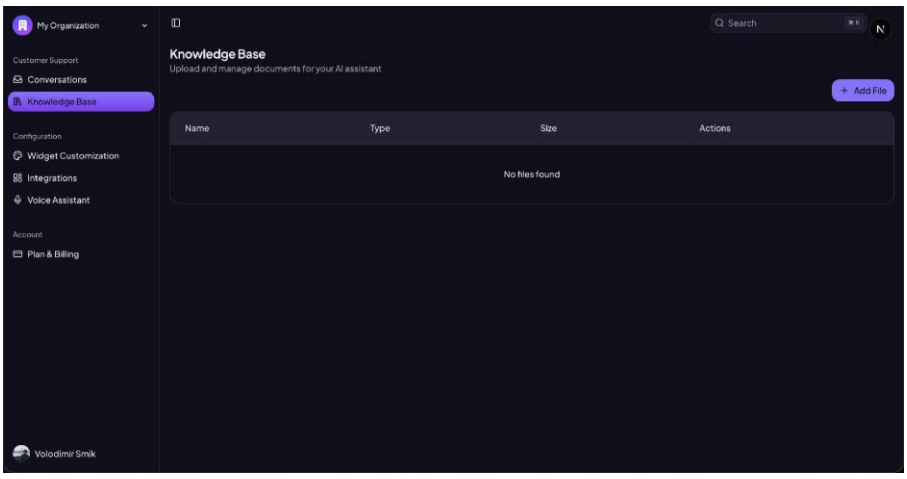

Рисунок 3.4 — Сторінка керування базою знань

Сторінка інтеграцій формує фрагмент коду для вбудовування віджета у вебсайт організації. На рисунку 3.5 подано діалог інтеграції з готовим тегом сценарію, що містить ідентифікатор організації та підлягає розміщенню в розділі head сторінки-носія.

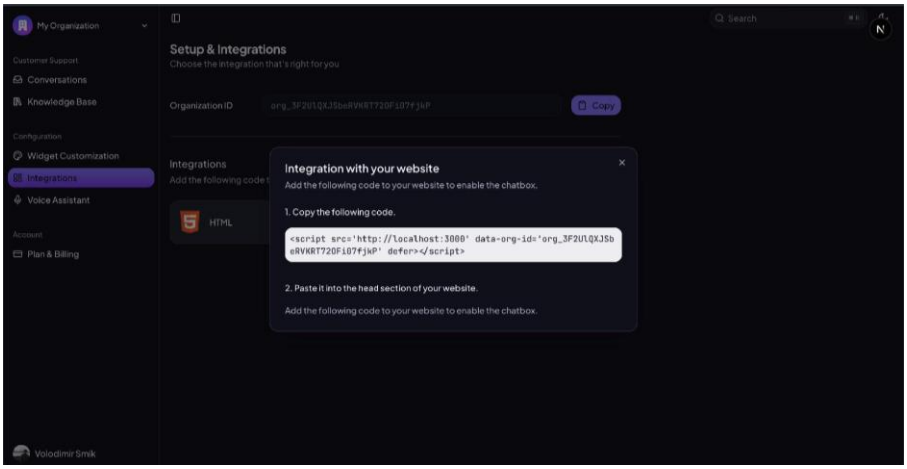

Рисунок 3.5 — Сторінка інтеграції з кодом вбудовування віджета

Голосовий канал взаємодії активується підключенням зовнішнього сервісу телефонії на сторінці голосового асистента (рисунок 3.6), що відкриває можливості голосових викликів, виділених телефонних номерів та автоматизованих сценаріїв.

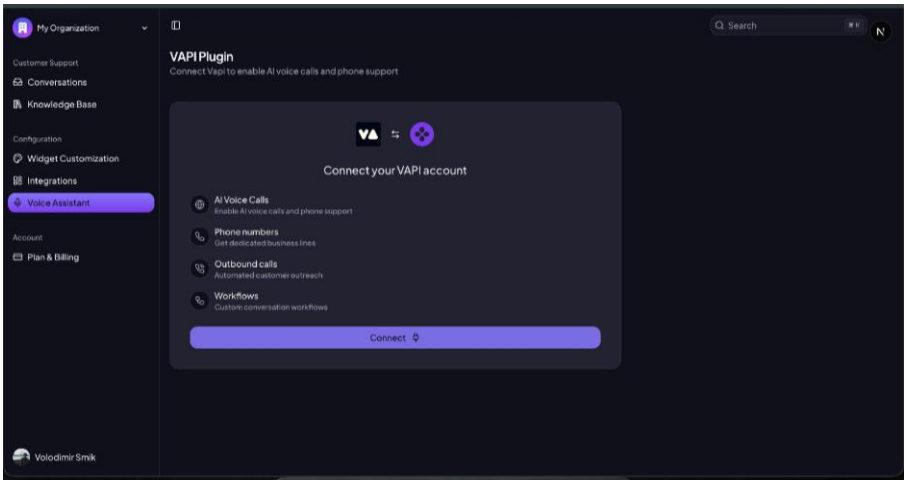

Рисунок 3.6 — Сторінка підключення голосового асистента

Основний сценарій роботи системи — автоматизоване опрацювання звернення кінцевого користувача — об’єднує описані інтерфейси в єдиний робочий потік.

Спершу організація-орендар одноразово підключає платформу: реєструє обліковий запис, налаштовує параметри віджета та вбудовує згенерований код

віджета на свій вебсайт. Вхідними даними на цьому етапі є налаштування організації, результатом — активний віджет на сайті.

Далі відвідувач сайту відкриває віджет та обирає текстовий канал підтримки. Якщо контактна сесія ще не створена, віджет відображає екран автентифікації, де відвідувач вводить ім’я та електронну адресу. Вхідними даними є контактні дані відвідувача, результатом — створена контактна сесія, прив’язана до організації.

Після цього відвідувач формулює запитання у вікні чату. Введене повідомлення є основними вхідними даними системи. Сервер створює діалог, передає повідомлення агентові штучного інтелекту, який виконує семантичний пошук у базі знань організації та формує відповідь природною мовою. Сформована відповідь є вихідними даними системи й реактивно відображається в інтерфейсі віджета.

Якщо агент не може вирішити звернення самостійно, він виконує інструментальний виклик ескалації, що переводить діалог у стан очікування втручання оператора. Оператор бачить ескальоване звернення у списку звернень та долучається до діалогу. Таким чином, типові звернення опрацьовуються автоматично, а складні передаються людині, що й становить основну цінність системи.

Узагальнено, вхідними даними системи є повідомлення користувача (текстові або голосові), контактні дані відвідувача та документи бази знань організації, а вихідними — згенеровані агентом відповіді, стан діалогів та сповіщення операторові про ескалацію.

3.2 Види та план тестування

Перший рівень забезпечення якості — статичний аналіз коду засобами інструмента ESLint, що виявляє потенційні дефекти, порушення узгодженості стилю та небезпечні конструкції без виконання програми. Конфігурацію статичного аналізу централізовано винесено в окремий пакунок спільного використання @workspace/eslint-config, що успадковується всіма застосунками монорепозиторію, чим забезпечується єдиний набір правил для всієї кодової бази. Базова конфігурація поєднує рекомендовані набори правил для JavaScript, TypeScript та узгодження з форматувальником, а для застосунків на основі Next.js додатково підключено правила перевірки коректності використання хуків React, відповідності рекомендаціям Next.js та показникам Core Web Vitals. Запуск статичного аналізу інтегровано в конвеєр завдань Turborepo, що дозволяє перевіряти всю кодову базу єдиною командою.

Другий рівень — статична перевірка типів засобами компілятора TypeScript у суворому режимі (strict). Завдяки наскрізній типовій узгодженості, за якої типи серверних функцій та моделі даних автоматично генеруються платформою Convex та споживаються клієнтським кодом, перевірка типів охоплює не лише внутрішню коректність клієнтських компонентів, а й узгодженість контрактів взаємодії клієнта із сервером. Будь-яка невідповідність між очікуваним та фактичним типом даних — наприклад, звернення до неіснуючого поля документа або передавання аргументу некоректного типу до серверної функції — виявляється на етапі компіляції, що унеможливлює відповідний клас помилок під час виконання. Конфігурацію компілятора також централізовано в пакунку @workspace/typescript-config.

Третій рівень — функціональне тестування, що полягає у перевірці відповідності поведінки системи функціональним вимогам шляхом виконання ключових сценаріїв використання та зіставлення фактичного результату з очікуваним.

Для функціонального тестування розроблено набір тестових сценаріїв, що охоплюють основні функціональні вимоги до платформи. Кожен сценарій визначає послідовність дій користувача, очікуваний результат та критерій його досягнення. Результати функціонального тестування за основними сценаріями наведено в таблиці 3.1.

Таблиця 3.1 – Результати функціонального тестування

Тип тестування	Очікуваний результат	Результат
Автентифікація оператора	Доступ до панелі надається лише після успішної аутентифікації	Пройдено
Перегляд списку звернень	Відображається список звернень організації з можливістю фільтрування за станом	Пройдено
Опрацювання звернення агентом	Агент формує відповідь на основі бази знань організації	Пройдено
Ескалація звернення оператору	Діалог переходить у стан очікування втручання оператора	Пройдено
Реактивне оновлення інтерфейсу	Нове повідомлення відображається без перезавантаження сторінки	Пройдено
Налаштування параметрів віджета	Змінені параметри застосовуються та зберігаються	Пройдено
Ізоляція даних орендарів	Оператор має доступ лише до даних власної організації	Пройдено
Адаптивність інтерфейсу	Інтерфейс коректно відображається на різних класах пристроїв	Пройдено

За результатами функціонального тестування підтверджено коректність роботи системи в усіх перевірених сценаріях. Виявлені під час тестування незначні недоліки, що стосувалися переважно граничних випадків та відображення інтерфейсу, було усунено в робочому порядку. Поєднання статичного аналізу коду,

статичної перевірки типів та функціонального тестування за сценаріями забезпечило належний рівень якості програмної системи та підтвердило її відповідність сформульованим функціональним вимогам.

3.3 Розгортання програмної системи та системні вимоги

Розгортання платформи зумовлене її дворівневою архітектурою та організацією кодової бази у форматі монорепозиторію. Підготовку застосунків до розгортання здійснюють засобами системи інкрементального збирання Turborepo, що складає спільні пакунки та застосунки в топологічно впорядкованій послідовності з повторним складанням лише змінених частин.

Розгортання серверної частини на платформі Convex полягає у публікації серверних функцій та схеми даних, після чого виконання та масштабування забезпечуються автоматично, без ручного адміністрування серверів. Клієнтський застосунок розгортають як вебзастосунок, що обслуговує статично згенеровані та серверно-відрендерені сторінки. Конфіденційні параметри конфігурації задають через змінні середовища і не включають до кодової бази.

Системні вимоги поділяються на вимоги з боку кінцевого користувача та вимоги до середовища розгортання. З боку користувача для роботи з вебінтерфейсом достатньо сучасного вебоглядача та з'єднання з мережею Інтернет; встановлення додаткового програмного забезпечення не потрібне. Вимоги до середовища розгортання охоплюють наявність середовища виконання Node.js для складання застосунків, менеджера залежностей pnpm у зафіксованій версії та облікових записів відповідних зовнішніх сервісів.

Верифікація програмної системи полягає у підтвердженні відповідності реалізованої функціональності сформульованим у першому розділі вимогам. Зіставлення реалізації з функціональними вимогами засвідчує, що платформа забезпечує опрацювання звернень інтелектуальним агентом, ескалацію складних випадків операторові, керування станом діалогу, текстовий та голосовий канали взаємодії, операторську панель керування зверненнями, налаштування віджета й

керування інтеграціями та автентифікацію операторів, що відповідає визначеному переліку функціональних вимог.

Перевірку досягнення нефункціональних вимог здійснено на рівні застосованих архітектурних рішень. Вимогу багатоорендарності реалізовано через ізоляцію даних організацій на рівні схеми даних із дискримінацією записів за ідентифікатором організації. Вимогу продуктивності й реактивності задоволено застосуванням реактивної моделі синхронізації стану платформи Convex, що забезпечує автоматичне оновлення інтерфейсу без ручного повторного опитування сервера. Вимогу типової безпеки реалізовано через наскрізну типову узгодженість «сховище — клієнт» із верифікацією контрактів на етапі компіляції.

За результатами цього розділу здійснено тестування розробленої системи із застосуванням статичного аналізу коду, статичної перевірки типів та функціонального тестування за сценаріями, розглянуто процедуру розгортання платформи й сформульовано системні вимоги та виконано верифікацію відповідності системи поставленим вимогам. Підтверджено працездатність системи та її готовність до практичного застосування.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОХОРОНА ПРАЦІ

Розроблення програмного забезпечення передбачає тривалу роботу фахівця за персональним комп'ютером, що супроводжується впливом на його організм низки чинників виробничого середовища. У цьому розділі розглянуто питання безпеки життєдіяльності та охорони праці [32] стосовно діяльності розробника програмного забезпечення. Проаналізовано ергономічні проблеми, пов'язані з роботою за комп'ютером, та визначено гігієнічні вимоги до організації й обладнання робочого місця, дотримання яких забезпечує збереження здоров'я та працездатності працівника.

4.1 Ергономічні проблеми безпеки життєдіяльності

Розроблення програмного забезпечення є видом професійної діяльності, що передбачає тривале перебування працівника за персональним комп’ютером у відносно нерухомій робочій позі з високим рівнем зорового та нервово-емоційного навантаження. Саме тому ергономічні аспекти безпеки життєдіяльності мають безпосереднє значення для збереження здоров’я та працездатності розробника. Ергономіка як наука вивчає функціональні можливості й особливості людини в процесі праці з метою створення таких умов, знарядь та організації робочого процесу, що відповідають фізіологічним і психологічним характеристикам людини. Метою ергономічного підходу є узгодження засобів праці, робочого середовища та трудового процесу з можливостями людини, що забезпечує безпеку, збереження здоров’я та високу ефективність діяльності.

Праця розробника програмного забезпечення належить до категорії розумової праці операторського типу, для якої характерне переважання зорового аналізатора в отриманні інформації, мала рухова активність та значне навантаження на опорно-руховий апарат унаслідок тривалого перебування у фіксованій позі. До основних ергономічних проблем такої діяльності належать зорове перевантаження, статичне м’язове напруження, а також нервово-емоційне

навантаження, пов’язане з високою концентрацією уваги та відповідальністю за результат.

Зорове навантаження є одним з найвагоміших чинників. Тривала робота з відеодисплейним терміналом супроводжується постійним напруженням м’язів ока, частим переведенням погляду між екраном, документами та клавіатурою, а також сприйняттям зображення, сформованого із світних точок. Це може спричиняти зорову втому, що проявляється відчуттям різі в очах, головним болем, тимчасовим зниженням гостроти зору. Сукупність таких порушень отримала назву комп’ютерного зорового синдрому. Ергономічними засобами запобігання зоровому перевантаженню є раціональне розташування екрана відносно очей, забезпечення належного освітлен [30]ня робочої зони, усунення відблисків на екрані та дотримання регламентованих перерв у роботі.

Статичне м’язове напруження виникає внаслідок тривалого перебування працівника в одноманітній сидячій позі. Відсутність рухової активності призводить до перевантаження м’язів шиї, плечового поясу, спини та погіршення кровообігу, що з часом може спричиняти захворювання опорно-рухового апарату. Окремим проявом є перенапруження кисті та передпліччя внаслідок одноманітних рухів під час роботи з клавіатурою та маніпулятором, відоме як синдром зап’ясткового каналу. Запобігання цим порушенням досягається ергономічною організацією робочого місця, що забезпечує фізіологічно правильну робочу позу, а також чергуванням праці й відпочинку з виконанням фізичних вправ.

Нервово-емоційне навантаження зумовлене високою інтенсивністю розумової праці, потребою тривалої концентрації уваги, опрацюванням значних обсягів інформації та роботою в умовах обмеженого часу. Надмірне нервово-емоційне навантаження призводить до розвитку втоми, зниження продуктивності та якості праці, а за тривалої дії — до перевтоми та погіршення загального стану здоров’я. Ергономічними заходами зниження такого навантаження є раціональна організація режиму праці й відпочинку, психофізіологічне розвантаження та створення сприятливого мікроклімату в колективі.

Таким чином, урахування ергономічних проблем безпеки життєдіяльності є необхідною умовою збереження здоров’я та працездатності розробника програмного забезпечення. Комплексне розв’язання цих проблем досягається належною організацією робочого місця, дотриманням гігієнічних вимог та раціональним режимом праці, що детально розглянуто в наступному підрозділі.

4.2 Гігієнічні вимоги до організації та обладнання робочих місць з відеодисплейними терміналами

Організація робочого місця користувача відеодисплейного термінала (ВДТ) регламентується державними санітарн [29]ими нормами і правилами, що встановлюють гігієнічні вимоги до приміщень, обладнання робочих місць та умов праці. Дотримання цих вимог спрямоване на запобігання несприятливому впливу на здоров’я працівника шкідливих і небезпечних виробничих чинників, що супроводжують роботу з обчислювальною технікою. Оскільки розроблення програмного забезпечення передбачає постійне використання персонального комп’ютера, дотримання гігієнічних вимог до робочого місця з ВДТ є безпосередньо застосовним до умов праці розробника.

Вимоги до виробничого приміщення передбачають достатню площу та об’єм на одне робоче місце. Згідно з чинними санітарними нормами, площа на одне робоче місце користувача ВДТ має становити не менше 6 квадратних метрів, а об’єм — не менше 20 кубічних метрів. Приміщення мають бути обладнані системами опалення, кондиціонування повітря або припливно-витяжною вентиляцією, що забезпечують підтримання оптимальних параметрів мікроклімату. Не допускається розміщення робочих місць з ВДТ у підвальних приміщеннях.

Параметри мікроклімату робочої зони мають відповідати оптимальним значенням для приміщень з ВДТ. Для холодного періоду року оптимальна температура повітря становить від 22 до 24 градусів Цельсія, для теплого — від 23 до 25 градусів Цельсія. Відносна вологість повітря має перебувати в межах від 40

до 60 відсотків, а швидкість руху повітря не повинна перевищувати 0,1 метра за секунду. Підтримання таких параметрів запобігає перегріванню або переохолодженню організму та забезпечує комфортні умови праці.

Освітлення робочого місця є одним з найважливіших гігієнічних чинників, оскільки безпосередньо впливає на зорове навантаження. Освітлення має бути змішаним — природним та штучним. Природне світло повинно надходити переважно збоку, ліворуч від користувача. Штучне освітлення робочої поверхні має забезпечувати освітленість у межах від 300 до 500 люксів. Для уникнення відблисків на екрані та осліплення працівника застосовують розсіяне освітлення, а джерела світла розташовують так, щоб вони не потрапляли в поле зору та не відбивалися від поверхні екрана. Робоче місце розташовують відносно вікон таким чином, щоб уникнути прямого та відбитого блиску.

Вимоги до організації безпосередньо робочого місця стосуються взаємного розташування його елементів відповідно до ергономічних принципів. Екран відеодисплейного термінала має розташовуватися на оптимальній відстані від очей користувача, що становить від 600 до 700 міліметрів, але не ближче ніж 500 міліметрів, з урахуванням розміру символів. Верхній край екрана має бути на рівні очей або трохи нижче, що забезпечує природне положення голови та шиї. Клавіатуру розташовують на поверхні столу на відстані від 100 до 300 міліметрів від краю, зверненого до користувача, що дозволяє опертися передпліччям. Робоче крісло [31] має бути підйомно-поворотним, із регульованими висотою сидіння та кутом нахилу спинки, що дозволяє підтримувати фізіологічно раціональну робочу позу.

Раціональна робоча поза передбачає, що ступні працівника спираються на підлогу або підставку для ніг, гомілки розташовані вертикально, а стегна — горизонтально; кут згину в ліктьовому суглобі становить близько 90 градусів. Така поза мінімізує статичне навантаження на опорно-руховий апарат та запобігає розвитку втоми.

Окреме значення має дотримання режиму праці та відпочинку. Безперервна робота з відеодисплейним терміналом без регламентованої перерви не повинна

перевищувати дві години. Упродовж робочого дня встановлюються додаткові регламентовані перерви тривалістю від 10 до 15 хвилин, які доцільно використовувати для виконання комплексу вправ для очей, шиї, рук та тулуба з метою зниження зорового й статичного навантаження. Чергування праці та відпочинку є дієвим засобом запобігання перевтомі та збереження працездатності.

Отже, у цьому розділі розглянуто ергономічні проблеми безпеки життєдіяльності, що виникають під час роботи розробника програмного забезпечення за персональним комп'ютером, та визначено гігієнічні вимоги до організації й обладнання робочого місця з відеодисплейним терміналом. Дотримання наведених вимог щодо параметрів мікроклімату, освітлення, організації робочого місця та режиму праці й відпочинку забезпечує збереження здоров'я і працездатності працівника та запобігає розвитку професійних захворювань.

Важливим елементом профілактики є також проходження працівниками, зайнятими роботою з відеодисплейними терміналами, обов'язкових медичних оглядів. Попередній медичний огляд здійснюється під час прийняття на роботу, а періодичні — упродовж трудової діяльності, що дозволяє своєчасно виявити можливі відхилення у стані здоров'я, зумовлені впливом виробничих чинників, та вжити відповідних запобіжних заходів. Сукупність організаційних, гігієнічних та медико-профілактичних заходів забезпечує комплексний підхід до охорони праці розробника програмного забезпечення.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи здійснено повний цикл розробки програмного забезпечення — від аналізу предметної області та формування вимог до проєктування архітектури, реалізації, тестування й верифікації системи. Основним результатом роботи є розроблена масштабована B2B-платформа для автоматизації клієнтської підтримки на основі мультимодальних агентів штучного інтелекту, що поєднує серверну та клієнтську складові з підтримкою текстового й голосового каналів взаємодії.

На етапі аналізу предметної області розглянуто наявні рішення у сфері автоматизації клієнтської підтримки, проведено їх порівняльний аналіз та виявлено спільні обмеження — високу й складно прогнозовану вартість володіння, прив'язку до власних мовних моделей, залежність від постачальника та реалізацію голосового каналу як другорядної надбудови. На основі цього аналізу сформульовано функціональні та нефункціональні вимоги до платформи, визначено акторів і ключові варіанти використання, що склало підґрунтя для проєктування системи.

Для реалізації платформи обрано стек технологій на основі мови TypeScript із забезпеченням наскрізної типової узгодженості від сховища даних до інтерфейсу користувача. Клієнтську частину реалізовано засобами фреймворку Next.js на основі бібліотеки React із застосуванням компонентного підходу, а серверну — на основі безсерверної реактивної платформи Convex. Кодову базу організовано у форматі монорепозиторію на основі менеджера залежностей pnpm із системою інкрементального збирання Turborepo, що забезпечило повторне використання коду між застосунками, узгодженість конфігурацій та детермінованість складання. Для контролю версій застосовано систему Git у поєднанні з платформою GitHub.

Процес розробки здійснювався відповідно до ітеративно-інкрементального підходу на засадах методології Agile, що забезпечило гнучкість в управлінні проєктом, можливість поступового нарощування функціональності окремими модулями та швидке реагування на уточнення вимог. Модульно-доменна

організація кодової бази та реактивна модель синхронізації стану дозволили досягти локальності змін та оновлення інтерфейсу в режимі, близькому до реального часу, без ручного повторного опитування сервера.

У межах реалізації спроєктовано модель даних з ізоляцією орендарів за принципом багатоорендарності, реалізовано серверний програмний інтерфейс із розмежуванням функцій за рівнем довіри та інтелектуальну підсистему опрацювання звернень на основі агента штучного інтелекту з генерацією, доповненою пошуком. Клієнтську частину побудовано як ієрархію автономних повторно використовуваних компонентів, що проілюстровано на прикладі презентаційного модуля. Окремо проведено оптимізацію та рефакторинг кодової бази — уніфікацію логіки реактивних запитів, впровадження єдиної конвенції типізації, декомпозицію монолітних компонентів та централізацію стильових констант, — що підвищило супроводжуваність системи.

Забезпечення якості платформи здійснено за допомогою багаторівневого підходу, що поєднав статичний аналіз коду засобами ESLint, статичну перевірку типів засобами компілятора TypeScript у суворому режимі та функціональне тестування за сценаріями. Статична перевірка типів, завдяки наскрізній типовій узгодженості «сховище — клієнт», дозволила виявляти неузгодженість контрактів взаємодії на етапі компіляції, а функціональне тестування підтвердило коректність роботи системи в основних сценаріях використання. Проведена верифікація засвідчила відповідність реалізованої функціональності сформульованим функціональним та нефункціональним вимогам.

Таким чином, виконана робота підтвердила актуальність обраних технологій та архітектурних підходів і продемонструвала їх ефективність у вирішенні поставлених завдань. Розроблена платформа відповідає сформульованим вимогам щодо масштабованості, багатоорендарності, типової безпеки та супроводжуваності, придатна до подальшого розвитку й промислового застосування.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. Київ : ДП «УкрНДНЦ», 2016. 16 с.
2. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів першого (бакалаврського) рівня вищої освіти за освітньо-професійною програмою «Інженерія програмного забезпечення» спеціальності 121. Тернопіль : ТНТУ ім. І. Пулюя, 2024. 92 с.
3. TypeScript Documentation : офіційна документація. URL: https://www.typescriptlang.org/docs/ (дата звернення: 16.06.2026).
4. React : офіційна документація. URL: https://react.dev/ (дата звернення: 16.06.2026).
5. Next.js Documentation : App Router. URL: https://nextjs.org/docs/app (дата звернення: 16.06.2026).
6. Next.js : Getting Started — Installation : офіційна документація. URL: https://nextjs.org/docs/app/getting-started/installation (дата звернення: 16.06.2026).
7. React Server Components : документація React. URL: https://react.dev/reference/rsc/server-components (дата звернення: 16.06.2026).
8. pnpm : Workspaces : офіційна документація. URL: https://pnpm.io/workspaces (дата звернення: 16.06.2026).
9. Turborepo Documentation : офіційна документація. URL: https://turborepo.com/docs (дата звернення: 16.06.2026).
10. ESLint : Documentation. URL: https://eslint.org/docs/latest/ (дата звернення: 16.06.2026).
11. Prettier [11] : Documentation. URL: https://prettier.io/docs/ (дата звернення: 16.06.2026).
12. Convex Documentation : офіційна документація. URL: https://docs.convex.dev/ (дата звернення: 16.06.2026).

13. Convex : Database Schemas : документація. URL: https://docs.convex.dev/database/schemas (дата звернення: 16.06.2026).
14. Convex : Queries, Mutations and Actions : документація. URL: https://docs.convex.dev/functions (дата звернення: 16.06.2026).
15. Convex Agent : документація компонента. URL: https://docs.convex.dev/agents (дата звернення: 16.06.2026).
16. Clerk : Next.js Quickstart (App Router) : документація. URL: https://clerk.com/docs/nextjs/getting-started/quickstart (дата звернення: 16.06.2026).
17. Vapi : Documentation : офіційна документація. URL: https://docs.vapi.ai/ (дата звернення: 16.06.2026).
18. AWS Secrets Manager : User Guide : документація. URL: https://docs.aws.amazon.com/secretsmanager/ (дата звернення: 16.06.2026).
19. Tailwind [19] CSS : Documentation. URL: https://tailwindcss.com/docs (дата звернення: 16.06.2026).
20. Glova B., Mudryk I. Application of Deep Learning in Neuromarketing Studies of the Effects of Unconscious Reactions on Consumer Behavior. 2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP), Lviv, Ukraine, 2020. P. 337–340. DOI: 10.1109/DSMP47368.2020.9204192.
21. Bryk O., Mudryk I., Holubovskyi M., Stoianov Y. Machine learning models and methods aspects of processing unstructured data. Proceedings of the 1st International Workshop on Bioinformatics and Applied Information Technologies (BAIT 2024) : CEUR Workshop Proceedings. 2024. P. 64–74. ISSN 1613-0073.
22. Глух О., Мудрик І. Методи та підходи до автоматичної генерації інтерфейсних елементів у веброзробці на основі великих мовних моделей. Інформаційні моделі, системи та технології : матеріали XIII науково-технічної конференції (17 грудня 2025 р.). Тернопіль : ТНТУ, 2025. С. 168.

ДОДАТКИ

ДОДАТОК А

Графічні матеріали

У цьому додатку наведено графічні матеріали, що детально ілюструють архітектуру та модель даних платформи. На рисунки додатка є посилання в основному тексті роботи.

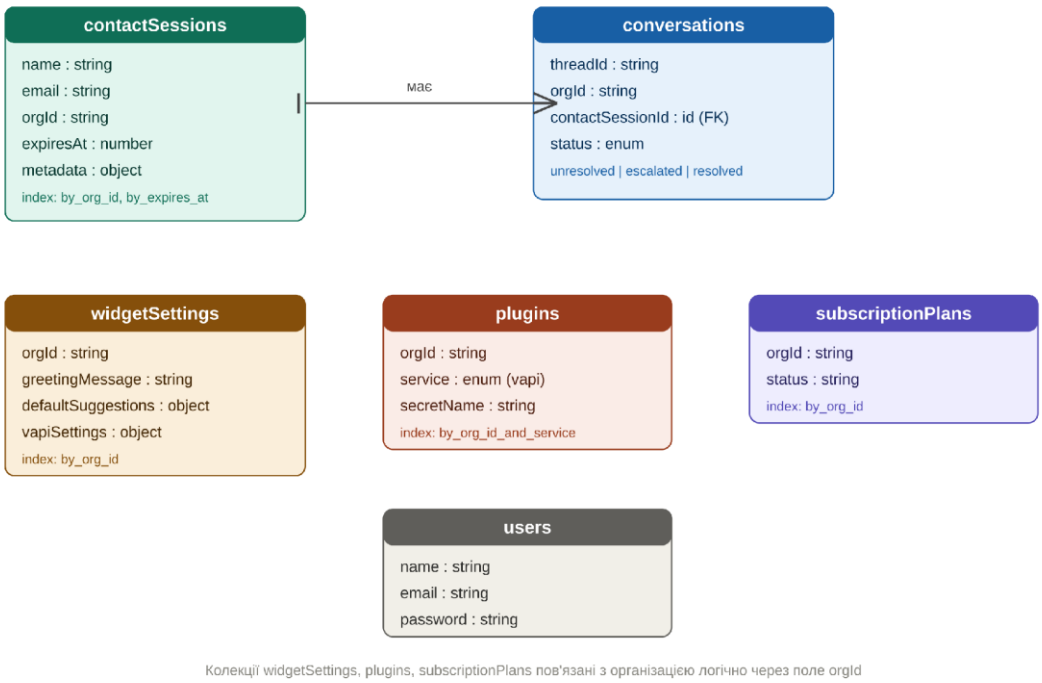

Рисунок А.1 — ER-діаграма моделі даних платформи

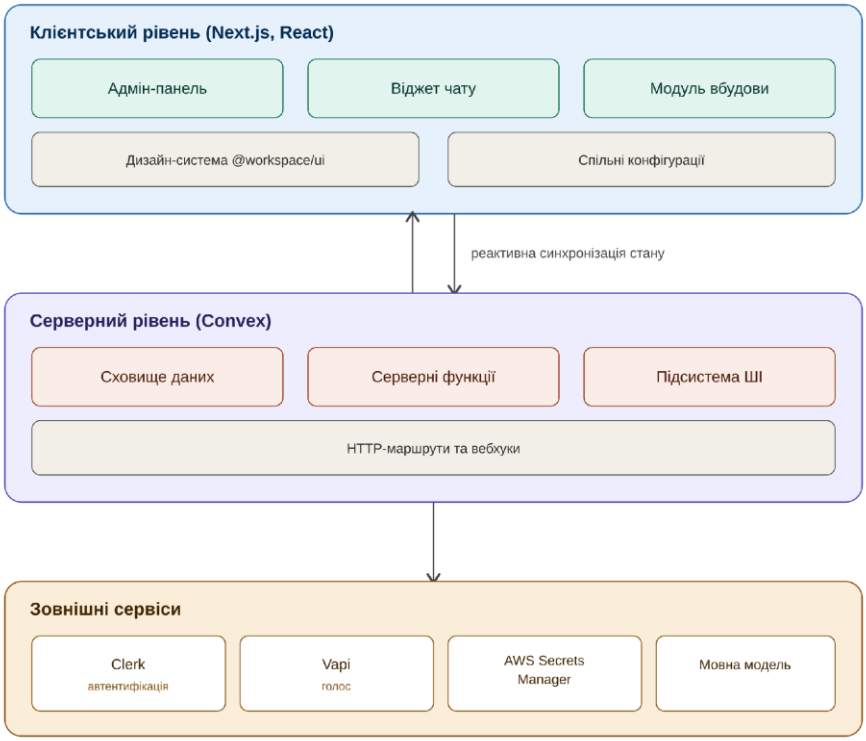

Рисунок А.2 — Структурна схема архітектури платформи

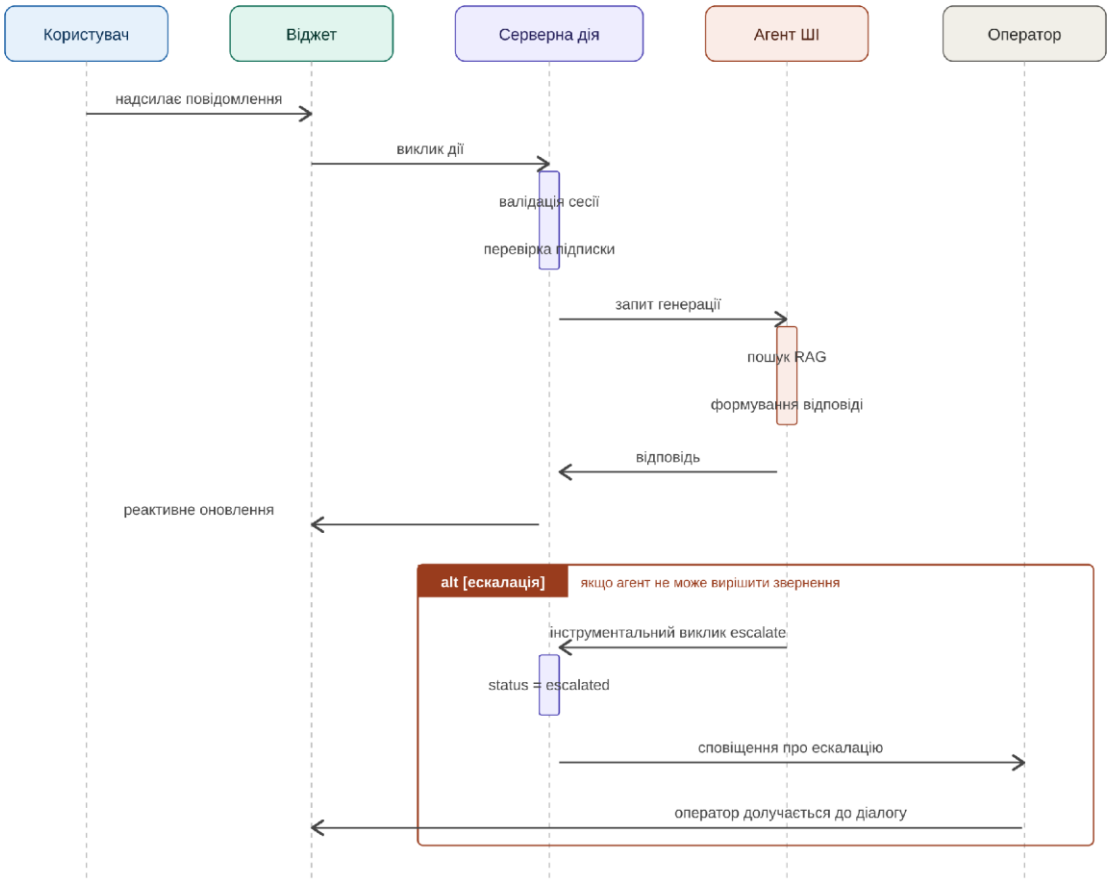

Рисунок А.3 — Діаграма послідовності опрацювання звернення з гілкою ескалації

ДОДАТОК Б

Фрагменти програмного коду

У цьому додатку наведено повні лістинги ключових модулів платформи, подані в основному тексті роботи у скороченому вигляді. На лістинги додатка є посилання в основному тексті роботи.

Лістинг Б.1 — Декларативна схема бази даних (schema.ts)

```
import { defineSchema, defineTable } from 'convex/server';
import { v } from 'convex/values';

export default defineSchema({
  users: defineTable({
    name: v.string(),
    email: v.string(),
    password: v.string()
  }),

  contactSessions: defineTable({
    name: v.string(),
    email: v.string(),
    orgId: v.string(),
    expiresAt: v.number(),
    metadata: v.optional(
      v.object({
        userAgent: v.optional(v.string()),
        ipAddress: v.optional(v.string()),
        language: v.optional(v.string()),
        platform: v.optional(v.string()),
        screenResolution: v.optional(
          v.object({ width: v.number(), height: v.number() })
        ),
        timezone: v.optional(v.string()),
        referrer: v.optional(v.string()),
        currentUrl: v.optional(v.string())
```

```
    })
  )
})
  .index('by_org_id', ['orgId'])
  .index('by_expires_at', ['expiresAt']),

conversations: defineTable({
  threadId: v.string(),
  orgId: v.string(),
  contactSessionId: v.id('contactSessions'),
  status: v.union(
    v.literal('unresolved'),
    v.literal('escalated'),
    v.literal('resolved')
  )
})
  .index('by_org_id', ['orgId'])
  .index('by_contact_session_id', ['contactSessionId'])
  .index('by_status_and_org_id', ['status', 'orgId'])
  .index('by_thread_id', ['threadId']),

plugins: defineTable({
  orgId: v.string(),
  service: v.union(v.literal('vapi')),
  secretName: v.string()
})
  .index('by_org_id', ['orgId'])
  .index('by_org_id_and_service', ['orgId', 'service']),

widgetSettings: defineTable({
  orgId: v.string(),
  greetingMessage: v.string(),
  defaultSuggestions: v.object({
    suggestion1: v.optional(v.string()),
    suggestion2: v.optional(v.string()),
    suggestion3: v.optional(v.string())
```

```
    }),
    vapiSettings: v.object({
      phoneNumber: v.optional(v.string()),
      assistantId: v.optional(v.string())
    })
  }).index('by_org_id', ['orgId']),

  subscriptionPlans: defineTable({
    orgId: v.string(),
    status: v.string()
  }).index('by_org_id', ['orgId'])
});
```

Лістинг Б.2 — Контейнер-композитор LandingView

```
export const LandingView = () => {
  return (
    <div className='mx-auto flex w-full max-w-screen-lg
                    flex-col gap-8 px-4 py-6'>
      <LandingHero />
      <LandingFeatureGrid />
      <LandingQuickActions />
    </div>
  );
};
```

Лістинг Б.3 — Користувацький хук useViewportQuery

```
export function useViewportQuery() {
  const [isOpen, setIsOpen] = useState(false);

  useEffect(() => {
    const mediaQuery =
      window.matchMedia('(max-width: 768px)');
    setIsOpen(mediaQuery.matches);

    const handler = (e: MediaQueryListEvent) => {
      setIsOpen(e.matches);
```

```
    };

    mediaQuery.addEventListener('change', handler);
    return () =>
      mediaQuery.removeEventListener('change', handler);
  }, []);

  return { isOpen };
}
```

Лістинг Б.4 — Компонент SidebarNavGroup

```
type SidebarNavGroupAttributes = {
  heading: string;
  entries: NavItem[];
  currentPath: string;
};

export const SidebarNavGroup = ({
  heading, entries, currentPath
}: SidebarNavGroupAttributes) => {
  return (
    <SidebarGroup>
      <SidebarGroupLabel>{heading}</SidebarGroupLabel>
      <SidebarMenu>
        {entries.map((entry) => {
          const selected = currentPath.startsWith(entry.url);
          return (
            <SidebarMenuItem key={entry.url}>
              <SidebarMenuButton
                asChild
                tooltip={entry.title}
                isActive={selected}
                className={cn(selected && activeEntryClass)}
              >
                <Link href={entry.url}>
                  <entry.icon className='size-4' />
```

```
                    <span>{entry.title}</span>
                  </Link>
                </SidebarMenuButton>
              </SidebarMenuItem>
            );
          })}
        </SidebarMenu>
      </SidebarGroup>
    );
};
```