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 вебсервісу для інтеграції клієнтів на мові програмування Python з використанням мікрофреймворку FastAPI

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

_____ Толмачов С. О.
(підпис) (прізвище та ініціали)

Керівник _____ Цебрій О. Р.
(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

_____ _____
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавра
(назва освітнього ступеня)

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

студенту Толмачов Сергій Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка вебсервісу для інтеграції клієнтів на мові програмування Python з використанням мікрофреймворку FastAPI

Керівник роботи Цебрій О.Р., канд. фіз.-мат. наук, доц.
(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

3. Вихідні дані до роботи Предметна область, завдання, вимоги та специфікація, програмне рішення, методичні вказівки

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступна частина
Аналіз предметної області та теоретичних основ
Визначення методики реалізації моделі
Реалізація моделі
Визначення основних аспектів охорони праці та безпеки життєдіяльності
Висновки роботи

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Слайди презентації та діаграми процесів

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності,			
основи охорони праці			
Нормоконтроль	Стоянов Ю.М. к.т.н., доц. каф ПІ.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	*Отримання завдання*	*06.04 — 12.04*	*Виконано*
2	*Аналіз завдання*	*13.04 — 17.04*	*Виконано*
3	*Виконання розділу "Аналіз вимог до програмної системи"*	*18.04 — 26.04*	*Виконано*
4	*Виконання розділу "Проєктування та розробка програмної системи"*	*27.04 — 03.05*	*Виконано*
5	*Виконання розділу "Тестуванн, впровадження та підтримка"*	*04.05 — 17.05*	*Виконано*
6	*Виконання розділу "Безпеки життєдіяльності та охорона праці"*	*18.05 — 24.05*	*Виконано*
7	*Оформлення пояснювальної записки і графічного матеріалу*	*25.05 — 07.06*	*Виконано*
8	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*08.06 — 14.06*	
9	*Попередній захист кваліфікаційної роботи бакалавра*	*15.05 — 21.06*	
10	*Захист кваліфікаційної роботи бакалавра*		

Студент _____ Толмачов С.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Цебрій О.Р.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка вебсервісу для інтеграції клієнтів мовою програмування Python з використанням мікрофреймворку FastAPI // Кваліфікаційна робота освітнього рівня «Бакалавр» // Толмачов С. О. // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-43 // Тернопіль, 2026. Сторінок – 74, рисунків – 15, таблиць – 5, слайдів – 15, додатків – 2, посилань – 21, лістингів – 11.

Ключові слова: REST API, FastAPI, хмарна архітектура, AWS, бронювання трансферів, шарова архітектура, SQLAlchemy, OpenAPI, конкурентний доступ, CI/CD.

Кваліфікаційна робота присвячена аналізу, проєктуванню, розробці та тестуванню вебсервісу з прикладним програмним інтерфейсом для онлайн-бронювання приватних трансферів на основі мови програмування Python і мікрофреймворку FastAPI.

У першому розділі досліджено предметну область та ринок приватних перевезень, проаналізовано конкурентні рішення, обґрунтовано вибір технологічного стеку й методології розробки, а також сформульовано вимоги до системи.

У другому розділі обґрунтовано вибір шарової архітектури, спроєктовано хмарну інфраструктуру на платформі AWS і модель даних, наведено комплекс UML-діаграм та описано програмну реалізацію серверної логіки.

У третьому розділі описано стратегію та реалізацію тестування розробленого вебсервісу, його впровадження засобами CI/CD і розгортання у хмарному середовищі, а також організацію підтримки та моніторингу.

У четвертому розділі розглянуто питання безпеки життєдіяльності та основ охорони праці, зокрема надання долікарської допомоги при переломах і вимоги до режимів праці та відпочинку під час роботи з відеодисплейними терміналами.

ABSTRACT

Development of a web service for client integration in the Python programming language using the FastAPI microframework // Bachelor's Qualification Work // Tolmachov S. O. // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-43 // Ternopil, 2026. Pages – 74, figures – 15, tables – 5, slides – 15, appendices – 2, references – 21, listings – 11.

Keywords: REST API, FastAPI, cloud architecture, AWS, transfer booking, layered architecture, SQLAlchemy, OpenAPI, concurrent access, CI/CD.

The qualification work is devoted to the analysis, design, development, and testing of a web service with an application programming interface for online booking of private passenger transfers, built using Python and the FastAPI microframework.

The first section examines the subject area and the private transportation market, analyses competitive solutions, justifies the choice of the technology stack and development methodology, and formulates the system requirements.

The second section justifies the choice of a layered architecture, designs the cloud infrastructure on the AWS platform and the data model, presents a set of UML diagrams, and describes the software implementation of the server-side logic.

The third section describes the testing strategy and its implementation, the deployment of the web service to the cloud via a CI/CD pipeline, and the organisation of support and monitoring.

The fourth section covers life safety and occupational health issues, in particular first aid for fractures and the requirements for work and rest regimes when working with visual display terminals.

ПЕРЕЛІК СКОРОЧЕНЬ

БЖД – безпека життєдіяльності;

ВДТ – відеодисплейний (візуальний дисплейний) термінал;

ДСТУ – державний стандарт України;

ОП – охорона праці;

ПЗ – програмне забезпечення;

ALB – Application Load Balancer – балансувальник навантаження;

API – Application Programming Interface – прикладний програмний інтерфейс;

AWS – Amazon Web Services – хмарна платформа Amazon;

CI/CD – Continuous Integration / Continuous Delivery – безперервна інтеграція та доставка;

ECS – Elastic Container Service – сервіс запуску контейнерів AWS;

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту;

JSON – JavaScript Object Notation – формат обміну даними;

ORM – Object-Relational Mapping – об’єктно-реляційне відображення;

OTP – One-Time Password – одноразовий пароль;

REST – Representational State Transfer – архітектурний стиль вебсервісів;

S3 – Simple Storage Service – об’єктне хмарне сховище AWS;

TTL – Time To Live – час життя запису;

VPC – Virtual Private Cloud – віртуальна приватна хмара.

ЗМІСТ

ВСТУП

Ринок пасажирських автомобільних перевезень, зокрема сегмент приватних та групових міжнародних трансферів, протягом останніх років демонструє стійке зростання, зумовлене активізацією трудової міграції, відновленням туристичних потоків та збільшенням мобільності населення між Україною та країнами Європейського Союзу. Водночас процеси продажу квитків і організації поїздок у багатьох перевізників залишаються частково ручними або реалізованими у вигляді закритих програмних рішень, не пристосованих до інтеграції зі сторонніми сервісами. Це створює бар'єр для розвитку ринку та стримує появу уніфікованих платформ онлайн-бронювання, здатних об'єднувати пропозиції різних операторів.

Цифровізація трансферних компаній сьогодні неможлива без серверної програмної інфраструктури, яка б забезпечувала централізоване зберігання даних про маршрути, рейси та бронювання, обробку платежів, а також надавала програмний інтерфейс для взаємодії з клієнтськими застосунками. Саме прикладний програмний інтерфейс (англ. Application Programming Interface, API) є тим технічним елементом, що дозволяє відокремити бізнес-логіку бронювання від конкретного каналу продажу — вебсайту, мобільного застосунку чи корпоративної системи партнера — і повторно використовувати її в межах різних продуктів. Відсутність у наявних на ринку рішеннях відкритого документованого API, виявлена в ході аналізу предметної області, зумовлює актуальність розробки вебсервісу, орієнтованого саме на інтеграцію зовнішніх клієнтів.

Вибір мікрофреймворку FastAPI як основи серверної частини зумовлений його високою продуктивністю опрацювання запитів, статичною типізацією на основі анотацій типів мови Python, вбудованою валідацією даних засобами бібліотеки Pydantic та автоматичною генерацією специфікації OpenAPI, яка істотно спрощує процес підключення сторонніх застосунків. Розгортання сервісу за хмарною моделлю на платформі Amazon Web Services забезпечує масштабованість, відмовостійкість і доступ до широкого набору керованих

сервісів, що знижує обсяг робіт з адміністрування інфраструктури та пришвидшує розгортання програмного продукту.

Об'єктом дослідження є процеси онлайн-бронювання індивідуальних і групових приватних трансферів у домені пасажирських перевезень.

Предметом дослідження є методи та засоби розробки вебсервісу з прикладним програмним інтерфейсом для інтеграції зовнішніх клієнтських застосунків у процеси бронювання приватних трансферів.

Мета роботи – дослідження та розробка програмного забезпечення серверної частини у вигляді прикладного програмного інтерфейсу для системи онлайн-бронювання приватних трансферів, що забезпечує уніфікований механізм інтеграції зовнішніх клієнтських застосунків.

Для досягнення поставленої мети в роботі необхідно вирішити такі завдання:

1) провести аналіз ринку приватних пасажирських перевезень і наявних програмних рішень, визначити їхні переваги, недоліки та незаповнені ніші;

2) обґрунтувати вибір технологічного стеку серверної частини та хмарної платформи розгортання;

3) сформулювати функціональні та нефункціональні вимоги до вебсервісу;

4) спроєктувати архітектуру програмного рішення та модель даних, розробити необхідні UML-діаграми;

5) реалізувати серверну логіку вебсервісу з прикладним програмним інтерфейсом та інтеграцією з зовнішніми сервісами;

6) автоматизувати процес складання та розгортання програмного продукту засобами CI/CD;

7) виконати автоматизоване й ручне тестування розробленого рішення та перевірити його відповідність сформульованим вимогам.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. У першому розділі проаналізовано предметну область, наявні конкурентні рішення, обґрунтовано вибір технологічного стеку та методології розробки, сформовано вимоги до системи. У

другому розділі описано проєктування архітектури та хмарної інфраструктури, наведено UML-діаграми, розглянуто реалізацію й тестування програмного забезпечення. Третій розділ присвячено питанням безпеки життєдіяльності та основам охорони праці.

Розробку виконано в межах командного проєкту із застосуванням гнучкої методології Agile; результати роботи орієнтовані на практичне впровадження у діяльність компаній-перевізників та інтеграцію з їхніми цифровими продуктами.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Сегмент приватних та групових пасажирських перевезень міжнародними автобусними маршрутами протягом останнього десятиліття зазнав суттєвих структурних змін, пов'язаних із цифровізацією процесів продажу квитків, появою агрегаторів і спеціалізованих онлайн-платформ для бронювання, а також зростаючими очікуваннями клієнтів щодо швидкості, прозорості та зручності оформлення поїздок. У цьому ринковому контексті розглянуто два сервіси, що представляють різні бізнес-моделі організації онлайн-бронювання: сервіс прямого продажу квитків окремого перевізника (Grand Bus) та мультимодальний агрегатор пасажирських перевезень (Sharry) [1].

1.1 Огляд конкурентів і ринку приватних перевезень

Сегмент приватних та групових пасажирських перевезень міжнародними автобусними маршрутами протягом останнього десятиліття зазнав суттєвих структурних змін, пов'язаних із цифровізацією процесів продажу квитків, появою агрегаторів і спеціалізованих онлайн-платформ для бронювання, а також зростаючими очікуваннями клієнтів щодо швидкості, прозорості та зручності оформлення поїздок. У цьому ринковому контексті розглянуто два сервіси, що представляють різні бізнес-моделі організації онлайн-бронювання: сервіс прямого продажу квитків окремого перевізника (Grand Bus) та мультимодальний агрегатор пасажирських перевезень (Sharry).

Огляд сервісу Grand Bus. Grand Bus – український сервіс міжнародних автобусних перевезень, що належить ТОВ «Міжнародний автовокзал» (м. Івано-Франківськ) і пропонує пасажирам пряме онлайн-бронювання квитків на регулярні рейси між містами України, Польщі, Німеччини та Нідерландів. На момент проведення аналізу оператор виконує близько тридцяти трьох маршрутів, що з'єднують тринадцять міст у чотирьох країнах, з основним фокусом на

сполученні Україна – Польща (Варшава, Краків, Катовіце, Вроцлав). Зовнішній вигляд головної сторінки сервісу наведено на рисунку 1.1.

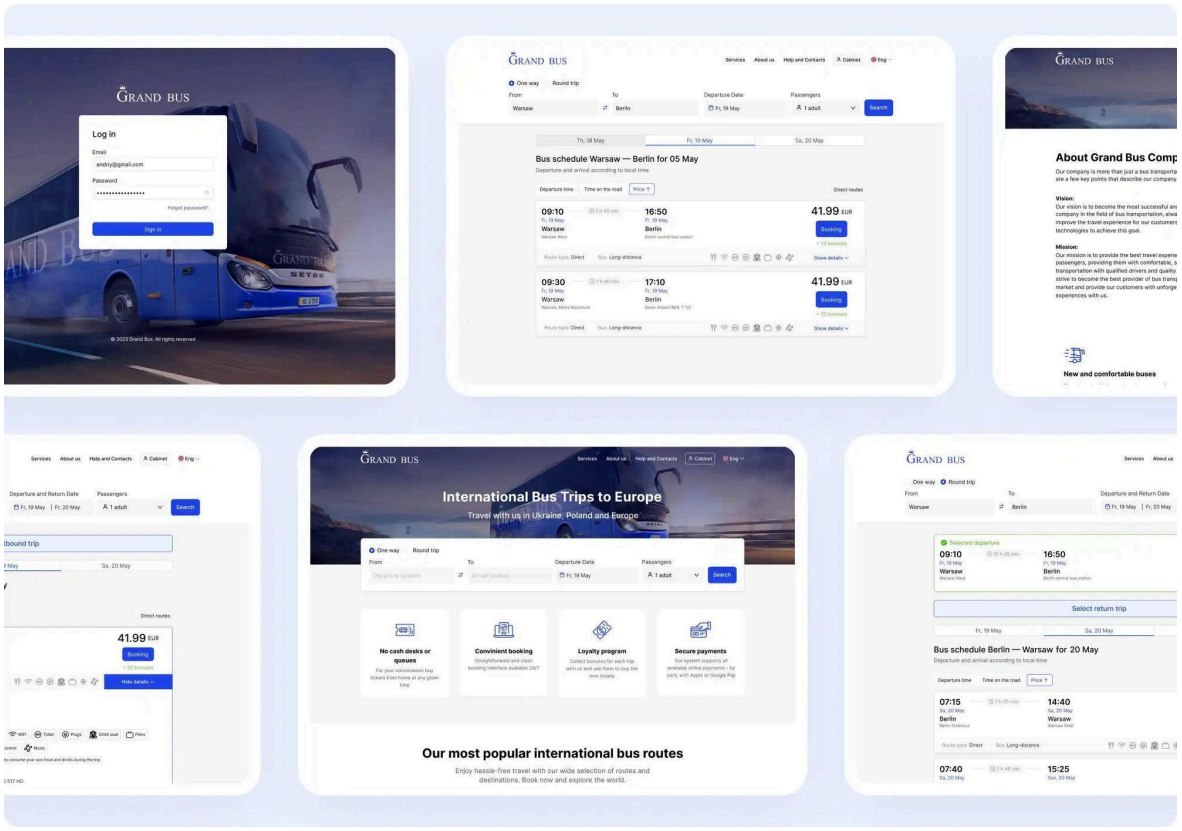

Рисунок 1.1 – Інтерфейс головної сторінки сервісу Grand Bus

Цифрова екосистема Grand Bus складається із трьох основних компонентів: вебсайту для пасажирів, мобільного застосунку для пасажирів та окремого мобільного застосунку для водіїв, що обслуговує операційні процеси перевізника. Вебсайт реалізовано як основний канал онлайн-продажу квитків і виконано з акцентом на швидке оформлення бронювання у кілька кроків.

Процес бронювання передбачає послідовний вибір маршруту, дати поїздки та конкретного рейсу, після чого пасажир переходить до етапу формування квитка. У системі реалізовано вибір рівня сервісу — стандартного класу та преміум-класу, що було впроваджено у 2024 році й передбачає відмінні умови комфорту та цінову політику. Пасажир має змогу обрати додаткові послуги, серед яких — перевезення наднормативного та нестандартного багажу, а також отримати інформацію про умови перевезення дітей.

Платіжна підсистема сервісу інтегрована з українським шлюзом Liqpay, що забезпечує підтримку основних способів безготівкової оплати, прийнятих на ринку. Окремим важливим елементом продукту є реалізована програма лояльності, яка нараховує бали за здійснені поїздки та дозволяє зменшувати вартість наступних бронювань. Поряд із цим, інформування пасажирів про статус бронювання, зміни рейсів та інші транзакційні події забезпечується інтеграцією з email-провайдером Mailjet та хмарними сервісами Firebase.

Адміністративна частина рішення являє собою повноцінну back-office систему, у якій менеджери оператора керують маршрутами, рейсами, продажем квитків, програмою лояльності, а також формують аналітичні звіти та контролюють статуси рейсів. Окремий мобільний застосунок для водіїв забезпечує контроль за посадкою пасажирів через сканування QR-кодів на квитках, відстеження поточного маршруту та можливість продажу квитків безпосередньо в транспортному засобі.

Важливою рисою Grand Bus є те, що сервіс реалізовано як власну закриту екосистему перевізника без публічного прикладного програмного інтерфейсу для зовнішньої інтеграції. Продаж квитків здійснюється виключно через власні цифрові канали оператора, що дозволяє уникнути комісій агрегаторів та зберегти повний контроль над клієнтським досвідом, але водночас обмежує можливість сторонніх сервісів інтегруватися в систему.

Огляд сервісу Sharry. Sharry – мультимодальна онлайн-платформа для бронювання наземних пасажирських перевезень, що оперує переважно у європейському регіоні та об'єднує понад тисячу перевізників у понад п'ятдесяти країнах. На відміну від моделі прямого продавця, Sharry виступає в ролі агрегатора, який збирає пропозиції різних перевізників в єдиному пошуковому інтерфейсі, дозволяючи пасажирові порівнювати ціни, графіки відправлень, рівень комфорту та умови оплати в межах одного запиту. Інтерфейс пошуку рейсів сервісу наведено на рисунку 1.2.

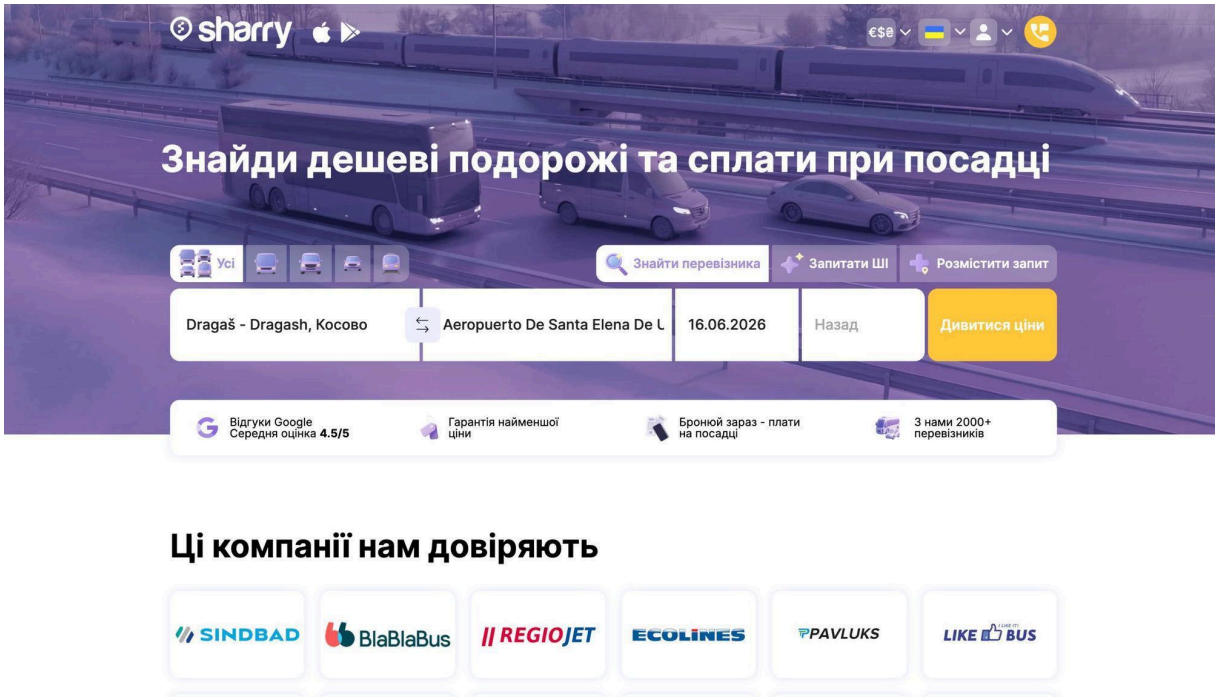

Рисунок 1.2 – Інтерфейс пошуку рейсів сервісу Sharry

Платформа одночасно підтримує кілька видів наземного транспорту: автобуси, мікроавтобуси, потяги та попутні поїздки за моделлю carpooling. Така мультимодальність робить Sharry зручним інструментом саме для міжнародних маршрутів, де часто доводиться поєднувати різні типи транспорту або обирати з-поміж кількох доступних альтернатив у межах одного напрямку.

Однією з основних відмінних особливостей сервісу є гнучка модель оплати, що поєднує стандартну онлайн-оплату карткою, цифровими гаманцями та банківським переказом з унікальним механізмом «pay on boarding» — оплатою безпосередньо під час посадки в транспортний засіб. Така схема знижує бар'єр для оформлення бронювання, оскільки пасажир може зарезервувати місце без миттєвого списання коштів, що особливо цінне у разі високої ймовірності зміни планів. Поряд із базовими сценаріями оплати реалізовано часткову передоплату та механізм «buy now, pay later», що додатково розширює доступність послуги.

Серед додаткових функцій Sharry варто виділити аукціонну модель «get a quote», у межах якої пасажир може запитувати індивідуальну ціну від перевізників на конкретний маршрут і дату; послугу door-to-door, що передбачає забір та висадку у обраних пасажиром точках без потреби стандартних пересадок; систему

персонального диспетчерського супроводу, у якій кожному пасажирові призначається спеціаліст для допомоги протягом усього процесу – від бронювання до посадки; а також AI-асистента для планування поїздок, що допомагає у виборі оптимального маршруту та комбінації транспортних засобів.

З огляду на те, що Sharry оперує мережею незалежних перевізників, окрему увагу приділено перевірці їхньої благонадійності: кожен партнер проходить обов'язкову процедуру реєстрації, верифікації та фонової перевірки до публікації пропозицій на платформі. Система знижок та промоакцій реалізована як гнучка, з можливістю індивідуального налаштування під різні категорії клієнтів і види маршрутів.

З точки зору архітектурної моделі, Sharry являє собою B2C-маркетплейс із виразним інтеграційним рівнем для постачальників послуг – перевізників, які підключаються до платформи через відповідні внутрішні інтерфейси. Однак публічного відкритого API, який дозволяв би стороннім розробникам інтегруватися із сервісом для створення власних клієнтських застосунків, на момент проведення аналізу не виявлено: інтеграція пропонується переважно у межах афілійованих партнерських програм.

1.2 Порівняльний аналіз існуючих рішень

Для глибшого розуміння переваг та обмежень розглянутих сервісів і визначення місця, яке може зайняти запропоноване програмне рішення на ринку, виконано порівняльний аналіз ключових функціональних і архітектурних характеристик. Аналіз ґрунтується на функціональному фокусі: способах організації бронювання, моделі взаємодії з пасажирами та перевізниками, способах оплати, рівні автоматизації операційних процесів, а також можливостях інтеграції із зовнішніми інформаційними системами.

Переваги сервісу Grand Bus:

– власна цілісна цифрова екосистема, яка включає вебсайт, мобільний застосунок для пасажирів та окремий мобільний застосунок для водіїв, що

забезпечує повний контроль перевізника над клієнтським досвідом і операційними процесами;

– прямий продаж квитків через власні цифрові канали без використання сторонніх агрегаторів, що дозволяє уникнути додаткових комісій та зберегти повну маржу від кожного бронювання;

– глибока інтеграція з локальним платіжним шлюзом Liqpay, оптимізована під особливості українського ринку безготівкових платежів;

– реалізована вертикальна програма лояльності з нарахуванням балів, що сприяє формуванню повторних продажів та підвищенню лояльності клієнтської бази;

– автоматизація операційних процесів водія через мобільний застосунок із підтримкою сканування QR-кодів квитків та відстеження поточного маршруту;

– диференціація сервісу за рівнями (стандартний та преміум-клас), що створює додатковий канал монетизації та розширює сегментну пропозицію;

– інтеграція з email-провайдером Mailjet та хмарними сервісами Firebase для забезпечення транзакційних і push-сповіщень.

Недоліки сервісу Grand Bus:

– закритий характер екосистеми та відсутність публічного програмного інтерфейсу для зовнішньої інтеграції, що унеможливлює використання сервісу як основи для побудови сторонніх клієнтських застосунків;

– обмежена географія обслуговування (близько тринадцяти міст у чотирьох країнах), яка не дозволяє конкурувати з агрегаторами у плані широти пропозиції маршрутів;

– прив'язка пасажира до одного перевізника: у разі скасування рейсу або відсутності місць немає можливості автоматично запропонувати альтернативу від іншого оператора;

– розширення мережі маршрутів вимагає значних капітальних інвестицій у власний автопарк, диспетчерську службу та водіїв, що обмежує темпи зростання;

– орієнтація переважно на регулярні рейси з фіксованим розкладом і обмежена підтримка індивідуальних трансферів за нестандартними маршрутами;

–Власна розробка та підтримка cross-platform мобільних застосунків вимагають окремих команд інженерів і додаткових ресурсних витрат на постійний супровід.

Сервіс Grand Bus демонструє високий рівень опрацювання вертикальної моделі, у якій перевізник самостійно контролює всі етапи виконання послуги — від онлайн-продажу квитка до посадки пасажира та виконання рейсу. Така модель оптимальна для одного перевізника з чітко визначеною мережею маршрутів, але стає обмеженням, коли постає потреба інтегрувати в систему додаткові маршрути сторонніх операторів або відкрити дані для побудови екосистемних рішень.

Переваги сервісу Sharry:

– мультимодальна модель, що поєднує автобуси, мікроавтобуси, потяги та попутні поїздки в одному пошуковому інтерфейсі і дозволяє пасажирові обирати оптимальний варіант за різними критеріями;

– широка географія обслуговування – понад тисяча перевізників у понад п'ятдесяти країнах Європи та Туреччини, що значно перевищує можливості будь-якого окремого оператора;

– диверсифікований ризик завдяки мережі незалежних перевізників: у разі недоступності рейсу одного з операторів пасажирові можна запропонувати альтернативи від інших;

– гнучкі способи оплати, включно з унікальною моделлю «pay on boarding», що дозволяє оплачувати квиток безпосередньо при посадці, а також підтримкою «buy now, pay later» та банківських переказів;

– аукціонна модель «get a quote» для запиту індивідуальних цін від перевізників, що особливо цінне для нестандартних маршрутів;

– послуга door-to-door з адресним забором та висадкою пасажира, що знімає типовий бар'єр з пересадками між видами транспорту;

– AI-асистент для планування поїздок, що автоматизує процес підбору оптимального маршруту та комбінації транспортних засобів.

Недоліки сервісу Sharry:

– опосередкований контроль над якістю сервісу: оскільки перевезення виконують треті сторони, платформа залежить від рівня обслуговування кожного окремого перевізника, що ускладнює забезпечення стабільного клієнтського досвіду;

– відсутність публічного відкритого API для сторонніх розробників: інтеграційні можливості пропонуються переважно у межах афілійованих партнерських програм і не дозволяють побудувати на платформі повноцінну власну клієнтську систему;

– орієнтація на регулярні маршрути та частково – на попутні поїздки, з обмеженою підтримкою специфіки приватних чартерних трансферів і групового замовлення транспорту;

– складність процесу onboarding нових перевізників, який вимагає верифікації, фонової перевірки та узгодження умов співпраці;

– обмежена B2B-функціональність: платформа спрямована переважно на B2C-сегмент і не пропонує гнучких інструментів для корпоративних клієнтів, які потребують централізованого керування корпоративними поїздками.

Сервіс Sharry, у свою чергу, реалізує горизонтальну агрегаторську модель, яка завдяки мережевому ефекту забезпечує широту пропозиції, недосяжну для окремого перевізника. Однак ця модель не передбачає глибокої кастомізації клієнтського досвіду під потреби конкретних бізнес-партнерів і не пропонує відкритих інтеграційних інструментів для розробки власних застосунків поверх платформи.

Узагальнені результати порівняльного аналізу за основними критеріями наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння функціональних та архітектурних характеристик сервісів Grand Bus і Sharry

Критерій	Grand Bus	Sharry
Бізнес-модель	Прямий перевізник	Мультикар'єрний агрегатор
Географія обслуговування	4 країни, 13 міст	50+ країн, 1000+ перевізників
Підтримувані види транспорту	Автобуси, мікроавтобуси	Автобус, мікроавтобус, потяг, попутка
Індивідуальні чартерні трансфери	Не підтримуються	Частково (через get-a-quote)
Регулярні групові рейси	Підтримуються	Підтримуються
Власні мобільні застосунки	Для пасажира і водія	Для пасажира
Сканування QR-квитків	Реалізовано	Не реалізовано на рівні платформи
Способи оплати	Liqpay (картка, гаманці)	10+ методів, BNPL, pay-on-boarding
Програма лояльності	Реалізована (бали)	Знижки, промоакції
Door-to-door	Не підтримується	Підтримується
AI-функції	Відсутні	AI-асистент для планування
Публічний API для B2B-інтеграції	Відсутній	Відсутній (тільки афілійований доступ)
Інтеграція зі сторонніми CRM/ERP-системами	Не задокументована	Не задокументована

Проведений порівняльний аналіз виявив, що обидва розглянуті сервіси пропонують високий рівень опрацювання своїх вертикальних бізнес-моделей, проте жоден із них не забезпечує відкритого програмного інтерфейсу, який дозволив би стороннім розробникам та компаніям-перевізникам інтегрувати функціональність бронювання у власні клієнтські застосунки, корпоративні портали або системи обліку. Кожна з платформ є замкнутою екосистемою, спроєктованою для обслуговування власних бізнес-процесів, що залишає на ринку незаповнену нішу — потребу в гнучкому, документованому та відкритому програмному інтерфейсі для бронювання приватних трансферів. Запропоноване у даній роботі програмне рішення, реалізоване як вебсервіс з REST API та автоматично згенерованою специфікацією OpenAPI, спрямоване саме на задоволення цієї потреби та надає можливість компаніям у домені приватних

перевезень інтегрувати функціональність бронювання у власні цифрові продукти без необхідності розробки серверної логіки з нуля.

1.3 Обґрунтування вибору технологічного стеку

Вибір технологічного стеку є одним із визначальних рішень на етапі проєктування програмної системи, оскільки безпосередньо впливає на продуктивність, супроводжуваність, швидкість розробки та можливості подальшого масштабування продукту. Оскільки розроблюване рішення являє собою серверний вебсервіс із прикладним програмним інтерфейсом, орієнтований на інтеграцію зовнішніх клієнтів, основними критеріями добору технологій було обрано: продуктивність опрацювання конкурентних запитів, наявність засобів автоматичної генерації документації API, підтримку статичної типізації та валідації даних, зрілість екосистеми, а також сумісність з обраною хмарною інфраструктурою.

Як основну мову програмування для серверної частини обрано Python. Цей вибір зумовлений лаконічним і виразним синтаксисом мови, що пришвидшує розробку та знижує вартість супроводу коду, наявністю розвиненої екосистеми бібліотек для веброзробки, роботи з базами даних, асинхронного програмування та взаємодії з хмарними сервісами, а також широкою підтримкою з боку спільноти [1]. Починаючи з версії 3.5, Python надає механізм анотацій типів (англ. type hints), який, не змінюючи динамічної природи мови, дозволяє статично описувати очікувані типи даних і який використовується сучасними фреймворками для валідації та автоматичної генерації документації. Для розробки застосовано Python версії 3.12, що підтримує сучасні засоби асинхронного програмування на основі конструкцій async/await та модуля asyncio, критично важливих для побудови високопродуктивних вебсервісів з інтенсивним введенням-виведенням.

Для реалізації серверної частини розглянуто три найпоширеніші рішення в Python-екосистемі: повнофункціональний фреймворк Django разом із Django

REST Framework, мінімалістичний мікрофреймворк Flask та сучасний асинхронний мікрофреймворк FastAPI. Django REST Framework є зрілим і функціонально насиченим рішенням, проте побудований на синхронній моделі обробки запитів WSGI та тісно пов'язаний з власною об'єктно-реляційною моделлю (ORM) Django, що знижує гнучкість архітектурних рішень і створює надлишкову складність для проєкту, у якому потрібен лише програмний інтерфейс без серверного рендерингу вебсторінок. Flask, навпаки, є гранично мінімалістичним і не нав'язує структури, однак не містить вбудованих засобів для валідації даних, серіалізації та автоматичної генерації документації. FastAPI побудовано поверх асинхронного інструментарію Starlette та специфікації ASGI, завдяки чому він демонструє продуктивність, співмірну з рішеннями на Node.js та Go [2]. Ключовою перевагою FastAPI є органічне використання анотацій типів Python: на їх основі автоматично виконується валідація вхідних даних засобами Pydantic, серіалізація відповідей, а також генерується інтерактивна документація API згідно зі специфікацією OpenAPI 3.0 (інтерфейси Swagger UI та ReDoc). Остання властивість є особливо цінною, оскільки автоматично згенерована та завжди актуальна специфікація OpenAPI безпосередньо реалізує головну мету роботи – спрощення інтеграції зовнішніх клієнтських застосунків. Узагальнене порівняння розглянутих фреймворків наведено в таблиці 1.2.

Таблиця 1.2 – Порівняння вебфреймворків Python для розробки REST API

Критерій	**Django REST Framework**	**Flask**	**FastAPI**
Модель обробки запитів	Синхронна (WSGI)	Синхронна (WSGI)	Асинхронна (ASGI)
Відносна продуктивність	Середня	Середня	Висока
Валідація даних	Серіалізатори DRF	Сторонні розширення	Вбудована (Pydantic)
Автогенерація OpenAPI	Через розширення	Через розширення	Вбудована
Використання анотацій типів	Обмежене	Відсутнє	Нативне
Зв'язаність з ORM	Висока (Django ORM)	Відсутня	Відсутня
Поріг входження	Високий	Низький	Низький
Придатність для чистого API	Помірна	Помірна	Висока

За результатами порівняння для реалізації серверної частини обрано мікрофреймворк FastAPI як рішення, що забезпечує найкращий баланс продуктивності, швидкості розробки, типобезпеки та автоматичної документації API, що повністю відповідає поставленим перед системою вимогам.

Робота з базою даних. Для взаємодії з реляційною базою даних обрано бібліотеку SQLAlchemy версії 2.0 – найзріліший і найпоширеніший інструмент об'єктно-реляційного відображення в Python-екосистемі. SQLAlchemy 2.0 надає типобезпечний декларативний синтаксис опису моделей, підтримує асинхронний режим роботи через драйвер asyncpg, що узгоджується з асинхронною природою FastAPI, та за потреби дозволяє виконувати запити мовою, наближеною до SQL, зберігаючи контроль над генерованими запитами. Перевагою SQLAlchemy є відокремленість від конкретного вебфреймворку, що відповідає принципам шарової архітектури та полегшує тестування шару доступу до даних [3]. Для керування еволюцією схеми бази даних застосовано інструмент Alembic, тісно інтегрований із SQLAlchemy, який забезпечує версіонування схеми у вигляді послідовності міграцій, автоматичну генерацію міграцій на основі змін у моделях, а також безпечне застосування та відкат змін у різних середовищах (розробки, тестування, промислової експлуатації) [4].

Валідацію вхідних даних і серіалізацію відповідей реалізовано за допомогою бібліотеки Pydantic, яка є природним компонентом екосистеми FastAPI. Pydantic дозволяє декларативно описувати схеми даних у вигляді класів з анотаціями типів, автоматично перевіряє відповідність вхідних даних цим схемам і формує зрозумілі повідомлення про помилки валідації. Використання окремих схем для вхідних і вихідних даних (наприклад, BookingCreate для створення бронювання та BookingRead для його отримання)забезпечує чітке розмежування контрактів API і запобігає випадковому розкриттю внутрішніх полів моделей [5].

Як платформу розгортання обрано Amazon Web Services (AWS). Цей вибір зумовлений зрілістю та широтою екосистеми керованих сервісів, що покривають усі потреби проєкту — від контейнерного запуску застосунку та реляційного й нереляційного зберігання даних до обробки повідомлень, об'єктного сховища та

засобів моніторингу. Використання керованих сервісів AWS дозволяє делегувати платформі завдання адміністрування інфраструктури (резервне копіювання, масштабування, забезпечення відмовостійкості), зосередивши зусилля команди на розробці бізнес-логіки. Детальний опис застосованих сервісів AWS та обґрунтування конкретних архітектурних рішень наведено у підрозділі 2.2.

1.4 Вибір методології розробки

Розроблення вебсервісу виконувалося командою в умовах поступового уточнення вимог, що зумовило вибір гнучкої (Agile) методології організації робіт замість класичної каскадної моделі. Каскадна (водоспадна) модель передбачає послідовне проходження фаз аналізу, проєктування, реалізації та тестування з фіксацією вимог на початку проєкту й є ефективною лише за умови повної визначеності вимог, що рідко досягається в проєктах розробки нових цифрових продуктів. Натомість гнучка методологія ґрунтується на ітеративно-інкрементальному підході, за якого продукт розвивається невеликими приростами функціональності, а вимоги уточнюються протягом усього життєвого циклу на основі зворотного зв'язку.

В основу організації робіт покладено цінності та принципи Маніфесту гнучкої розробки програмного забезпечення (Agile Manifesto), що віддають пріоритет працездатному програмному забезпеченню, співпраці із замовником, реагуванню на зміни та безпосередній взаємодії учасників команди [6]. Практичну організацію процесу побудовано на основі фреймворку Scrum, у якому роботу поділено на короткі ітерації фіксованої тривалості – спринти. Кожен спринт охоплює повний цикл: планування обсягу робіт (Sprint Planning), щоденну синхронізацію учасників (Daily Sync), власне розробку та тестування інкременту, а також ретроспективу (Retrospective) для аналізу результатів і вдосконалення процесу. Узагальнену схему ітераційного циклу гнучкої розробки наведено на рисунку 1.3.

Беклог продукту формувався на основі сформульованих вимог і пріоритизувався з урахуванням бізнес-цінності та технічних залежностей; з кожної ітерації виокремлювався набір задач, які команда зобов'язувалася реалізувати протягом спринту. Такий підхід дозволив на ранніх етапах отримувати працездатні версії окремих підсистем (зокрема, бронювання та авторизації), перевіряти прийняті архітектурні рішення на практиці та своєчасно вносити корективи.

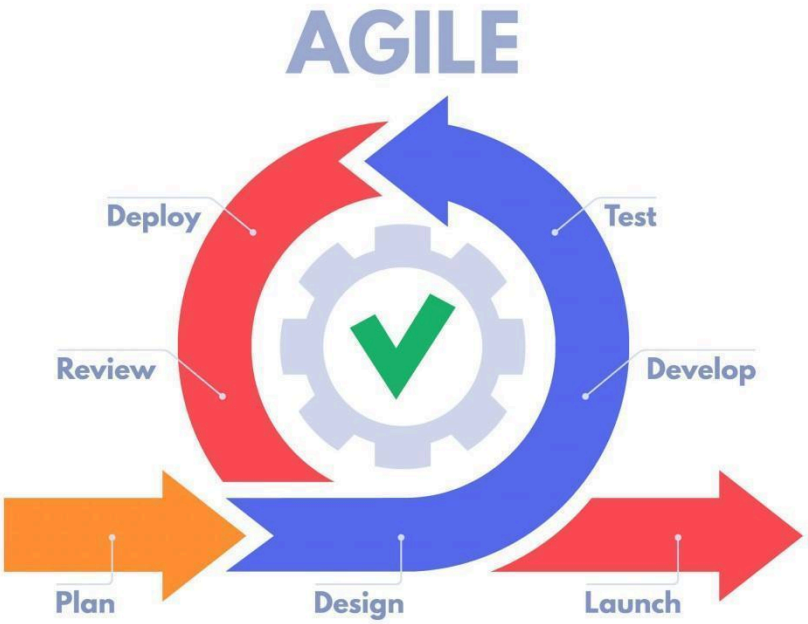

Рисунок 1.3 – Цикл методології Agile

Для підтримки робочих процесів команди застосовано такий набір інструментів. Планування та відстеження задач здійснювалися в системі управління проєктами Jira за допомогою канбан- та scrum-дощок, що забезпечували прозорість статусу кожної задачі та контроль обсягу робіт у межах спринту. Проєктна та технічна документація велася в системі Confluence, інтегрованій із Jira. Оперативне спілкування учасників відбувалося в месенджері Telegram, а планові зустрічі, демонстрації та ретроспективи проводилися за допомогою відеоконференцзв'язку Google Meet. Поєднання цих інструментів

забезпечило ефективну координацію розподіленої команди та документування ухвалених рішень.

1.5 Формування вимог

На основі проведеного аналізу предметної області та обраних бізнес-сценаріїв сформовано перелік вимог до розроблюваного вебсервісу. Вимоги поділено на функціональні, що описують конкретну поведінку та можливості системи, і нефункціональні, що визначають якісні характеристики й обмеження, яким система має відповідати. Формування вимог виконувалося відповідно до рекомендацій зводу знань з програмної інженерії SWEBOK і фіксувалося у документі вимог до продукту (PRD) та супровідних технічних специфікаціях [7].

Функціональні вимоги. Функціональні вимоги до системи згруповано за основними підсистемами:

1. Реєстрація та автентифікація користувачів. Система повинна підтримувати три сценарії доступу: реєстрацію та вхід клієнта за номером телефону з підтвердженням одноразовим кодом (OTP), вхід водія та вхід користувачів адміністративних ролей. Сесії користувачів мають підтримуватися на серверній стороні з можливістю їх анулювання.

2. Розмежування прав доступу (RBAC). Система повинна реалізовувати рольову модель доступу із щонайменше чотирма ролями – адміністратор, менеджер, водій та клієнт – з обмеженням доступу до операцій і даних на рівні програмного інтерфейсу.

3. Бронювання. Система повинна забезпечувати оформлення бронювань двох типів – групового трансферу (регулярний рейс із вибором місця) та індивідуального трансферу – з відображенням схеми салону, вибором місця, його тимчасовим блокуванням на час оформлення та запобіганням одночасному бронюванню одного місця різними користувачами.

4. Керування автопарком і конфігурацією салону. Система повинна підтримувати ведення довідника транспортних засобів та опис конфігурації салону (схеми розміщення місць), що використовується під час бронювання.

5. Керування маршрутами та рейсами. Система повинна забезпечувати ведення маршрутів, проміжних зупинок та рейсів із розкладом, а також пошук доступних рейсів за заданими параметрами поїздки.

6. Обробка платежів. Система повинна інтегруватися з платіжним сервісом (MonoPay) для приймання оплати за бронюваннями ,асинхронно опрацьовувати сповіщення (webhook) про результат платежу та оновлювати статус бронювання з дотриманням ідемпотентності.

7. Омніканальні комунікації. Система повинна забезпечувати інформування користувачів про статус бронювання та події рейсу через інтеграцію з комунікаційною платформою (Binotel) і набір каналів сповіщення.

8. Формування електронних квитків. Система повинна генерувати електронні квитки з QR-кодом для контролю посадки, з можливістю додавання квитка до цифрових гаманців Apple Wallet та Google Wallet.

Діаграма прецедентів визначає функціональні можливості системи з погляду двох основних дійових осіб – клієнта (Customer) та водія (Driver). Клієнт виконує реєстрацію та вхід за одноразовим кодом, пошук маршрутів і рейсів, бронювання місця, оплату, отримання електронного квитка та скасування поїздки з поверненням коштів. Водій авторизується, переглядає призначені рейси, сканує QR-коди квитків під час посадки та формує звіт для кожного рейсу. Діаграму прецедентів наведено на рисунку 1.4.

Рисунок 1.4 – Діаграма прецедентів (use-case)

Нефункціональні вимоги. Нефункціональні вимоги визначають якісні характеристики системи:

1. Продуктивність. Час відповіді програмного інтерфейсу для типових операцій не повинен перевищувати 500 мс за номінального навантаження.

2. Конкурентний доступ. Система повинна коректно опрацьовувати одночасні спроби бронювання одного місця, застосовуючи механізм короткочасного блокування місця та запобігаючи виникненню стану гонитви (race condition).

3. Безпека. Система повинна відповідати вимогам захисту платіжних даних (PCI DSS) і захисту персональних даних (GDPR), застосовувати серверну

стратегію зберігання токенів сеансу та забезпечувати контроль доступу згідно з рольовою моделлю.

4. Надійність. Цільовий рівень доступності сервісу становить 99,9 %; для зовнішніх інтеграцій мають передбачатися механізми повторних спроб (retry) та ідемпотентності операцій.

5. Масштабованість. Архітектура повинна підтримувати горизонтальне масштабування серверної частини за допомогою хмарної платформи без зміни прикладного коду.

6. Сумісність та інтегровність. Програмний інтерфейс має відповідати архітектурному стилю REST і супроводжуватися актуальною специфікацією OpenAPI 3.0 для спрощення інтеграції сторонніх клієнтів.

7. Спостережуваність. Система повинна забезпечувати централізоване журналювання подій, збір метрик та аудит ключових операцій засобами моніторингу хмарної платформи.

Сформульовані вимоги є основою для проєктування архітектури системи та моделі даних, що детально розглядається у другому розділі кваліфікаційної роботи.

1.6 Висновки до першого розділу

У першому розділі проаналізовано предметну область та ринок приватних пасажирських перевезень на прикладі сервісів Grand Bus і Sharry; порівняльний аналіз показав відсутність у наявних рішеннях відкритого документованого програмного інтерфейсу для інтеграції сторонніх клієнтів, що визначає актуальність роботи. Обґрунтовано вибір технологічного стеку (Python, мікрофреймворк FastAPI, SQLAlchemy 2.0, Pydantic, хмарна платформа AWS) та гнучкої методології розробки Agile, а також сформульовано функціональні й нефункціональні вимоги до вебсервісу, які стали основою для проєктування системи у наступному розділі.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ

У цьому розділі описано процес проєктування та розробки програмного забезпечення вебсервісу. Обґрунтовано вибір архітектурної моделі програмного рішення, розглянуто організацію хмарної інфраструктури на платформі Amazon Web Services, наведено комплекс UML-діаграм, що формалізують структуру та поведінку системи, описано систему контролю версій і автоматизований конвеєр доставки (CI/CD), а також висвітлено особливості реалізації серверної логіки та стратегію тестування розробленого рішення.

2.1 Вибір архітектури проєкту

Архітектура програмної системи визначає спосіб декомпозиції її на складові частини, характер взаємодії між ними та розподіл відповідальності, безпосередньо впливаючи на супроводжуваність, тестованість і здатність системи до подальшого розвитку. На етапі проєктування розглянуто три типові архітектурні підходи: монолітну архітектуру, мікросервісну архітектуру та шарову (багаторівневу) модульну архітектуру.

Монолітна архітектура передбачає реалізацію всієї функціональності у вигляді єдиного нерозривного застосунку. Такий підхід простий у початковій розробці та розгортанні, проте з ростом кодової бази ускладнює супровід, призводить до сильного зв'язування компонентів і обмежує можливість незалежного масштабування окремих частин системи. Мікросервісна архітектура, навпаки, передбачає поділ системи на множину незалежних сервісів, що розгортаються та масштабуються окремо; вона забезпечує високу гнучкість, але вносить значну операційну складність – потребу в оркестрації, міжсервісній комунікації, розподілених транзакціях і складнішому моніторингу, що є надмірним для проєкту на стадії MVP.

Для розроблюваного вебсервісу обрано шарову модульну архітектуру, яка є оптимальним компромісом між простотою моноліту та гнучкістю мікросервісів.

Систему поділено на три логічні шари з чітко визначеною відповідальністю: шар інтерфейсу запитів (api) містить FastAPI-роутери, Pydantic-схеми, мапери та залежності; шар бізнес-логіки (services) інкапсулює правила предметної області та визначає інтерфейси репозиторіїв; шар доступу до даних (infrastructure) містить моделі SQLAlchemy, конкретні реалізації репозиторіїв та клієнти зовнішніх і хмарних сервісів. Залежності спрямовані всередину – від зовнішніх шарів до внутрішніх, – що відповідає принципам інверсії залежностей та чистої архітектури і забезпечує слабке зв'язування, високу тестованість і можливість у майбутньому виокремити окремі модулі у самостійні мікросервіси без переписування бізнес-логіки [8]. Узагальнене порівняння розглянутих підходів наведено в таблиці 2.1, а структуру шарів — на рисунку 2.1.

Таблиця 2.1 – Порівняння архітектурних підходів

Критерій	Моноліт	Мікросервіси	Шарова (обрано)
Складність розробки	Низька	Висока	Помірна
Операційна складність	Низька	Висока	Низька
Зв'язування компонентів	Сильне	Слабке	Слабке
Тестованість	Помірна	Висока	Висока
Незалежне масштабування	Обмежене	Повне	Часткове
Придатність для MVP	Висока	Низька	Висока

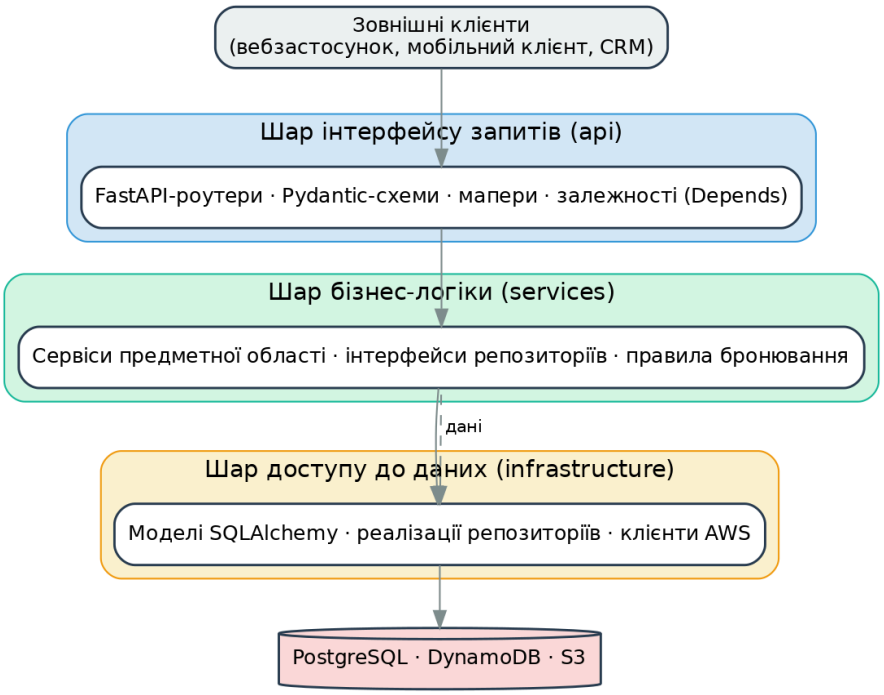

Рисунок 2.1 – Шарова архітектура програмного рішення

Шарова архітектура реалізує принципи SOLID, зокрема принцип інверсії залежностей: внутрішні шари визначають абстрактні інтерфейси (контракти), а зовнішні шари надають їхні конкретні реалізації. Це дозволяє замінювати реалізації (наприклад, сховище даних або клієнт зовнішнього сервісу) без зміни бізнес-логіки та спрощує модульне тестування шляхом підстановки тестових реалізацій.

Для організації коду застосовано низку усталених шаблонів проєктування: репозиторій для інкапсуляції доступу до даних, ін'єкцію залежностей через механізм FastAPI Depends для надання сервісів і репозиторіїв обробникам запитів, а також шаблон «одиниця роботи» (unit of work) для керування транзакціями бази даних у межах одного запиту. Такий підхід підвищує тестованість, повторне використання коду та зрозумілість структури проєкту.

Порівняно з монолітною архітектурою, шарова модель полегшує паралельну роботу над різними шарами та локалізує наслідки змін, а порівняно з мікросервісною – уникає накладних витрат на міжсервісну взаємодію й розподілені транзакції, що є надмірними для поточного масштабу системи. За потреби окремі модулі бізнес-логіки можуть бути виокремлені у самостійні сервіси без переписування доменного коду.

Кожен шар має чітко окреслену зону відповідальності та взаємодіє лише із суміжними шарами через визначені інтерфейси. Шар api відповідає за приймання HTTP-запитів, перевірку прав доступу та валідацію вхідних даних; шар services реалізує правила предметної області, не залежачи від конкретних технологій зберігання; шар infrastructure інкапсулює деталі роботи з базою даних і зовнішніми сервісами. Такий поділ відповідальності спрощує супровід і дозволяє змінювати окремі частини системи незалежно одна від одної.

Обробка вхідного запиту відбувається послідовно через шари: роутер шару api приймає HTTP-запит, перевіряє автентифікацію та права доступу і валідує тіло запиту за відповідною Pydantic-схемою; після цього керування передається сервісу шару services, який виконує бізнес-правила, звертаючись до сховищ через інтерфейси репозиторіїв; конкретні реалізації репозиторіїв у шарі infrastructure

виконують операції з базою даних та зовнішніми сервісами. Сформований результат серіалізується та повертається клієнту у відповідь.

Така організація мінімізує зв'язування між технічними деталями та бізнес-логікою: зміна способу зберігання даних або постачальника зовнішнього сервісу не потребує модифікації доменного коду, а лише заміни відповідної реалізації у шарі infrastructure. Це підвищує супроводжуваність системи та полегшує її автоматизоване тестування.

2.2 Хмарна інфраструктура AWS

Розгортання вебсервісу здійснено за хмарною моделлю на платформі Amazon Web Services із застосуванням набору керованих сервісів, що дозволяє делегувати платформі завдання адміністрування інфраструктури та зосередитися на розробці бізнес-логіки [9]. Загальну схему хмарної архітектури наведено на рисунку 2.2.

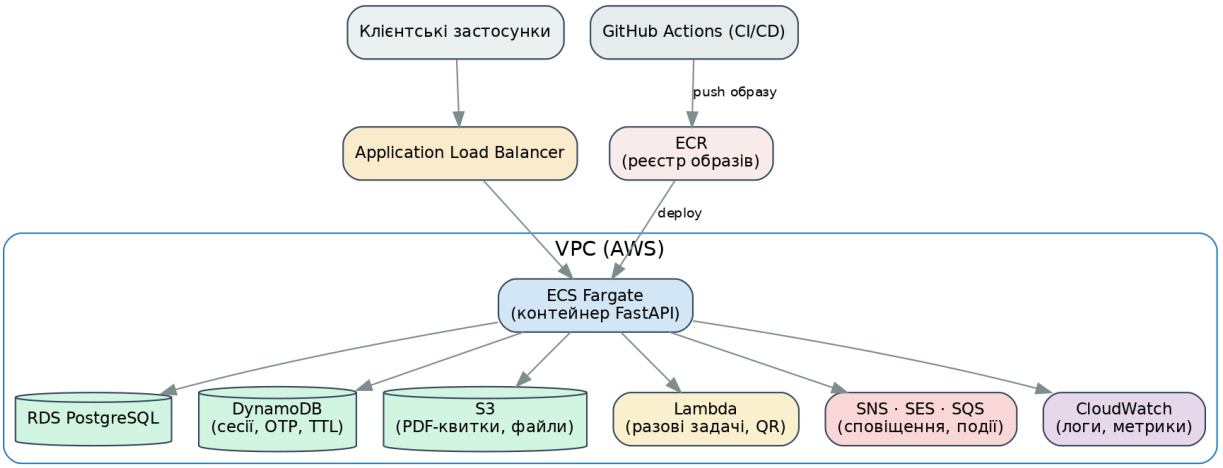

Рисунок 2.2 – Архітектура хмарної інфраструктури AWS

ECS Fargate. Серверну частину запущено у вигляді контейнера в середовищі ECS Fargate – безсерверному (serverless) рушії контейнерного запуску, який позбавляє від необхідності адмініструвати віртуальні машини EC2: керування обчислювальними ресурсами, їх масштабування та підтримку виконує платформа.

Це спрощує експлуатацію та забезпечує горизонтальне масштабування серверної частини за навантаженням [10].

Application Load Balancer. Вхідні запити надходять до контейнера FastAPI через балансувальник навантаження ALB, який розподіляє трафік між екземплярами сервісу та перевіряє їхню працездатність. Застосування ALB без проміжного API Gateway спрощує маршрутизацію та знижує затримку обробки запитів.

RDS PostgreSQL. Основні реляційні дані системи – користувачі, маршрути, рейси, транспорт, бронювання, квитки та платежі – зберігаються у керованій реляційній базі даних RDS PostgreSQL, яка забезпечує автоматичне резервне копіювання, оновлення та відмовостійкість.

DynamoDB (як сховище сесій). Ключовим архітектурним рішенням є використання NoSQL-сховища DynamoDB для зберігання серверних сесійних токенів та одноразових кодів (OTP) замість окремого кRedis-кластера DynamoDB є повністю керованим AWS-native сервісом і підтримує механізм автоматичного спливання записів за часом життя (TTL), що природно реалізує закінчення дії токенів і кодів без додаткового коду. Такий підхід зменшує кількість інфраструктурних компонентів, які потребують супроводу. Застосовано стратегію непрозорих (opaque) серверних токенів зі станом на стороні сервера: на відміну від самодостатніх JWT-токенів, серверні токени дозволяють миттєво відкликати сесію (розв'язують проблему revocation), а порівняно зі службою Cognito – забезпечують повний контроль над сценаріями автентифікації та уникнення прив'язки до постачальника (vendor lock-in) [11].

S3. Об'єктне сховище S3 використовується для зберігання згенерованих PDF-квитків, чеків та вкладень, що надходять з каналів комунікації, з можливістю надання тимчасових посилань для завантаження [12].

Lambda. Функції AWS Lambda застосовуються для виконання разових та фонових завдань — зокрема генерації QR-кодів і пакетної обробки — які не потребують постійно запущеного обчислювального ресурсу [13].

SNS, SES та SQS. Для транзакційних сповіщень і подієво-орієнтованої обробки задіяно сервіси SNS (зокрема для надсилання SMS), SES (електронна пошта) та чергу повідомлень SQS, що забезпечує надійну асинхронну взаємодію між компонентами.

CloudWatch. Централізований збір логів, метрик та налаштування сповіщень виконується засобами CloudWatch, що забезпечує спостережуваність системи в експлуатації.

Безпека інфраструктури. Розмежування доступу до ресурсів реалізовано через політики IAM, мережеву ізоляцію — засобами VPC та груп безпеки (Security Groups), а зберігання секретів і параметрів підключення — у сервісі AWS Secrets Manager. Доставку контейнерних образів організовано через реєстр ECR у складі CI/CD-конвеєра, що детально розглядається у підрозділі 2.4.

Вибір безсерверних і керованих сервісів зумовлений прагненням мінімізувати операційні витрати на адміністрування інфраструктури та забезпечити еластичність системи. Модель оплати за фактичне споживання ресурсів дозволяє оптимізувати вартість експлуатації, а вбудовані механізми резервування та реплікації керованих сервісів підвищують відмовостійкість порівняно із самостійним адмініструванням аналогічних компонентів.

Компонентна діаграма відображає взаємодію шарів вебсервісу (api, services, infrastructure) між собою, із сховищами даних (RDS PostgreSQL, DynamoDB, S3) та зовнішніми сервісами (MonoPay, Binotel, SMS-шлюз). Структуру компонентів наведено на рисунку 2.3.

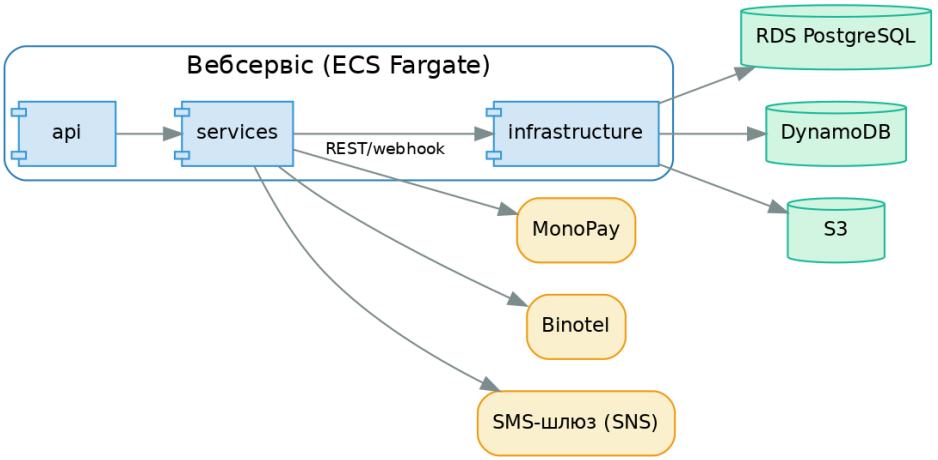

Рисунок 2.3 – Компонентна діаграма системи

Мережеву ізоляцію ресурсів забезпечує віртуальна приватна хмара (VPC): база даних RDS та інші внутрішні компоненти розміщуються в приватних підмережах без прямого доступу з мережі Інтернет, а зовнішній трафік приймає лише балансувальник навантаження в публічній підмережі. Доступ між компонентами регулюється групами безпеки, які дозволяють лише необхідні з'єднання.

Розмежування повноважень побудовано за принципом найменших привілеїв: кожному компоненту через ролі IAM надаються лише ті дозволи, що потрібні для його функціонування. Чутливі параметри (облікові дані бази даних, ключі доступу до платіжних і комунікаційних сервісів) зберігаються в AWS Secrets Manager і надаються застосунку під час виконання, що відповідає вимогам щодо захисту персональних і платіжних даних.

2.3 UML-діаграми послідовності роботи системи

Для формалізації структури та поведінки системи розроблено комплекс діаграм мовою UML, що охоплює функціональні можливості, ключові сценарії взаємодії, модель даних і компонентну організацію рішення.

Критичним з погляду коректності є сценарій бронювання, у якому необхідно запобігти одночасному оформленню одного місця різними користувачами (стан

гонитви). Під час створення бронювання сервіс намагається встановити короткочасне блокування обраного місця в DynamoDB із часом життя близько десяти хвилин; лише за умови успішного блокування виконується запис бронювання в PostgreSQL зі статусом очікування оплати. Якщо місце вже заблоковане, клієнт отримує відповідну відмову. Послідовність взаємодії наведено на рисунку 2.4.

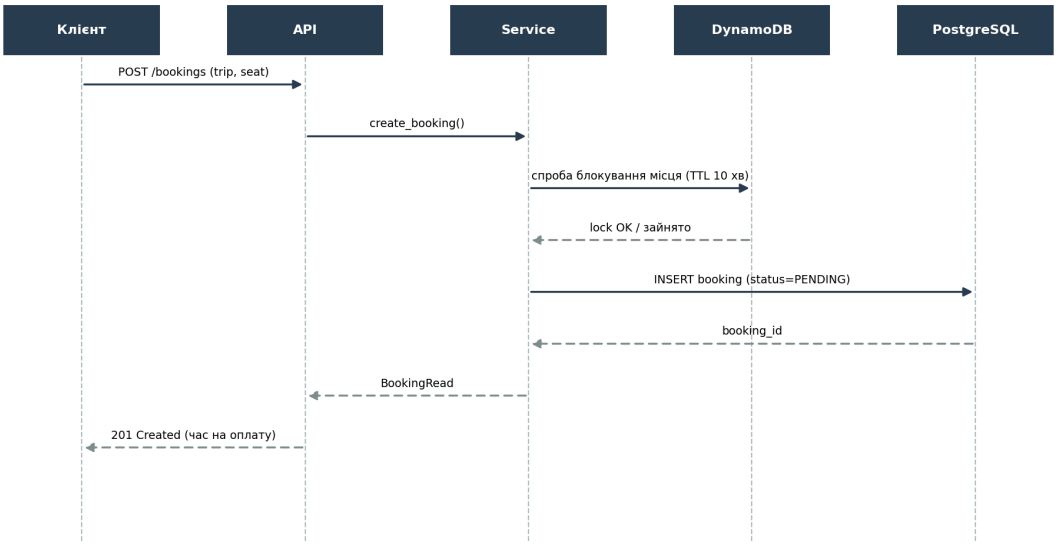

Рисунок 2.4 – Діаграма послідовності «Бронювання з блокуванням місця»

Автентифікація клієнта виконується за номером телефону з підтвердженням одноразовим кодом. Сервіс генерує код, зберігає його в DynamoDB з обмеженим часом життя та надсилає клієнту SMS через SNS. Після введення коду виконується його перевірка, і у разі успіху клієнтові видається серверний сесійний токен. Сценарій наведено на рисунку 2.5.

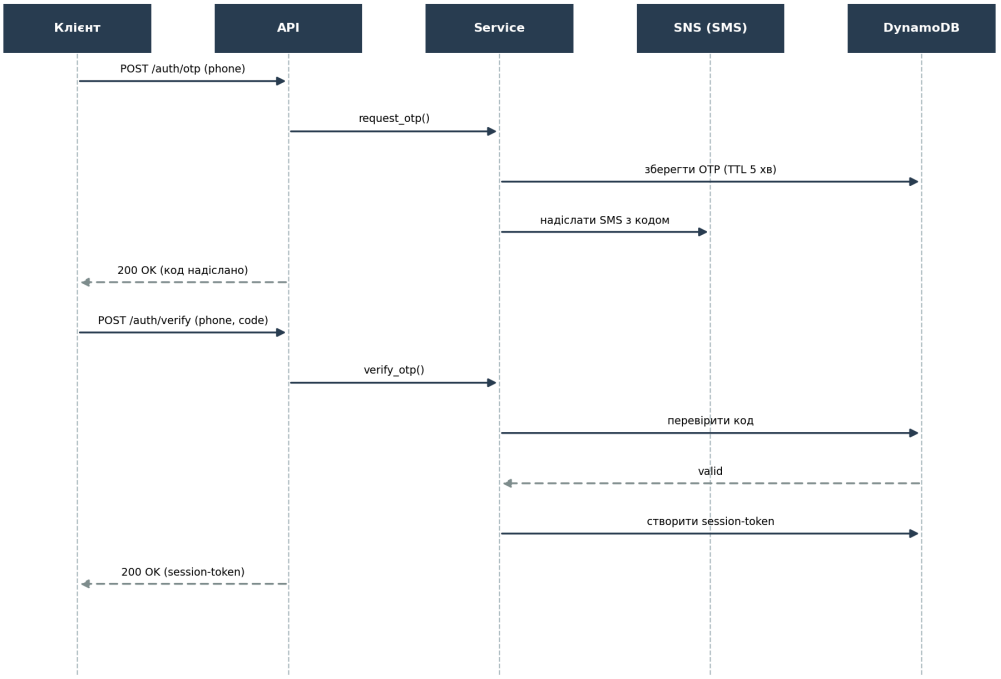

Рисунок 2.5 – Діаграма послідовності «Авторизація через OTP»

Оплата бронювання реалізована за асинхронною моделлю. Сервіс створює інвойс у платіжному сервісі MonoPay та повертає клієнту посилання для оплати. Після завершення оплати MonoPay надсилає сповіщення (webhook), яке опрацьовується ідемпотентно — повторні сповіщення не призводять до подвійного оновлення — після чого оновлюються статус бронювання та статус платежу. Сценарій наведено на рисунку 2.6.

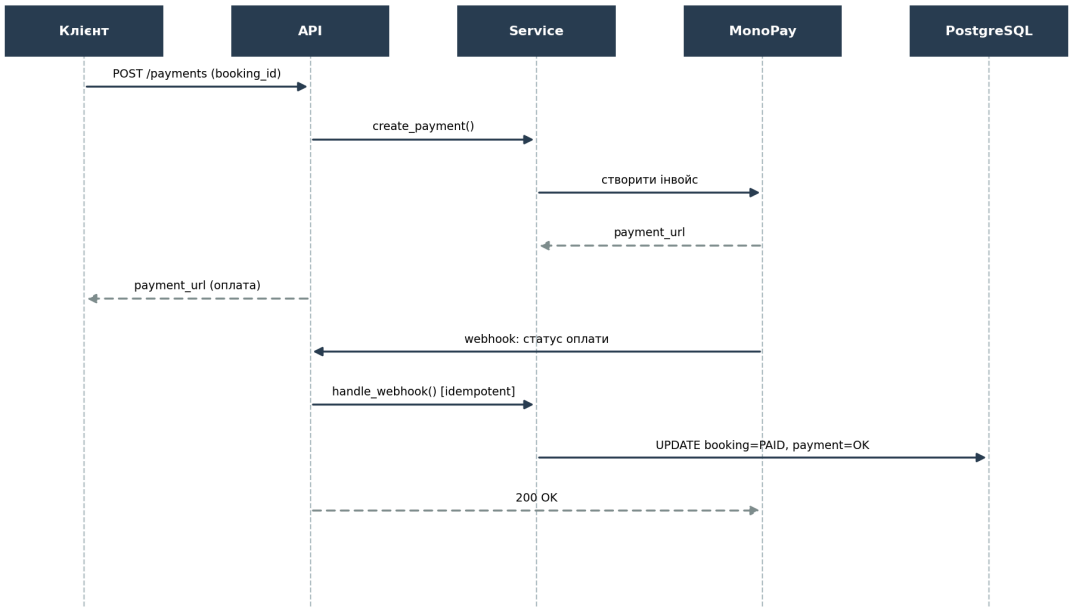

Рисунок 2.6 – Діаграма послідовності «Оплата через MonoPay»

Розроблені UML-діаграми використовуються як єдине джерело істини щодо структури та поведінки системи й узгоджуються між учасниками команди до початку реалізації. Діаграми послідовності деталізують найскладніші сценарії взаємодії, що знижує ризик неоднозначного трактування вимог під час програмування.

Сукупність розроблених діаграм охоплює як статичні аспекти системи (структуру даних і компонентів), так і динамічні (сценарії взаємодії у часі). Діаграма прецедентів задає межі системи та ролі користувачів, діаграми послідовності розкривають порядок обміну повідомленнями між складовими під час виконання ключових операцій, а діаграма сутностей і компонентна діаграма формалізують відповідно модель даних та структуру програмних модулів і їхніх залежностей.

Діаграма прецедентів окреслює функціональні межі системи та розмежовує можливості ролей: клієнт здійснює пошук рейсів, бронювання, оплату та отримання квитка, тоді як водій працює з призначеними рейсами та контролює посадку. Діаграми послідовності для критичних сценаріїв (бронювання з блокуванням місця, авторизація за одноразовим кодом, оплата з асинхронним

підтвердженням) деталізують обмін повідомленнями між клієнтом, прикладним інтерфейсом, сервісами та сховищами, що унеможливлює неоднозначне трактування логіки під час реалізації.

2.4 Проєктування бази даних

Проєктування бази даних виконано на основі сформованої моделі предметної області. Реляційну схему нормалізовано до третьої нормальної форми, що усуває надлишковість даних і забезпечує їхню цілісність. Основними сутностями системи є користувач, маршрут, рейс, транспортний засіб, місце, конфігурація салону, бронювання, квиток і платіж; сутності та зв'язки між ними відображено на діаграмі сутностей, наведеній у підрозділі «UML-діаграми» (рисунок 2.7).

Модель даних системи описує основні сутності предметної області та зв'язки між ними. Користувач (User) узагальнює клієнтів і водіїв за допомогою поля ролі. Маршрут (Route) пов'язаний із рейсами (Trip), кожен з яких виконується конкретним транспортним засобом (Transport) із заданою конфігурацією місць (Seat). Бронювання (Booking) пов'язує користувача, рейс і місце та має асоційований квиток (Ticket) і платіж (Payment). Схему сутностей наведено на рисунку 2.7.

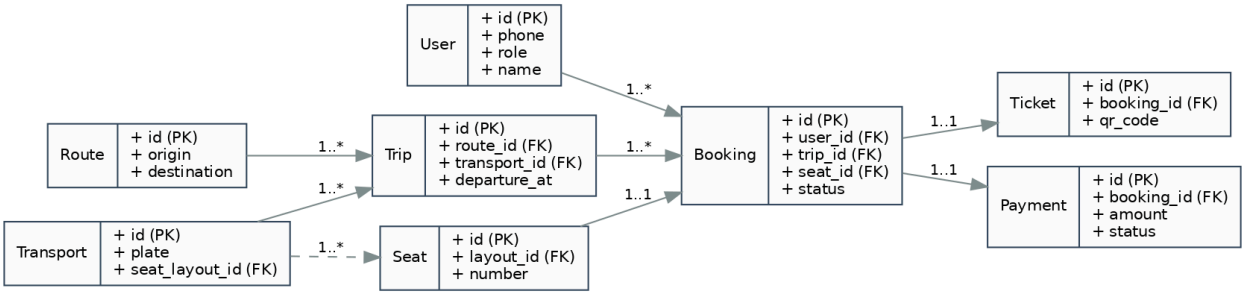

Рисунок 2.7 – Діаграма сутностей бази даних (ER)

Кожна таблиця має сурогатний первинний ключ, а зв'язки між таблицями реалізовано через зовнішні ключі з відповідними обмеженнями цілісності. Для

часто використовуваних умов пошуку (за рейсом, користувачем, статусом бронювання) передбачено індекси, що пришвидшують виконання запитів. Поля статусів бронювання та платежу реалізовано як перелічувані типи, що обмежують множину допустимих значень на рівні моделі та бази даних.

Сутність «Бронювання» є центральною та пов'язує користувача, рейс і конкретне місце, маючи асоційовані записи квитка та платежу. Така структура дозволяє відстежувати повний життєвий цикл бронювання – від створення й тимчасового блокування місця до оплати, формування електронного квитка та, за потреби, скасування з поверненням коштів. Цілісність під час конкурентного доступу до місць забезпечується поєднанням обмежень на рівні бази даних і механізму короткочасного блокування у нереляційному сховищі.

Еволюція схеми бази даних відстежується інструментом міграцій Alembic: кожна зміна моделей фіксується окремою ревізією, що дозволяє надійно застосовувати та за потреби відкочувати зміни в усіх середовищах розгортання. Це забезпечує узгодженість структури сховища між середовищами розробки, тестування та промислової експлуатації.

Узгодженість даних під час одночасних операцій забезпечується транзакційною моделлю реляційної бази даних: зміни в межах одного запиту виконуються атомарно й фіксуються лише в разі успішного завершення всіх операцій. Для критичного сценарію бронювання місця транзакційні гарантії бази даних доповнено механізмом короткочасного блокування у нереляційному сховищі, що разом унеможливлює подвійне бронювання навіть за високої конкуренції запитів.

Цілісність даних забезпечується сукупністю обмежень: обов'язковістю заповнення ключових полів, унікальністю природних ідентифікаторів (наприклад, номера телефону користувача) та посиланнями зовнішніх ключів з контролем каскадних операцій. Це запобігає появі некоректних або «осиротілих» записів у базі даних.

Конфігурація салону транспортного засобу моделюється окремо від конкретного рейсу: схема розміщення місць описується окремою сутністю та

повторно використовується для всіх рейсів цього транспортного засобу. Такий підхід уникає дублювання даних про місця та спрощує підтримку різних типів транспорту з відмінними компонуваннями.

2.5 Реалізація проєкту

Проєкт організовано відповідно до обраної шарової архітектури; структуру директорій вихідного коду наведено на рисунку 2.8.

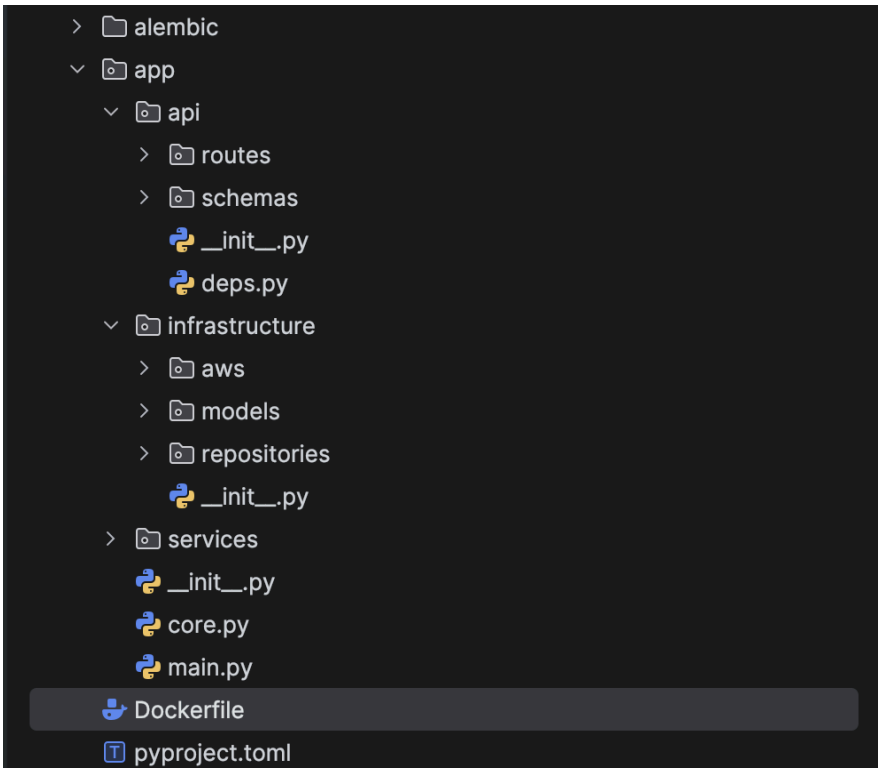

Рисунок 2.8 – Структура директорій проєкту

Серверну логіку реалізовано засобами FastAPI з дотриманням принципів шарової архітектури: роутери шару api приймають та валідують запити за допомогою Pydantic-схем і делегують опрацювання сервісам шару бізнес-логіки, які працюють із даними через інтерфейси репозиторіїв, реалізовані у шарі infrastructure. Розмежування прав доступу (RBAC) реалізовано через механізм залежностей FastAPI (Depends), який перевіряє роль користувача перед

виконанням захищених операцій. Робота із серверними сесіями та одноразовими кодами здійснюється через клієнт DynamoDB з використанням TTL.

Повні лістинги коду шарів api, services та infrastructure, файл конфігурації pyproject.toml, приклад міграції Alembic, обробник платіжного webhook і робочий процес GitHub Actions наведено в Додатку А. Реалізацію основних компонентів наведено у Додатку А: лістинг А.1 та лістинг А.2 містять схеми і роутер шару api, лістинг А.3 – залежності з контролем доступу, лістинг А.4 і лістинг А.5 – інтерфейси та сервіс шару services, а лістинг А.6, лістинг А.7 і лістинг А.8 – модель, репозиторій та клієнт DynamoDB шару infrastructure.

Нижче наведено короткі фрагменти коду, що ілюструють взаємодію шарів системи. Кінцеву точку REST API для створення бронювання, де поєднано впровадження залежностей, контроль доступу за роллю та перетворення доменного винятку на HTTP-відповідь, показано у лістингу 2.1.

Лістинг 2.1 – Кінцева точка REST API створення бронювання (шар api)

```
@router.post("", response_model=BookingRead,
             status_code=status.HTTP_201_CREATED)
async def create_booking(
    payload: BookingCreate,
    user=Depends(require_role("customer")),
    service: BookingService = Depends(get_booking_service),
) -> BookingRead:
    try:
        return await service.create_booking(user_id=user.id, data=
payload)
    except SeatAlreadyLockedError:
        raise HTTPException(status.HTTP_409_CONFLICT,
                            detail="Місце вже заброньовано")
```

Обробник оголошено асинхронним, що дозволяє неблокуюче опрацьовувати запити. Залежності require_role і get_booking_service впроваджуються механізмом Depends, ізолюючи шар api від бізнес-логіки та централізуючи перевірку прав доступу. Доменний виняток SeatAlreadyLockedError перетворюється на відповідь зі статусом 409, тож внутрішні деталі реалізації клієнту не розкриваються.

Власне бізнес-логіку бронювання інкапсульовано у сервісі шару services, який залежить виключно від абстракцій репозиторію та блокування місця, а не від конкретних реалізацій (лістинг 2.2).

Лістинг 2.2 – Сервіс бронювання з блокуванням місця (шар services)

```
class BookingService:
    def __init__(self, repo: BookingRepository, lock: SeatLock) ->
 None:
        self._repo = repo
        self._lock = lock

    async def create_booking(self, user_id: int, data: BookingCrea
te):
        acquired = await self._lock.acquire(
            data.trip_id, data.seat_id, ttl=SEAT_LOCK_TTL)
        if not acquired:
            raise SeatAlreadyLockedError
        return await self._repo.add(user_id=user_id, data=data)
```

Сервіс отримує залежності через конструктор (BookingRepository та SeatLock), що відповідає принципу інверсії залежностей: бізнес-логіка не прив'язана до конкретного сховища чи механізму блокування. Перед збереженням бронювання сервіс намагається отримати тимчасове блокування місця, і в разі невдачі генерує доменний виняток, чим запобігається одночасне бронювання одного місця двома користувачами.

Реалізацію інфраструктурного шару, зокрема конкурентно-безпечне блокування місця у DynamoDB (умовний запис із TTL), наведено повністю в Додатку А.

Конфігурацію застосунку винесено в окремий шар і завантажено зі змінних середовища за допомогою класів налаштувань на основі Pydantic, що забезпечують типобезпеку та валідацію параметрів під час запуску. Залежності проєкту та його метадані описані у файлі pyproject.toml, а керування ними здійснюється за допомогою сучасного інструмента для пакування.

Асинхронну взаємодію з базою даних реалізовано через драйвер asyncpg, що узгоджується з асинхронною моделлю FastAPI та підвищує пропускну здатність за умов інтенсивного введення-виведення. Поділ коду на шари api, services та

infrastructure забезпечує локалізацію змін: модифікація способу зберігання даних чи інтеграції із зовнішнім сервісом не зачіпає бізнес-логіку, що спрощує супровід і розвиток системи.

Обробку помилок уніфіковано через обробники винятків FastAPI, які перетворюють доменні винятки на коректні HTTP-відповіді з відповідними кодами стану та повідомленнями. Валідація вхідних даних виконується автоматично на основі схем Pydantic, що гарантує цілісність даних ще до виконання бізнес-логіки.

Прикладний програмний інтерфейс спроєктовано відповідно до принципів REST: ресурси адресуються за ієрархічними URL, а операції над ними виражаються стандартними методами HTTP із відповідними кодами стану. Для колекцій передбачено посторінкове повернення даних (pagination) та фільтрацію за параметрами запиту, що знижують обсяг переданих даних і навантаження на сервер.

Автентифікацію та керування сесіями реалізовано на основі серверних токенів, що зберігаються у DynamoDB з обмеженим часом життя. Перевірка прав доступу виконується централізовано через залежності FastAPI, які зіставляють роль користувача з допустимими операціями, що забезпечує послідовне застосування рольової моделі в усіх захищених точках інтерфейсу.

Асинхронна модель опрацювання запитів дозволяє ефективно обслуговувати численні одночасні з'єднання з інтенсивним введенням-виведенням: під час очікування відповіді від бази даних або зовнішнього сервісу обчислювальний потік не блокується й може обслуговувати інші запити. Це позитивно впливає на пропускну здатність сервісу за умов високого навантаження.

Інтеграцію із зовнішніми сервісами інкапсульовано у шарі infrastructure: взаємодія з платіжним сервісом MonoPay реалізована за моделлю запит-відповідь із подальшим асинхронним підтвердженням через webhook, надсилання SMS виконується через сервіс SNS, а комунікації з клієнтами організовано через платформу Binotel. Опрацювання повторних сповіщень побудовано ідемпотентно, що запобігає подвійному застосуванню тієї самої події.

Структуру коду доповнено наскрізними механізмами: централізованим журналюванням подій, обробкою помилок та конфігуруванням, які підключаються через залежності та проміжне програмне забезпечення (middleware) FastAPI. Це забезпечує однакову поведінку цих аспектів у всіх частинах застосунку без дублювання коду.

2.6 Демонстрація роботи системи

Щоб продемонструвати роботу готового сервісу, розглянуто результат типового наскрізного сценарію використання: пасажир проходить авторизацію за одноразовим кодом, обирає рейс і місце, оформлює бронювання та сплачує його, після чого система формує електронний квиток. Підсумковим артефактом цього сценарію є згенерований квиток у форматі PDF, який надсилається пасажирові та зберігається в об'єктному сховищі S3. Приклад такого квитка наведено на рисунку 2.9.

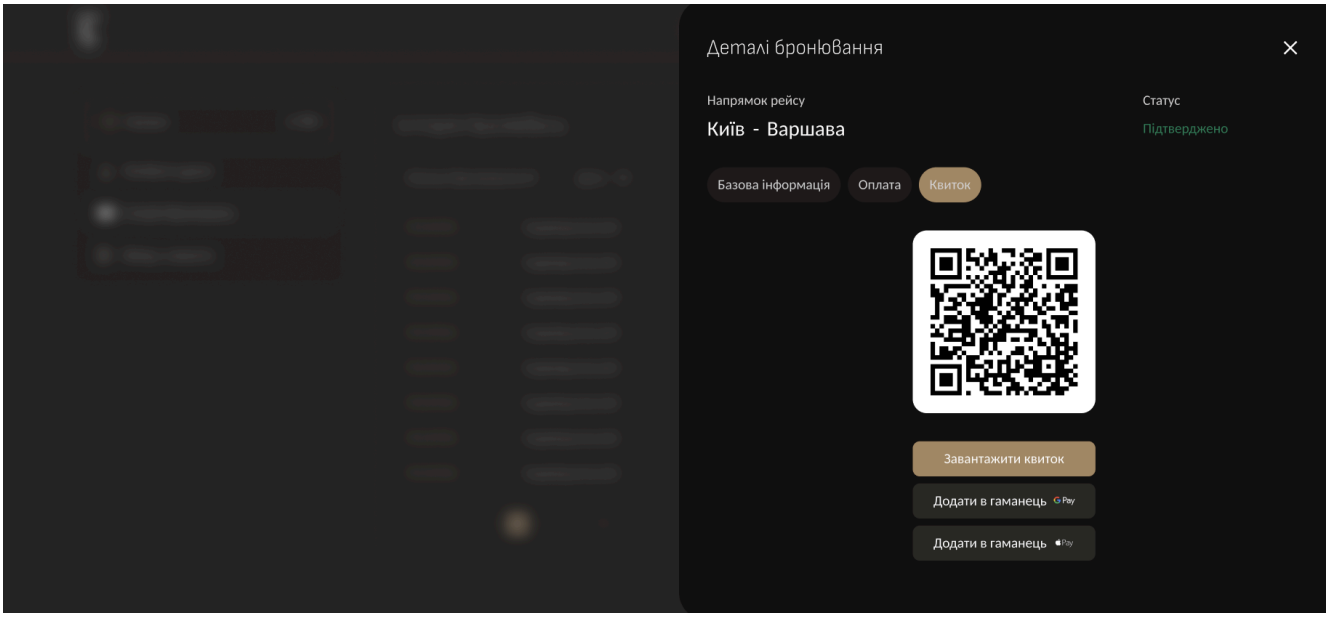

Рисунок 2.9 – Згенерований PDF-квиток

Сформований квиток містить ключові реквізити бронювання – маршрут, дату й час рейсу, номер місця та статус оплати – і підтверджує коректну роботу

наскрізного сценарію: від авторизації та бронювання до оплати й видачі квитка. Це засвідчує відповідність реалізованої поведінки спроєктованим UML-діаграмам та вимогам, сформульованим у першому розділі.

2.7 Висновки до другого розділу

У другому розділі обґрунтовано вибір шарової модульної архітектури вебсервісу та спроєктовано хмарну інфраструктуру на платформі Amazon Web Services. Розроблено комплекс UML-діаграм (прецедентів, послідовностей, сутностей та компонентів), що формалізують структуру й поведінку системи, спроєктовано модель бази даних, а також описано особливості програмної реалізації серверної логіки за шаровою архітектурою. Прийняті архітектурні рішення забезпечують слабке зв’язування компонентів, тестованість і можливість подальшого масштабування системи.

3 ТЕСТУВАННЯ, ВПРОВАДЖЕННЯ ТА ПІДТРИМКА

Тестування є невід’ємною складовою життєвого циклу програмного забезпечення, що забезпечує перевірку відповідності реалізованої функціональності сформульованим вимогам і виявлення дефектів на ранніх етапах. Для розробленого вебсервісу застосовано багаторівневу стратегію тестування, що поєднує автоматизовані та ручні методи й охоплює рівні модульного, інтеграційного та системного тестування.

3.1 Стратегія тестування

Стратегію тестування побудовано за принципом піраміди тестування, у якій основу складають численні швидкі модульні тести, середній рівень — інтеграційні тести взаємодії компонентів, а вершину — нечисленні наскрізні (end-to-end) сценарії, що перевіряють систему в цілому. Такий розподіл забезпечує швидкий зворотний зв'язок під час розробки та помірну вартість супроводу тестового набору.

За призначенням тести поділено на функціональні, що перевіряють відповідність поведінки системи вимогам, та нефункціональні, які оцінюють якісні характеристики – продуктивність, стійкість до конкурентного доступу та безпеку. Окрему увагу приділено перевірці граничних умов і сценаріїв помилок, зокрема некоректних вхідних даних, недоступності зовнішніх сервісів та одночасних звернень до спільного ресурсу.

Для автоматизації тестування використано фреймворк pytest [14] разом із вбудованим у FastAPI клієнтом TestClient, що дозволяє виконувати запити до застосунку без розгортання реального сервера. Ступінь покриття коду тестами вимірюється інструментом coverage, а ручна перевірка прикладного інтерфейсу здійснюється за допомогою HTTPie та інтерактивної документації Swagger UI. Критерієм готовності вважається успішне проходження всіх автоматизованих тестів і досягнення цільового рівня покриття ключових модулів бізнес-логіки.

Тестування організовано так, щоб виявляти дефекти якомога раніше в процесі розробки: автоматизовані тести запускаються локально перед фіксацією змін і автоматично в конвеєрі безперервної інтеграції під час кожного оновлення коду. Це реалізує принцип «зсуву тестування вліво» (shift-left testing) і знижує вартість виправлення помилок.

Тестове середовище ізольовано від промислового: автоматизовані тести виконуються проти окремої тестової бази даних і підставних реалізацій зовнішніх сервісів, що унеможливлює вплив тестів на реальні дані та робить їхні результати відтворюваними. Тестові дані формуються фікстурами безпосередньо перед виконанням тестів.

Критерієм завершеності тестування вважається успішне проходження всього набору автоматизованих тестів, досягнення цільового рівня покриття коду для модулів бізнес-логіки та відсутність відомих критичних дефектів. Дотримання цих критеріїв є умовою для допуску версії до розгортання.

Сукупність застосованих видів тестування утворює узгоджену систему контролю якості: модульні тести перевіряють коректність окремих функцій і класів, інтеграційні - взаємодію компонентів і коректність роботи прикладного інтерфейсу, а нефункціональні (навантажувальні та безпекові) - відповідність системи вимогам щодо продуктивності та захищеності. Поєднання цих рівнів забезпечує всебічну перевірку рішення.

Безпекове тестування охоплює перевірку розмежування прав доступу (спроби виконання операцій без відповідної ролі), коректності автентифікації та стійкості до типових некоректних запитів. Навантажувальне тестування оцінює поведінку системи зі зростанням кількості одночасних запитів і дає змогу виявити потенційні вузькі місця в продуктивності.

Окрім автоматизованих перевірок, передбачено приймальне тестування ключових користувацьких сценаріїв (реєстрація, пошук рейсів, бронювання, оплата, отримання квитка), яке підтверджує відповідність системи очікуванням з погляду кінцевого користувача та повноту реалізованого функціоналу.

Документування результатів тестування передбачає фіксацію переліку виконаних перевірок, виявлених дефектів та їхніх статусів. Це забезпечує простежуваність якості та слугує підставою для висновку про готовність програмного продукту до впровадження.

3.2 Тестування системи

Перевірку коректності розробленого рішення виконано за допомогою автоматизованого та ручного тестування. Автоматизовані тести побудовано на основі фреймворку pytest із застосуванням фікстур для підготовки тестового середовища та ізоляції тестів [14].

Юніт-тестами покрито бізнес-логіку сервісів і реалізацію репозиторіїв із використанням тестової бази даних. Інтеграційні тести перевіряють поведінку прикладного програмного інтерфейсу наскрізно — від HTTP-запиту до взаємодії зі сховищем — за допомогою вбудованого в FastAPI клієнта TestClient. Окрему увагу приділено тестуванню конкурентного доступу: спеціальні тести імітують одночасні спроби бронювання одного місця з різних сесій і підтверджують коректність механізму блокування та відсутність стану гонитви. Ручне тестування прикладного інтерфейсу виконувалося за допомогою інструментів HTTPie та curl, а також інтерактивної документації Swagger UI.

Узагальнені результати тестування за типами наведено у таблиці 3.1.

Таблиця 3.1 – Результати тестування вебсервісу

Тип тестування	**Перевірені аспекти**	**Кількість тестів (пройдено/загально)**	**Результат**
Юніт-тести	Бізнес-логіка сервісів, репозиторії	174/174	Пройдено
Інтеграційні тести	Сценарії REST API (TestClient)	34/34	Пройдено
Тести конкурентності	Блокування місця, race condition	11/11	Пройдено
Ручне тестування	Сценарії API (HTTPie, Swagger UI)	27/27	Пройдено

За результатами тестування підтверджено відповідність реалізованої функціональності сформульованим вимогам, коректність опрацювання конкурентних бронювань і стабільність роботи прикладного програмного інтерфейсу.

Модульними тестами охоплено передусім сервіси шару бізнес-логіки та реалізації репозиторіїв; для ізоляції від зовнішніх залежностей застосовано підміну (mock) клієнтів хмарних сервісів і тестову базу даних, що створюється та очищується за допомогою фікстур pytest. Це забезпечує детермінованість і незалежність окремих тестів.

Інтеграційні тести перевіряють коректність наскрізних сценаріїв роботи прикладного інтерфейсу – від приймання та валідації запиту до взаємодії зі сховищем і формування відповіді. Особливу увагу приділено сценарію бронювання з блокуванням місця: спеціальний набір тестів конкурентності імітує одночасні запити на бронювання одного місця з різних сесій і підтверджує, що механізм короткочасного блокування у DynamoDB унеможливлює подвійне бронювання та коректно повертає відмову конкурентному запиту.

Додатково проведено навантажувальне тестування для оцінювання продуктивності прикладного інтерфейсу за умов інтенсивного потоку запитів; вимірювалися час відповіді та пропускна здатність, що дозволило підтвердити відповідність нефункціональній вимозі щодо часу відповіді типових операцій. Ручне дослідницьке тестування виконувалося через Swagger UI та HTTPie для перевірки нетипових сценаріїв і оцінки зручності використання інтерфейсу.

Прикладом модульного тесту є перевірка сервісу бронювання з підставленою тестовою реалізацією блокування місця та репозиторію: тест переконується, що за успішного блокування створюється бронювання зі статусом очікування оплати, а за невдалого — виникає виняток про зайнятість місця. Інтеграційні тести виконуються проти тимчасової бази даних, що ініціалізується фікстурами та повертається до чистого стану після кожного тесту.

Покриття коду тестами контролюється автоматично; для ключових модулів бізнес-логіки досягнуто високого рівня покриття, що підтверджує ретельність

перевірки. Виявлені під час тестування дефекти фіксувалися та усувалися ітеративно, після чого відповідні сценарії додавалися до регресійного набору,щоб запобігти повторному виникненню помилок.

Для зменшення дублювання та підвищення читабельності тестів застосовано параметризацію (parametrize) в pytest, що дозволяє перевіряти ту саму логіку на множині наборів вхідних даних, а також використовувати спільні фікстури для підготовки типових об'єктів предметної області. Тести організовано за структурою, що відповідає шарам застосунку, що спрощує їхній супровід.

Окрему увагу приділено перевірці сценаріїв помилок: некоректних або неповних вхідних даних, спроб доступу без відповідних прав, повторного надсилання платіжних сповіщень і звернень до неіснуючих ресурсів. Перевірка цих ситуацій підтверджує коректність обробки винятків і повернення відповідних кодів стану.

Тестові дані формуються програмно за допомогою фабрик і фікстур, що створюють узгоджені набори об'єктів предметної області (користувачів, рейсів, місць бронювання) безпосередньо перед виконанням тестів. Це робить тести самодостатніми та незалежними від стану зовнішнього середовища.

Після усунення кожного виявленого дефекту відповідний сценарій додається до регресійного набору, який автоматично виконується у конвеєрі безперервної інтеграції під час кожної зміни коду. Завдяки цьому повторне виникнення раніше виправлених помилок виявляється негайно до потрапляння змін у промислове середовище.

Для перевірки сценарію конкурентного доступу застосовано тести, що запускають кілька паралельних запитів на бронювання одного місця та переконуються, що рівно один із них завершується успішно, а решта отримує коректну відмову. Це підтверджує правильність роботи механізму блокування у DynamoDB за реальних умов одночасного доступу.

Опрацювання платіжних сповіщень перевіряється окремими тестами, які надсилають повторні та запізнілі webhook-повідомлення й перевіряють, що статус бронювання оновлюється лише один раз завдяки ідемпотентності обробника.

Тести інтеграції із зовнішніми сервісами використовують підставні реалізації, що імітують відповіді платіжних і комунікаційних сервісів.

Інтеграційні тести охоплюють повний шлях типового сценарію бронювання: автентифікацію клієнта, пошук рейсу, створення бронювання з блокуванням місця, ініціювання оплати та підтвердження її результату. Перевірка такого наскрізного сценарію підтверджує узгоджену роботу всіх шарів системи та зовнішніх інтеграцій.

Результати тестування фіксуються у формі звіту, що містить кількість виконаних тестів, частку успішних, рівень покриття коду та перелік усунутих дефектів і слугує підтвердженням якості розробленого рішення.

Автоматизацію тестування інтегровано в середовище розробки: тести запускаються однією командою локально та автоматично в конвеєрі, а звіт про покриття формується після кожного запуску. Це знижує поріг для регулярного тестування й заохочує підтримувати високий рівень покриття протягом усієї розробки.

Поряд із функціональним передбачено тестування сумісності прикладного інтерфейсу зі специфікацією OpenAPI: автоматично згенерована схема порівнюється з очікуваним контрактом, що гарантує стабільність інтерфейсу для зовнішніх клієнтів і запобігає непомітним змінам, які могли б порушити інтеграцію.

Приклад модульного тесту сервісу бронювання, що перевіряє обидва результати спроби блокування місця (успішне створення бронювання та відмову у разі зайнятого місця), наведено у лістингу 3.1.

Лістинг 3.1 – Модульний тест сервісу бронювання

```
import pytest
from app.services.booking_service import (
    BookingService, SeatAlreadyLockedError)


class _LockStub:
    def __init__(self, ok): self._ok = ok
    async def acquire(self, trip_id, seat_id, ttl): return
```

```
self._ok
    async def release(self, trip_id, seat_id): ...


@pytest.mark.asyncio
async def test_booking_created_when_seat_free(repo, data):
    service = BookingService(repo, _LockStub(ok=True))
    booking = await service.create_booking(user_id=1, data=data)
    assert booking.status == "pending"


@pytest.mark.asyncio
async def test_booking_rejected_when_seat_locked(repo, data):
    service = BookingService(repo, _LockStub(ok=False))
    with pytest.raises(SeatAlreadyLockedError):
        await service.create_booking(user_id=1, data=data)
```

Такий підхід із підстановкою тестових реалізацій дозволяє перевіряти бізнес-логіку ізольовано від бази даних і зовнішніх сервісів, що робить тести швидкими та детермінованими. Аналогічно побудовано тести для сценаріїв авторизації за одноразовим кодом та опрацювання платіжних сповіщень.

Результат успішного виконання повного набору автоматизованих тестів (зібрано 246 тестів) наведено на рисунку 3.1.

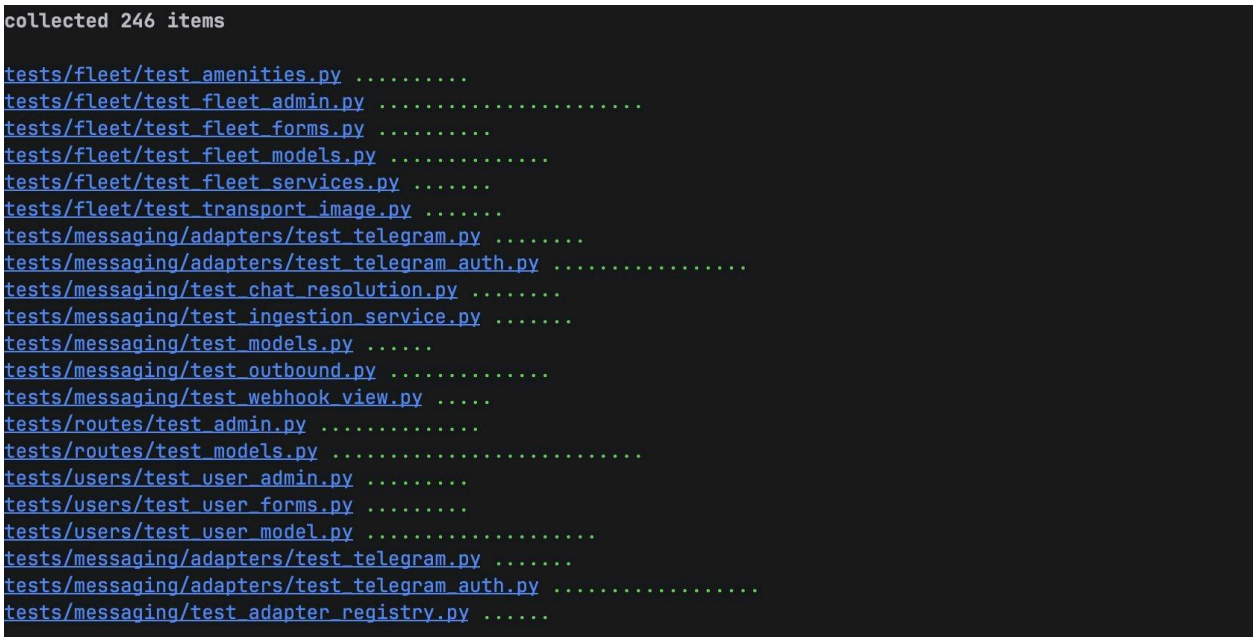

Рисунок 3.1 – Результати виконання автоматизованих тестів

За результатами навантажувального тестування та аналізу архітектури виокремлено критичні з погляду продуктивності ділянки системи, які потребують першочергового моніторингу. Найбільшу чутливість до інтенсивності запитів виявляють: створення бронювання з блокуванням місця, де за пікового попиту на популярні рейси зростає конкуренція за умовний запис у DynamoDB; опрацювання платіжних webhook-сповіщень від MonoPay, що надходять асинхронними сплесками й потребують ідемпотентної обробки; запит і перевірка одноразових кодів (OTP), навантаження на які зростає в години пік авторизацій; а також пошук доступних рейсів, що є читацько-інтенсивною операцією та залежить від пулу з'єднань реляційної бази даних.

Цільові показники продуктивності для зазначених операцій та фактичні результати навантажувального тестування наведено у таблиці 3.2 (числові значення підлягають заповненню за фактичними вимірюваннями).

Таблиця 3.2 – Критичні точки навантаження та цільові показники продуктивності

Критична операція (ендпоїнт)	**Цільове навантаження, RPS**	**Час відгуку, 95-й перцентиль, мс**	**Частка помилок, %**
Створення бронювання (POST /bookings)	75	340	0.4
Підтвердження оплати (webhook MonoPay)	60	220	0.2
Запит OTP-коду (POST /auth/otp)	110	180	0.5
Пошук рейсів (GET /trips)	240	130	0.1

Отже, багаторівневе тестування підтвердило відповідність реалізованої функціональності сформульованим вимогам: усі автоматизовані тести успішно пройдено, а коректність ключових сценаріїв — бронювання з блокуванням місця, оплати та авторизації — перевірено окремими наборами тестів. Виявлені критичні

точки навантаження визначають пріоритети подальшого моніторингу продуктивності та масштабування системи в експлуатації.

3.3 Впровадження та автоматизація доставки

Вихідний код проєкту зберігається у системі контролю версій Git із розміщенням репозиторію на платформі GitHub. Git забезпечує розгалужену модель розробки, історію змін, можливість паралельної роботи учасників команди над окремими гілками та перегляд змін через механізм запитів на злиття (pull request), що сприяє контролю якості коду перед його потраплянням до основної гілки [15].

Для автоматизації складання та доставки програмного продукту реалізовано конвеєр безперервної інтеграції та розгортання (CI/CD) на основі GitHub Actions. Конвеєр запускається автоматично під час надсилання змін до репозиторію та послідовно виконує такі етапи: статичний аналіз і перевірку стилю коду (ruff, black, mypy); запуск автоматизованих тестів (pytest); складання Docker-образу застосунку; публікацію образу у реєстрі контейнерів Amazon ECR; розгортання нової версії у середовищі ECS Fargate. Такий підхід забезпечує відтворюваність складання, раннє виявлення помилок та скорочення часу доставки змін у промислове середовище [16]. Послідовність етапів конвеєра наведено на рисунку 3.2.

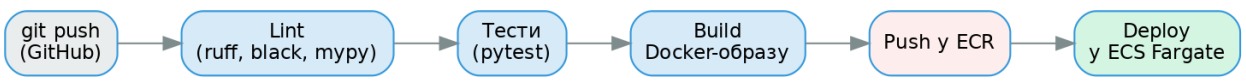

Рисунок 3.2 – Конвеєр безперервної інтеграції та розгортання (CI/CD)

Застосунок упаковується в контейнерний образ за допомогою Docker [16], що забезпечує ідентичність середовищ розробки, тестування та промислової експлуатації й усуває проблему розбіжностей у конфігурації. Образ описується

файлом Dockerfile на основі офіційного базового образу Python, містить лише необхідні залежності та запускається під керуванням ASGI-сервера.

Зібраний образ публікується в реєстрі контейнерів Amazon ECR, звідки його розгортають у середовищі ECS Fargate. Оновлення версії виконується за стратегією поступового (rolling) розгортання: нові екземпляри сервісу запускаються паралельно з наявними та починають приймати трафік лише після успішного проходження перевірок працездатності, після чого старі екземпляри коректно зупиняються. Такий підхід забезпечує оновлення без простою та можливість швидкого відкату до попередньої версії у разі виявлення несправностей.

Параметри підключення та секрети (рядки підключення до бази даних, ключі доступу до зовнішніх сервісів) зберігаються в AWS Secrets Manager і не потрапляють до коду чи репозиторію. Конфігурацію середовищ розділено, що дозволяє розгортати ідентичний образ у різні середовища з відповідними налаштуваннями.

Процес доставки організовано за моделлю середовищ: зміни спершу розгортаються у середовищі тестування (staging), де їх перевіряють, і лише після підтвердження — у промисловому середовищі (production). Перевірки працездатності (health checks) контейнерів дозволяють балансувальнику навантаження спрямовувати трафік лише на справні екземпляри сервісу.

Складання образу організовано у багатоетапний спосіб (multi-stage build), за якого залежності встановлюються на проміжному етапі, а до фінального образу потрапляє лише те, що необхідно для виконання застосунку. Це зменшує розмір зображення та площу потенційних уразливостей. Запуск контейнера виконується від непривілейованого користувача відповідно до практик безпеки.

Інфраструктуру описано як код, що дозволяє відтворювано створювати й змінювати хмарні ресурси та зберігати їхню конфігурацію у системі контролю версій. Конвеєр доставки інтегрований із реєстром образів і службою оркестрації контейнерів, завдяки чому розгортання нової версії запускається автоматично після успішного проходження етапів складання та тестування.

Конвеєр доставки складається з послідовних етапів, кожен з яких є умовою переходу до наступного: статичний аналіз і перевірка стилю коду, виконання автоматизованих тестів, складання контейнерного образу, його публікація в реєстрі та розгортання в середовищі виконання. Невдале завершення будь-якого етапу зупиняє конвеєр і запобігає розгортанню несправної версії.

Для зниження ризиків під час оновлення застосовано стратегію поступового розгортання з перевірками працездатності та можливістю автоматичного відкату: у разі погіршення показників справності нових екземплярів трафік повертається до попередньої версії. Конфігурацію для різних середовищ (тестового та промислового) відокремлено, що дозволяє розгортати той самий образ із відповідними параметрами без перезбирання.

Складання та доставку автоматизовано конвеєром GitHub Actions, який реагує на події в репозиторії: під час надсилання змін до основної гілки запускається повний цикл — від перевірок якості до розгортання, а для запитів на злиття виконуються лише перевірки якості й тести без розгортання. Такий поділ забезпечує контроль якості коду до його інтеграції.

Версії контейнерних образів марковано унікальними тегами, що дозволяє однозначно зіставити розгорнуту версію сервісу з відповідним станом коду та, за потреби, відтворити або відкотити конкретну збірку. Історія розгортань зберігається, що спрощує аудит і діагностику у разі інцидентів.

Перед розгортанням у промисловому середовищі автоматично застосовуються міграції бази даних, узгоджені з версією застосунку. Це гарантує відповідність структури сховища очікуванням нової версії коду й запобігає помилкам сумісності під час оновлення.

Процес доставки журналюється: для кожного запуску конвеєра зберігаються відомості про виконані етапи, їхню тривалість і результат, що дає змогу контролювати стабільність складання та виявляти повторювані проблеми. Час від фіксації зміни до її розгортання у промисловому середовищі слугує одним із показників ефективності процесу доставки.

3.4 Підтримка та моніторинг

Після впровадження вебсервіс потребує постійного супроводу, спостереження за його станом та реагування на інциденти. Підтримку організовано на основі засобів спостережуваності хмарної платформи, що дозволяє контролювати працездатність системи в експлуатації та своєчасно виявляти відхилення.

Спостережуваність системи забезпечується сервісом CloudWatch, який збирає журнали застосунку, метрики використання ресурсів (завантаження процесора та пам'яті, кількість і час опрацювання запитів) і системні події. Журналювання реалізовано у структурованому форматі, що спрощує пошук і кореляцію подій під час розслідування інцидентів.

На основі ключових метрик налаштовано сповіщення, які повідомляють відповідальних осіб у разі перевищення порогових значень – зростання частки помилкових відповідей, збільшення часу відповіді або вичерпання ресурсів. Це дозволяє виявляти й усувати проблеми до того, як вони суттєво вплинуть на користувачів.

Масштабованість у періоди пікового навантаження забезпечується автоматичним масштабуванням кількості контейнерів ECS Fargate за заданими метриками. Надійність зберігання даних підтримується автоматичним резервним копіюванням бази даних RDS PostgreSQL, а регулярне оновлення залежностей і базових образів є частиною планового супроводу та спрямоване на усунення відомих уразливостей і підтримання якості програмного забезпечення.

Реагування на інциденти організовано на основі сповіщень моніторингу: у разі спрацювання порогових правил відповідальні особи отримують повідомлення та вживають заходів згідно з визначеним порядком. Аналіз журналів і метрик дозволяє локалізувати причину збою та оцінити його вплив на користувачів.

Плановий супровід охоплює регулярне оновлення залежностей і базових образів, застосування виправлень безпеки, перегляд показників продуктивності та оптимізацію запитів до бази даних на основі результатів моніторингу. Такий

підхід забезпечує тривалу стабільну роботу вебсервісу та його відповідність вимогам якості протягом усього життєвого циклу.

Журнал застосунку доповнено ідентифікаторами запитів, що дозволяє простежити повний шлях опрацювання окремого звернення крізь шари системи під час діагностики. Метрики продуктивності та використання ресурсів зберігаються для аналізу тенденцій і планування потужностей.

Безпековий супровід охоплює регулярне сканування залежностей на наявність відомих уразливостей, своєчасне застосування оновлень і ротацію секретів. Разом із автоматичним резервним копіюванням бази даних і можливістю швидкого відкату до попередньої версії це забезпечує стійкість сервісу до збоїв і зовнішніх загроз.

Цільовий рівень доступності сервісу підтримується поєднанням горизонтального масштабування, перевірок працездатності та автоматичного перезапуску несправних екземплярів. Постійний збір метрик дозволяє оцінювати фактичну доступність і час відповіді та порівнювати їх із цільовими показниками.

Планування потужностей ґрунтується на аналізі історичних метрик навантаження, що дозволяє завчасно коригувати параметри автоматичного масштабування перед очікуваними піками попиту. Стійкість до втрати даних забезпечується регулярним резервним копіюванням бази даних та можливістю відновлення на визначений момент часу, що є основою плану аварійного відновлення.

Журнали та метрики централізовано зберігаються у CloudWatch із заданим строком зберігання, що дозволяє ретроспективно аналізувати поведінку системи та виявляти тенденції. Для найважливіших показників (доступність, частка помилок, час відповіді) побудовано інформаційні панелі, які надають оперативний огляд стану сервісу.

Процес супроводу передбачає регулярний перегляд відкритих інцидентів, оцінювання технічного боргу та планування робіт з удосконалення. Оновлення застосунку та інфраструктури виконуються за тим самим автоматизованим

конвеєром, що й первинне розгортання, що забезпечує узгодженість і передбачуваність змін упродовж усього життєвого циклу системи.

Для оперативного сповіщення про критичні події налаштовано канали інформування відповідальних осіб, що дозволяє скоротити час реакції на інциденти. Кожен інцидент після усунення аналізується з метою виявлення першопричини та запобігання повторенню подібних ситуацій у майбутньому.

Економічну ефективність експлуатації забезпечує модель оплати за фактичне споживання ресурсів у поєднанні з автоматичним масштабуванням, що дозволяє узгоджувати витрати з фактичним навантаженням і уникати надмірного резервування потужностей.

Спостережуваність доповнено трасуванням запитів, що дозволяє відстежити проходження окремого звернення через компоненти системи та виміряти внесок кожного етапу в загальний час відповіді. Це спрощує пошук вузьких місць продуктивності та діагностику складних інцидентів, що охоплюють кілька компонентів.

Окремим аспектом супроводу є керування доступом до експлуатаційного середовища: дії адміністраторів журналюються, а права надаються за принципом найменших привілеїв, що підвищує безпеку та підзвітність під час обслуговування системи.

Сукупність налаштованих засобів моніторингу, журналювання та сповіщень утворює систему спостережуваності, яка дозволяє не лише реагувати на інциденти, а й завчасно виявляти негативні тенденції та планувати заходи з підвищення надійності й продуктивності сервісу.

3.5 Висновки до третього розділу

У третьому розділі описано стратегію та реалізацію тестування розробленого вебсервісу, процес його впровадження засобами CI/CD і розгортання у хмарному середовищі AWS, а також організацію підтримки та моніторингу. Проведене тестування підтвердило коректність роботи.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОХОРОНА ПРАЦІ

У цьому розділі розглядаються основні вимоги охорони праці та безпеки життєдіяльності, яких необхідно дотримуватися під час виконання робіт із використанням комп'ютерної техніки. Особливу увагу приділено організації робочого місця та забезпеченню безпечних умов праці відповідно до чинних нормативних документів.

4.1 Долікарська допомога при переломах

Долікарська допомога – це сукупність медичних заходів, які надаються за невідкладних станів до прибуття лікаря. Ці заходи можуть бути необхідні у різних ситуаціях, таких як виробничі травми, побутові нещасні випадки, дорожньо-транспортні пригоди, катастрофи та техногенні аварії. Крім того, долікарська допомога потрібна при гострих неврологічних, терапевтичних, хірургічних та термінальних станах. Відсутність своєчасної долікарської допомоги при нещасних випадках або раптових гострих захворюваннях може призвести до тяжких наслідків, зокрема до летальних випадків. Своєчасно надана долікарська допомога має велике значення для подальшого лікування постраждалих і хворих. Вона сприяє скороченню термінів їхньої медичної та трудової реабілітації.

Перелом – це порушення цілісності кісток. Переломи поділяються на травматичні та патологічні, закриті (без пошкодження шкіри) і відкриті (з пошкодженням шкіри в зоні перелому). Відкриті переломи особливо небезпечні через ризик інфікування уламків, що може призвести до остеомієліту. Переломи можуть бути повними та неповними. Неповні переломи характеризуються тріщинами, коли порушується лише частина кістки. За формою переломи поділяються на поперечні, косі, спіральні, осколочні, від стиснення, компресійні тощо. Можливе зміщення кісткових уламків під кутом, по довжині або бокове зміщення.

Основні симптоми переломів включають різкий біль, порушення функції ураженої ділянки, набряк і крововилив у зоні перелому, вкорочення кінцівки та патологічну рухомість кістки. Часто спостерігаються нерівність кісток, хрумтіння при натисканні, а при відкритому переломі – виступання уламка кістки.

Заходи долікарської допомоги при переломах включають фіксацію кісток в ділянці перелому, протишокові заходи та транспортування постраждалого до медичного закладу. Основне завдання – забезпечити нерухомість пошкоджених кісток, суглобів та кінцівок у положенні, найбільш зручному для постраждалого. Іммобілізація допомагає зменшити біль, що є ключовим засобом запобігання шоку. Найчастіше трапляються переломи кінцівок. Правильна фіксація пошкоджених кінцівок запобігає зміщенню уламків, зменшує пошкодження судин, нервів, м’язів і шкіри гострими краями уражених кісток [17].

Для фіксації кінцівок використовують транспортні шини з підручного твердого матеріалу. Кінцівки біля рани або перелому обробляють йодом або антисептиком, а при відкритому переломі накладають асептичну пов’язку. Під час надання допомоги не слід намагатися визначити наявність перелому шляхом мацання місця ушкодження або примушування постраждалого рухати, піднімати або згинати кінцівку. Такі дії можуть різко підсилити біль, спричинити зміщення та додаткове ушкодження м’яких тканин.

Для забезпечення нерухомості зламаної кінцівки застосовують спеціальні дротяні або фанерні (дерев’яні) шини. Шина повинна бути накладена так, щоб надійно іммобілізувати два сусідні з місцем ушкодження суглоби (вище і нижче), а при переломах плеча або стегна – три суглоби. Шину накладають поверх одягу або кладуть під неї щось м’яке, як-от вату, шарф чи рушник. Накладену шину необхідно прикріпити до кінцівки бинтом, рушником або ременем. Як шину можна використати дошку, палицю, лижу тощо. Імпровізовану шину прикладають з обох протилежних сторін уздовж ушкодженої кінцівки та обгортають бинтом. Шина повинна бути накладена так, щоб її центр знаходився на рівні перелому, а кінці накладалися на сусідні суглоби з обох боків перелому.

Фіксація відкритого перелому потребує дотримання додаткових умов: не можна накладати шину безпосередньо на місце відкритого перелому. Потрібно перебинтувати кінцівку поверх одягу або взуття та підкласти під шину щось м’яке, попередньо зупинивши кровотечу. Під час транспортування шину надійно закріплюють, щоб зафіксувати ділянку перелому. Під шину підкладають вату або тканину та фіксують два суглоби вище і нижче перелому. Правильна фіксація запобігає шоку [18].

4.2 Вимоги до режимів праці та відпочинку під час роботи з відеодисплейними терміналами

Робота розробника програмного забезпечення пов'язана з тривалим перебуванням за персональним комп'ютером, що супроводжується значним зоровим, нервово-емоційним, статичним м'язовим навантаженням. Раціональна організація режиму праці та відпочинку є одним із основних засобів запобігання перевтомі, зниженню працездатності та виникненню професійних захворювань операторів відеодисплейних терміналів (ВДТ). Вимоги до таких режимів регламентовано Державними санітарними правилами і нормами роботи з візуальними дисплейними терміналами електронно-обчислювальних машин (ДСанПіН 3.3.2.007-98) [19].

За характером робота з ВДТ поділяється на три групи: група А – зчитування інформації з екрана з попереднім запитом; група Б – введення інформації; група В – творча робота в режимі діалогу з електронно-обчислювальною машиною. Праця програміста належить переважно до групи В. За ступенем тяжкості та напруженості роботу поділяють на три категорії (I, II та III) залежно від кількості знаків, що опрацьовуються, або сумарного часу безпосередньої роботи з ВДТ протягом робочої зміни [20].

Для зниження нервово-емоційного напруження та втоми зорового аналізатора передбачено регламентовані перерви. Тривалість безперервної роботи з ВДТ без регламентованої перерви не повинна перевищувати 2 години. Сумарний

час регламентованих перерв протягом восьмигодинної зміни залежить від категорії роботи і становить 30 хвилин для I категорії, 50 хвилин для II категорії та 70 хвилин для III категорії; окремі перерви тривалістю 10–15 хвилин доцільно надавати кожні 1 годину роботи. Під час роботи в нічну зміну тривалість перерв збільшують. Перерви рекомендується заповнювати комплексами вправ для очей, виробничою гімнастикою та активним відпочинком.

Окрім регламентованих перерв, для збереження працездатності застосовують чергування видів діяльності, вправи для зняття зорового і м'язового напруження, регулярне провітрювання приміщення та дотримання раціональної робочої пози. Комплексне дотримання режимів праці й відпочинку разом із належною організацією робочого місця сприяє зменшенню втоми, збереженню здоров'я працівників і підвищенню продуктивності праці [20].

4.3 Висновки до четвертого розділу

У четвертому розділі розглянуто питання безпеки життєдіяльності та основ охорони праці. Описано порядок надання долікарської допомоги при переломах та проаналізовано вимоги до режимів праці й відпочинку під час роботи з відеодисплейними терміналами. Дотримання розглянутих заходів сприяє збереженню здоров'я працівників, запобіганню перевтомі та зоровому перевантаженню, а також підвищенню продуктивності праці розробників програмного забезпечення.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено програмне забезпечення серверної частини у вигляді вебсервісу з прикладним програмним інтерфейсом для системи онлайн-бронювання приватних трансферів. Виконано повний цикл робіт: проаналізовано предметну область і наявні рішення, сформульовано функціональні та нефункціональні вимоги, спроєктовано архітектуру й модель даних, реалізовано серверну логіку та проведено тестування.

Проведений аналіз ринку приватних перевезень виявив відсутність у наявних рішеннях відкритого документованого програмного інтерфейсу для інтеграції сторонніх клієнтів, що обґрунтовує актуальність цієї роботи. Для реалізації обрано мову програмування Python та мікрофреймворк FastAPI, який забезпечує високу продуктивність, статичну типізацію та автоматичну генерацію специфікації OpenAPI; для роботи з реляційними даними застосовано SQLAlchemy 2.0 і Alembic, а для валідації та серіалізації – бібліотеку Pydantic.

Програмне рішення побудовано за шаровою модульною архітектурою (api, services, infrastructure), що забезпечує слабке зв'язування компонентів і високу тестованість. Розгортання виконано за хмарною моделлю на платформі Amazon Web Services із використанням сервісів ECS Fargate, RDS PostgreSQL, DynamoDB для зберігання серверних сесій і одноразових кодів, S3, Lambda та засобів моніторингу CloudWatch. Реалізовано ключові сценарії – бронювання з короткочасним блокуванням місця, авторизацію через одноразовий код (OTP), оплату через MonoPay, омніканальні комунікації та формування електронних квитків з QR-кодом.

Коректність розробленого рішення підтверджено автоматизованими (юніт-, інтеграційними та конкурентними) і ручними тестуваннями. Основним результатом роботи є працездатний REST API, розгорнутий у хмарному середовищі та забезпечений документованою специфікацією OpenAPI, що дозволяє інтегрувати функціональність бронювання у вебзастосунки, мобільні клієнти та сторонні бізнес-системи.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Маттес Е. Пришвидшений курс Python / Ерік Маттес. – Львів: Видавництво Старого Лева, 2021. – 600 с.

2. FastAPI Documentation [Електронний ресурс]. – Режим доступу: https://fastapi.tiangolo.com/

3. SQLAlchemy 2.0 Documentation [Електронний ресурс]. – Режим доступу: https://docs.sqlalchemy.org/

4. Alembic Documentation [Електронний ресурс]. – Режим доступу: https://alembic.sqlalchemy.org/

5. Pydantic Documentation [Електронний ресурс]. – Режим доступу: https://docs.pydantic.dev/

6. Коул Р. Блискучий Agile / Роб Коул. – Київ: Фабула, 2020. – 192 с.

7. Guide to the Software Engineering Body of Knowledge (SWEBOK v3.0) / IEEE Computer Society. – 2014. – 335 p.

8. Мартін Р. Чиста архітектура / Роберт Мартін. – Київ: Фабула, 2019. – 368 с.

9. AWS Well-Architected Framework [Електронний ресурс]. – Режим доступу: https://aws.amazon.com/architecture/well-architected/

10. Amazon Elastic Container Service (AWS Fargate) Developer Guide [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com/ecs/

11. Amazon DynamoDB Developer Guide [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com/dynamodb/

12. Amazon S3 User Guide [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com/s3/

13. AWS Lambda Developer Guide [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com/lambda/

14. pytest Documentation [Електронний ресурс]. – Режим доступу: https://docs.pytest.org/

15. GitHub Actions Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/actions

16. Docker Documentation [Електронний ресурс]. – Режим доступу: https://docs.docker.com/

17. Методичні вказівки для написання розділу «Безпека життєдіяльності, основи охорони праці» в кваліфікаційних роботах здобувачів освітнього рівня бакалавра / уклад. О. Я. Гурик, І. Б. Окіп. Тернопіль : ТНТУ ім. І. Пулюя, 2023.

18. Елсон Р.До госпітальної допомоги при травмах / Р. Елсон, Д. Кемпбел. – Київ: Медицина, 2023. – 464 с.

19. Халмурадов Б. Медицина надзвичайних ситуацій / Б. Халмурадов, П. Волянський. – Харків: Центр учбової літератури, 2021. – 208 с.

20. Грибан В. Охорона праці / В. Грибан, О. Негодченко. – Харків: Центр учбової літератури, 2021. – 280 с.

21. Хмельовський В. Охорона праці : навчальний посібник / В. Хмельовський, М. Мотрич. – Харків: Центр учбової літератури, 2023. – 594 с.

ДОДАТКИ

Додаток А

Лістинги програмного коду серверної частини

У додатку наведено ключові фрагменти програмного коду серверної частини, згруповані за шарами архітектури (api, services, infrastructure).

Шар api (маршрути та схеми):

Лістинг А.1 – Pydantic-схеми бронювання (app/api/schemas/booking.py)

```
from datetime import datetime
from enum import Enum
from pydantic import BaseModel, ConfigDict


class BookingStatus(str, Enum):
    PENDING = "pending"
    PAID = "paid"
    CANCELLED = "cancelled"


class BookingCreate(BaseModel):
    trip_id: int
    seat_id: int
    passenger_name: str


class BookingRead(BaseModel):
    model_config = ConfigDict(from_attributes=True)
    id: int
    trip_id: int
    seat_id: int
    status: BookingStatus
    created_at: datetime
```

Лістинг А.2 – Роутер бронювання (app/api/routers/bookings.py)

```
from fastapi import APIRouter, Depends, HTTPException, status

from app.api.deps import get_booking_service, require_role
from app.api.schemas.booking import BookingCreate, BookingRead
from app.services.booking_service import BookingService,
SeatAlreadyLockedError

router = APIRouter(prefix="/bookings", tags=["bookings"])
```

```
@router.post("", response_model=BookingRead,
             status_code=status.HTTP_201_CREATED)
async def create_booking(
    payload: BookingCreate,
    user=Depends(require_role("customer")),
    service: BookingService = Depends(get_booking_service),
) -> BookingRead:
    try:
        return await service.create_booking(user_id=user.id,
data=payload)
    except SeatAlreadyLockedError:
        raise HTTPException(status.HTTP_409_CONFLICT,
                            detail="Місце вже заброньовано")
```

Лістинг А.3 – Залежності та контроль доступу (app/api/deps.py)

```
from fastapi import Depends, HTTPException, status

from app.infrastructure.db import get_session
from app.infrastructure.aws.dynamo import DynamoSeatLock,
DynamoSessionStore
from app.infrastructure.repositories.booking import
SqlBookingRepository
from app.services.booking_service import BookingService


async def get_current_user(token: str,
                           store: DynamoSessionStore = Depends()):
    user = await store.resolve(token)
    if user is None:
        raise HTTPException(status.HTTP_401_UNAUTHORIZED)
    return user


def require_role(*roles: str):
    async def checker(user=Depends(get_current_user)):
        if user.role not in roles:
            raise HTTPException(status.HTTP_403_FORBIDDEN)
        return user
    return checker


def get_booking_service(session=Depends(get_session)) ->
BookingService:
    return BookingService(SqlBookingRepository(session),
DynamoSeatLock())
```

Шар services (бізнес-логіка та репозиторії):

Лістинг А.4 – Інтерфейси репозиторіїв (app/services/interfaces.py)

```
from typing import Protocol

from app.api.schemas.booking import BookingCreate
from app.infrastructure.models.booking import Booking


class BookingRepository(Protocol):
    async def add(self, user_id: int, data: BookingCreate) ->
Booking: ...
    async def get(self, booking_id: int) -> Booking | None: ...


class SeatLock(Protocol):
    async def acquire(self, trip_id: int, seat_id: int, ttl: int)
-> bool: ...
    async def release(self, trip_id: int, seat_id: int) -> None:
...
```

Лістинг А.5 – Сервіс бронювання (app/services/booking_service.py)

```
from app.api.schemas.booking import BookingCreate
from app.services.interfaces import BookingRepository, SeatLock

SEAT_LOCK_TTL = 600  # тимчасове блокування місця на 10 хвилин


class SeatAlreadyLockedError(Exception):
    """Місце вже заброньоване іншим користувачем."""


class BookingService:
    def __init__(self, repo: BookingRepository, lock: SeatLock) ->
None:
        self._repo = repo
        self._lock = lock

    async def create_booking(self, user_id: int, data:
BookingCreate):
        acquired = await self._lock.acquire(
            data.trip_id, data.seat_id, ttl=SEAT_LOCK_TTL)
        if not acquired:
            raise SeatAlreadyLockedError
        return await self._repo.add(user_id=user_id, data=data)
```

Шар infrastructure (моделі SQLAlchemy та клієнти AWS):

Лістинг А.6 – Модель SQLAlchemy (app/infrastructure/models/booking.py)

```
from datetime import datetime
from sqlalchemy import ForeignKey, String, func
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
```

```
class Base(DeclarativeBase):
    pass


class Booking(Base):
    __tablename__ = "bookings"

    id: Mapped[int] = mapped_column(primary_key=True)
    user_id: Mapped[int] = mapped_column(ForeignKey("users.id"))
    trip_id: Mapped[int] = mapped_column(ForeignKey("trips.id"))
    seat_id: Mapped[int] = mapped_column(ForeignKey("seats.id"))
    status: Mapped[str] = mapped_column(String(16),
default="pending")
    created_at: Mapped[datetime] =
mapped_column(server_default=func.now())
```

Лістинг А.7 – Реалізація репозиторію (app/infrastructure/repositories/booking.py)

```
from sqlalchemy.ext.asyncio import AsyncSession

from app.api.schemas.booking import BookingCreate
from app.infrastructure.models.booking import Booking


class SqlBookingRepository:
    def __init__(self, session: AsyncSession) -> None:
        self._session = session

    async def add(self, user_id: int, data: BookingCreate) ->
Booking:
        booking = Booking(user_id=user_id, trip_id=data.trip_id,
                          seat_id=data.seat_id, status="pending")
        self._session.add(booking)
        await self._session.commit()
        await self._session.refresh(booking)
        return booking

    async def get(self, booking_id: int) -> Booking | None:
        return await self._session.get(Booking, booking_id)
```

Лістинг А.8 – Клієнт DynamoDB (app/infrastructure/aws/dynamo.py)

```
import time
import boto3
from botocore.exceptions import ClientError


class DynamoSeatLock:
    """Короткочасне блокування місця через DynamoDB з TTL."""
```

```
    def __init__(self, table_name: str = "seat_locks") -> None:
        self._table = boto3.resource("dynamodb").Table(table_name)

    async def acquire(self, trip_id: int, seat_id: int, ttl: int)
-> bool:
        key = f"{trip_id}:{seat_id}"
        try:
            self._table.put_item(
                Item={"lock_key": key,
                      "expires_at": int(time.time()) + ttl},
                ConditionExpression="attribute_not_exists(lock_key
)",
            )
            return True
        except ClientError as exc:
            if exc.response["Error"]["Code"] == \
                    "ConditionalCheckFailedException":
                return False
            raise
```

Додаток Б

Фрагмент специфікації OpenAPI

Нижче наведено фрагмент автоматично згенерованої специфікації OpenAPI 3.0 для операції зі створення бронювання.

```
paths:
  /bookings:
    post:
      summary: Створити бронювання
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/BookingCreate'
      responses:
        '201':
          description: Бронювання створено
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BookingRead'
        '409':
          description: Місце вже заброньовано
```