МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії
програмного забезпечення

_____ Євген ДАВИДЕНКО

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

ІНТЕЛЕКТУАЛЬНИЙ ВЕБСЕРВІС ДЛЯ ГЕНЕРАЦІЇ ПЕРСОНАЛІЗОВАНИХ НАСТІЛЬНИХ ІГОР

Спеціальність 121 Інженерія програмного забезпечення
Освітня програма «Інженерія програмного забезпечення»

Здобувач _____ **Роберт АЙРАПЕТЯН**

«__» _____ 2026 р.

Керівник роботи
д-рка техн. наук,
професорка _____ **Альона ШВЕД**

«__» _____ 2026 р.

Миколаїв – 2026

Завдання на виконання кваліфікаційної роботи

Чорноморський національний університет імені Петра Могили

Факультет	Комп'ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступінь	Бакалавр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри інженерії програмного забезпечення

_____ Євген ДАВИДЕНКО

«____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну бакалаврську роботу здобувача

Айрапетяна Роберта

---

1. Тема кваліфікаційної роботи Інтелектуальний вебсервіс для генерації персоналізованих настільних ігор затверджена наказом ректора ЧНУ ім. Петра Могили № 345 від «26» грудня 2025 р.
2. Строк представлення кваліфікаційної роботи «__» червня 2026 р.
3. Очікуваний результат роботи та початкові дані, якщо такі потрібні.

Очікуваним результатом роботи є інтелектуальний вебсервіс для генерації персоналізованих настільних ігор, який забезпечує формування ігрового контенту, відображення результатів генерації у вебінтерфейсі та збереження сформованих матеріалів в обліковому записі користувача.

Перелік питань, що підлягають розробці: дослідження предметної галузі та аналіз існуючих рішень; формування функціональних і нефункціональних вимог до вебсервісу; проєктування алгоритму генерації, інформаційних потоків, архітектури вебсервісу та інформаційної моделі бази даних; реалізація клієнтської та серверної частин вебсервісу; реалізація модуля фонової обробки та поетапного процесу генерації для формування персоналізованої гри і її цифрових матеріалів; впровадження системи автентифікації та розмежування доступу; реалізація механізмів відображення, збереження та експорту результатів генерації; опрацювання механізму доступу до генерації з урахуванням безкоштовного та платного режимів; проведення тестування вебсервісу.

Перелік графічних матеріалів:

Презентація

4. Консультанти:

Консультант	**Кафедра (організація)**	**Частина роботи**

Дата видачі завдання «12» січня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

Тема: **Інтелектуальний вебсервіс для генерації персоналізованих настільних ігор**

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розроблення та затвердження завдання на виконання КБР	12.01.2026	13.01.2026	Виконано
2.	Огляд літератури за темою роботи	19.01.2026	23.01.2026	Виконано
3.	Складання календарного плану КБР	26.01.2026	27.01.2026	Виконано
4.	Аналіз предметної області та визначення вимог до вебзастосунку	02.02.2026	06.02.2026	Виконано
5.	Розроблення проєктних рішень до КБР	09.02.2026	27.02.2026	Виконано
6.	Моделювання та конструювання ПЗ (проєктування бази даних, API)	02.03.2026	13.03.2026	Виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування	16.03.2026	15.05.2026	Виконано
8.	Оформлення КБР та презентації	18.05.2026	25.05.2026	Виконано
9.	Відгук керівника КБР	26.05.2026	26.05.2026	Виконано
10.	Попередній захист КБР	27.05.2026	27.05.2026	Виконано
11.	Завершення оформлення КБР та презентації	01.06.2026	05.06.2026	Виконано
12.	Рецензування	08.06.2026	08.06.2026	Виконано
13.	Захист кваліфікаційної роботи			

Здобувач _____ **Роберт АЙРАПЕТЯН**

«__» _____ 2026 р.

Керівник роботи

д-рка техн. наук,
професорка _____ **Альона ШВЕД**

«__» _____ 2026 р.

АНОТАЦІЯ

до кваліфікаційної бакалаврської роботи

Інтелектуальний вебсервіс для генерації персоналізованих настільних ігор

Здобувач 409 гр.: Айрапетян Роберт

Керівник: д-рка техн. наук, професорка Швед Альона

Настільні ігри протягом останніх кількох років не втрачають своєї привабливості як спосіб спільного проведення часу. Люди грають у настільні ігри під час дружніх зустрічей, освітнього процесу або в командних проєктах. Інтерес викликають не лише відомі настільні ігри, а й спеціально спроєктовані ігри з персоналізацією. Саме тому все частіше виникає потреба створювати настільні ігри, які враховують конкретні умови та склад учасників. Через це актуальним є використання програмних засобів і методів штучного інтелекту, які можуть допомогти впорядкувати цей процес і зробити його доступнішим для користувачів без спеціального досвіду в геймдизайні.

Кваліфікаційна бакалаврська робота присвячена розробці інтелектуального вебсервісу, який дозволяє впорядкувати процес створення персоналізованих настільних ігор і зменшити час, необхідний для підготовки гри.

Науково-практичне значення обраної теми полягає в тому, що вебсервіс поєднує можливості генеративного штучного інтелекту з практичними потребами користувачів, які хочуть створити власну настільну гру. Такий підхід дає змогу не починати розробку гри з порожнього аркуша, а поступово отримувати концепцію, правила, компоненти та візуальні матеріали в межах одного процесу.

Метою кваліфікаційної роботи є створення інтелектуального вебсервісу для генерації персоналізованих настільних ігор, орієнтованого на користувачів, які можуть не мати досвіду у геймдизайні, але бажають отримати повноцінну гру.

Об’єктом кваліфікаційної роботи виступає процес генерації ігрового контенту для персоналізованих настільних ігор. Створення таких ігор охоплює формування концепції, текстового й візуального контенту, правил і допоміжних матеріалів.

Предметом кваліфікаційної роботи є моделі та методи штучного інтелекту для генерації ігрового контенту, а також засоби організації цього процесу у форматі інтелектуального вебсервісу.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків та переліку джерел посилання.

У вступі обґрунтовано актуальність розробки вебсервісу для генерації настільних ігор з персоналізацією, визначено мету, завдання, об’єкт та предмет кваліфікаційної роботи.

У першому розділі проведено детальний аналіз предметної області генерації настільних ігор, досліджено існуючі рішення та їх функціональні можливості.

У другому розділі обґрунтовано вибір засобів розробки, розглянуто моделі й методи генерації ігрового контенту та сформовано вимоги до вебсервісу.

У третьому розділі виконано проєктування системи з представленням UML-діаграм та схем.

У четвертому розділі наведено етапи процесу розробки вебсервісу та представлено результати тестування системи.

У висновках узагальнено результати виконаної роботи.

Кваліфікаційна робота викладена на 106 сторінках, вона містить 4 розділи, 65 ілюстрацій, 8 таблиць, 2 додатки, 33 джерела в переліку посилань.

Ключові слова: генерація ігрового контенту, генерація настільних ігор, інтелектуальний вебсервіс, персоналізація настільних ігор, штучний інтелект, NestJS, Next.js.

ABSTRACT

of the Bachelor's Thesis

Intelligent web service for generating personalized board games

Student of group 409: Robert Airapetian

Supervisor: Dr. Sc., Prof. Alyona Shved

Board games have not lost their appeal as a way of spending time together over the past several years. People play board games during friendly meetings, in educational settings, or in team projects. Interest is shown not only in well-known board games, but also in custom-made and specially designed personalized games. Therefore, there is an increasing need to create board games that take into account specific conditions and the composition of participants. For this reason, the use of software tools and artificial intelligence methods is relevant, as they can help structure this process and make it more accessible to users without specialized experience in game design.

The bachelor's qualification thesis is devoted to the development of an intelligent web service that makes it possible to organize the process of creating personalized board games and reduce the time required for game preparation.

The scientific and practical significance of the chosen topic lies in the fact that the proposed web service combines the capabilities of generative artificial intelligence with the practical needs of users who want to create their own board game. This approach makes it possible not to start game development from a blank page, but to gradually obtain the concept, rules, components, and visual materials within a single process.

The purpose of the qualification thesis is to create an intelligent web service for generating personalized board games, oriented toward users who may not have experience in game design but wish to obtain a complete game.

The object of the bachelor's qualification thesis is the process of generating game content for personalized board games. The creation of such games includes the formation of the concept, textual and visual content, rules, and auxiliary materials.

The subject of the qualification thesis is models and methods of artificial intelligence for game content generation, as well as means of organizing this process in the format of an intelligent web service.

The qualification thesis consists of an introduction, four chapters, conclusions, and a list of references.

The introduction substantiates the relevance of developing a web service for generating personalized board games and defines the purpose, tasks, object, and subject of the qualification thesis.

The first chapter provides a detailed analysis of the subject area of board game generation and examines existing solutions and their functional capabilities.

The second chapter justifies the choice of development tools, considers models and methods of game content generation, and formulates the requirements for the web service.

The third chapter presents the system design with UML diagrams and schemes.

The fourth chapter describes the stages of the web service development process and presents the results of system testing.

The conclusions summarize the results of the completed work.

The qualification thesis is presented on 106 pages; it contains 4 chapters, 65 figures, 8 tables, 2 appendices and 33 sources in the list of references.

Keywords: artificial intelligence, board game generation, board game personalization, game content generation, intelligent web service, NestJS, Next.js.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
ПЗ	–	програмне забезпечення
ШІ	–	штучний інтелект

API	–	Application Programming Interface
CSS	–	Cascading Style Sheets
HTTP	–	HyperText Transfer Protocol
HTTPS	–	HyperText Transfer Protocol Secure
JSON	–	JavaScript Object Notation
ORM	–	Object-Relational Mapping
PDF	–	Portable Document Format
PNG	–	Portable Network Graphics
REST	–	Representational State Transfer
SVG	–	Scalable Vector Graphics

ВСТУП

Настільні ігри протягом останніх кількох років не втрачають своєї привабливості як спосіб спільного проведення часу. Люди грають у настільні ігри під час дружніх зустрічей, освітнього процесу або у командних проєктах. Інтерес викликають не лише відомі настільні ігри, а й спеціально спроєктовані ігри з персоналізацією. Все частіше виникає потреба створювати настільні ігри, які враховують конкретні умови та склад гравців. У таких випадках стандартні настільні ігри часто не підходять, оскільки вони розраховані на інший склад учасників або тривалість гри.

Розроблення настільної гри виявляється значно складнішою справою, ніж може здатися на перший погляд. Навіть найпростіша за своєю суттю гра вимагає детально прописаного регламенту та продуманих етапів створення. Насамперед доводиться окреслити умови завершення партії, можливі сценарії розвитку подій, а також механізми взаємодії між ігровими компонентами.

Персоналізація є важливою, тому що більшість настільних ігор створюється без прив’язки до конкретної групи людей. Гра стає особливою, коли в ній є персонажі, ролі та сюжетні деталі, які логічно вписуються в процес гри. Персоналізація вручну є складною, тому що кожна нова компанія гравців потребує окремих матеріалів або правок уже готових. Без автоматизації масштабування такого процесу є складним завданням для звичайних користувачів.

Сучасні технології штучного інтелекту дозволяють автоматизувати значну частину цього процесу. Генеративні системи, якщо їх не організувати належним чином, зазвичай видають матеріал, який потребує ретельної перевірки та доопрацювання. Через це доцільно використовувати структурований підхід, у межах якого процес створення контенту поєднується з перевіркою узгодженості, уніфікацією та фіксацією даних.

Науково-практичне значення обраної теми полягає у розробленні інтелектуального вебсервісу, який дозволяє впорядкувати процес створення персоналізованих настільних ігор. Це призводить до зменшення часу, необхідного

для підготовки гри, спрощує процес для користувачів без досвіду геймдизайну. Також забезпечує формування структурованого результату, придатного до подальшого використання й розвитку.

Метою кваліфікаційної роботи є створення інтелектуального вебсервісу для генерації персоналізованих настільних ігор. Сервіс орієнтований на користувачів, які можуть не мати досвіду у геймдизайні, але бажають отримати повноцінний ігровий продукт для конкретної компанії або групи гравців.

Персоналізація передбачає налаштування ігрового процесу і змісту під конкретну групу гравців, з урахуванням їхніх ролей, характеристик і бажаного формату гри. Сервіс орієнтований на те, щоб надавати не просто ідеї, а практичні, готові до впровадження рішення. Основними вимогами до результату є зрозумілі правила, структурований опис гри та достатні матеріали для проведення в реальних умовах.

Відповідно до поставленої мети визначено такі **завдання**:

1) проаналізувати предметну область створення настільних ігор та визначити ключові етапи, які впливають на якість і працездатність результату;

2) дослідити існуючі підходи до автоматизації створення ігрового контенту та визначити, які з них доцільно використати у вебсервісі генерації персоналізованих ігор;

3) сформувати функціональні вимоги до вебсервісу, включаючи сценарії взаємодії користувача, параметри персоналізації, механізми генерації та повторної генерації, а також збереження і отримання результатів;

4) сформувати нефункціональні вимоги до системи, зокрема щодо безпеки даних, стабільності, продуктивності та базової якості результатів генерації;

5) спроєктувати архітектуру вебсервісу і модель даних, необхідну для зберігання параметрів користувача, опису гри, сформованих артефактів, історії генерацій та етапів їх виконання;

6) реалізувати робочу версію вебсервісу та перевірити його роботу на типових сценаріях створення, збереження і використання гри;

7) визначити підхід до первинної перевірки узгодженості правил і контенту, щоб мінімізувати очевидні суперечності в кінцевому результаті.

Отже, мета окреслює бажаний практичний результат у формі інтелектуального вебсервісу, а завдання формують послідовність дій, які дають змогу перейти від аналізу предметної області та вимог до проєктування, реалізації та тестування ефективності розробленого рішення.

Об’єктом кваліфікаційної роботи виступає процес генерації ігрового контенту для персоналізованих настільних ігор. Створення такого виду ігор є сукупністю взаємопов’язаних процесів: від формування концепції гри та створення текстового й візуального контенту до структурованого формування правил і допоміжних матеріалів. У цьому процесі виникає найбільша кількість проблем, зокрема суперечності між елементами контенту, невідповідність між початковим задумом і кінцевим результатом, а також складнощі з адаптацією під різні умови використання.

Предметом кваліфікаційної роботи є моделі та методи штучного інтелекту для генерації ігрового контенту, а також засоби організації цього процесу у форматі вебсервісу. Це включає підхід до організації процесу генерації, моделі представлення ігрових даних, використання алгоритмів автоматичного формування текстового й візуального контенту. У центрі уваги перебуває не вся діяльність розробника настільної гри, а саме ті її аспекти, які можуть бути автоматизовані, структуровані та реалізовані у вебсередовищі з використанням інтелектуальних технологій.

Таке розмежування допомагає чітко окреслити межі дослідження. Об’єкт визначає загальний контекст і проблемне поле. Предмет відображає конкретний напрям роботи, пов’язаний із розробленням інтелектуального вебсервісу для генерації персоналізованих настільних ігор. Саме в межах предмета формується технічне рішення, яке відповідає темі роботи і поставленій меті.

1 АНАЛІЗ ПРОЦЕСІВ СТВОРЕННЯ ПЕРСОНАЛІЗОВАНИХ НАСТІЛЬНИХ ІГОР

1.1 Актуальність розроблення інтелектуальних вебсервісів генерації персоналізованого ігрового контенту

Генеративний ШІ вже давно не виглядає як технологія лише для експериментів. Його використовують там, де раніше вручну формували багато варіантів тексту, зображень або інших елементів для цифрового продукту [1]. У таких задачах добре видно практичну користь цієї технології: вона прискорює підготовку матеріалів і дає більше варіантів для відбору. Водночас без структури й перевірки генерація може дати матеріал, який виглядає переконливо, але потребує доопрацювання. Тому в цій роботі генеративний ШІ розглядається не як окремий «автоматичний автор», а як частина керованого процесу створення контенту.

Поширення штучного інтелекту в організаціях показує, що він уже вийшов за межі допоміжного інструменту і все частіше стає частиною цифрової інфраструктури. Для розроблення вузькоспеціалізованих вебсервісів це має особливе значення, особливо для тих, які вимагають не лише творчого підходу, а й аналітичних навичок. На рис. 1.1 представлені дані про світове використання ШІ організаціями у період з 2023 по 2025 роки [2].

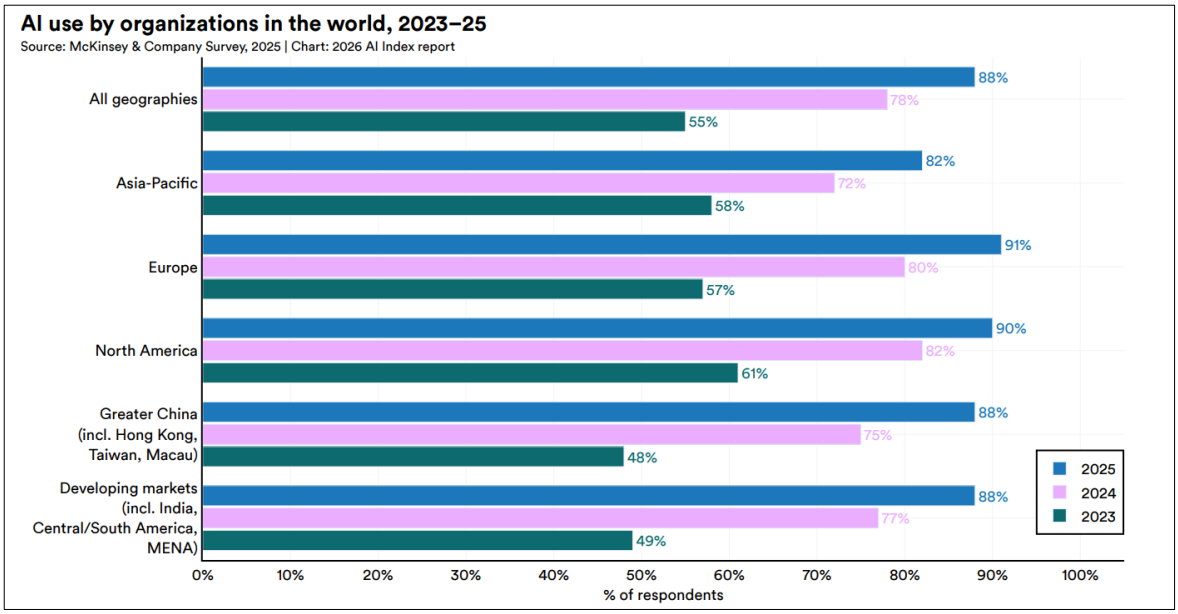

Рисунок 1.1 – Використання ШІ організаціями у світі у 2023–2025 роках

Частка організацій, що застосовують ШІ хоча б в одному напрямку, зросла з 55% у 2023 році до 78% у 2024 році та до 88% у 2025 році [2]. Це сильний показник швидкого впровадження штучного інтелекту в діяльність організацій. Еволюція цифрових платформ дає основу для появи вебсервісів, що використовують генеративні моделі ШІ, і це має довгостроковий характер.

Окремо потрібно подивитися саме на генеративний ШІ. У межах цієї роботи важливо, що він не лише обробляє дані, а може створювати новий контент. Саме тому нижче наведено порівняння його використання організаціями у світі у 2023 та 2024 роках (рис. 1.2) [3].

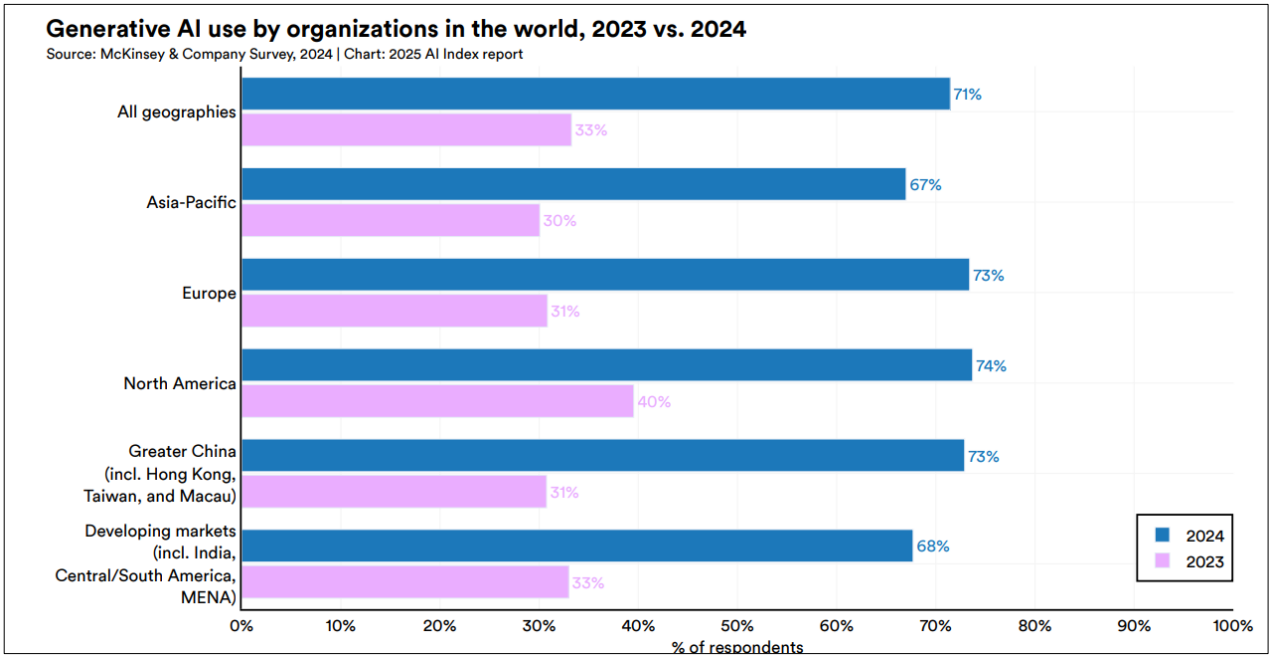

Рисунок 1.2 – Використання генеративного ШІ організаціями у світі

Як показують дані, у 2023–2024 роках динаміка впровадження генеративного ШІ була особливо швидкою порівняно із загальним використанням ШІ [2, 3]. Оскільки персоналізована настільна гра потребує створення нового контенту, а не тільки використання вже готових даних, цей механізм стає дуже важливим у межах кваліфікаційної роботи.

Наведена вище інформація є важливим елементом не лише загальної технологічної динаміки, а й свідчить про зрілість підходів до розроблення нових цифрових продуктів. Зараз генеративний ШІ все частіше використовують в

практичних випадках, коли важливими аспектами є швидкість підготовки, різноманітність контенту та особливо його адаптація. У такому випадку генерація персоналізованих настільних ігор – дійсно актуальне завдання. Йдеться не лише про загальне використання ШІ, а про розроблення сервісу, який може перетворити набір параметрів від користувачів на організований ігровий продукт. Такий продукт повинен мати чітку логіку, встановлені правила, компоненти та відповідне тематичне оформлення, яке особливо важливе для теми кваліфікаційної роботи.

Сучасні наукові дослідження вказують на те, що генеративний штучний інтелект демонструє свої найкращі результати не при окремому використанні, а в контексті організованого прикладного процесу. Цю технологію вже використовують для створення концепцій, розроблення різних ігрових елементів, виготовлення прототипів або підготовки додаткових матеріалів гри. Проте варто зазначити, що генеративні моделі не забезпечують автоматично високу якість отриманого результату [4, 5]. Під час розроблення складної гри, яка вимагає узгодження механік, правил, елементів і тематичного оформлення, лише одного вільного генерування недостатньо. У цьому контексті важливо поєднати всі можливості штучного інтелекту з чіткою структурою логіки вебсервісу. Це забезпечує правильну послідовність етапів та формує якісний ігровий продукт, який можна використати.

У межах кваліфікаційної роботи особливо важливим є те, що генеративний підхід не є ізольованим механізмом, він виступає частиною більш масштабної програмної системи. Для розроблення складного ігрового продукту необхідно не лише зібрати окремі текстові чи візуальні компоненти. Необхідним є забезпечення їхньої узгодженості, логічної структури та потенціалу для подальшого використання. Практична цінність інтелектуального вебсервісу виходить за межі лише застосування генеративних моделей. Ключові аспекти такого вебсервісу – взаємодія з контрольованою обробкою, а також здатність зберігати результати, генерувати повторно і поступово формувати фінальний продукт.

Такі висновки мають особливе значення для теми кваліфікаційної роботи. Створення персоналізованої настільної гри не зводиться до генерації окремого

тексту або окремого зображення. Необхідно досягти узгодженості, в якій концепція гри, її правила, механіка, картки, ролі та візуальні елементи гармонійно доповнюють одне одного. Важливо врахувати не лише тему генерації ШІ, а й аспекти розроблення вебсервісу. Сервіс повинен організовувати процес, зберігати отримані результати, дозволяти повторну генерацію та забезпечувати зручну взаємодію з користувачами.

Актуальність обраної теми можна звести до таких основних чинників:

– стрімке зростання практичного використання ШІ в організаціях і цифрових продуктах;

– підвищення доступності моделей ШІ;

– поширення генеративного ШІ за межі експериментального середовища;

– підвищення попиту на персоналізований цифровий контент;

– відсутність цілісних вебсервісів, орієнтованих на автоматизовану генерацію персоналізованих настільних ігор.

На практиці це означає перехід від окремих експериментів із генеративними моделями до прикладних сервісів, які мають давати користувачу зрозумілий результат. Тут важлива не тільки швидкість створення контенту. Користувач повинен отримати не набір розрізнених фрагментів, а матеріали, які можна переглянути, зберегти й використати як основу для гри. Саме тому в цій роботі акцент зроблено не лише на генерації окремих текстових чи візуальних елементів, а на побудові цілісного сервісного підходу. Він дозволяє повторно використовувати введені параметри, зберігати проміжні результати, зменшувати кількість ручних правок і поступово покращувати вже сформований контент. У такій моделі генерація стає частиною впорядкованого сценарію, а не одноразовою відповіддю моделі.

Розроблення інтелектуального вебсервісу для генерації персоналізованих настільних ігор є актуальним і практично доцільним напрямом на даному етапі розвитку технологій. Таким чином, поєднання нових технологій генеративного штучного інтелекту з необхідністю створення структурованих продуктів для користувачів має велику цінність.

1.2 Особливості створення персоналізованих настільних ігор як об’єкта автоматизації

Процес розроблення настільної гри містить багато повторюваних і взаємопов’язаних етапів, тому краще розглядати їх як об’єкт автоматизації. Персоналізовані ігри мають складну багатокомпонентну структуру, тому письмовий опис або колода карт – це не остаточна версія, яка придатна для безпосереднього застосування. Ігровий продукт, який є повноцінним, потребує ретельного формування концепції, що включає в себе вибір відповідних ігрових механік, формалізацію правил, конкретну специфікацію матеріалів та забезпечення єдиного візуального стилю. Не менш важливою є адаптація змісту під конкретну компанію гравців. Складна структура персоналізованих настільних ігор надає як можливості, так і виклики для автоматизації процесу їх розроблення [6].

У контексті цієї роботи об’єкт дослідження доцільно розглядати саме як процес генерації ігрового контенту для персоналізованих настільних ігор, оскільки існує багато практичних проблем зі створенням повного набору правил, компонентів та візуальних матеріалів на основі загальної ідеї. Увага приділяється використанню моделей та методів штучного інтелекту, а також організації цього процесу у вигляді вебсервісу. Таке уточнення дає змогу розглядати розроблення персоналізованої гри не просто як абстрактну творчу задачу, а як організований процес, який можна автоматизувати, аналізувати та проєктувати програмно.

У роботах про генеративний ШІ в ігровому дизайні добре простежується думка, що найбільше користі він дає там, де потрібно швидко підготувати ідеї, кілька варіантів контенту або перший прототип. Це справді прискорює роботу, особливо на ранніх етапах, коли ще потрібно шукати форму майбутньої гри. Але згенерований матеріал ще не можна автоматично вважати готовим результатом. Його потрібно впорядкувати, перевірити й привести до такої форми, у якій він уже може використовуватися на практиці. Для персоналізованої настільної гри це особливо помітно, бо користувач очікує не випадкові фрагменти правил, назв чи описів, а цілісний ігровий продукт зі зрозумілою логікою. Інакше генерація

залишається лише допоміжною чернеткою, а не повноцінною основою для гри [4, 5].

Характерна ознака персоналізованої настільної гри – її здатність об’єднувати формалізовану логіку та творчу варіативність. Якість гри визначається за такими характеристиками, як чіткі правила, зрозумілі умови початку і завершення партії, визначені ролі гравців і несуперечлива система дій. Крім цього, гра має бути веселою та захопливою, задаючи тон і жанр під вимоги групи гравців. Але вільна генерація призведе до нестабільного результату, а редактор шаблонів теж не може повністю вирішити таку задачу через відсутність персоналізації.

У практичному сенсі процес створення персоналізованої настільної гри охоплює такі основні етапи:

– формування концепції гри, вибір теми, жанрової рамки, атмосфери, рівня серйозності або розважальності;

– побудова механік, визначення взаємодії між гравцями, типу ходів, системи ресурсів, кооперативної або змагальної моделі;

– формування правил, опис послідовності дій, винятків, умов перемоги та завершення партії;

– генерування компонентів, створення карток, ролей, предметів, поля, токенів, допоміжних елементів;

– створення візуальної частини, добір стилю, ілюстрацій, кольорової логіки та читабельності елементів;

– персоналізація, адаптація змісту до кількості гравців, їхніх ролей, побажань, характеру події або конкретної компанії;

– перевірка узгодженості, приведення результату до цілісного вигляду та усунення очевидних суперечностей.

Кожен з цих етапів має своє значення, проте в контексті персоналізованої настільної гри їх варто розглядати не як відокремлені процеси. Наприклад, концепція визначає основні механіки, які, у свою чергу, впливають на правила. Ці правила формують вимоги до гри та її компонентів, а візуальні елементи повинні гармонійно доповнювати загальний зміст і відповідати логіці гри. Окремі

інструменти швидко стають проблемою. Для персоналізованої гри важливо, щоб всі етапи збиралися не по різних місцях, а поступово складалися в один зрозумілий результат. З огляду на це автоматизація має охоплювати не лише створення окремих елементів гри, а й підтримувати їхню узгодженість [4, 6].

Окремої уваги потребує візуальна складова гри. Проведений аналіз джерел інформації показує, що генеративні технології вже активно використовуються для створення ігрових персонажів, концепт-арту та інших візуальних елементів [7]. Їх застосування стосується не лише зовнішнього вигляду, а й характерних рис, атрибутів та загального образу. Водночас у цій частині зазвичай виникає проблема, пов’язана з передбачуваністю результатів, єдністю стилю та подальшою узгодженістю зображень із логікою самого процесу гри [8]. Візуальну генерацію слід розглядати не як окремий декоративний елемент, а як важливу частину більш широкого процесу розробки гри, де вона підпорядкована загальному задуму, правилам і механікам.

У сучасних цифрових інструментах для настільного геймдизайну добре видно одну слабку сторону – вони рідко працюють як єдиний процес. Один сервіс зручний для компонентів, інший потрібен для тестування, окремо доводиться вести правила, матеріали або підготовку до спільної роботи. Користувач ніби й має потрібні інструменти, але змушений постійно переносити дані, звіряти матеріали між собою і підлаштовуватися під різну логіку роботи. Для досвідченого дизайнера це ще можна сприймати як нормальний робочий процес, але для звичайного користувача така розрізненість швидко стає перешкодою. Увага зміщується з ідеї гри на технічні дрібниці, хоча саме цього вебсервіс має уникати [9–12].

Персоналізована настільна гра як об’єкт автоматизації не зводиться до одного типу контенту, бо в ній одночасно поєднуються зміст, механіка, візуальна частина та користувацька взаємодія. Саме ця багатошаровість задає складність проблеми і водночас пояснює, чому її доцільно вирішувати через інтелектуальний вебсервіс. Наступний раціональний крок – аналіз сучасних програмних рішень, які вже використовуються у предметній області.

1.3 Аналіз сучасних програмних рішень

Після аналізу характеристик персоналізованих настільних ігор з точки зору автоматизації доцільно перейти до огляду актуальних програмних рішень, що застосовуються у суміжній предметній області. Цей огляд має на меті не лише розглянути різноманітні доступні рішення, а й виявити обмеження їх функціональних можливостей. Розглядаються три варіанти з різними методами підтримки в розробці настільного геймдизайну: Component Studio для вебпроєктування компонентів, nanDECK для автоматизації сценаріїв для карткових наборів та Tabletop Simulator для моделювання та плейтестів готових ігор. Усі три рішення показують, що сучасне програмне забезпечення досягло значного рівня розвитку. Проте кожне з них підтримує процес створення настільної гри переважно на окремому етапі розроблення.

Для більш об’єктивного аналізу сучасних рішень слід врахувати три основні категорії критеріїв:

1) функціональне призначення, тобто які саме етапи створення гри підтримує система;

2) рівень автоматизації, тобто чи обмежується рішення оформленням окремих артефактів, чи бере участь у побудові змісту гри;

3) ступінь цілісності результату, тобто чи дозволяє інструмент отримати повноцінний продукт, а не лише окремі його частини.

Оцінка програмного забезпечення має ґрунтуватися на цих критеріях, адже вони дають змогу чітко визначити передбачувані функції та реальне використання, а також відповідність автоматизованого процесу створення настільних ігор. Такий підхід дозволяє перейти від простого опису наявних рішень до їх змістовного порівняння в межах завдань цієї кваліфікаційної роботи. Увага приділяється не лише технічним можливостям окремих систем, а й тому, наскільки вони придатні для формування цілісного ігрового результату. Такі системи можуть бути не прямими аналогами, але вони повинні охоплювати ключові принципи розробки та засоби організації процесу генерації ігрового контенту.

Component Studio

Component Studio слід розглядати як спеціалізовану веборієнтовану платформу, призначену для розробки компонентів настільних ігор (рис. 1.3). Її функціональна спрямованість пов’язана зі створенням карт, плиток, жетонів та інших друкованих або цифрових компонентів, що використовуються в грі. Згідно з офіційними матеріалами, платформа підтримує генерування компонентів із даних, експорт у різні формати, форматування PDF-файлів для самостійного друку, отримання окремих PNG-файлів, підтримка прямого завантаження у The Game Crafter, а також додатково експорт до сервісу Tabletop Simulator [9, 10]. Це все разом дозволяє класифікувати його як інструмент, який корисно використовувати для оформлення та підготовки ігрових матеріалів для подальшого застосування.

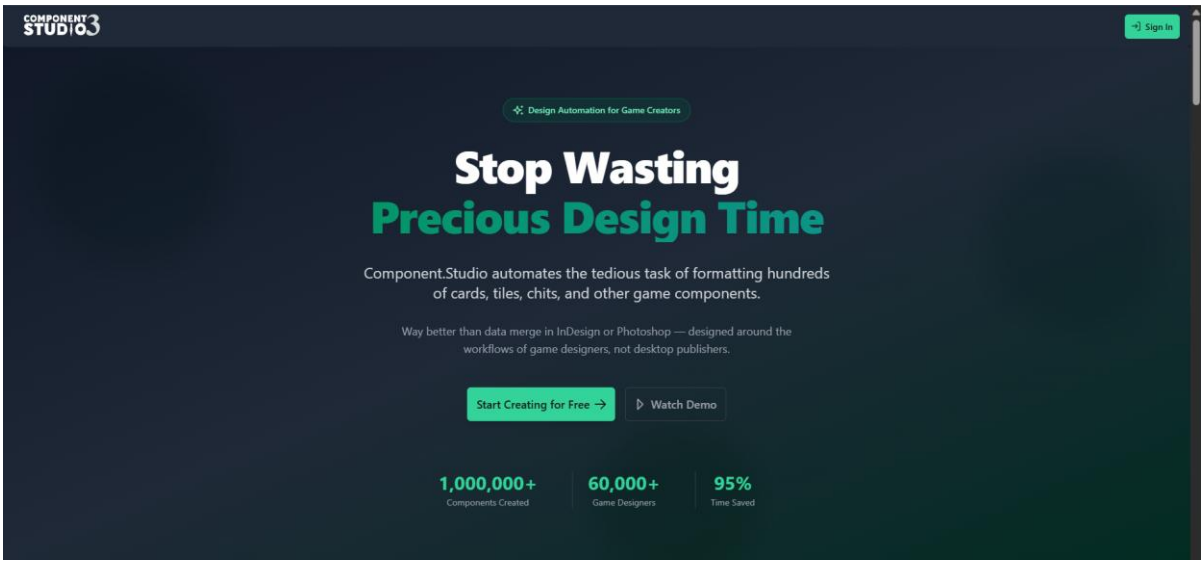

Рисунок 1.3 – Вигляд інтерфейсу Component Studio

Сервіс орієнтований на роботу з наборами компонентів, які можна гнучко створювати, змінювати та експортувати для друку або цифрового тестування. Найбільша його практична цінність проявляється на тих етапах, коли механіка гри вже визначена і потрібно швидко підготувати значну кількість візуально узгоджених матеріалів. По суті, Component Studio добре вирішує одну конкретну задачу, а саме формування та підготовку компонентів для розробників настільних ігор. Водночас функціональність платформи не виходить за межі цієї задачі. Узагальнені характеристики Component Studio наведено в табл. 1.1.

Таблиця 1.1 – Опис Component Studio

Параметр	Характеристика
Назва	Component Studio
Розробник	The Game Crafter
Тип рішення	вебсервіс проєктування ігрових компонентів
Архітектурний підхід	хмарний вебсервіс
Мова реалізації	веборієнтована платформа, точний стек не розкривається у відкритих офіційних матеріалах
Функції	– генерування компонентів на основі даних; – експорт окремих PNG-файлів; – формування PDF-файлів для друку; – пряме завантаження в The Game Crafter; – експорт у форматах, придатних для Tabletop Simulator; – одночасне формування кількох типів експорту.
Переваги	– вебформат роботи та зручність створення великої кількості однотипних елементів; – придатність до друку та цифрового плейтесту; – інтеграція з суміжними сервісами екосистеми The Game Crafter.
Обмеження	– не генерує концепцію гри, механіки й правила; – орієнтована переважно на професійних дизайнерів, а не звичайних користувачів.
Джерело інформації	https://component.studio/ https://www.thegamecrafter.com/

Процес друку та тестування справді став значно простішим завдяки цьому сервісу, але є важливе обмеження, яке варто враховувати. Платформа не взаємодіє з ігровою концепцією, не формує механічну структуру і не розробляє правила. Саме тому Component Studio є орієнтиром, але не функціональним аналогом системи, що розробляється. Її функції краще розглядати як підтримку на певному етапі підготовки ігрових матеріалів, а не як інструмент для повного циклу розроблення.

nanDECK

nanDECK займає принципово іншу нішу в інструментарії настільного геймдизайну (рис. 1.4). Component Studio в основному займається вебпроєктуванням та візуальним відображенням, тоді як nanDECK зосереджується на використанні скриптів для автоматизації карткових наборів. Офіційне визначення програмного забезпечення вказує на те, що воно розроблено для Windows і призначене для оптимізації проєктування та друку колод карт, що робить його особливо корисним на етапах прототипування та тестування [11]. Згідно з переліком можливостей, nanDECK пропонує інтеграцію з Excel, OpenOffice, Google Sheets та CSV. Серед його функціоналу є внутрішній редактор, рендеринг у реальному часі, віртуальний стіл, візуальний редактор, комбінаторний модуль і Monte-Carlo симулятор. Такий набір інструментів робить nanDECK одним із найуніверсальніших рішень для автоматизації створення карткових систем.

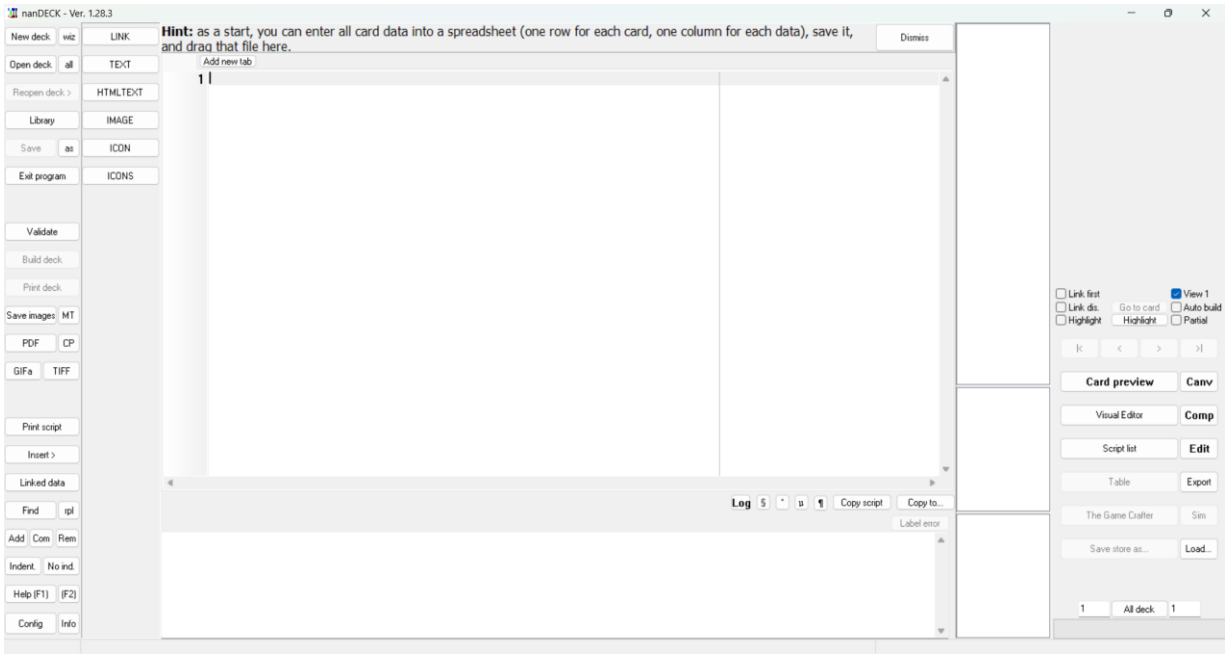

Рисунок 1.4 – Вигляд інтерфейсу nanDECK

Важливо підкреслити, що nanDECK орієнтований не на спрощене редагування компонентів, а більше акцентує увагу на технічному підході. Користувач фактично управляє структурою карт за допомогою сценаріїв, таблиць даних і параметрів генерації. Узагальнені характеристики nanDECK наведено в табл. 1.2.

Таблиця 1.2 – Опис nanDECK

Параметр	Характеристика
Назва	nanDECK
Розробник	Nand
Тип рішення	локальний застосунок для Windows
Архітектурний підхід	локальний настільний інструмент
Мова реалізації	у відкритих офіційних матеріалах не акцентується, для користувача застосовується власний сценарний синтаксис nanDECK
Функції	– генерація колод і графічних елементів на основі сценаріїв; – робота з Excel, OpenOffice, Google Sheets і CSV; – рендеринг у реальному часі; – візуальний редактор; – віртуальний стіл для плейтесту; – комбінаторний модуль і Monte-Carlo симулятор; – шаблони для друкарських сервісів; – створення комбінованих зображень для Tabletop Simulator.
Переваги	– висока гнучкість; – потужна автоматизація карткових систем; – корисність для прототипування; – підтримка складних сценаріїв побудови колод
Обмеження	– високий поріг входу; – орієнтація на технічно підготовленого користувача; – відсутність генерації концепції та правил; – відсутність вебсервісного сценарію повного циклу.
Джерело інформації	https://nandeck.com/

Таким чином, nanDECK – це добре розвинений і технічно потужний інструмент для автоматизації процесу створення карткових наборів. Проте його можливості не охоплюють повний цикл розробки гри. Щоб створити повноцінний персоналізований ігровий продукт, який буде зручним для користувачів без досвіду в геймдизайні, цього інструменту недостатньо. Найбільшою практичною цінністю є саме задача автоматизації карткових компонентів.

Tabletop Simulator

Tabletop Simulator (рис. 1.5) представляє третій тип сучасних програмних рішень. На відміну від двох попередніх рішень, він орієнтований не просто на створення компонентів, а саме на цифрове відтворення, демонстрацію та тестування ігор. Платформа має широкий спектр можливостей для створення ігор. Головні функції платформи: імпорт ресурсів, інтеграція зі Steam Workshop, 3D-моделювання та робота зі скриптуванням мовою Lua. Користувач може зберігати та завантажувати свої ігрові об’єкти, використовувати hotseat-режим і адмініструвати гру [12]. Саме тому Tabletop Simulator є потужним інструментом для цифрового моделювання.

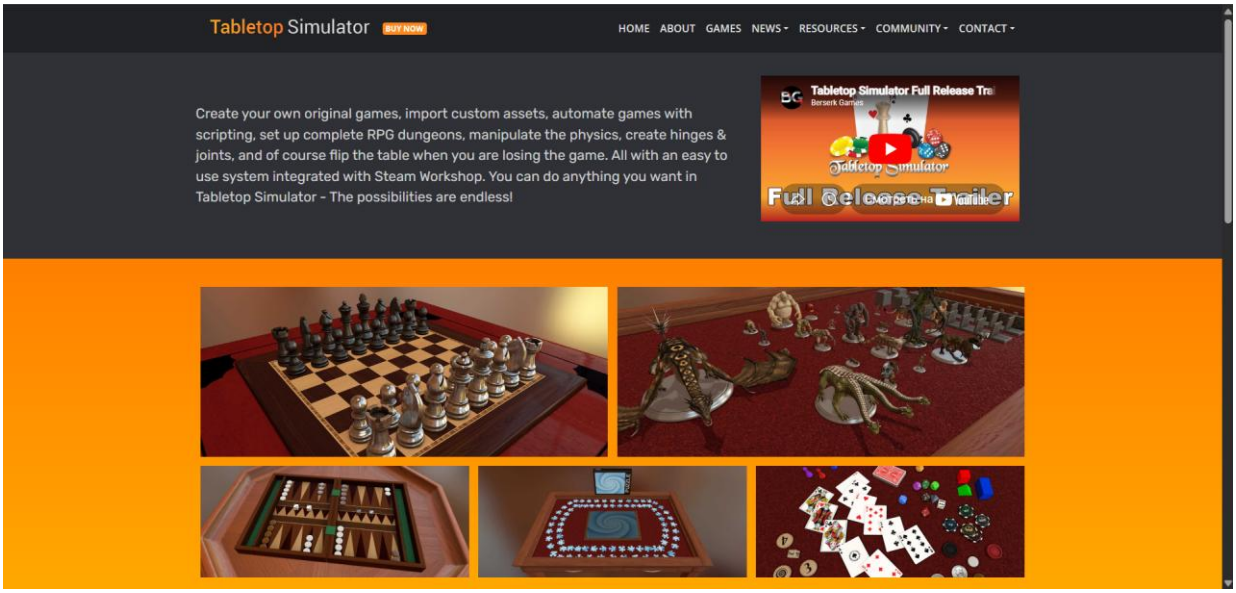

Рисунок 1.5 – Вигляд інтерфейсу Tabletop Simulator

Ця платформа надає якісне середовище для взаємодії з ігровими елементами, командного геймплею та випробування різних механік. Тому вона є особливо цінною під час тестування прототипів, але не на стадії початкового створення гри. Основну роботу, що стосується змісту, механіки та компонентів, слід заздалегідь завершити поза платформою. Саме тому Tabletop Simulator варто розглядати як засіб для кінцевого цифрового моделювання. У контексті цього дослідження він виступає важливим додатковим інструментом, хоча не є повноцінним функціональним аналогом системи, що розробляється. Узагальнені характеристики Tabletop Simulator наведено в табл. 1.3.

Таблиця 1.3 – Опис Tabletop Simulator

Параметр	Характеристика
Назва	Tabletop Simulator
Розробник	Berserk Games
Тип рішення	локальна платформа цифрової симуляції настільних ігор
Архітектурний підхід	локальне багатокористувацьке середовище
Мова реалізації	у відкритих офіційних матеріалах не розкривається; для автоматизації підтримується скриптування мовою Lua
Функції	– середовище фізичного моделювання; – багатокористувацький режим до 10 користувачів; – Steam Workshop; – імпорт власних матеріалів і 3D-моделей; – скриптування мовою Lua; – збереження, завантаження об’єктів і готових ігор; – hotseat-режим; – інструменти адміністратора.
Переваги	– потужне середовище плейтесту; – підтримка модифікацій і кастомного контенту; – багатокористувацький формат; – висока придатність до цифрової демонстрації прототипу.
Обмеження	– не генерує концепцію гри; – не формує механіки та правила; – вимагає попередньо підготовлених матеріалів; – не виконує персоналізовану ШІ-генерацію.
Джерело інформації	https://www.tabletopsimulator.com/

З точки зору теми кваліфікаційної роботи Tabletop Simulator не є інструментом для автоматичного створення ігор. Його роль інша – цифрове моделювання вже готового прототипу. Саме тому його доцільніше відносити до суміжних рішень, які підтримують заключний етап роботи над грою, але не торкаються питань концепції, механіки, правил і компонентів. Платформа не замінює процес створення гри, вона лише дає змогу перевірити те, що вже створено. А це інша задача, ніж та, яку вирішує система, що розробляється.

Узагальнення результатів аналізу сучасних програмних рішень

Аналіз сучасних програмних рішень показав, що наявні інструменти для розробки настільних ігор справді досягли високого рівня зрілості. Кожен із них добре справляється зі своєю задачею. Проте є одна спільна риса, яка простежується у всіх розглянутих рішеннях. Розглянуті програмні рішення залишаються вузькоспеціалізованими. Одні допомагають з компонентами, інші з тестуванням, але жодне не охоплює повний цикл. Саме це і стало підставою для розроблення інтелектуального вебсервісу, який об’єднує основні етапи створення гри в єдиному середовищі, має чіткий сценарій для користувача і дає на виході готовий продукт, а не набір розрізнених артефактів.

Висновки до розділу 1

У першому розділі виконано аналіз процесів створення персоналізованих настільних ігор. Це дало змогу підтвердити актуальність обраної теми й окреслити характерні риси досліджуваного об’єкта та оцінити сучасний стан програмних рішень у суміжній предметній області. Аналіз показав, що розвиток генеративного штучного інтелекту відкриває нові можливості для створення прикладних вебсервісів, але у створенні ігрового продукту головним є не просто застосування генеративних моделей, а їхня інтеграція в структурований і злагоджений процес.

Визначено, що персоналізована настільна гра є багатокомпонентним продуктом, що включає в себе узгоджену концепцію, механіки, правила, компоненти та візуальне оформлення. Аналіз сучасних програмних рішень виявив, що існуючі інструменти мають зрілу функціональність, проте здебільшого вони концентруються на вузьких аспектах і не пропонують повного циклу автоматизованого процесу створення гри в межах єдиного сервісу. Отже, результати проведеного дослідження підтверджують актуальність і доцільність розроблення інтелектуального вебсервісу, спрямованого на розроблення завершеного ігрового продукту.

2 ВИБІР ЗАСОБІВ РОЗРОБКИ ТА ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

2.1 Аналіз сучасного інструментарію розроблення інтелектуальних вебсервісів

Після аналізу предметної області стало зрозуміло, що сервіс не можна зробити як звичайний CRUD-застосунок (Create, Read, Update, Delete), бо генерація триває довго і має не один етап. Система повинна включати клієнтський інтерфейс, модель даних, фонову обробку тривалих завдань, серверний API, підключення до моделей штучного інтелекту та інструменти для їх обробки, а також відтворювану інфраструктуру. У цій роботі технології обиралися під конкретні задачі, наприклад користувач запускає генерацію, API створює задачу, worker-модуль виконує її у фоні, а результат зберігається в базі даних.

Запит користувача проходить не один рівень системи. Спочатку frontend, потім API, база даних, черга і worker-модуль, а далі діють модулі взаємодії з генеративними моделями та інфраструктурний шар. Оскільки генерація не завершується одразу, її не можна тримати всередині одного HTTP-запиту. Система запускає послідовний процес, який містить обробку генерації, збереження проміжних даних та повернення остаточного результату.

Вибір інструментарію здійснено відповідно до наступних критеріїв:

- узгоджені типи даних між frontend, backend і worker-модулем;
- єдина мова програмування для ключових компонентів системи;
- модульна серверна логіка без надлишкової складності;
- надійна реляційна модель із контрольованими міграціями;
- винесення тривалих операцій ШІ за межі HTTP-запиту;
- локальна інтеграція з текстовою та візуальною генерацією;
- відтворюване середовище запуску;
- можливість розширювати систему без переходу до мікросервісів.

Для організації проєкту було обрано монорепозиторій на основі pnpm workspace та Turborepo [13, 14]. Застосунок складається з web, API та worker-модуля, які разом утворюють один продукт і відповідають за різні функції.

Головна причина вибору монорепозиторію – підтримання спільної структури, конфігурації та залежностей в одному просторі. Це логічний підхід для проєкту, бо він зменшує ймовірність того, що інтерфейс, API та фоновий процес випадково відійдуть один від одного.

Next.js, React та TypeScript утворюють клієнтську частину. Вебсервіс базується на Next.js, який відповідає за організацію маршрутів сторінок та загальну логіку створення інтерфейсу. TypeScript необхідний для покращення надійності системи, оскільки він дозволяє типізувати дані, якими обмінюються різні рівні вебсервісу. Важливими під час розроблення є саме типи, бо форма гри, DTO-запити (Data Transfer Object), статуси задач і результати генерації мають бути сумісні не лише на рівні значень, а й на рівні типів [15–17].

Використання користувацького сценарію є основою для вибору технологічного складу frontend. Вебсервіс пропонує не тільки звичний формат з однією відповіддю. Користувач має проходити покроковий процес на сторінці, який включає в себе введення параметрів гри, запуск генерації та перегляд статусу виконання. Після того як користувач зробив всі ці дії, він отримує результат. Для таких задач інтерфейс має вміти управляти станом, мати компонентну структуру та забезпечувати надійну взаємодію з серверною частиною.

Для стилізації інтерфейсу використано Tailwind CSS. Він потрібен, щоб забезпечити гнучке оформлення сторінки, адаптивність і підтримку єдиного візуального стилю без надмірного дублювання CSS-коду. Інтерфейс має вести користувача через створення гри без зайвої плутанини, показувати поточний етап, збирати дані поступово й нормально адаптуватися до різних екранів [18].

NestJS використовується як основа серверної частини вебсервісу [19]. Backend у проєкті бере на себе запуск і контроль генерації. Саме тут запускається процес генерації. Сервер приймає параметри від користувача, потім перевіряє їх, далі створює запис про генерацію, передає завдання до черги й повертає ідентифікатор для відстеження статусу. Якщо всю серверну логіку залишити тільки в маршрутах, код швидко стане важким для підтримки. У проєкті є різні частини: генерація, користувачі, базові операції, інтеграція зі штучним інтелектом та інші

компоненти. Найкраще їх відокремити одразу, а не поєднувати в одному рівні коду. Тому модульна архітектура NestJS тут є практичним рішенням, а не просто особливістю фреймворку.

Для взаємодії frontend і backend використано REST API. Це добре підходить до логіки сервісу, бо основні запити не є складними вкладеними вибірками. Користувач запускає генерацію, перевіряє статус, відкриває результат або проходить перевірку доступу. У таких сценаріях REST залишається простим і достатнім рішенням. Дослідження NestJS з GraphQL також показує, що REST краще працює для простих запитів і навантажених сценаріїв [20]. Розроблюваний вебсервіс в основному функціонує зі сценаріями, такими як створення задачі, отримання статусу та отримання результату. REST дає важливі речі – простоту, прозорість і технічну точність, що робить його кращим для таких задач.

PostgreSQL у проєкті потрібен як основне джерело даних [21]. Крім того, PostgreSQL утримує зв’язки між сутностями. Система працює з кількома пов’язаними сутностями, серед них користувачі, сесії, параметри генерації, статуси виконання, текстові результати, візуальні артефакти, платіжні дії та нарешті готові ігри. Наприклад, завершена гра повинна бути пов’язана з певним процесом генерації, а цей процес – з користувачем та його запитом. У цій архітектурі важливо стежити за дотриманням цілісності зв’язків, а не лише за збереженням. Транзакції, зв’язки та строга схема важливіші за звичайне збереження JSON-подібних даних.

Для роботи з базою даних використовується Prisma ORM [22]. Prisma потрібна для опису моделі даних, роботи із запитами до PostgreSQL та керування міграціями. Окремо важливо, що вона підтримує типізований доступ до даних. Для серверної частини помилка в моделі даних не є локальною проблемою. Якщо некоректно описано поле генерації або зв’язок між готовою грою і користувачем, наслідок може проявитися вже на наступних етапах pipeline (генераційного конвеєра). Наприклад, якщо змінюється поле в Generation або Game, це швидше проявляється в коді, а не тільки під час виконання. Це не усуває всі помилки, але зменшує ризик передати в запит неправильне поле або використати властивість, якої немає в моделі.

Описувати серверну частину тільки через API-маршрути було б неповно. Частина її логіки закріплена нижче – у таблицях, моделях і зв'язках між користувачами, генераціями, платіжними подіями та готовими іграми. Тому під час аналізу архітектури варто враховувати не тільки контролери й сервіси NestJS, а й структуру бази даних. Тому таблиці й представлення варто розглядати не як другорядні елементи, а як частину архітектури програмного забезпечення [23].

У серверній частині також виділено логіку автентифікації, авторизації та доступу до генерацій. Повний цикл охоплює реєстрацію, вхід, перевірку ролей і доступ до створення генерацій. Генерація в цьому сервісі є не просто технічною дією. Процес генерації потребує ресурсів, тому доступ до неї регулюється бізнес-правилами. Для комерційних сценаріїв застосовується Stripe. Система включає створення платіжних сеансів, обробку webhook-подій, облік придбаних кредитів генерації і підтримку підписки.

Одна з ключових технічних частин проєкту – фонове виконання задач. Генерацію гри недоцільно виконувати всередині одного HTTP-запиту. У такому разі сервер довго утримує запит, а будь-яка затримка моделі ШІ може зірвати весь процес. Тому в проєкті використовуються Redis, BullMQ та окремий worker-модуль, написаний на TypeScript [24, 25].

Redis – швидкий шар даних у пам'яті для черги, тоді як BullMQ відповідає за постановку, обробку та моніторинг задач. Окремий worker-модуль винесено для того, щоб API не виконував довгі операції ШІ самостійно. Такий розподіл узгоджується з підходом, за якого етапи pipeline можна контролювати й масштабувати окремо. У дослідженні щодо pipeline обробки даних акцент робиться на тому, що модульна архітектура дозволяє масштабувати компоненти окремо. Такий підхід покращує стійкість до збоїв, забезпечує гнучкість та полегшує процес моніторингу [26]. У цьому сервісі це проявляється просто. HTTP-рівень швидко повертає відповідь, а генерація виконується окремо.

З точки зору архітектури, систему точніше описувати як модульну багатокомпонентну з окремим worker-контуром, а не як повноцінну мікросервісну. Компоненти не виділені в незалежні бізнес-сервіси з власними циклами

розгортання, але й простим монолітом це назвати не можна, адже web, api і worker-модуль мають різні зони відповідальності і не змішують їх між собою.

Така архітектура дає системі хорошу гнучкість і дозволяє масштабувати окремі частини незалежно. Водночас вона вимагає уважніше ставитися до динамічного масштабування, продуктивності, витрат і безпеки [27]. Як показують дослідження переходу від монолітів до мікросервісів, мікросервісний підхід справді підсилює масштабованість і стійкість до збоїв [28]. Але разом із цим з'являється інша сторона: складніше налаштовувати моніторинг, контролювати залежності й підтримувати систему в цілому. У цій роботі немає потреби ділити систему на багато окремих сервісів, бо основна логіка залишається спільною. Поділ на web, API і worker-модуль достатньо розмежовує відповідальність без перевантаження.

Текстова генерація в проєкті винесена на Ollama. Важливо, що модель працює локально, а не через зовнішню платформу. API-доступ дозволяє підключити її до генераційного процесу і при цьому зберегти контроль над середовищем розроблення та демонстрації. Окремо логіка оркестрації сконцентрована на серверній частині та робочих процесах. У генераційному процесі Ollama використовується для формування тематичного наповнення [29].

Для генерації зображень використовується ComfyUI. Він є окремим виконавчим рівнем, не пов'язаним із вебінтерфейсом напряму. Його принципова відмінність у тому, що процес генерації зображень описується явним workflow-графом (графом робочого процесу) із вузлів і зв'язків [30]. У системі ComfyUI створює візуальні матеріали, які потім передаються у контрольований рендеринг. Без контрольованого процесу цей інструмент не є самостійним для задачі генерації зображень.

Інфраструктурний рівень забезпечено за допомогою Docker, який у проєкті потрібен для запуску PostgreSQL і Redis. Було обрано Docker, бо він робить запуск PostgreSQL і Redis передбачуваним на різних машинах та зменшує залежність від локальних налаштувань середовища [31]. Узагальнення використаного інструментарію наведено в табл. 2.1.

Таблиця 2.1 – Інструментарій розроблення інтелектуального вебсервісу

Рівень системи	**Інструмент**	**Архітектурна відповідальність**	**Аргумент вибору**
Організація кодової бази	pnpm workspace, Turborepo	монорепозиторій, спільні пакети, запуск задач	узгоджений розвиток web, api і worker в одному репозиторії
Клієнтський рівень	Next.js, React, TypeScript	сторінки, компоненти, стан інтерфейсу, типи даних	сильна компонентна модель і спільна типізація з backend
Серверний рівень	NestJS	модулі, контролери, сервіси, координація процесу	модульна структура добре відповідає логіці сервісу
API-рівень	REST API	запуск генерації, отримання статусу, отримання результату	оптимальний формат для простих і прямих сценаріїв
Модель даних	PostgreSQL	зберігання користувачів, генерацій, статусів, результатів	природна реляційна модель для зв’язаних сутностей
Доступ до даних	Prisma	схема, міграції, типобезпечні запити	зменшує кількість помилок і пов’язує БД з TypeScript-моделлю
Асинхронне виконання	Redis, BullMQ	черга задач і диспетчеризація	розводить HTTP-рівень і тривалу генерацію
Виконання задач	worker на TypeScript	поетапна обробка генерації та оновлення стану	ізолює важкі операції від основного API
Текстовий ШІ-рівень	Ollama	локальні виклики мовної моделі через API	контрольоване середовище без залежності від зовнішнього хмарного сервісу

Кінець таблиці 2.1

Візуальний ШІ-рівень	ComfyUI	workflow-граф для генерації зображень	керований і відтворюваний конвеєр візуальної генерації
Інфраструктура	Docker	контейнерний запуск залежностей	стабільне локальне середовище для PostgreSQL і Redis

Обраний стек працює як практична основа для сервісу, де кожен інструмент має свою роль, а система не перевантажується зайвими шарами. Це залишає можливість розвивати проєкт далі, поступово покращувати генерацію, візуальні матеріали та персоналізацію без повної перебудови архітектури з нуля. Це важливо, бо проєкт ще можна розвивати після першої робочої версії. Нові можливості можна додавати поступово, не переробляючи весь сервіс заново. Визначений стек дає основу для специфікації вимог до ПЗ, а також аналізу моделей і методів генерації ігрового контенту.

2.2 Аналіз моделей і методів генерації ігрового контенту

Після вибору інструментів розроблення потрібно окремо розглянути моделі і методи, які можуть використовуватися для генерації ігрового контенту. Зведення до одного універсального методу створення настільної гри не є точним підходом. У задачі розробки ігор потрібні одночасно зміст, правила, механіки, картки, візуальні елементи для цих карток та зв’язок між ними. У межах цієї роботи розглядаються не лише генеративні моделі, а ще й методи організації результату. Це дозволить забезпечити стабільну та придатну до використання гру, яка не потребує кардинальних змін після генерації. Для таких задач, які були названі, доцільно розглянути мовні моделі, дифузійні моделі, процедурну генерацію, шаблонний підхід та їх поєднання.

У наукових дослідженнях генеративні моделі розглядаються як інструменти для створення нових текстів, графіки та мультимедійних матеріалів [1]. Створення

ігрового контенту не зводиться лише до створення нових варіантів. Важливо також мати механізми контролю якості, гарантувати різноманітність та забезпечити можливість управління результатами, отриманими від моделі [32, 33]. У реалізації проєкту створення гри побудовано як послідовний генераційний pipeline.

Перша група для розгляду – **мовні моделі**. У межах кваліфікаційної роботи вони корисні в задачах створення текстової та змістової частини гри. Мовні моделі швидко створюють різноманітні варіанти тексту гри та добре працюють зі стилем або тематикою.

У роботах про генеративний ШІ в ігровому дизайні добре простежується думка, що найбільше користі він дає там, де потрібно швидко підготувати ідеї, кілька варіантів контенту або перший прототип. Це справді прискорює роботу, особливо на ранніх етапах, коли ще потрібно шукати форму майбутньої гри. Але згенерований матеріал ще не можна автоматично вважати готовим результатом. Його потрібно впорядкувати, перевірити й привести до такої форми, у якій він уже може використовуватися на практиці. Для персоналізованої настільної гри це особливо помітно, бо користувач очікує не випадкові фрагменти правил, назв чи описів, а цілісний ігровий продукт зі зрозумілою логікою. Інакше генерація залишається лише допоміжною чернеткою, а не повноцінною основою для гри [4, 5].

Мовна модель у проєкті застосовується для створення тематичного елемента гри. Вона формує ігровий світ, атмосферу, ролі гравців, назви ресурсів, дій, зон, колод та типів карт, а також короткі текстові описи. Уся гра не визначається моделлю самостійно, вона створюється в межах уже заданої структури механіки.

У pipeline відповідь мовної моделі не передається далі як звичайний суцільний текст. Відповіді моделі мають повертатися у форматі, зручному для автоматичного опрацювання, зокрема у вигляді JSON-структури. На практиці це спрощує перевірку результату. Можна одразу побачити, чи модель не пропустила поле, не додала зайвий текст або не повернула значення у неправильному форматі.

Другу групу становлять **дифузійні моделі**. У межах цієї роботи дифузійні моделі доцільно розглядати як моделі для генерації картинок, зокрема для карток,

ігрового поля та інших елементів гри. Важливим є підхід, де не змішуються самі дифузійні моделі та засоби їх організації. Наприклад, як було описано раніше, інструмент ComfyUI допомагає організувати workflow, тобто це включає в себе запити, виконання генерації, отримання результату та передавання цих візуальних матеріалів далі в систему.

Проте не варто покладати всю візуальну частину на одну генеративну модель. Окреме зображення вона часто створює переконливо, але серія матеріалів швидко виявляє слабкі місця, а саме змінюється стиль, губиться функція елемента, а картинка починає жити окремо від правил. Це і є характерною ознакою такого виду технологій і слабким місцем, що потребує підходу, де потрібно контролювати результат дифузійної моделі.

Уже розглянуті моделі дають основу для переходу до методів організації ігрового контенту. Першим методом є **процедурна генерація**. У контексті настільної гри процедурна генерація може відповідати за кількість раундів, типи дій, структуру карток, базові параметри компонентів, логіку поля, зони, ресурси або умови завершення партії. Дослідження з процедурної генерації контенту в іграх показують, що такий підхід важливий саме там, де потрібно отримувати різні варіанти контенту, але не втрачати контроль над його структурою [32, 33].

Основною цінністю процедурного підходу до генерації є забезпечення стабільності. Наприклад, якщо є певний ресурс або дія в грі, то вони мають бути пов’язані між собою у правилах, картках, на самому ігровому полі та в системі підрахунку. Якщо це не зробити і не контролювати зв’язки, то гарантовано гра буде цікавою за темою, але нелогічною і незрозумілою як повноцінна система. Але є і недоліки, а саме відсутність живої персоналізації. Може бути створений каркас гри, але без мовної і візуальної генерації результатом буде «суха» і не персоналізована гра для групи гравців.

Окремої уваги потребує **шаблонний підхід до збірки гри**. Варто помітити, що у рамках створення настільної гри це не є ознакою слабкої генерації. Шаблон дає змогу втримати гру в робочому стані та не дати грі розпастись за логікою. Це включає в себе базову механіку, кількість етапів, типи взаємодій між гравцями,

структуру карток та ігрового поля тощо. Система не отримує хаотичний набір ідей і не починає все з нуля, а навпаки спирається на зрозумілий каркас.

Саме завдяки шаблонному підходу генеративні моделі стають керованими. Обидва аспекти є важливими для теми кваліфікаційної роботи. Така комбінація є відображенням стабільності і персоналізації. Шаблон наповнюється темою завдяки мовній моделі, включаючи назви, описи та атмосферу гри. Візуальна генерація може створювати матеріали для цього змісту. Процедурна логіка стежить за кількістю і структурою елементів. Шаблон не обмежує персоналізацію, а задає рамки, у яких вона стає надійною.

Найбільш доцільним для розроблюваного вебсервісу є **гібридний підхід до генерації** ігрового контенту. Це не просто одночасне використання деяких інструментів, а більш глибокий підхід з розподілом ролей між цими інструментами. У такому випадку генеративні моделі не працюють ізольовано, а наповнюють уже підготовлену структуру цікавим змістом і візуальними елементами. Програмна логіка контролює послідовність цих етапів генерації, формату даних і логічний зв’язок між результатами. При такій комбінації вся якість гри не залежить лише від однієї відповіді моделі. Гібридний підхід робить процес більш керованим. Гра наповнюється ігровим і цікавим змістом, але водночас зберігає механічну основу, логіку компонентів і придатність до подальшого використання.

Отже, для генерації персоналізованої настільної гри недостатньо використати тільки мовну модель, тільки дифузійну модель або тільки процедурний метод. Гра потребує поєднання творчого наповнення і структурного контролю. Саме гібридний підхід дає можливість сформувати не набір розрізнених фрагментів, а цілісний ігровий продукт, у якому тема, правила, компоненти та візуальні матеріали працюють разом. Розглянуті моделі та методи генерації ігрового контенту визначають логіку системи та її підтримувані функції. На основі цього доцільно перейти до специфікації вимог до ПЗ. Вона має враховувати коректну роботу системи, взаємодію з користувачем, безпеку даних та інші важливі характеристики сервісу.

2.3 Специфікація вимог до програмного забезпечення

1. Призначення та межі проєкту

1.1 Призначення системи

Вебсервіс призначений для автоматизованої генерації персоналізованих настільних ігор на основі параметрів, заданих користувачем. Система забезпечує формування концепції гри, правил, ігрових компонентів, карток, ігрового поля, візуальних матеріалів та експортного пакета цифрових файлів.

Сервіс орієнтований на користувачів, які не мають професійного досвіду в геймдизайні, але хочуть отримати структурований ігровий продукт для конкретної групи гравців, тематики або сценарію використання.

1.2 Погодження, ухвалені в програмній документації

– специфікація вимог базується на технічному завданні проєкту;

– погоджено асинхронну обробку генерації через worker-модуль;

– погоджено використання генеративного ШІ для створення текстових і візуальних матеріалів;

– результати генерації зберігаються в системі й можуть бути повторно переглянуті користувачем.

1.3 Межі проєкту ПЗ

Проєкт охоплює розробку вебінтерфейсу, API-сервера, worker-модуля генерації, бази даних, механізму авторизації, платіжного доступу, перегляду результатів, експорту цифрових матеріалів і галереї прикладів згенерованих ігор.

Проєкт не охоплює фізичний друк і доставку компонентів, розробку нативних мобільних застосунків та повну автоматичну гарантію балансу гри без подальшого тестування людиною. Система не охоплює всі види настільних ігор.

2. Загальний опис

2.1 Сфера застосування

Вебсервіс використовується у навчальних активностях, командних подіях, творчих експериментах або демонстраційних проєктах. Результат роботи системи – цифровий набір ігрових матеріалів, які готові до застосування та друку.

2.2 Характеристики користувачів

Система розрахована на такі категорії користувачів:

– FREE-користувач – зареєстрований користувач із базовим доступом, зокрема до першої безкоштовної генерації;

– PREMIUM-користувач – зареєстрований користувач із розширеним доступом на основі підписки або придбаних кредитів генерації.

– адміністратор, який має доступ до функцій керування користувачами, генераціями та архівом ігор. Він не належить до комерційних режимів доступу.

Незареєстрований відвідувач може переглядати загальнодоступні сторінки сервісу, сторінки входу та реєстрації, а також публічні приклади ігор у галереї.

2.3 Загальна структура і склад системи

Система складається з таких основних компонентів:

– клієнтська частина – вебінтерфейс на Next.js, React і TypeScript;

– серверна частина – REST API на NestJS;

– worker-модуль – фонова обробка завдань генерації;

– база даних PostgreSQL із доступом через Prisma ORM;

– черга завдань Redis/BullMQ;

– ШІ-рівень для текстової та візуальної генерації;

– платіжна інтеграція Stripe;

2.4 Загальні обмеження

– для роботи сервісу потрібне стабільне інтернет-з’єднання;

– тривалість генерації залежить від складності параметрів, навантаження на worker-модуль і доступності ШІ-компонентів;

– результат може потребувати додаткового тестування користувачем;

– система формує цифрові матеріали, але не виконує фізичний друк гри.

3. Функції системи

3.1 Реєстрація та авторизація користувачів

3.1.1 Опис функції

Функція забезпечує створення облікового запису, вхід у систему, перевірку поточної сесії та вихід користувача.

3.1.2 Вхідна і вихідна інформація

Вхідна інформація: ім’я користувача, електронна пошта, пароль.

Вихідна інформація: статус операції, дані користувача, активна сесія, режим доступу FREE або PREMIUM.

3.1.3 Функціональні вимоги

– система повинна забезпечувати реєстрацію користувача;

– система повинна перевіряти коректність ел. пошти, пароля та імені;

– система повинна зберігати пароль у захищеному вигляді;

– система повинна забезпечувати вхід і вихід користувача;

– система повинна перевіряти сесію перед доступом до захищених функцій;

– система повинна обмежувати кількість невдалих спроб входу.

3.2 Формування параметрів гри

3.2.1 Опис функції

Функція забезпечує введення початкових параметрів, на основі яких система створює персоналізовану настільну гру.

3.2.2 Вхідна і вихідна інформація

Вхідна інформація: назва гри, кількість гравців, тривалість, жанр, тональність, режим генерації, додаткові побажання, профілі гравців.

Вихідна інформація: сформована чернетка гри та набір параметрів для запуску генерації.

3.2.3 Функціональні вимоги

– система повинна надавати покроковий майстер створення гри;

– система повинна підтримувати вибір кількості гравців, тривалості, жанру та тональності;

– система повинна отримувати актуальні параметри генерації з backend;

– система повинна перевіряти правильність введених даних;

– система повинна зберігати чернетку під час заповнення форми;

– система повинна відображати підсумок параметрів.

3.3 Запуск і виконання генерації

3.3.1 Опис функції

Функція забезпечує запуск генерації, перевірку доступу користувача, створення завдання та його виконання у фоновому worker-модулі.

3.3.2 Вхідна і вихідна інформація

Вхідна інформація: ідентифікатор користувача, параметри гри, режим доступу, дані про безкоштовну генерацію, кредити або підписку.

Вихідна інформація: ідентифікатор генерації, статус, прогрес, список етапів, повідомлення про помилку або успішний результат.

3.3.3 Функціональні вимоги

– система повинна запускати генерацію лише для авторизованого користувача;

– система повинна перевіряти право користувача на генерацію в межах FREE або PREMIUM;

– система повинна резервувати доступ перед запуском генерації;

– система повинна створювати запис генерації та її етапів;

– система повинна передавати завдання до черги worker-модуля;

– система повинна виконувати генерацію за етапами CONCEPT, MECHANICS, RULES, COMPONENTS, IMAGES, VALIDATION, ASSEMBLY;

– система повинна оновлювати статус і прогрес генерації;

– система повинна завершувати генерацію статусом COMPLETED або FAILED.

3.4 Формування ігрових матеріалів

3.4.1 Опис функції

Функція забезпечує створення основного змісту гри: концепції, правил, механік, компонентів, карток, поля та візуальних матеріалів.

3.4.2 Вхідна і вихідна інформація

Вхідна інформація: параметри гри, механічний каркас гри, тема, профілі гравців, налаштування ШІ-рівня.

Вихідна інформація: правила, презентаційний опис, компоненти, картки, ігрове поле, планшети гравців, SVG/PNG-матеріали, JSON-структури.

3.4.3 Функціональні вимоги

– система повинна формувати концепцію та тему гри;

– система повинна створювати структуру ігрової механіки;

– система повинна генерувати правила гри;
– система повинна формувати перелік компонентів і карток;
– система повинна створювати візуальні матеріали карток;
– система повинна формувати ігрове поле;
– система повинна формувати планшети гравців, якщо це передбачено структурою гри;
– система повинна узгоджувати текстові та візуальні матеріали.

3.5 Перегляд результатів і повторна генерація

3.5.1 Опис функції

Функція забезпечує перегляд поточного стану генерації, фінального результату та повторний запуск генерації на основі попередніх параметрів.

3.5.2 Вхідна і вихідна інформація

Вхідна інформація: ідентифікатор генерації, сесія користувача.

Вихідна інформація: статус генерації, прогрес, етапи, правила, компоненти, картки, поле, повідомлення про помилки.

3.5.3 Функціональні вимоги

– система повинна відображати статус генерації;
– система повинна періодично оновлювати прогрес;
– система повинна показувати активний і завершені етапи;
– система повинна відображати фінальний результат генерації;
– система повинна дозволяти повторний запуск генерації;
– система повинна обмежувати доступ до результату власнику генерації.

3.6 Збереження ігор і галерея

3.6.1 Опис функції

Функція забезпечує збереження результатів генерації та формування галереї публічних прикладів ігор.

3.6.2 Вхідна і вихідна інформація

Вхідна інформація: результат генерації, метадані гри, користувач, режим видимості.

Вихідна інформація: збережена гра, список власних ігор, список публічних прикладів.

3.6.3 Функціональні вимоги

– система повинна зберігати завершену генерацію як ігровий результат;

– система повинна зберігати основні метадані гри;

– система повинна підтримувати режими видимості;

– система повинна за замовчуванням зберігати користувацькі ігри як приватні;

– система повинна підтримувати галерею публічних прикладів згенерованих ігор.

3.7 Експорт цифрових матеріалів

3.7.1 Опис функції

Функція забезпечує формування експортного пакета матеріалів гри для перегляду, архівування, редагування або підготовки до друку поза межами системи.

3.7.2 Вхідна і вихідна інформація

Вхідна інформація: завершений результат генерації, правила, компоненти, картки, поле, візуальні матеріали.

Вихідна інформація: PDF-пакет, ZIP-архів, Markdown-правила, JSON-файли, SVG/PNG-матеріали, маніфест рендерингу.

3.7.3 Функціональні вимоги

– система повинна формувати експортний пакет після завершення генерації;

– система повинна включати до пакета правила, компоненти, картки, поле та візуальні матеріали;

– система повинна зберігати посилання на сформовані файли;

– система повинна забезпечувати доступ до матеріалів власнику;

– система повинна забезпечувати зв’язок між результатом і файлами.

3.8 Платіжний доступ

3.8.1 Опис функції

Функція забезпечує допоміжний комерційний доступ до генерацій. Платіжний модуль дає змогу монетизувати генерацію через кредити та підписку.

3.8.2 Вхідна і вихідна інформація

Вхідна інформація: користувач, тип оплати, платіжна сесія, статус підписки, кількість кредитів.

Вихідна інформація: посилання на оплату, оновлений режим доступу, кількість доступних генерацій, історія платіжних подій.

3.8.3 Функціональні вимоги

– система повинна підтримувати першу безкоштовну генерацію;
– система повинна підтримувати купівлю генераційних кредитів;
– система повинна підтримувати PREMIUM-підписку;
– система повинна створювати платіжні сеанси через Stripe;
– система повинна обробляти результат оплати;
– система повинна зберігати платіжні події;
– система повинна відокремлювати платіжний модуль від генерації.

4. Вимоги до інформаційного забезпечення

4.1 Джерела і зміст вхідної інформації

Джерелами вхідної інформації є користувач, клієнтська частина, API-сервер, worker-модуль, сервіси ШІ та платіжний сервіс. Основними даними є параметри гри, дані користувача, статус доступу, проміжні результати генерації та платіжні події.

4.2 Нормативно-довідкова інформація

До нормативно-довідкової інформації належать підтримувані жанри, тональності, режими генерації, етапи pipeline, статуси генерації, типи доступу, статуси платежів, типи ресурсів і режими видимості гри.

4.3 Вимоги до способів організації, збереження та ведення інформації

– дані повинні зберігатися в PostgreSQL;
– доступ до даних повинен здійснюватися через Prisma ORM;
– користувачі, сесії, ігри, генерації, етапи, матеріали гри та платіжні події повинні зберігатися як пов’язані сутності;
– вхідні параметри та результати генерації повинні зберігатися у структурованому вигляді;

– шляхи до згенерованих файлів повинні зберігатися як частина результату.

5. Вимоги до технічного забезпечення

– клієнтська частина повинна працювати на ПК, ноутбуках, планшетах і смартфонах із сучасним браузером;

– спеціалізоване обладнання з боку користувача не потрібне;

– серверна частина повинна забезпечувати роботу API, бази даних, черги та worker-модуля;

– для візуальної генерації можуть використовуватися додаткові обчислювальні ресурси;

– інфраструктура повинна підтримувати PostgreSQL і Redis;

– обмін даними повинен виконуватися через HTTP/HTTPS.

6. Вимоги до програмного забезпечення

6.1 Архітектура програмної системи

Система повинна бути побудована за клієнт-серверною архітектурою з окремим worker-модулем для фонової генерації. Архітектура повинна забезпечувати розділення інтерфейсу, бізнес-логіки, збереження даних, платіжного доступу та генераційного pipeline.

6.2 Системне програмне забезпечення

Для роботи системи використовуються Node.js, PostgreSQL, Redis, Docker та середовище виконання ШІ-компонентів.

6.3 Мережне програмне забезпечення

Система повинна підтримувати HTTP/HTTPS-запити між клієнтом і сервером. У промисловому середовищі передавання персональних, сесійних і платіжних даних повинно виконуватися через HTTPS.

6.4 Програмне забезпечення ведення інформаційної бази

Основною системою керування базами даних є PostgreSQL. Схема бази даних повинна підтримувати користувачів, сесії, ігри, генерації, етапи генерації, матеріали гри, платіжні сеанси, підписки та платіжні події.

6.5 Мова і технології розробки ПЗ

– frontend: Next.js, React, TypeScript, Tailwind CSS, Framer Motion;

– backend: NestJS, TypeScript, Prisma;

– worker-модуль: TypeScript, Redis/BullMQ;

– база даних: PostgreSQL;

– платіжна інтеграція: Stripe;

– інфраструктура: Docker, pnpm workspace, Turborepo;

– ШІ-рівень: Ollama та візуальний генераційний/рендеринг-конвеєр.

7. Вимоги до зовнішніх інтерфейсів

7.1 Інтерфейс користувача

– вебінтерфейс повинен бути доступним через сучасний браузер;

– система повинна мати сторінки входу, реєстрації, створення гри, статусу генерації, результату та галереї;

– інтерфейс повинен бути адаптивним;

– повідомлення про помилки, обмеження доступу та оплату повинні бути зрозумілими для користувача.

7.2 Апаратний інтерфейс

Система не потребує спеціалізованого апаратного інтерфейсу. Для роботи достатньо пристрою з браузером і доступом до інтернету.

7.3 Програмний інтерфейс

Система повинна надавати REST API для авторизації, перевірки сесії, оплати, запуску генерації, отримання статусу, перегляду результатів, повторної генерації, доступу до матеріалів та перегляду публічних прикладів.

7.4 Комунікаційний протокол

Основним протоколом взаємодії є HTTP/HTTPS. Для захищених даних у промисловому середовищі повинен використовуватися HTTPS.

8. Властивості програмного забезпечення

8.1 Доступність

Система повинна бути доступною через браузер без встановлення окремого застосунку. Завершені генерації повинні залишатися доступними власнику після повторного входу.

8.2 Супроводжуваність

Кодова база повинна бути організована як монорепозиторій з окремими frontend, backend і worker-компонентами. Генераційна логіка повинна бути поділена на етапи й сервіси.

8.3 Переносимість

Клієнтська частина повинна працювати в актуальних версіях Chrome, Firefox, Edge і Safari. Серверна частина повинна запускатися в середовищі Node.js, а PostgreSQL і Redis – через Docker або сумісну серверну інфраструктуру.

8.4 Продуктивність

Типові API-запити повинні виконуватися без помітної затримки. Тривалі операції генерації повинні виконуватися у фоновому режимі через чергу завдань.

8.5 Надійність

Система повинна зберігати статус генерації та її етапів. У разі помилки має зберігатися повідомлення про причину збою. Збережений результат не повинен втрачатися після оновлення сторінки або повторного входу.

8.6 Безпека

– паролі повинні зберігатися у хешованому вигляді;

– сесійні токени повинні зберігатися у вигляді хешів;

– захищені API-методи повинні перевіряти активну сесію;

– доступ до генерацій і файлів повинен перевіряти належність користувачеві;

– webhook-події платіжного сервісу повинні перевірятися за підписом;

– доступ до даних повинен здійснюватися через ORM для зниження ризику SQL-ін’єкцій.

8.7 Якість згенерованого результату

Правила, компоненти, картки та поле повинні описувати одну цілісну гру. Назви ресурсів, зон, дій і карток повинні бути узгодженими. Візуальний стиль повинен відповідати темі й тональності гри. Повний баланс гри може потребувати додаткового плейтесту.

9. Інші вимоги

– система повинна підтримувати подальше розширення жанрів, тональностей і шаблонів;

– система повинна підтримувати розвиток галереї публічних прикладів;

– система повинна дозволяти додавання нових форматів експорту;

– система повинна мати можливість інтеграції з іншими моделями ШІ;

– платіжна модель повинна розвиватися без зміни основного генераційного pipeline.

Висновки до розділу 2

У другому розділі було представлене обґрунтування технічної основи для розроблення інтелектуального вебсервісу, яке включає клієнтську частину, серверну логіку, систему бази даних, фонову обробку задач, генерацію ШІ та зберігання отриманих результатів. Обраний стек відповідає структурі системи.

Окремо визначено гібридний підхід до генерації ігрового контенту. Мовна модель формує тематичне й текстове наповнення, а стабільність результату підтримується механічною основою гри, структурованими даними та процедурним формуванням матеріалів. Дифузійна модель обрана для візуальних матеріалів гри.

Також сформовано специфікацію вимог до програмного забезпечення, яка визначає призначення системи, межі проєкту, основні функції, вимоги до даних, інтерфейсів, безпеки, продуктивності та якості результату. Таким чином, другий розділ створює основу для подальшого проєктування і програмної реалізації вебсервісу.

3 ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОГО ВЕБСЕРВІСУ ДЛЯ ГЕНЕРАЦІЇ ПЕРСОНАЛІЗОВАНИХ НАСТІЛЬНИХ ІГОР

3.1 Моделювання функціональних сценаріїв роботи вебсервісу

Після визначення вимог до програмного забезпечення можна перейти до проєктування вебсервісу. Спочатку доцільно описати функціональні сценарії. Для демонстрації взаємодії користувача з вебсервісом корисні функціональні сценарії, особливо ті, що показують можливості системи. Перехід користувача від ознайомлення до підготовки параметрів гри, запуску генерації, перегляду результату, роботи з бібліотекою ігор є важливим в рамках розроблення.

Основна увага приділяється зовнішній поведінці системи. Діаграма варіантів використання дає чітку основу для моделювання функціональних сценаріїв. У вебсервісі доцільно виділити три ролі, а саме гість (відвідувач), зареєстрований користувач та адміністратор, де кожна роль має різний рівень доступу.

Зручним сценарієм є те, що гість може заповнити параметри настільної гри до того моменту, коли буде ініційовано запуск генерації. Але потрібна авторизація для фактичного запуску через те, що результат має бути збережений у профілі користувача. Діаграма варіантів використання для гостя наведена на рисунку 3.1.

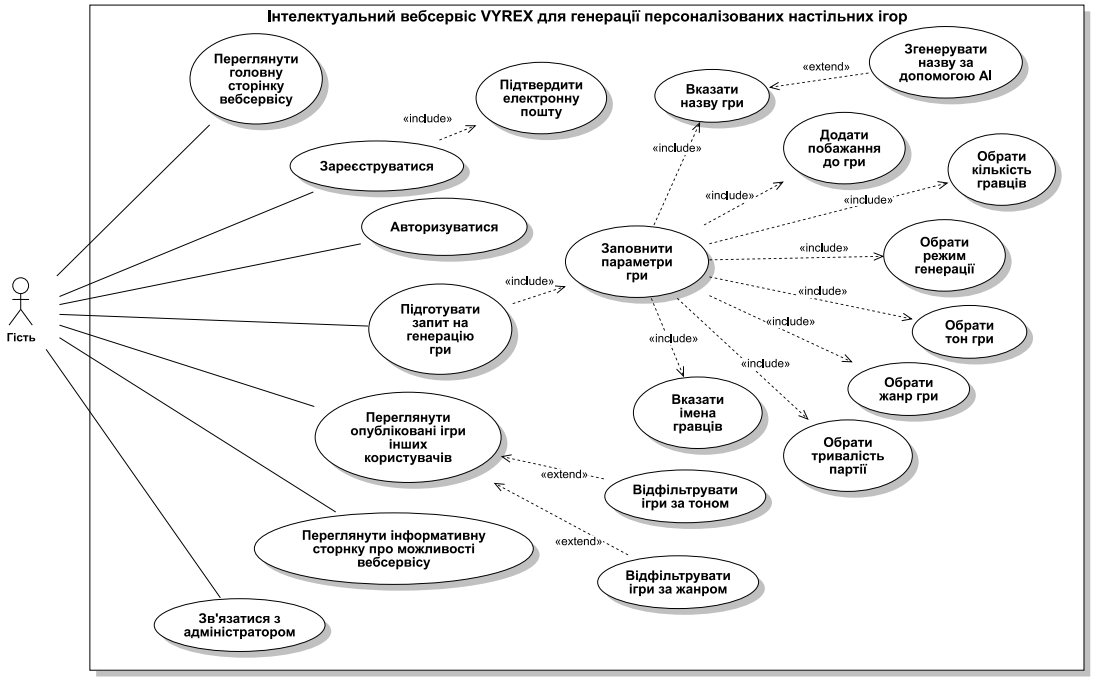

Рисунок 3.1 – Діаграма варіантів використання для відвідувача

Зареєстрований користувач вже має повний доступ до основного сценарію, де створюється настільна гра. Користувач може підготувати параметри гри, запускати генерацію, переглядати хід роботи та проміжні результати під час процесу генерації, отримувати результат, відкривати згенеровані файли у профілі, повторювати генерацію та працювати з бібліотекою ігор, які він створив. Особливо корисним є сценарій керування профілем, тарифами та бібліотекою. Діаграма варіантів використання для зареєстрованого користувача наведена на рисунку 3.2.

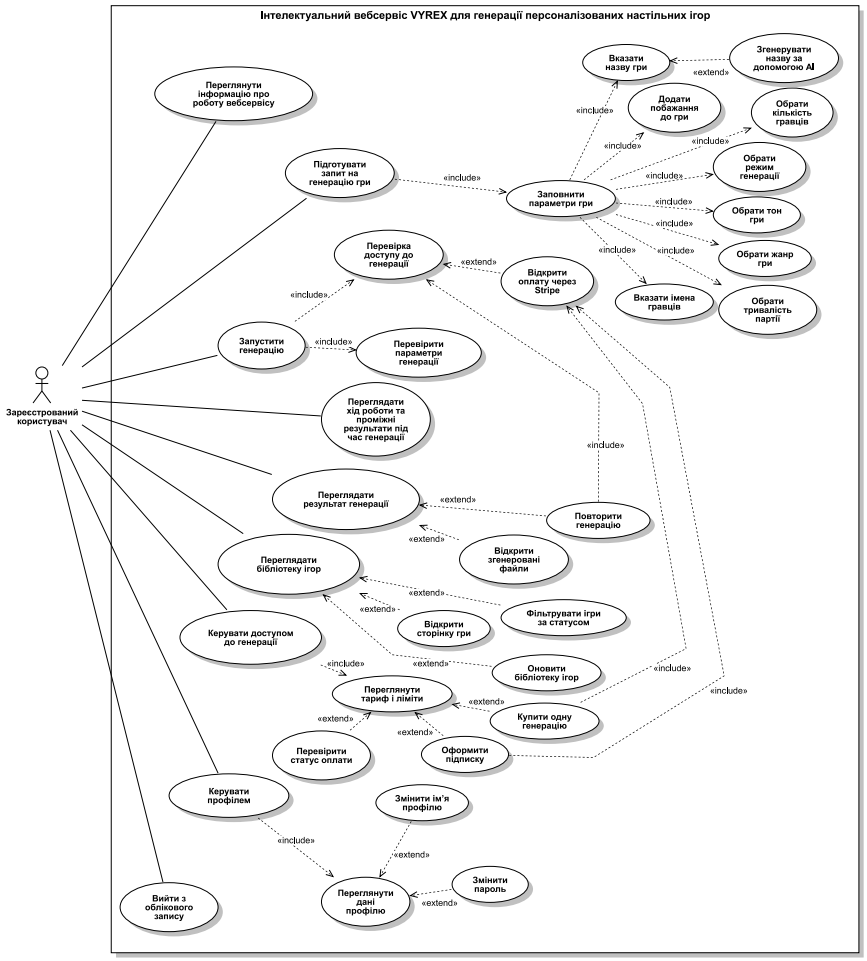

Рисунок 3.2 – Діаграма варіантів використання для зареєстрованого користувача

Роль адміністратора дуже важлива і охоплює контроль роботи вебсервісу. Щоб адміністратор міг повноцінно виконувати свою роль, для нього передбачено перегляд ігор користувачів без деталей. Також є керування активними генераціями, зупинка процесу генерації, керування користувачами та адміністрування ігор. Це дає підтримку стабільної роботи системи та контроль над створеними матеріалами. Діаграма варіантів використання для адміністратора наведена на рисунку 3.3.

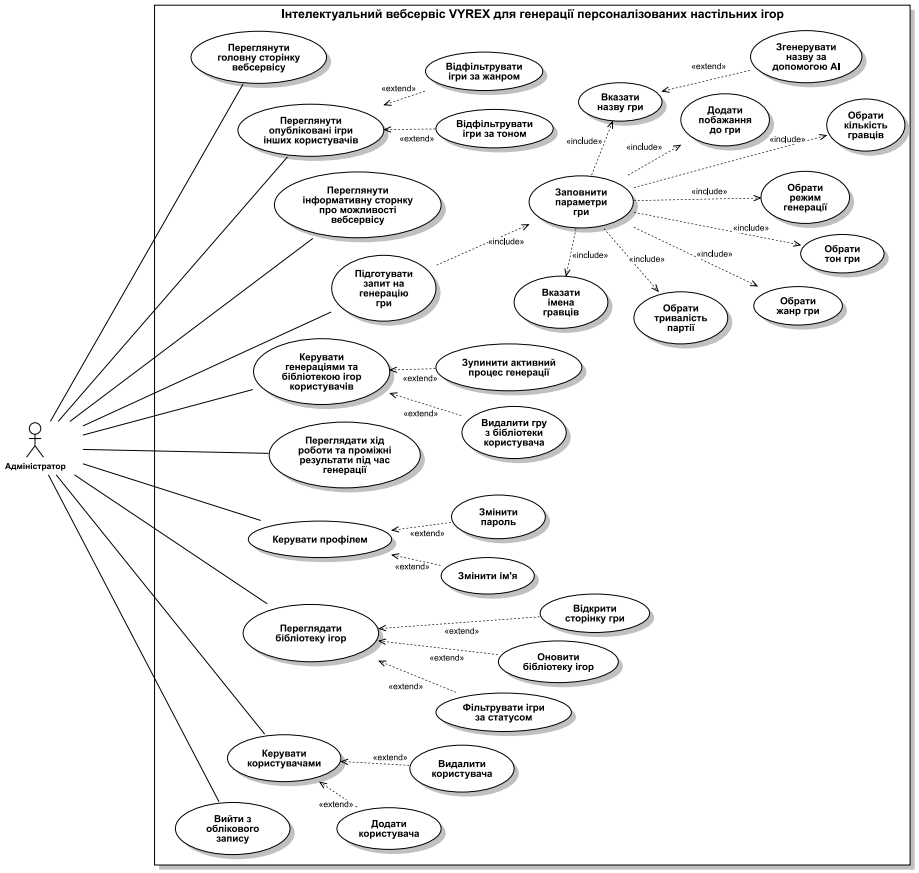

Рисунок 3.3 – Діаграма варіантів використання для адміністратора

Групування у діаграмах сценаріїв використання виконано за принципом ролей і призначення сценаріїв. Частина сценаріїв є загальнодоступною для всіх відвідувачів вебсервісу. Інша частина виконується тільки після авторизації як передумова. Окремо виділяються адміністративні сценарії, які потрібні для керування користувачами, генераціями та іграми. Для системи це є дуже важливим, оскільки створення гри тісно пов’язане зі збереженням результатів у профілі конкретного користувача, перевіркою та контролем лімітів генерації, тобто бізнес-логікою.

Наведені діаграми відображають основні сценарії роботи з вебсервісом для різних ролей. Відображено не лише основний сценарій генерації, а й допоміжні дії, без яких сервіс не буде повноцінним і очікуваним. Зв’язки «include» використовуються для певних дій, де вони є обов’язковою частиною більшого сценарію, наприклад перевірка параметрів або доступу. Зв’язки «extend» показують додаткові можливості, які виконуються не завжди, а лише за певної умови. Для кращого пояснення діаграм сценарії доцільно згрупувати за призначенням. Це дає

змогу окремо показати загальнодоступні дії, захищені сценарії, адміністративні можливості та дії, які виконуються тільки за певних умов. Узагальнення функціональних сценаріїв вебсервісу наведено в таблиці 3.1, що дає чіткі межі та повне розуміння кожної групи сценаріїв.

Таблиця 3.1 – Узагальнення функціональних сценаріїв вебсервісу

Група сценаріїв	**Основні дії**	**Роль**	**Результат**
Ознайомлення	Перегляд сторінок сервісу та опублікованих ігор	Гість, зареєстрований користувач	Розуміння можливостей сервісу
Обліковий запис	Реєстрація, авторизація, підтвердження пошти, вихід	Гість, зареєстрований користувач	Доступ до персональних функцій
Підготовка запиту	Заповнення параметрів гри та генерація назви	Гість, зареєстрований користувач	Чернетка запиту на генерацію
Генерація гри	Перевірка параметрів і доступу, запуск та перегляд ходу	Зареєстрований користувач	Створення гри
Результат і бібліотека	Перегляд гри, файлів, повторна генерація, робота з бібліотекою	Зареєстрований користувач	Доступ до створених ігор
Доступ і оплата	Перегляд тарифів, купівля генерації, підписка	Зареєстрований користувач	Право на генерацію
Адміністрування	Керування користувачами, іграми та активними генераціями	Адміністратор	Контроль роботи сервісу

Адміністратор є авторизованим користувачем із розширеними правами, тому може виконувати користувацькі сценарії. Передбачено особливі дії управління, які недоступні іншим ролям. Для адміністратора немає дій оплати. Коли адміністратор запускає створення гри, система враховує його роль і не перевіряє доступ до генерації.

Ключовим сценарієм є саме створення персоналізованої настільної гри, бо він є основною темою вебсервісу. Поєднання користувацького інтерфейсу,

перевірки лімітів, pipeline генерації, БД і збереження робить такий сценарій найскладнішим у системі. Навколо цього сценарію побудовані й інші дії. Виходячи з цього, доцільно детальніше розглянути сценарій створення гри з використанням повної форми Use Case (табл. 3.2).

Таблиця 3.2 – Опис основного сценарію створення настільної гри

Use case section	**Comment**
Use Case Name	Створити персоналізовану настільну гру
Scope	Інтелектуальний вебсервіс для генерації персоналізованих настільних ігор
Level	User-goal – отримання готової настільної гри
Primary Actor	Зареєстрований користувач
Stakeholders and interests	1) користувач – отримати гру за власними параметрами; 2) користувач – мати доступ до результату після генерації; 3) система – перевірити право на генерацію; 4) система – зберегти гру та пов’язані матеріали.
Preconditions	Вебсервіс відкрито. Користувач авторизований у вебсервісі та має доступ до генерації.
Success guarantee	1) система створює настільну гру за введеними параметрами; 2) результат зберігається у профілі користувача; 3) користувач може переглянути готову гру в бібліотеці.
Main Success Scenario	1) користувач переходить до створення нової гри; 2) система відображає форму параметрів; 3) користувач вводить назву, жанр, тон, кількість гравців і тривалість; 4) користувач за потреби додає профілі гравців і побажання; 5) система перевіряє обов’язкові поля; 6) користувач підтверджує запуск генерації; 7) система перевіряє авторизацію та доступ; 8) система запускає створення гри; 9) користувач переглядає статус виконання; 10) система формує концепцію, правила, компоненти та візуальні матеріали; 11) система зберігає готову гру; 12) користувач переглядає результат; 13) користувач відкриває гру в бібліотеці.

Кінець таблиці 3.2

Extensions	– користувач не авторизований: система пропонує увійти або створити обліковий запис; – недостатньо даних: система показує, які поля потрібно доповнити; – відсутній доступ до генерації: система пропонує оплату або підписку; – помилка генерації: система зберігає статус помилки і дозволяє повторний запуск з попередніми параметрами.
Special Requirements	Інтерфейс має бути зрозумілим для користувача. Система повинна показувати статус генерації, контролювати доступ і зберігати результат.
Technology and Data Variations List	1) гра створюється з нуля за параметрами, що вибрав користувач; 2) доступ може надаватися через безкоштовну генерацію, кредити або підписку; 3) результат містить опис гри, правила, компоненти та візуальні матеріали; 4) створені ігри зберігаються в бібліотеці користувача.
Frequency of Occurrence	Орієнтовно 60 % від основних сценаріїв авторизованого користувача, оскільки створення гри є центральною функцією вебсервісу.
Miscellaneous	Сценарій є основним для вебсервісу, оскільки поєднує підготовку параметрів, генерацію, контроль доступу та збереження результату.

Таблиця 3.2 уточнює основний сценарій створення гри, а тому можна зробити висновок, що створення гри не є просто однією дією. Можливі відхилення теж були враховані окремо в сценарії. Наприклад, коли користувач не авторизований і система має перенаправити на сторінку входу або реєстрації. Якщо доступ до генерації відсутній, система пропонує користувачу оплату однієї генерації або підписку. При помилці під час генерації система повинна обов’язково зберегти статус цієї помилки і дозволити користувачу повторно запустити генерацію з попередніми параметрами.

Після визначення того, як користувач запускає створення гри, потрібно розглянути, як саме система обробляє цей запит усередині. Тому наступний етап присвячено алгоритму генерації персоналізованої настільної гри.

3.2 Алгоритм генерації персоналізованої настільної гри

Змодельовані функціональні сценарії дають змогу перейти до алгоритму, який виконується після того, як користувач запускає генерацію настільної гри. Задача цього алгоритму полягає у перетворенні параметрів користувача у готову персоналізовану гру. Така гра включає в себе наступні ключові етапи: правила, компоненти, візуальні матеріали та експортний пакет, який можна роздрукувати.

У системі генерація не є однією синхронною дією. Через виконання декількох ресурсомістких етапів при створенні гри, асинхронність є важливою. Цими етапами є: підготовка концепції гри, формування механіки, генерація правил, створення компонентів, рендеринг карток й ігрового поля, а також збереження результату. Через це доцільно алгоритм побудувати саме як асинхронний pipeline. Наприклад, запис генерації створює backend, а окремий worker-модуль виконує основну обробку, не перевантажуючи основний сервер.

Важливим є обов’язкова перевірка вхідних параметрів перед самим запуском генерації. Поля, які мають бути заповнені: назва гри, жанр, тон, кількість гравців, тривалість гри. Додатково користувач може вказати профілі гравців та власні побажання. Наприклад, у випадку, коли кількість профілів не відповідає кількості гравців або ключових параметрів немає, вхідні дані додатково нормалізуються. Тобто pipeline отримує строго структуровані дані. Це допомагає не витрачати ресурси на неповний або некоректний запит.

Діаграма діяльності (рис. 3.4) показує повний шлях генерації настільної гри, починаючи від етапу, коли користувач запускає створення гри. Першим кроком є отримання системою параметрів, перевірка коректності цих параметрів. Якщо дані неповні або помилкові, то система не переходить до наступних етапів, поки не буде введено очікувані параметри. Це супроводжується повідомленням про помилку валідації, після чого користувач може виправити й повторити спробу. Генераційний pipeline не повинен запускатися з неповним або суперечливим набором даних. Діаграма зосереджена на типовому сценарії запуску генерації після підготовки запиту, тому службовий доступ адміністратора у ній не деталізується.

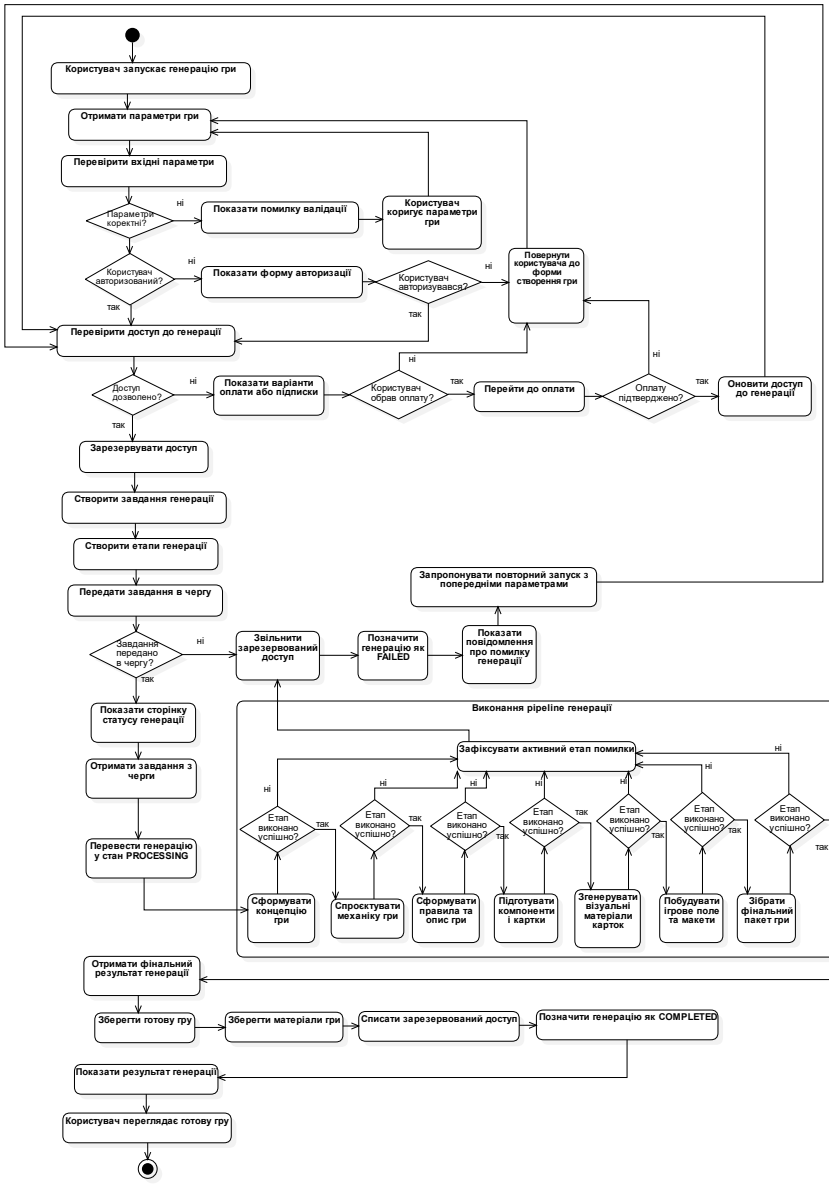

Рисунок 3.4 – Діаграма діяльності процесу генерації настільної гри

Не менш важливим можна виділити саме перевірку авторизації. У випадку, коли користувач не авторизований, система відкриває форму входу. Успішна авторизація повертає користувача до форми створення гри, а після цього процес знову проходить стандартні перевірки. Навіть після входу користувача система має повторно отримати актуальні параметри, а потім перевірити їх для завершення підготовки запуску генерації.

Ключовим кроком перед запуском генерації є перевірка доступу до генерації, включаючи підписку та безкоштовну генерацію. Система резервує доступ та переходить до завдання самої генерації, якщо доступ дозволено і всі перевірки пройдені. Якщо користувач вичерпав безкоштовну генерацію, система пропонує

варіанти оплати або підписки. При підтвердженні оплати система оновлює права доступу та знову здійснює перевірку. Такий підхід забезпечує запуск генерації тільки після фактичного підтвердження права на використання ресурсів вебсервісу.

Наступний етап після перевірки доступу до генерації та резервування – це створення завдання генерації та етапів її виконання. Передавання завдання у чергу запускає нову перевірку. Як показано на діаграмі, ця перевірка потрібна для визначення, чи успішно передано це завдання у чергу. Якщо нічого не передалось, то зарезервований доступ звільняється, а генерація отримує статус FAILED. Користувач теж отримує повідомлення про помилку. Враховуючи це, система запобігає ситуації, коли доступ втрачено, а насправді процес генерації навіть не почався. Такі випадки одразу перехоплюються, а користувач не витрачає марно свої кредити для запуску нових ігор.

Після того як завдання потрапило в чергу, користувач може переглядати сторінку статусу генерації, де демонструються актуальні етапи та хід роботи системи. У той час worker-модуль отримує завдання з черги та переводить генерацію у стан PROCESSING. Це є початком основного алгоритму, а саме виконання pipeline генерації. Він є найскладнішим і найтривалішим етапом, який поєднує у собі послідовні кроки.

Перші два важливі кроки, які робить система – формування концепції гри, під час якого визначається її механічна основа, після чого третім кроком стає формування правил гри та її опис. Четвертий крок у pipeline – підготовка компонентів і карток. П’ятим і дуже чутливим кроком є генерація візуальних матеріалів зі застосуванням дифузійної моделі та керуванням ходом виконання. Шостим кроком виступає побудова ігрового поля з макетами, а останнім кроком є збірка фінального пакету гри, який завершує pipeline. На кожному етапі передбачено перевірку на успішність виконання. У випадку успішного завершення, система переходить далі. Але якщо виникла якась проблема на будь-якому кроці, система фіксує активний етап помилки, після чого звільняється зарезервований доступ, а генерація отримує статус FAILED.

Таке рішення дає прозорість у роботі системи, а помилка не залишається прихованою всередині pipeline. Система обов'язково фіксує, на якому саме етапі виникла помилка, а користувач отримує повідомлення, що завершення є невдалим. Повторний запуск з попередніми параметрами дає змогу ще раз виконати генерацію, а кредит за невдалу спробу не списується з користувача.

У разі успішного виконання всіх етапів, система отримує фінальний результат. Готова гра зберігається в базі даних. Зарезервований доступ списується і генерація отримує статус COMPLETED. На практиці це означає, що доступ витрачається виключно після отримання сформованого результату та успішного збереження. Для користувача такий підхід дає надійність, а для системи керованість. Особливо важливо це забезпечити через той факт, що у вебсервісі є платні генерації. Основні етапи pipeline та їх призначення наведено в таблиці 3.3, що допоможе узагальнити та більш просто зрозуміти основні етапи в генерації, які застосовує система.

Таблиця 3.3 – Основні етапи pipeline генерації настільної гри

Етап	**Призначення**	**Результат етапу**
CONCEPT	Формування концепції гри	Тематична основа майбутньої гри
MECHANICS	Фіксація готовності механічного каркаса	Базова логіка дій, ресурсів і взаємодії
RULES	Формування правил та опису гри	Правила гри і текстовий опис результату
COMPONENTS	Підготовка компонентів і карток	Набір ігрових компонентів
IMAGES	Генерація візуальних матеріалів карток	Зображення та файли карткових матеріалів
VALIDATION	Формування ігрового поля та пов'язаних макетів	Поле, макети та узгоджений візуальний набір
ASSEMBLY	Збірка фінального пакета гри	Готовий результат генерації та експортний пакет

Під час етапу CONCEPT система обирає механічну основу, будує механічний каркас гри та формує тематичне наповнення. Наступний статус MECHANICS

фіксує готовність сформованої механічної структури у загальному прогресі генерації. Крок VALIDATION – контрольований етап після створення карткових матеріалів, який потрібно визначити не як звичайну перевірку правил. У системі він пов’язаний із побудовою ігрового поля та формуванням макетів. Це означає, що на цьому етапі система формує ігрове поле та пов’язані з ним макети на основі вже підготовлених матеріалів.

Алгоритм не зводиться до одного запиту до мовної моделі, інакше це був би неконтрольований набір розрізнених матеріалів. Генерація складається з кількох контрольованих етапів, де кожен етап має своє призначення і результат. Це дає системі можливість не лише розробити текстову концепцію, але й завершити її у вигляді повного набору для гри.

Отже, алгоритм генерації побудовано як керований асинхронний pipeline. Після визначення алгоритму доцільно перейти до моделювання інформаційних потоків, оскільки саме вони показують, як під час генерації взаємодіють користувач, клієнтська частина, backend, черга, worker та інші компоненти системи.

3.3 Моделювання інформаційних потоків під час генерації гри

Після визначення алгоритму ще одним важливим етапом проєктування вебсервісу є моделювання інформаційних потоків під час генерації гри. Важливо продемонструвати не лише послідовність, але й обмін інформацією між різними частинами системи. Одного модуля недостатньо, щоб створити ігровий контент для настільної гри. Взаємодія є необхідною між frontend, backend, базою даних, чергою задач, worker-процесом, сервісами ШІ, рендерингом та файловим сховищем.

Діаграма послідовності добре демонструє взаємодію між цими компонентами системи. Відображаються лінії життя, порядок повідомлень, передані дані та результати, які повертаються після кожного етапу. Для читабельності та уникнення перевантаження складний процес генерації поділено на наступні частини: запуск генерації та постановка задачі у чергу, виконання pipeline, отримання готового результату та його відображення. На рис. 3.5 наведено першу діаграму послідовності запуску генерації та постановки задачі в чергу.

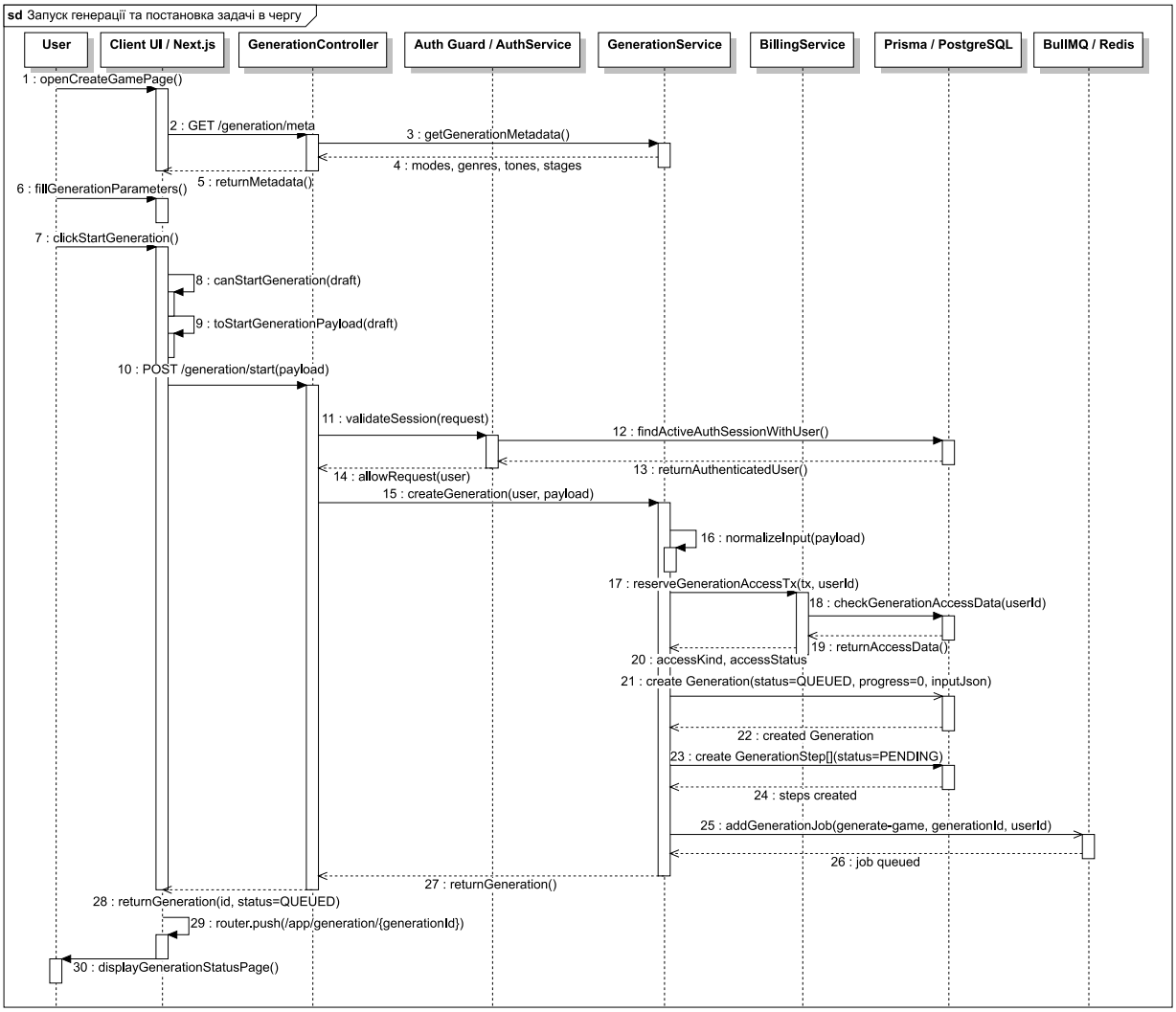

Рисунок 3.5 – Діаграма послідовності запуску генерації, постановка задачі в чергу

Перша діаграма є відображенням початкового обміну даними між інтерфейсом користувача, backend, базою даних та чергою задач. Спочатку frontend отримує метадані генерації, до яких входять режими, жанри, тони та перелік етапів генерації. Вони потрібні для формування конкретної форми для створення гри.

Наступним кроком після заповнення форми виступає формування тіла запиту на генерацію, де основними даними є параметри гри, профілі гравців та додаткові побажання гравців. Далі вже payload передається до GenerationController, де додатково проходить перевірка сесії користувача вебсервісу.

Коли перевірка в БД проходить успішно, створюється запис Generation. У ньому зберігаються дані, такі як статус QUEUED, початковий прогрес та вхідні параметри гри у форматі JSON. Для подальшого відстеження стану кожного етапу створюються GenerationStep.

Після цих процедур створення записів GenerationService передає задачу в BullMQ/Redis. Особливістю цього процесу є те, що у чергу надсилається не повний набір параметрів, а лише короткі job data: generationId. PostgreSQL зберігає повні дані. Це дає перевагу у компактності повідомлення в черзі, а база даних є основним джерелом стану генерації.

У разі успішної постановки задачі створений об’єкт Generation відправляється у frontend від GenerationService (backend). Клієнтська частина отримує generationId та переходить на сторінку статусу, що дуже зручно для користувача спостерігати за виконанням та розуміти, на якому етапі генерація.

Друга діаграма (рис. 3.6) показує інформаційний обмін під час тривалої генерації. Є низка ліній життя на діаграмі, але основна лінія на цьому етапі – GenerationWorker. Він отримує задачу з черги та виконує основну та найтривалішу обробку. Першим кроком worker-модуля є отримання generationId у Redis через BullMQ, після чого він звертається до бази даних і читає запис генерації, що був записаний у JSON-форматі як inputJson. Як тільки ці процедури зроблені, статус генерації змінюється на PROCESSING, а вхідні дані передаються далі.

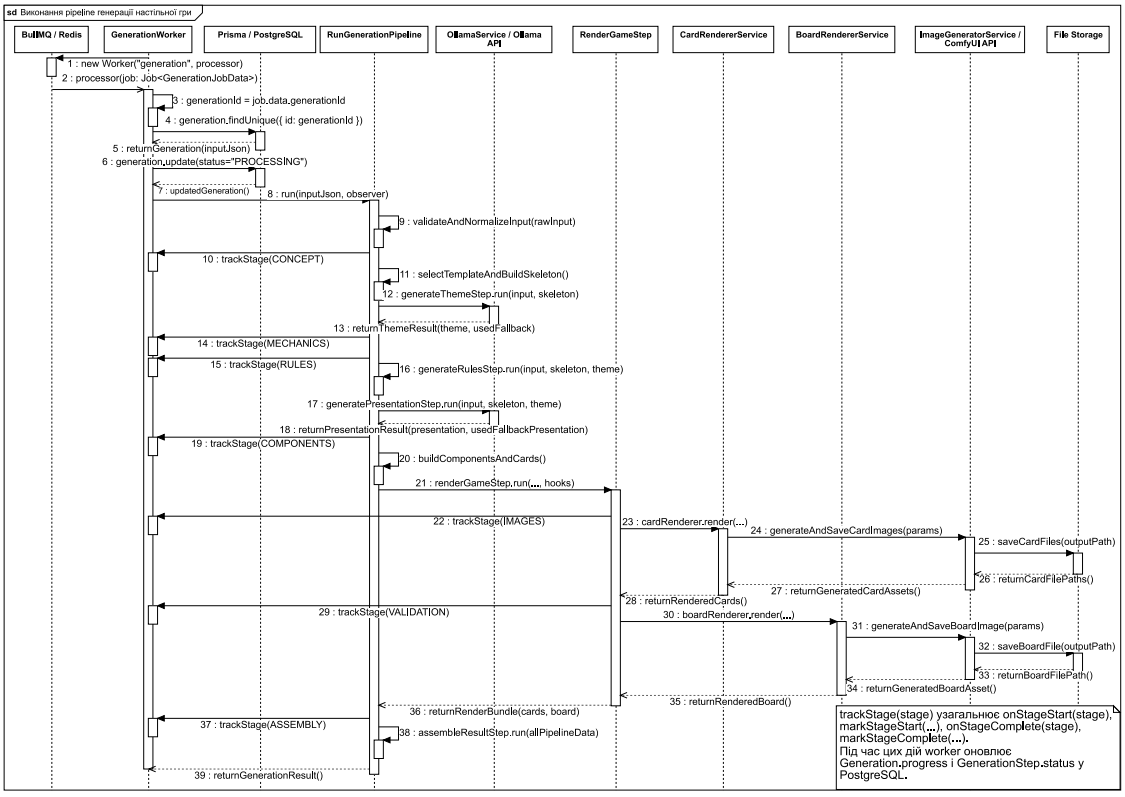

Рисунок 3.6 – Діаграма послідовності виконання pipeline генерації настільної гри

У цій діаграмі показано основний інформаційний потік даних без надмірної деталізації, зокрема внутрішніх сервісів. Фактичне виконання генерації містить набагато більше допоміжних операцій.

Усередині pipeline нормалізуються вхідні дані та крок за кроком перетворюються в набір проміжних структур. Формування GameSkeleton задає основу гри, а саме ресурси, дії, зони, колоди і компоненти. Для узгодження назви, світу гри, стилю, тону і тематичних назв елементів використовується сформований ThemePack на основі параметрів користувача і каркаса.

Сервіси правил і презентаційного опису потрібні для формування текстової частини генерації. Результатами стають RulesPack і PresentationPack, які містять правила, підготовку до гри, пояснення дій гравців, короткий опис гри та презентаційний текст. При цьому правила формуються програмно на основі механічного каркаса і теми гри, а звернення до Ollama API використовується для презентаційного тексту. До мовної моделі передається заздалегідь підготовлений контекст. У відповідь система отримує структурований результат. Якщо відповідь є некоректною, запускається повторна обробка, а коли придатного результату немає, використовуються резервні дані.

Як тільки текстовий блок вже готовий, наступним етапом формується AssetsPack і CardContentPack. Для візуальної генерації застосовуються стильові обмеження, промпти та негативні промпти. Це все формується безпосередньо у AssetsPack. CardContentPack містить колоди, картки, тексти правил, типи карток, вартість дій, винагороди та службові дані для рендерингу.

RenderGameStep розділяє потік на два напрями, у які входять картки та ігрове поле. Для карток передаються CardContentPack, ThemePack і GameSkeleton. На їх основі формується макет, візуальні ознаки, промпти для зображень і фінальні SVG-картки. Для поля передаються GameSkeleton, ThemePack, картки та профілі гравців. Ці дані є основою для формування макета поля, поверхні, SVG-макетів та матеріалів гравців.

ImageGeneratorService і ComfyUI API виконують роботу, яка пов’язана з генерацією зображень. Принцип роботи такий, що до ComfyUI передаються

промпти, негативні промпти, workflow, seed, розмір зображення та параметри генерації. А вже після завершення повертаються створені зображення від ComfyUI, які, у свою чергу, зберігаються у файловому сховищі. Особливо важливо, що у pipeline повертаються не самі файли, а шляхи до файлів.

Після того як рендеринг завершено, далі формується RenderBundle, що містить результат створення карток, ігрового поля, SVG-матеріалів, планшетів гравців і стильового набору. Етап ASSEMBLY – фінальний крок у pipeline, де проміжні структури збираються в GenerationResult, який по суті є результатом всієї генерації та збірки. До основних елементів входять: параметри генерації, профілі гравців, каркас, тема, правила, опис, компоненти, картки, результати рендерингу, шляхи до файлів і експортний пакет. Pipeline можна вважати закритим, що дає основу для збереження. На рисунку 3.7 наведено діаграму послідовності збереження результату генерації та отримання готової гри.

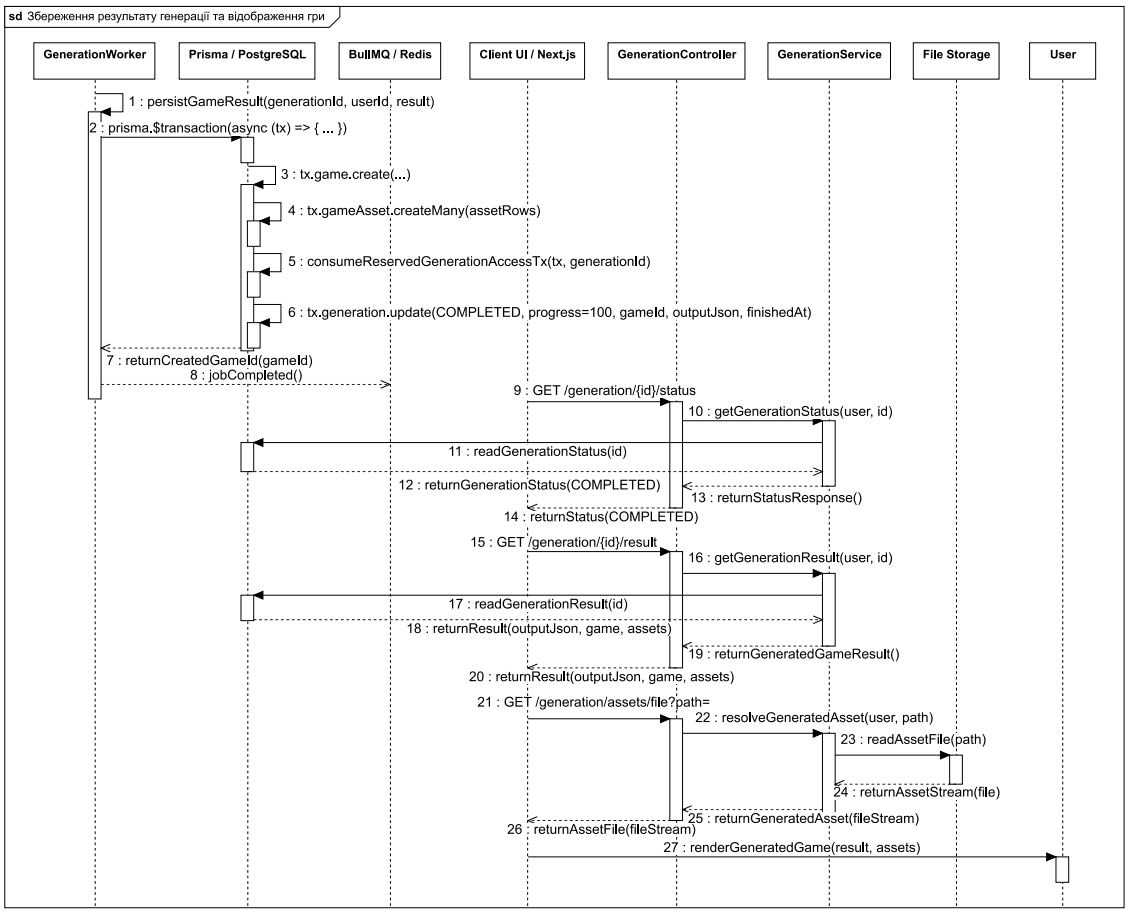

Рисунок 3.7 – Діаграма послідовності збереження результату генерації та відображення гри

На діаграмі видно, що після завершення всіх кроків pipeline worker-модуль передає GenerationResult на збереження. Створюється запис Game однією транзакцією. Далі додаються пов’язані записи GameAsset, списується зарезервований доступ, а генерація переходить у статус COMPLETED.

Клієнтська частина періодично відправляє запит для отримання статусу генерації. У разі повернення серверною частиною COMPLETED, повний результат відправляється у клієнтську частину, а потрібні матеріали завантажуються з файлового сховища. Результат цих операцій – це фактичне відображення готової гри користувачу. Дані ігор залишаються доступними для повторного відкриття з бібліотеки. Для подальшого проєктування потрібно визначити, як ці дані організовуються та пов’язуються в БД.

3.4 Проєктування інформаційної моделі бази даних

Моделювання інформаційних потоків визначило послідовність та обмін між різними частинами системи, тому доцільно буде показати, як ці дані організуються в БД. Спроєктована логічна схема БД (рис. 3.8) демонструє основні сутності вебсервісу та зв’язки між ними. В інформаційній моделі використовуються облікові записи користувачів, сесії, платіжні доступи, сам процес генерації, етапи pipeline, готові ігри та матеріали цих створених ігор.

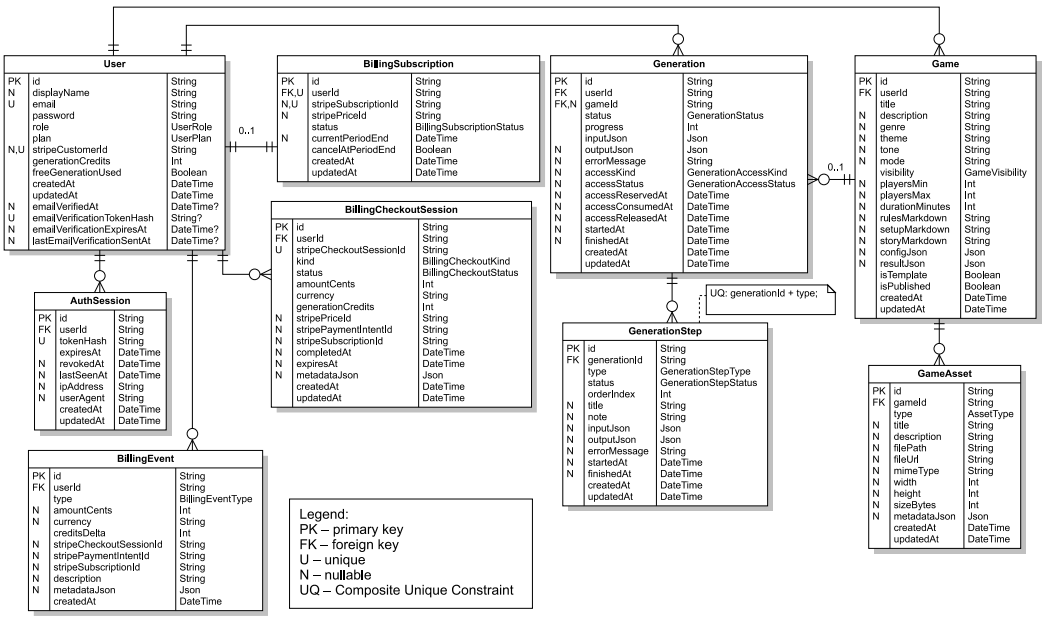

Рисунок 3.8 – Логічна схема бази даних вебсервісу

Таблиці бази даних можна поділити на кілька груп. Користувач та авторизація складають першу таку групу, до якої належать таблиці User та AuthSession. У таблиці User зберігаються основні дані про користувача, а саме його роль, план доступу, кількість генераційних кредитів і ознаку використання безкоштовної генерації та перевірки пошти. У свою чергу AuthSession відповідає за активні сесії, хеш токена, час завершення сесії, відкликання сесії та службову інформацію про пристрій користувача. Основною перевагою такого поділу є те, що не змішуються профіль користувача з даними конкретних входів у систему.

Другою групою можна назвати таблиці, які відповідають за платіжний доступ. BillingSubscription, BillingCheckoutSession та BillingEvent створюють таку групу таблиць. Збереження інформації про підписку користувача забезпечується таблицею BillingSubscription, тоді як BillingCheckoutSession фіксує створені платіжні сесії, їх статус, суму, валюту і кількість придбаних генерацій. BillingEvent – журнал платіжних подій і змін доступу. Підсумовуючи такий поділ, платіжна логіка винесена в окремий блок, що дозволяє не перевантажувати таблиці, які відповідають за генерацію або збереження готових ігор.

Третя група описує сам процес генерації, основою якого є таблиця Generation. Вона потрібна для збереження статусів генерації, прогресу, вхідних параметрів, результату, повідомлень про помилку та інформацію, що стосується доступу до генерації. Особливість цієї таблиці полягає у тому, що поля accessKind і accessStatus забезпечують контроль того, за рахунок чого користувач запускає генерацію. Це може бути або безкоштовний запуск, або кредити чи підписка. Для відстеження резервування, списання або звільнення резерву використовуються часові поля.

Таблиця GenerationStep показує деталізацію pipeline через те, що там зберігаються окремі етапи генерації, порядок, статус, вхідні та вихідні дані, помилки та час виконання. Унікальним обмеженням на діаграмі продемонстровано поєднання generationId разом з type. Це зроблено для того, щоб не дозволити створити два однакові етапи у рамках однієї генерації, що дає контроль для стану.

Четвертою групою таблиць є ті, що пов’язані з готовим результатом генерації. Наприклад, у випадку успішної генерації створюється запис Game, де

зберігається вже готова настільна гра, яка включає в себе весь ігровий контент. Поля visibility та isPublished дозволяють розділяти приватні ігри користувачів та публічні приклади для галереї, що є особливістю сервісу.

Матеріали гри винесені в таблицю GameAsset, яка має зв’язок з Game і зберігає файли, які належать до конкретної гри. Такий підхід потрібен тому, що одна гра може мати багато пов’язаних матеріалів. Якщо зберігати їх прямо в таблиці Game, то структура швидко стала б незручною і перевантаженою.

Користувач, генерація та готова гра – це ключові елементи інформаційної моделі, а основні зв’язки побудовані навколо них. Наприклад, один користувач може мати багато сесій, генерацій, ігор і платіжних записів. Одна генерація, ініційована користувачем, складається з декількох етапів GenerationStep, а після успішного завершення вона має зв’язок з однією згенерованою грою. Ще один приклад, коли гра може мати багато GameAsset, тобто багато матеріалів гри.

Інформаційна модель поєднує JSON-поля та реляційні зв’язки. Реляційна частина використовується для цілісності основних даних, а JSON-поля використовуються для генеративного контенту, який потребує змін. Це стосується параметрів, проміжних результатів, конфігурації гри, фінального результату та метаданих файлів. База даних стає строгою для службових даних і гнучкою для згенерованої гри.

У схемі присутні індекси, enum-типи та правила роботи зі зв’язками, які не показано безпосередньо у діаграмі, щоб уникнути перевантаження рисунку. Наприклад, індекси потрібні для швидшого пошуку за статусом, користувачем або датою створення. Enum-типи задають допустимі значення для ролей, статусів, типів доступу, платежів і матеріалів гри.

Якщо підсумовувати проєктування інформаційної моделі, стає зрозуміло, що вона підтримує повний цикл роботи вебсервісу. Спочатку користувач запускає генерацію, а система починає проходити етапи pipeline. Готова гра зберігається в БД, а всі матеріали прив’язуються до неї. Отже, такий розподіл робить базу даних основою для контролю стану системи, стає джерелом для збереження результатів та дає міцну основу для розвитку вебсервісу.

3.5 Проєктування архітектури та розгортання програмної системи

Попередні етапи проєктування показали основні частини вебсервісу, що дає чітку основу для їх поєднання в одну програмну систему. Основна увага приділяється загальній архітектурі компонентів, а також їх розміщенню у середовищі виконання.

Діаграма компонентів (рис. 3.9) розкриває поділ системи, який охоплює кілька основних рівнів. Перший з них – це PresentationLogic, який виконує роль взаємодії з користувачем. До складу презентаційної логіки належать сторінки головного екрана, авторизація, створення гри, статуси генерації, результати, галерея, профіль користувача та платежі. PresentationLogic напряму не працює з БД, але передає запити до backend та отримує у відповідь потрібні дані.

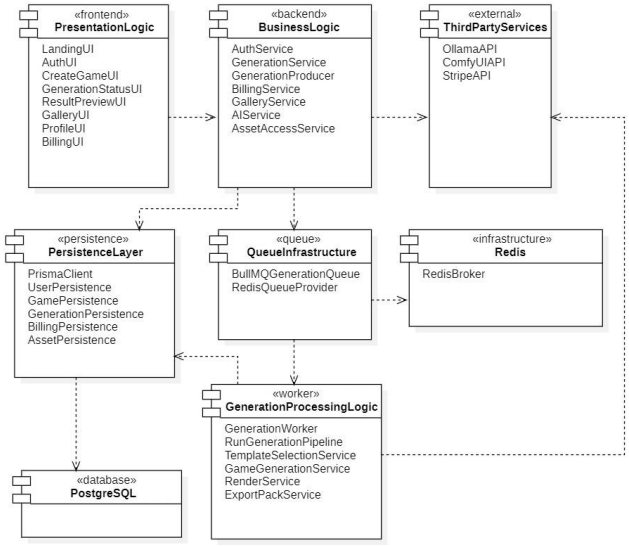

Рисунок 3.9 – Діаграма компонентів системи

Основа серверної частини – це рівень BusinessLogic. Його роль у тому, що він об’єднує сервіси авторизації, генерації, платежів, галереї, доступу до матеріалів та постановки задач у чергу. У цьому рівні відбувається перевірка користувача та контроль прав на генерацію, а після цього робиться запис генерації та формується задача для worker-модуля.

Окремої уваги потребує PersistenceLayer, який відповідає за роботу з даними через Prisma. Через цей рівень система звертається до PostgreSQL. Це контрольований шар доступу до сутностей, який не дає працювати з БД хаотично.

Компоненти QueueInfrastructure і Redis відповідають за передачу задач у фонову обробку. Через це backend не виконує тривалу генерацію самостійно. Замість цього він лише створює задачу і передає її в чергу, цю задачу отримує GenerationProcessingLogic, де розміщено worker-модуль, генераційний pipeline, сервіси рендерингу та формування експортного пакета.

Окремим блоком є ThirdPartyServices, що включає в себе зовнішні сервіси. Наприклад, Ollama API, ComfyUI API та Stripe API застосовуються в цьому рівні. Це не частина основної бізнес-логіки, а саме текстова або візуальна генерація та оплати. Діаграма компонентів показує логіку побудови системи на рівні програмних частин, але для повного розуміння потрібно також показати, як ці частини розміщуються у середовищі виконання (рис. 3.10).

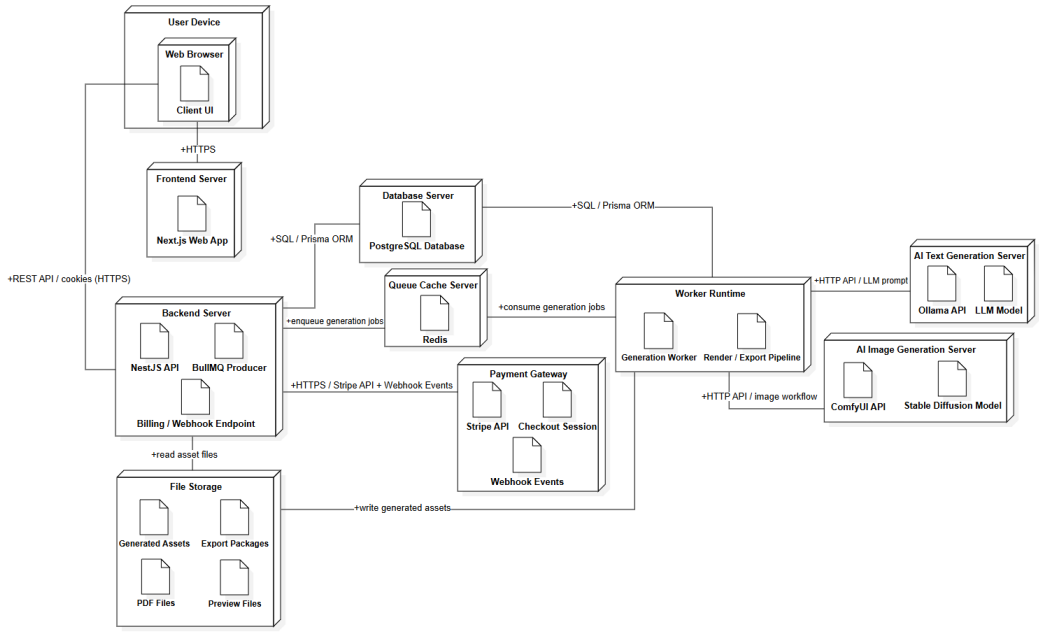

Рисунок 3.10 – Логічна діаграма розгортання системи

Діаграма на рисунку 3.10 показана у логічному вигляді через те, що система запускається на одному локальному пристрої. Такий формат не показує фізичні сервери. Це вузли самої архітектури, але особливістю є те, що їх можна розгорнути окремо в іншому корпоративному середовищі.

Браузер є основним засобом роботи користувача з системою, що розробляється. Через браузер користувач відкриває клієнтський інтерфейс, який розміщується на frontend-сервері як Next.js Web App. REST API і cookies – це міст

між frontend та backend. А backend-сервер містить в собі NestJS API, а також BullMQ Producer та API-методи платіжного модуля і webhook-подій.

PostgreSQL є окремим сервером БД, що зберігає користувачів, ігри, матеріали, сесії та платіжні дані. Redis у свою чергу виконує роль сервера для черг та кешу, який передає задачі генерації від backend до worker-модуля.

Окремо винесено рівень, який взаємодіє з ШІ. Наприклад, для текстової генерації використовується сервер текстової ШІ-генерації з Ollama API та LLM-моделлю (Large Language Model). Це дозволяє уникнути перевантаження backend та виконувати важкі задачі генерації у відокремленому середовищі. Для візуальної генерації використовується сервер візуальної ШІ-генерації з ComfyUI API та моделлю Stable Diffusion.

Файлове сховище використовується для збереження створених матеріалів. Окремий платіжний шлюз відповідає за Stripe API, платіжні сеанси та webhook-події. Серверна частина отримує ці події і оновлює доступ до генерації користувача.

Отже, можна зробити висновок, що архітектура вебсервісу побудована як модульна багатокомпонентна система. У такому розподілі система стає зрозумілою для підтримки та дає поле для розвитку окремих частин без зміни архітектури.

Висновки до розділу 3

У третьому розділі було спроєктовано основні частини вебсервісу. Спочатку розглянуто основні сценарії користувача, а потім уже внутрішню логіку системи, що охоплює алгоритм генерації, інформаційні потоки, структуру бази даних, архітектуру та розгортання системи.

Проєктування продемонструвало, що система побудована як асинхронний багатокомпонентний інтелектуальний вебсервіс. Він не зводиться до одного запиту до ШІ-моделі. Користувач задає параметри гри, backend перевіряє дані й доступ, worker-модуль виконує генераційний pipeline, а готовий результат зберігається разом із матеріалами гри. У цьому розділі сформовано проєктну основу для програмної реалізації вебсервісу та подальшої перевірки його роботи.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБСЕРВІСУ

4.1 Структура програмної реалізації вебсервісу

Програмна реалізація вебсервісу виконана у форматі монорепозиторію. Це має характерну особливість у тому, що усі основні частини системи знаходяться в одному репозиторії, але розділені на окремі застосунки. Такий підхід є зручним для такого проєкту, що розробляється через те, що є такі частини як frontend, backend, worker-модуль і спільні пакети. Загальна структура показана на рисунку 4.1.

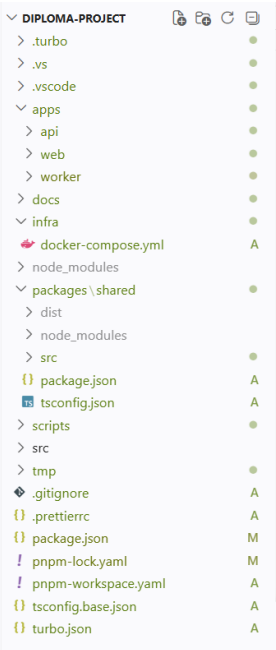

Рисунок 4.1 – Загальна структура вебсервісу

У корені проєкту є конфігураційні файли: package.json, pnpm-workspace.yaml, turbo.json і tsconfig.base.json, що потрібні для запуску, роботи workspace та спільних налаштувань TypeScript. Окремо присутня директорія infra, у якій розміщується конфігурація Docker Compose, що потрібна для роботи PostgreSQL і Redis. Директорія apps – основна реалізація вебсервісу. У ній утворюються три ключові частини: web, api та worker-модуль. Це окремі частини системи, що відповідають за різні функції проєкту.

Клієнтська частина функціонує у межах директорії apps/web, яка реалізована через поєднання Next.js, React і TypeScript. Цей клієнтський шар містить в собі такі

елементи, як захищені сторінки, сторінка статусу генерації, профіль користувача, галерея ігор та сторінка авторизації. Щоб звернутись до серверної частини, використовуються файли, які розташовані у src/lib.

Серверна частина розташована у директорії apps/api. Реалізація була виконана на NestJS, а у структурі присутні модулі auth, generation, billing, queues і prisma. Кожен модуль виконує свою роль, наприклад, авторизацію, запуск генерації та інші дії. Окремо prisma-схема та міграції знаходяться у apps/api/src/prisma.

Особливо важливим для вебсервісу є фоновий модуль генерації, який розміщується у директорії apps/worker. Його роль критично важлива через те, що він виконує задачі, які потребують часу та не повинні запускатись безпосередньо в серверній частині. Фоновий модуль містить pipeline, кроки генерації, сервіси, render, шаблони, типи та обробник задач BullMQ. Окремо в проєкті є директорія packages/shared. Вона виконує роль спільного пакета для частин системи, що дозволяє не змішувати код клієнта, сервера і worker-модуля.

Реалізована загальна структура повністю відповідає спроєктованій архітектурі, яка була представлена в попередньому розділі. Відокремлення кожної частини вебсервісу є зручним та більш керованим підходом. Це дозволяє розглянути частини вебсервісу детально та окремо.

4.2 Реалізація клієнтської частини вебсервісу

Клієнтська частина вебсервісу реалізована в apps/web. Основна логіка frontend зосереджена в маршрутах src/app, API-клієнтах src/lib та компонентах захищеної частини. Розглядаються саме механізми роботи клієнтської частини.

Реалізація багатокрокового майстра створення гри

Створення гри зроблено у форматі покрокового майстра. Основний компонент знаходиться у файлі WizardFlow.tsx, а стан форми винесений у useCreateDraft.ts і types.ts. Це дає можливість окремо зберігати дані форми, перевіряти кроки та формувати фінальний запит на генерацію гри. Кожен крок працює з частиною чернетки. Важливо, що форма створення гри зберігає тільки ті дані, які потрібні для запуску генерації (рис. 4.2).

```ts
export type CreateDraft = {
  title: string;
  genre: Genre | null;
  players: number | null;
  playerProfiles: DraftPlayerProfile[];
  duration: number | null;
  tone: Tone | null;
  mode: Mode;
  notes: string;
};

export type StepKey =
  | "players"
  | "duration"
  | "tone"
  | "genre"
  | "notes"
  | "title"
  | "review";
```

Рисунок 4.2 – Структура стану майстра створення гри

Наприклад, кількість гравців і профілі гравців зберігаються окремо. Це дає чітку межу, щоб система могла сформувати правильний список гравців перед відправленням на backend. Також кроки майстра винесені в окремий масив STEPS і через це порядок проходження форми контролюється централізовано.

Стан чернетки зберігається через власний hook useCreateDraft (рис. 4.3). Як тільки сторінка завантажилась, він відновлює попередні дані, а після зміни автоматично зберігає їх. Це дозволяє не втрачати введені параметри під час роботи з майстром.

```ts
export function useCreateDraft() {
  const [draft, setDraft] = useState<CreateDraft>(() => defaultDraft());
  const [hydrated, setHydrated] = useState(false);

  useEffect(() => {
    const frame = window.requestAnimationFrame(() => {
      setDraft(loadDraft());
      setHydrated(true);
    });

    return () => {
      window.cancelAnimationFrame(frame);
    };
  }, []);

  useEffect(() => {
    if (!hydrated) return;
    saveDraft(draft);
  }, [draft, hydrated]);

  const progress = useMemo(() => {
    const done = REQUIRED_STEPS.filter((s) => isStepDone(s, draft)).length;
    return Math.round((done / REQUIRED_STEPS.length) * 100);
  }, [draft]);

  function resetAll() {
    clearDraft();
    setDraft(defaultDraft());
  }

  return { draft, setDraft, progress, resetAll, hydrated };
}
```

Рисунок 4.3 – Збереження стану майстра

У фрагменті, наведеному на рис. 4.3, видно, що чернетка спочатку створюється зі стандартними значеннями. Перший рендер дає основу для завантаження даних зі сховища. А ознака hydrated, у свою чергу, запобігає випадковому запису порожнього стану, що є дуже важливим і практичним. Автоматична зміна чернетки виконується кожного разу.

У WizardFlow.tsx є також перехід між кроками майстра (рис. 4.4). Виконується перевірка поточного кроку перед кожним переходом. Якщо дані введені некоректно, система показує користувачу повідомлення, а він залишається на тому ж етапі. Це дозволяє запобігти запуск генерації з неповними даними.

```
340  const goNext = () => {
341    if (!canGoNext) {
342      setGenerationError(stepValidationError || "Complete current step before moving forward.");
343      return;
344    }
345
346    setGenerationError("");
347    setStep((current) => nextStep(current));
348  };
```

Рисунок 4.4 – Перехід між кроками майстра

Цей фрагмент показує просту, але важливу логіку. Кнопка переходу не просто змінює активний крок, а спочатку перевіряє стан поточного етапу. У випадку, коли перевірка пройдена, помилка очищається, а майстер переходить далі.

Валідація та формування даних запиту для генерації

Frontend виконує клієнтську перевірку даних перед самим запуском генерації (рис. 4.5). Це не є повною заміною валідації на стороні backend, але блокує неповний запит навіть до серверної обробки. Мінімальний набір для запуску генерації охоплює назву, жанр, тон, кількість гравців і тривалість.

```
83  export function canStartGeneration(draft: CreateDraft): draft is CreateDraft & {
84    title: string;
85    genre: Genre;
86    tone: Tone;
87    players: number;
88    duration: number;
89  } {
90    return (
91      draft.title.trim().length > 0 &&
92      !!draft.genre &&
93      !!draft.tone &&
94      typeof draft.players === "number" &&
95      typeof draft.duration === "number"
96    );
97  }
```

Рисунок 4.5 – Перевірка готовності чернетки до запуску генерації

Функція canStartGeneration не повертає лише boolean. Уточняються і типи чернетки. TypeScript вже знає, що title, genre, tone, players і duration заповнені, тому зменшується кількість перевірок безпосередньо у коді. Це дозволяє клієнтській частині працювати з визначеними даними перед створенням тіла запиту.

Окремо перевіряються правила конкретних кроків (рис. 4.6). Наприклад, кількість гравців має бути від 2 до 8. Тривалість партії має бути у межах від 10 до 180 хвилин. Такі перевірки одразу прив’язані до кроку майстра, де вводяться дані.

```
220  const stepValidationError = useMemo(() => {
221    if (step === "players") {
222      if (typeof draft.players !== "number") return "Select players count to continue.";
223      if (draft.players < 2 || draft.players > 8) return "Players count must stay between 2 and 8.";
224      return "";
225    }
226
227    if (step === "duration") {
228      if (typeof draft.duration !== "number") return "Select duration to continue.";
229      if (draft.duration < 10 || draft.duration > 180) {
230        return "Duration must stay between 10 and 180 minutes.";
231      }
232      return "";
233    }
234
235    if (step === "tone") {
236      if (!draft.tone) return "Choose tone to continue.";
237      if (supportedTones.length > 0 && !supportedTones.includes(draft.tone)) {
238        return "Selected tone is not currently available. Please choose another tone.";
239      }
240      return "";
241    }
```

Рисунок 4.6 – Перевірка значень поточного кроку

Фрагмент на рис. 4.6 виконує роль перевірки якості введення. Це забезпечує, що користувач не може перейти далі, якщо значення виходять за допустимі межі, що зменшує кількість помилкових запитів до backend. Після перевірки для backend іде перетворення чернетки у тіло запиту (рис. 4.7). Через те, що внутрішній стан форми не повинен напряму відправлятися на сервер, це доцільно розглядати як окремий крок. Тіло запиту відповідає контракту StartGenerationRequest. На цьому етапі frontend перетворює дані, які ввів користувач, у формат генераційного запиту.

```
99   export function toStartGenerationPayload(draft: CreateDraft): StartGenerationPayload {
100    if (!canStartGeneration(draft)) {
101      throw new Error("Draft is incomplete and cannot be sent for generation.");
102    }
103
104    return {
105      title: draft.title.trim(),
106      genre: draft.genre,
107      tone: draft.tone,
108      mode: draft.mode,
109      playersCount: draft.players,
110      playerProfiles: toPlayerProfilePayload(draft.players, draft.playerProfiles),
111      durationMinutes: draft.duration,
112      notes: draft.notes.trim(),
113      templateId: null,
114    };
115  }
```

Рисунок 4.7 – Формування даних запиту для backend

Реалізація запуску генерації через API

API-запити винесені в окремий шар. Базова функція формує URL backend, додає JSON-заголовки, передає cookies і зводить помилки до єдиного формату (рис. 4.8). Таким чином, всі API-функції працюють однаково.

```
export async function requestJson<T>(path: string, init?: RequestInit): Promise<T> {
  let response: Response;

  try {
    response = await fetch(`${getApiBaseUrl()}${path}`, {
      ...init,
      headers: {
        "Content-Type": "application/json",
        ...(init?.headers ?? {}),
      },
      credentials: "include",
      cache: "no-store",
    });
  } catch (error) {
    throw new ApiRequestError(0, error, {
      message: "VYREX is not available right now. Try again in a moment.",
      code: "API_UNAVAILABLE",
    });
  }

  const raw = await response.text();
  const payload = raw ? safelyParseJson(raw) : null;

  if (!response.ok) {
    throw new ApiRequestError(
      response.status,
      payload,
      extractErrorDetails(payload, `Request failed with status ${response.status}.`),
    );
  }

  return payload as T;
}
```

Рисунок 4.8 – Базова функція запиту до backend

Ключовим параметром є credentials: "include". Його роль дуже зрозуміла, а саме, щоб браузер передавав cookies сесії разом із запитом. Без цього захищені backend-методи не змогли б визначити користувача. У generation-api.ts описані функції для запуску генерації, отримання статусу та результату (рис. 4.9). Підхід такий, що компоненти не викликають fetch напряму, а працюють через готові функції API-шару. Це робить зв’язок frontend з backend більш контрольованим.

```
export async function startGeneration(payload: StartGenerationRequest) {
  return requestJson<GenerationRecord>("/generation/start", {
    method: "POST",
    body: JSON.stringify(payload),
  });
}

export async function fetchGenerationStatus(generationId: string) {
  return requestJson<GenerationRecord>(`/generation/${generationId}/status`, {
    method: "GET",
  });
}

export async function fetchGenerationResult(generationId: string) {
  return requestJson<GenerationRecord>(`/generation/${generationId}/result`, {
    method: "GET",
  });
}
```

Рисунок 4.9 – API-функції генерації

Основний запуск генерації виконується безпосередньо в WizardFlow.tsx. Важливим рішенням є повторна перевірка чернетки і метаданих, а після успішного створення генерації користувач переходить на сторінку, де відображається статус (рис. 4.10). У випадку, коли користувач не авторизований або не має доступу, йому система показує відповідне модальне вікно.

```
364   const launchGeneration = useCallback(async () => {
365   setGenerationError("");
366
367   if (!canStartGeneration(draft)) {
368     setGenerationError("The draft is incomplete. Finish the required steps before creating the game.");
369     return;
370   }
371
372   setGenerating(true);
373
374   try {
375     const generation = await startGeneration(toStartGenerationPayload(draft));
376     router.push(`/app/generation/${generation.id}`);
377   } catch (error) {
378     if (error instanceof ApiRequestError && error.status === 401) {
379       setAuthGateOpen(true);
380       return;
381     }
382
383     if (error instanceof ApiRequestError && error.status === 403) {
384       setBillingGateOpen(true);
385       return;
386     }
387
388     setGenerationError(error instanceof Error ? error.message : "Failed to start generation.");
389   } finally {
390     setGenerating(false);
391   }
392 }, [draft, metadata, refreshSession, router]);
```

Рисунок 4.10 – Запуск генерації з майстра

Фрагмент показує весь клієнтський запуск генерації. Якщо успішно все виконалось, використовується router.push, що переводить користувача на /app/generation/{id}. Помилки 401 і 403 обробляються не разом, бо вони мають різну поведінку в інтерфейсі.

Моніторинг генерації та обробка станів

Сторінка генерації реалізована так, що отримує generationId з маршруту і починає періодично запитувати стан генерації. Для цього використовується періодичне опитування через setTimeout, інтервал якого становить 2500 мс. Фрагмент, показаний на рис. 4.11 – основа моніторингу генерації. Коли статус ще незавершений, сторінка робить план на наступний запит. А у випадку, коли статус позначається як COMPLETED, frontend одразу завантажує повний результат. FAILED або CANCELED запускає перехід сторінки у стан помилки, де виводиться відповідне повідомлення. Ще однією важливою дією є очищення таймеру, якщо користувач залишає сторінку. Disposed блокує оновлення стану після завершення асинхронного запиту, що захищає сторінку від неправильного оновлення стану.

```
36    const load = async () => {
37      setLoadState((current) => (current === "idle" ? "loading" : "polling"));
38
39      try {
40        const nextGeneration = await fetchGenerationStatus(generationId);
41        if (disposed) return;
42
43        setGeneration(nextGeneration);
44        if (nextGeneration.outputJson) {
45          setResultRecord(nextGeneration);
46        }
47        setError("");
48
49        if (nextGeneration.status === "COMPLETED") {
50          const completed = await fetchGenerationResult(generationId);
51          if (disposed) return;
52
53          setResultRecord(completed);
54          setLoadState("success");
55          return;
56        }
57
58        if (nextGeneration.status === "FAILED" || nextGeneration.status === "CANCELED") {
59          setLoadState("error");
60          return;
61        }
62
63        timer = window.setTimeout(() => {
64          void load();
65        }, 2500);
66      } catch (loadError) {
67        if (disposed) return;
68
69        setError(
70          loadError instanceof Error
71            ? loadError.message
72            : "Failed to load generation progress.",
73        );
74        setLoadState("error");
75      }
76    };
```

Рисунок 4.11 – Запит статусу генерації

Якщо генерація завершена, frontend починає працювати з готовим outputJson, а не проміжним статусом. Система робить нормалізацію результату через normalizeOutput, а після чого сторінка визначає, чи придатний він для демонстрації (рис. 4.12). Передбачено і повторний запуск, що виконується через retryGeneration. Після створення нової генерації користувач переходить на новий маршрут статусу.

```
88    const output = useMemo(
89      () => normalizeOutput(resultRecord?.outputJson ?? generation?.outputJson),
90      [generation?.outputJson, resultRecord],
91    );
92    const activeStep = useMemo(() => getActiveStep(generation), [generation]);
93    const isCompleted = generation?.status === "COMPLETED" && !!output;
94
95    const handleRetry = async () => {
96      setRetrying(true);
97      setError("");
98
99      try {
100       const nextGeneration = await retryGeneration(generationId);
101       router.push(`/app/generation/${nextGeneration.id}`);
102     } catch (retryError) {
103       setError(
104         retryError instanceof Error
105           ? retryError.message
106           : "Failed to retry generation.",
107       );
108       setRetrying(false);
109     }
110   };
```

Рисунок 4.12 – Підготовка результату та повторний запуск генерації

Забезпечується захищений доступ до сторінок генерації через ProtectedRoute, а матеріали гри відкриваються через backend-запити. Отже, клієнтська частина реалізує повний технічний цикл роботи з генерацією, а не лише інтерфейс. Доцільно розглянути і серверну частину для повного розуміння процесів.

4.3 Реалізація серверної частини та структури класів

Серверна частина виступає координатором між клієнтською частиною, базою даних, платіжним модулем і worker-модулем. Основна задача полягає не тільки в прийманні HTTP-запитів, а ще у керуванні запуском генерації, перевірці сесії, перевірці прав доступу. Окрему важливу роль серверна частина виконує для стану генерації та передачі завдання у чергу.

На діаграмі класів (рис. 4.13) показано основні класи, що формують серверну частину вебсервісу. Persistence-сутності на діаграмі подано у скороченому вигляді, оскільки повна структура таблиць, ключів та зв’язків бази даних була наведена окремо на логічній схемі БД. Деякі допоміжні класи не були додані для уникнення перевантаження діаграми та збереження читабельності.

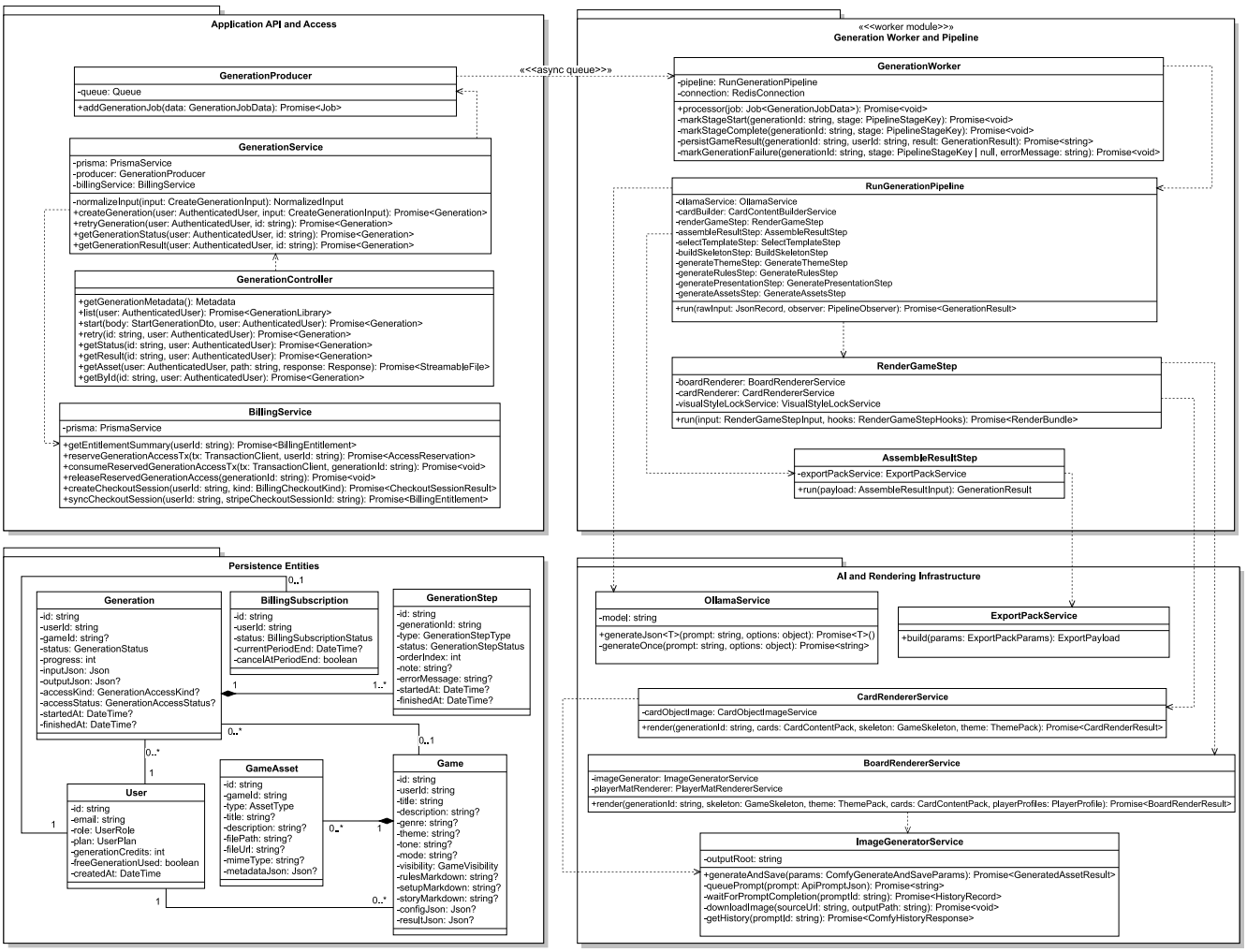

Рисунок 4.13 – Діаграма класів вебсервісу

Основний вхідний клас серверної частини – це GenerationController, який описує API для роботи з генерацією. До них належать запуск, повторний запуск, отримання статусу, отримання результату, робота з бібліотекою та доступ до файлів. Окремо потрібен SessionAuthGuard для захищених запитів і саме тому методи контролера отримують перевіреного користувача, а не лише тіло запиту. Завдяки цьому зменшується кількість ручних перевірок у самих методах.

Не менш важливим є серверна валідація, яка не винесена у випадкові перевірки всередині контролера. Використовується BodyValidationPipe для запуску генерації, що приймає функцію validateStartGenerationDto (рис. 4.14). Цей підхід дозволяє контролеру залишатися компактним, а перевірка вхідних даних відбувається до входження у бізнес-логіку. Це полегшує підтримку через те, що правила валідації зосереджені в одному місці і не розкидані по різних частинах.

```
27  export class GenerationController {
61    @UseGuards(SessionAuthGuard)
62    @Post('start')
63    async start(
64      @Body(
65        new BodyValidationPipe('generation start', validateStartGenerationDto),
66      )
67      body: StartGenerationDto,
68      @CurrentUser() user: AuthenticatedUser,
69    ) {
70      return this.generationService.createGeneration(user, body);
71    }
72
73    @UseGuards(SessionAuthGuard)
74    @Patch(':id/publication')
75    async updatePublication(
76      @Param('id') id: string,
77      @Body(new BodyValidationPipe('publication', validatePublicationDto))
78      body: PublicationDto,
79      @CurrentUser() user: AuthenticatedUser,
80    ) {
81      return this.generationService.updateGenerationPublication(
82        user,
83        id,
84        body.published,
85      );
86    }
87    @UseGuards(SessionAuthGuard)
88    @Post(':id/retry')
89    async retry(@Param('id') id: string, @CurrentUser() user: AuthenticatedUser) {
90      return this.generationService.retryGeneration(user, id);
91    }
92    @UseGuards(SessionAuthGuard)
93    @Get(':id/status')
94    async getStatus(
95      @Param('id') id: string,
96      @CurrentUser() user: AuthenticatedUser,
97    ) {
98      return this.generationService.getGenerationStatus(user, id);
99    }
100   @UseGuards(SessionAuthGuard)
101   @Get(':id/result')
102   async getResult(
103     @Param('id') id: string,
104     @CurrentUser() user: AuthenticatedUser,
```

Рисунок 4.14 – Реалізація захищених API-методів генерації

Цей фрагмент на рисунку 4.14 є показником стилю реалізації backend API. Це видно з того, що контролер не створює генерацію самостійно, він не працює напряму з чергою і не містить логіку платежів. Лише приймається запит і передається в GenerationService. Завдяки цьому, цей шар стає зрозумілим і простим.

GenerationService – це центральний клас серверної частини, де зосереджено логіку, яка має виконуватись перед потраплянням задачі у worker-модуль. Першим кроком виступає нормалізація параметрів. Потім відбувається запуск транзакції, де резервується доступ до генерації та створюється запис генерації. Після цих процедур створюються етапи генерації. Окремо важливо підкреслити те, що транзакція виконується з рівнем ізоляції Serializable. Наприклад, у випадку, коли користувач має одну безкоштовну генерацію або платний кредит, система не дозволяє два одночасні запуски. Якщо Prisma повертає помилку через конкурентне виконання транзакції, метод виконує повторну спробу.

Backend додає задачу в чергу через GenerationProducer як тільки завершується запис генерації (рис. 4.15). У випадку, коли задача не була додана, сервер звільняє доступ, який був зарезервований та переводить генерацію у стан помилки. Це має велику цінність, тому що генерація не вважається витраченою, якщо вона навіть ще не перейшла у стан обробки.

```
143   async createGeneration(
144     user: AuthenticatedUser,
145     input: CreateGenerationInput,
146   ) {
147     const normalized = this.normalizeInput(input);
148     const persistedInput = {
149       ...normalized,
150       userId: user.id,
151     };
152
153     const generation = await this.runSerializableTransaction(async (tx) => {
154       const accessReservation =
155         await this.billingService.reserveGenerationAccessTx(tx, user.id);
156
157       const createdGeneration = await tx.generation.create({
158         data: {
159           userId: user.id,
160           status: 'QUEUED',
161           progress: 0,
162           inputJson: persistedInput as unknown as Prisma.InputJsonValue,
163           ...accessReservation,
164         },
165       });
166
167       await tx.generationStep.createMany({
168         data: GENERATION_STAGE_BLUEPRINT.map((stage) => ({
169           generationId: createdGeneration.id,
170           type: stage.type,
171           status: 'PENDING',
172           orderIndex: stage.orderIndex,
173           title: stage.title,
174           note: stage.pendingMessage,
175         })),
176       });
177
178       return createdGeneration;
179     });
180
181     try {
182       await this.producer.addGenerationJob({
183         generationId: generation.id,
184         userId: user.id,
185       });
186     } catch (error) {
187       await this.billingService.releaseReservedGenerationAccess(generation.id);
188       await this.prisma.generation.update({
189         where: { id: generation.id },
190         data: {
191           status: 'FAILED',
192           finishedAt: new Date(),
193           errorMessage:
194             error instanceof Error
195               ? error.message.slice(0, 1000)
196               : 'Failed to enqueue generation.',
```

Рисунок 4.15 – Реалізація створення генерації та постановки задачі в чергу

Окремо реалізована повторна генерація retryGeneration є правильним підходом, оскільки система повторно не просить користувача ввести ті ж самі параметри. Замість цього backend бере попередній inputJson збереженої генерації. Створюється нова генерація через той самий метод createGeneration. Усі перевірки виконуються однаково для першого і повторного запуску.

Створений BillingService – це не звичайний допоміжний модуль оплати. Його роль значно складніша, тому що він бере участь у контролі доступу до генерації. Тобто сервіс перевіряє чи є безкоштовна генерація, кредити, активна підписка, денний ліміт та доступ адміністратора, який має безлімітні генерації. Саме тому є процес резервування доступу, замість миттєвого списання.

Для зв’язку з worker-модулем використовується клас GenerationProducer. Реалізація нескладна, але архітектурно важлива для системи. Основною функцією цього класу є приховування від основного сервісу роботи з BullMQ. Важливо, що GenerationService не бачить деталей Redis або очищення задач. Його роль – передавати тільки generationId, а GenerationProducer формує задачу generate-game. Ще одна важлива частина серверної реалізації – захищена видача згенерованих файлів (рис. 4.16). Клієнтська частина не відкриває файлову систему напряму.

```
async resolveGeneratedAsset(user: AuthenticatedUser, relativePath: string) {
  if (typeof relativePath !== 'string' || !relativePath.trim()) {
    throw new BadRequestException('Asset path is required.');
  }

  const cleaned = normalize(relativePath)
    .replace(/^(\.\.(\/|\\|$))+/, '')
    .replace(/\\/g, '/');
  const absolutePath = resolve(this.getGeneratedAssetsRoot(), cleaned);
  const root = this.getGeneratedAssetsRoot();

  if (!absolutePath.startsWith(root + sep) && absolutePath !== root) {
    throw new BadRequestException('Invalid asset path.');
  }

  if (
    user.role !== 'SUPER_ADMIN' ||
    user.email.toLowerCase() !== SUPER_ADMIN_EMAIL
  ) {
    const generations = await this.prisma.generation.findMany({
      where: {
        userId: user.id,
      },
      select: {
        outputJson: true,
      },
    });

    const ownsAsset = generations.some((generation) =>
      this.searchJsonValueForPath(generation.outputJson, cleaned),
    );

    if (!ownsAsset) {
      throw new ForbiddenException('Asset access denied.');
    }
  }

  await access(absolutePath);
  return {
    absolutePath,
    fileName: absolutePath.split(sep).pop() ?? 'asset',
    mimeType: this.guessMimeType(cleaned),
```

Рисунок 4.16 – Захищена видача згенерованих файлів

Цей фрагмент на рисунку 4.16 показує, що backend контролює доступ не тільки до записів у базі даних, а й до самих файлів результату. Генерація створює реальні SVG, PNG, PDF та ZIP-матеріали, саме тому це важливо для вебсервісу. Користувач отримує тільки ті файли, які реально належать до його генерацій. Для адміністратора реалізовано службовий доступ до згенерованих файлів через роль SUPER_ADMIN, що потрібно для контролю роботи вебсервісу.

Також реалізовано публікацію готових ігор до галереї (рис. 4.17). Для цього використовується окремий метод для оновлення видимості генерації. Backend дозволяє публікувати тільки ті генерації, які завершені та мають збережений результат і пов’язаний запис гри. Це не дає опублікувати неповну генерацію.

```
518     if (!generation) {
519       throw new NotFoundException('Generation not found.');
520     }
521
522     this.assertOwner(user, generation.userId);
523
524     if (generation.status !== 'COMPLETED' || !generation.outputJson) {
525       throw new BadRequestException(
526         'Only completed games can be published to Examples.',
527       );
528     }
529
530     if (!generation.game) {
531       throw new BadRequestException(
532         'This generation does not have a saved game record yet.',
533       );
534     }
535
536     const updatedGame = await this.prisma.game.update({
537       where: { id: generation.game.id },
538       data: {
539         isPublished: published,
540         visibility: published ? 'PUBLIC' : 'PRIVATE',
541       },
542       select: {
543         id: true,
544         isPublished: true,
545         visibility: true,
546       },
547     });
548
549     return {
550       id: generation.id,
551       gameId: updatedGame.id,
552       isPublished: updatedGame.isPublished,
553       visibility: updatedGame.visibility,
```

Рисунок 4.17 – Перевірка можливості публікації готової гри

Таким чином, серверна реалізація є центральним шаром, який узгоджує роботу основних частин вебсервісу. Контролер приймає запити, сервіс керує станом генерації, платіжний модуль перевіряє і резервує доступ, а GenerationProducer передає задачу в чергу. Важливо зазначити, що backend також відповідає за контроль доступу до створених файлів і не передає матеріали безпосередньо з файлової системи. Сервер також дозволяє публікувати ігри лише після того, як генерація завершиться успішно.

4.4 Реалізація worker-модуля та генераційного pipeline

Як вже було визначено, серверна частина створює генерацію та передає завдання до черги. Worker-модуль реалізовує це завдання, а саме перетворює параметри, які ввів користувач, на повний набір матеріалів настільної гри. До таких матеріалів відносяться структурована механіка гри, тематичне оформлення, правила гри, картки, ігрове поле, планшети гравців та сформовані файли.

Файли generation.worker.ts та run-generation.pipeline.ts є центральними у worker-модулі. Основна роль generation.worker.ts – це виконання задачі з черги, оновлення стану генерації та забезпечення збереження результату. А run-generation.pipeline.ts відповідає за послідовність генераційного pipeline.

Виконання задачі та фіксація стану генерації

RunGenerationPipeline отримує збережені параметри гри від worker-модуля. Це виконується після отримання задачі. Кожен етап у pipeline має свій обробник початку, завершення та проміжного результату. Особливістю є те, що стан генерації оновлюється безпосередньо під час її виконання (рис. 4.18).

```
653   const runPromise = pipeline.run(inputJson, {
654     onStageStart: async (stage) => {
655       activeStage = stage;
656       await markStageStart(generationId, stage);
657     },
658     onStageComplete: async (stage) => {
659       await markStageComplete(generationId, stage);
660     },
661     onStageSnapshot: async (stage, snapshot) => {
662       await storeStageSnapshot(generationId, stage, snapshot);
663     },
664   });
665   runPromise.catch((error) => {
666     if (error instanceof GenerationCanceledError) return;
667     logger.warn("Generation pipeline finished after cancellation or timeout", {
668       generationId,
669       jobId: job.id,
670       error: error instanceof Error ? error.message : String(error),
671     });
672   });
673
674   const result = await Promise.race([runPromise, timeoutPromise]);
675   if (pipelineTimeoutId) clearTimeout(pipelineTimeoutId);
```

Рисунок 4.18 – Виконання pipeline та оновлення стану генерації у worker-модулі

Цей фрагмент на рисунку 4.18 демонструє, що генерація не виконується як одна неподільна операція. Завдяки onStageStart і onStageComplete фіксується активний етап та його завершення у БД. А onStageSnapshot, у свою чергу, зберігає частково сформований результат. Promise.race у поєднанні з таймером виступає обмеженням часу виконання, що допомагає не залишати генерацію активною без контролю, наприклад, якщо генерація не завершується у встановлений час.

Формування початкової основи гри

Першим кроком є перевірка отриманих параметрів, вибір базової моделі гри, побудова початкового механічного каркасу гри та формування тематичного наповнення. Реалізація виконана у формі послідовного виконання SelectTemplateStep, BuildSkeletonStep і GenerateThemeStep (рис. 4.19).

```
74  async run(
75    rawInput: Record<string, unknown>,
76    observer?: RunGenerationPipelineObserver,
77  ): Promise<GenerationResult> {
78    const validated = validateAndNormalizeInput(rawInput);
79
80    if (!validated.ok || !validated.normalized) {
81      throw new Error(`Invalid generation input: ${validated.errors.join("; ")}`);
82    }
83
84    const input = validated.normalized;
85    logger.info("Pipeline started", { title: input.title, mode: input.mode });
86
87    await observer?.onStageStart?.("CONCEPT");
88    const selection = this.selectTemplateStep.run(input);
89    const skeleton = this.buildSkeletonStep.run(input, selection.template);
90    const { theme, usedFallback } = await this.generateThemeStep.run(input, skeleton);
91    await observer?.onStageSnapshot?.("CONCEPT", {
92      input,
93      playerProfiles: input.playerProfiles,
94      playerSections: [],
95      template: selection.template,
96      skeleton,
97      theme,
98      previewOnly: true,
99    });
100   await observer?.onStageComplete?.("CONCEPT");
```

Рисунок 4.19 – Формування концепції, механічного каркасу та теми гри

Генерація не починається, якщо не пройдена перевірка вхідних даних та не зроблена нормалізація. Після цих дій обирається одна з реалізованих механічних основ гри. До таких шаблонів входять: influence_control, mission_race, shared_crisis, resource_engine, risk_tradeoff або hidden_pressure. TemplateSelectorService виконує роль вибору механічної основи. Принцип роботи такий, що кожен доступний варіант отримує «оцінки» відповідності. Окремо враховуються жанр, тональність, тривалість партії, кількість гравців та ключові ознаки із назви гри, яку обрав користувач. Є перевірка і характерних сигналів певної механіки. Такими сигналами можуть бути, наприклад, назва гри, у якій є згадування маршрутів. У такому випадку буде більша ймовірність вибору саме змагального виконання місій, а не, наприклад, моделі ризику. Фрагмент таких перевірок наведено на рис. 4.20.

```
245     const family = templateFamilyFor(template.id);
246     const familyScore = familyWeights[family];
247     if (familyScore > 0) {
248       score += familyScore * 2;
249       reasons.push(`mechanic profile matched ${family}`);
250     }
251
252     const strongMissionStructure = hasStrongMissionStructure(combinedText);
253     const explicitPushYourLuckIntent = hasExplicitPushYourLuckIntent(combinedText);
254     const explicitNegotiationIntent =
255       input.genre.toLowerCase() === "negotiation" || hasExplicitNegotiationIntent(combinedText);
256
257     if (family === "mission" && strongMissionStructure) {
258       score += 8;
259       reasons.push("route-objective structure detected");
260     }
261
262     if (family === "risk" && strongMissionStructure && !explicitPushYourLuckIntent) {
263       score -= 6;
264       reasons.push("route-objective cues outweighed generic risk cues");
265     }
266
267     if (family === "risk" && explicitNegotiationIntent && !/push your luck|press your luck|gamble|gambling|risk[-\s]?reward/i.test(combinedText)) {
268       score -= 8;
269       reasons.push("negotiation intent outweighed generic tradeoff cues");
270     }
271
272     if (family === "control" && explicitNegotiationIntent) {
273       score += 10;
274       reasons.push("deal-making negotiation structure detected");
275     }
276
277     if (family === "hidden" && explicitNegotiationIntent && /secret|hidden|bluff|betray|covert|promise|auction/i.test(combinedText)) {
278       score += 7;
279       reasons.push("hidden negotiation pressure detected");
280     }
281
282     if (input.playersCount === 1 && template.supportsSolo) {
283       score += 2;
284     }
```

Рисунок 4.20 – Оцінювання відповідності механічної основи параметрам гри

Тобто, система не вибирає механічну основу випадковим способом, а обрана основа визначає механічний каркас гри, на базі якого формуються інші етапи. ThemeGeneratorService виконує роль тематичного наповнення, де у ньому реалізовано запит до локальної мовної моделі за допомогою OllamaService. Також очікується у ньому структурований JSON. У випадку, коли у відповіді структура некоректна, worker-модуль використовує резервний варіант (рис. 4.21).

```
688     if (!hasMinimumViableCore(llm)) {
689       logger.warn("Theme generation fell back completely", {
690         title: input.title,
691         templateId: skeleton.templateId,
692       });
693
694       const deterministicTheme: ThemePack = {
695         ...fallback,
696         resourceNames: skeleton.resources.some((resource) => resource.key === "prestige")
697           ? { ...fallback.resourceNames, prestige: fallback.scoreName }
698           : fallback.resourceNames,
699         actionCommands: Object.fromEntries(
700           actionKeys.map((key) => [key, buildDeterministicActionCommand(key, fallback, skeleton)]),
701         ),
702       };
703
704       return { theme: deterministicTheme, usedFallback: true };
705     }
```

Рисунок 4.21 – Використання резервної теми при некоректній відповіді

Мовна модель використовується лише для тематичного змісту. Від однієї відповіді не залежить вся генерація, що дає повний контроль. Наприклад, якщо ШІ-рівень поверне недостатньо повний результат, worker-модуль працює надалі з коректною структурованою темою, сформованою програмно як резерв.

Побудова структурованої механіки гри

Формування механіки виконується під час створення механічного каркасу, що реалізується до накладання тематичного змісту поверх нього. Це дозволяє не створювати тему довільно. Замість цього чітко прив’язується тема до узгоджених ресурсів, дій, колод, зон та компонентів.

Реалізовано окремо побудову механіки для кожного шаблону. Це зроблено у класі MechanicsExpanderService, де після формування конкретної моделі додаються універсальні компоненти, необхідні для гри. Наприклад, такими компонентами є маркер першого гравця, області для колод, трек очок та інші (рис. 4.22).

```
117  export class MechanicsExpanderService {
118    build(
119      input: NormalizedGameGenerationInput,
120      template: BaseGameTemplate,
121      partial: Pick<GameSkeleton, "rounds" | "playersCount" | "resources" | "actions" | "components">,
122    ): MechanicsExpansion {
123      const expansion = (() => {
124      switch (template.id) {
125        case "influence_control":
126          return this.buildInfluenceControl(input, partial);
127        case "mission_race":
128          return this.buildMissionRace(input, partial);
129        case "shared_crisis":
130          return this.buildSharedCrisis(input, partial);
131        case "resource_engine":
132          return this.buildResourceEngine(input, partial);
133        case "risk_tradeoff":
134          return this.buildRiskTradeoff(input, partial);
135        case "hidden_pressure":
136          return this.buildHiddenPressure(input, partial);
137        default:
138          return this.buildMissionRace(input, partial);
139      }
140      })();
141
142      return this.addUniversalPlayabilityGuarantees(input, partial, template.id, expansion);
143    }
```

Рисунок 4.22 – Вибір механічної моделі та додавання ігрових елементів

Важливо, що цей етап дає системі не просто текстову ідею гри, а саме структуровану модель механіки. Така механіка визначає, які взагалі існують ресурси, які колоди потрібні, які дії може виконати гравець та як побудовано ігрове поле. Додатково враховані визначення результату партії та приклад першого ходу. Така чітка структура надалі дає основу для правил, карток і поля.

Статус MECHANICS у pipeline використовується для фіксації готовності механічного каркаса у загальному прогресі генерації. Це означає, що worker-модуль не створює другу незалежну механіку після CONCEPT. Він позначає завершення вже побудованої структурної основи гри.

Формування правил і презентаційного опису

Перехід до правил виконується одразу, як тільки створено тему та механіку гри. Так як правила є ключовим елементом гри, без яких не буде узгодженої взаємодії між гравцями, вони не генеруються безпосередньо мовною моделлю. Клас RulesGeneratorService будує їх програмно з використанням механічного каркасу і теми, які були сформовані до цього. Такий підхід визначає відповідність між ресурсами, колодами та компонентами. Основні блоки правил створюються через набір функцій, які формують огляд гри, підготовку, послідовність раунду, перелік дій та інші частини регламенту (рис. 4.23).

```
export class RulesGeneratorService {
  generate(
    input: NormalizedGameGenerationInput,
    skeleton: GameSkeleton,
    theme: ThemePack,
  ): { rules: RulesPack; usedFallback: boolean } {
    const overview = polishRuleText(buildCanonicalOverview(skeleton, theme));
    const setupText = polishRuleText(
      linesToNumberedText(buildCanonicalSetupLines(skeleton, theme)),
    );

    const turnStructureText = polishRuleText(
      linesToNumberedText(buildCanonicalRoundFlowLines(skeleton, theme)),
    );
    const playAlgorithmText = polishRuleText(
      linesToNumberedText(buildCanonicalPlayAlgorithmLines(skeleton, theme)),
    );
    const playerAreasText = polishRuleText(
      linesToBullets(buildPlayerAreaRuleLines(input.playerProfiles)),
    );
    const systemProfileText = polishRuleText(
      linesToBullets(buildCanonicalSystemProfileLines(skeleton, theme)),
    );
    const exampleTurnText = polishRuleText(
      linesToNumberedText(buildCanonicalExampleTurnLines(skeleton, theme)),
    );

    const builtActions = buildCanonicalActions(skeleton, theme);
    const actions: RulesActionEntry[] = builtActions.map((action) => ({
      key: action.key,
      label: action.label,
      costText: action.costText,
      commandText: action.commandText,
      outcomeText: action.outcomeText,
      restrictions: action.restrictions,
    }));
```

Рисунок 4.23 – Програмне формування основних блоків правил гри

По цьому фрагменту видно, що програмно створюються правила до партії, ходу гравця, використання ресурсів, взаємодії з полем, підрахунку результатів і завершення гри. Вебсервіс об’єднує сформовані розділи в rulesMarkdown, який використовується на сторінці результату та під час експорту матеріалів. PresentationGeneratorService формує короткий опис гри та текст для її представлення користувачу. Ця частина може застосовувати мовну модель через те, що вона відповідає за подачу результату, а не за механічну правильність правил.

Створення карток і специфікацій візуальних матеріалів

Створення компонентів гри відбувається після формування правил. Наприклад, GenerateAssetsStep готує набір промптів і візуальних характеристик для майбутніх матеріалів гри. А формування колод карток відповідно до механіки гри виконується через CardContentBuilderService.

Список колод береться з самого skeleton.mechanics.cardDecks. Тобто кількість, типи й призначення карток визначаються сформованою ігровою моделлю. Під час побудови колод також використовується пам’ять повторів, щоб уникати однакових назв, правил і текстів карток (рис. 4.24).

```
export class CardContentBuilderService {
  build(theme: ThemePack, skeleton: GameSkeleton): CardContentPack {
    const duplicateMemory: CardDuplicateMemory = {
      summaries: new Set<string>(),
      rules: new Set<string>(),
      flavors: new Set<string>(),
      titleStems: new Set<string>(),
      titleStemCounts: new Map<string, number>(),
    };

    const decks = skeleton.mechanics.cardDecks.map((deck) =>
      this.buildDeck(theme, skeleton, deck.key, deck.label, deck.purpose, deck.count, deck.cardTypes, duplicateMemory),
    );

    return {
      decks,
      totalCards: decks.reduce((sum, deck) => sum + deck.cards.length, 0),
    };
  }
```

Рисунок 4.24 – Побудова колод карток на основі механіки гри

Система для кожної карти визначає назву, текст, ефект, тематичний опис, вартість, винагороду, відповідну зону та механічну функцію. Також врахована обробка повторів карток, яка визначена у CardContentBuilderService (рис. 4.25).

```
const finalTitleStem = titleStemKey(card.title);
seenTitleStemCounts.set(finalTitleStem, (seenTitleStemCounts.get(finalTitleStem) ?? 0) + 1);

if (seenFlavors.has(canonicalTextKey(card.flavorText)) || duplicateMemory.flavors.has(canonicalTextKey(card.flavorText))) {
  card.flavorText = resolveDuplicateFlavorText(card, seenFlavors, duplicateMemory.flavors, i + deckVariantSalt * 17);
}

if (seenSummaries.has(canonicalTextKey(card.summary)) || duplicateMemory.summaries.has(canonicalTextKey(card.summary))) {
  card.summary = resolveDuplicateSummaryText(card, seenSummaries, duplicateMemory.summaries, i + deckVariantSalt * 19);
}

if (seenRules.has(canonicalTextKey(card.ruleText)) || duplicateMemory.rules.has(canonicalTextKey(card.ruleText))) {
  card.ruleText = resolveDuplicateRuleText(card, seenRules, duplicateMemory.rules, i + deckVariantSalt * 23);
}

card = normalizeCardForSkeleton(card, skeleton, theme);
```

Рисунок 4.25 – Усунення повторів і нормалізація сформованих карток

У випадку, коли правило, опис або назва повторюються, відповідний текст змінюється до збереження картки у колоді. Після цього набір карток може передаватися до візуального рендерингу.

Генерація ілюстрацій карток та первинна перевірка якості

Кожна картка отримує конкретний центральний об’єкт, що відповідає її функції. Тобто worker-модуль формує візуальний намір і добирає конкретний об’єкт замість генерації складних сцен. Такий підхід використано для того, щоб зображення було зрозумілим для користувача і його можна було одразу розпізнати на готовій картці. Ще одним фактором є те, що дифузійні моделі працюють краще з одиночними об’єктами, аніж з абстрактними сценами.

Генерація виконується у CardObjectImageService через ComfyUI workflow card-object.json. Для зображення визначено розмір, модель, параметри семплінгу та seed. Після отримання PNG-файлу він одразу передається до валідатора (рис. 4.26).

```
108   const saved = await this.getImageGenerator().generateAndSave({
109     workflowFileName: "card-object.json",
110     nodeMap: CARD_OBJECT_WORKFLOW_NODE_MAP,
111     prompt,
112     negativePrompt,
113     width: 1024,
114     height: 1024,
115     steps: 10,
116     cfgScale: 3.5,
117     samplerName: "euler",
118     scheduler: "normal",
119     checkpointName: "sdxl_lightning_8step.safetensors",
120     category: "card-art",
121     generationId: params.generationId,
122     fileNameBase: `${params.card.id}-art-pass-${attempt}`,
123     seed: hashString(`${params.seedKey}|attempt:${attempt}`),
124   });
```

Рисунок 4.26 – Генерація ілюстрації картки через ComfyUI та запуск перевірки

На одне зображення може використатись максимум дві спроби генерації. Коли перший результат має проблеми, запускається повторне створення з посиленими обмеженнями у промпті. Це дозволяє автоматично відреагувати на типові помилки дифузійної генерації, не перериваючи весь процес формування гри. Наприклад, якщо на зображенні присутні зайві об’єкти, виконується друга спроба.

Перевірку ілюстрації покладено на CardObjectImageValidatorService, де він аналізує центральність об’єкта, кількість окремих фрагментів, заповнення країв, наявність рамкоподібної структури та перевантаженість зображення. Це ті проблеми, які виникають найчастіше у дифузійних моделях та заважають користувачу зрозуміти суть картки. Частину перевірок показано на рис. 4.27.

```
export class CardObjectImageValidatorService {
  async validate(filePath: string): Promise<CardObjectImageValidationResult> {
    const decoded = decodePng(await readFile(filePath));
    const analysis = this.analyze(decoded);
    const reasons: string[] = [];

    if (analysis.foregroundRatio < 0.04) {
      reasons.push("main object is too small");
    }

    if (analysis.foregroundRatio > 0.88) {
      reasons.push("foreground coverage is too large for a single isolated object");
    }

    if (analysis.centerOffset > 0.32) {
      reasons.push("main object is not centered enough");
    }

    if (analysis.componentCount > 18 && analysis.largestComponentRatio < 0.5) {
      reasons.push("too many separate foreground components were detected");
    }

    if (analysis.largestComponentRatio < 0.36) {
      reasons.push("no single dominant object silhouette was detected");
    }

    if (analysis.borderForegroundRatio > 0.56 && analysis.frameContinuity > 0.5) {
      reasons.push("frame-like border dominates the image");
    }

    if (analysis.cornerForegroundRatio > 0.24 && analysis.borderForegroundRatio > 0.38) {
      reasons.push("decorative frame corners were detected");
    }

    if (analysis.borderNoise > 46) {
      reasons.push("busy image edges suggest multiple objects or a background pattern");
```

Рисунок 4.27 – Перевірка згенерованої ілюстрації картки

Якщо зображення не проходить перевірку, worker-модуль зберігає причини та приймає рішення про повторну генерацію. Коли друга спроба теж не дала очікуваних результатів, використовується доступний резервний результат. Такі перевірки не оцінюють художню якість, але відсіюють типові проблеми.

Формування ігрового поля та планшетів гравців

У моделі прогресу цей етап має назву VALIDATION, однак у фактичній реалізації він пов’язаний із завершальним формуванням структурованого поля та пов’язаних матеріалів. Перевірка ілюстрацій карток уже виконується на попередньому етапі IMAGES, а тут worker-модуль формує кінцеву структуру гри.

Цей етап у фактичній реалізації пов’язаний із завершальним формуванням структурованого поля та матеріалів. Worker формує кінцеву просторову структуру гри, де BoardRendererService спочатку будує внутрішню модель гри та шари поля (рис. 4.28). Далі відбувається генерація тематичної поверхні поля через workflow board-surface.json. Готове зображення поєднується з структурними елементами. До таких елементів належать зони, переходи, колоди, треки та позначення. Додатково є окремі планшети гравців.

```
    playerProfiles: NormalizedPlayerProfile[],
  ): Promise<BoardRenderResult> {
    const gameModel = this.gameInterpreter.build(skeleton, theme, cards);
    const layout = this.layoutBuilder.build(skeleton, theme, gameModel, playerProfiles);

    const bible = this.styleBibleService.build(theme);
    const boardPrompt = {
      positive: buildBoardSurfacePrompt(theme, bible.paletteHints, layout.themeSkin),
      negative: buildBoardSurfaceNegativePrompt(bible.boardNegativeCore, theme),
    };

    const seed = hashString(
      [
        generationId,
        theme.title,
        theme.world,
        layout.title,
        layout.zones.map((zone) => zone.name).join("|"),
        playerProfiles.map((profile) => profile.name).join("|"),
      ].join("|"),
    );

    const saved = env.skipBoardImageRender
      ? undefined
      : await this.imageGenerator.generateAndSave({
          workflowFileName: "board-surface.json",
          nodeMap: BOARD_WORKFLOW_NODE_MAP,
          prompt: boardPrompt.positive,
          negativePrompt: boardPrompt.negative,
          width: 1536,
          height: 1024,
          steps: 12,
          cfgScale: 3.5,
          samplerName: "euler",
          scheduler: "normal",
          checkpointName: "sdxl_lightning_8step.safetensors",
          category: "boards",
          generationId,
          fileNameBase: "board-surface",
          seed,
        });
```

Рисунок 4.28 – Побудова макета та генерація тематичної поверхні ігрового поля

Поверхня поля – основа для фінальної композиції SVG-матеріалів (рис. 4.29). Згенероване зображення використовується для тематичної основи поля та не замінює структуру гри. Програмно формуються компоненти поля та ігрові зони, які залишаються тісно пов’язаними з механікою.

```
    const composedBoard = await this.composer.compose(board, generationId);
    const playerMats = (await this.playerMatRenderer.render(board, generationId))?.items ?? [];
    const composed = composedBoard ? { ...composedBoard, playerMats } : undefined;

    return {
      board: {
        ...board,
        finalSvgPath: composed?.svgPath,
        finalSvgRelativePath: composed?.svgRelativePath,
        playerMats,
      },
      composed,
    };
  }
}
```

Рисунок 4.29 – Формування фінального SVG-поля та планшетів гравців

Поле містить не лише тематичну картинку, а й підготовлені зони та області використання. Окремі планшети гравців можуть бути включені до експортного набору разом із картками та правилами. Об’єднання цих компонентів дає ігрове поле, яке відображає суть гри та атмосферу. Наприклад, вузли на полі теж використовують згенеровані зображення для більш атмосферного відображення.

Складання результату й експортного пакета

Останнім є етап, де всі створені дані збираються в один результат генерації. AssembleResultStep отримує параметри гри, механічну основу, тему, правила, презентаційний опис, картки та рендеровані матеріали. Починається процес з того, що PdfPackageService готує версію матеріалів для друку. Після цього ExportPackService формує структуру експорту, а ZipPackageService збирає повний архів гри (рис. 4.30). Ці сформовані матеріали утворюють комплект гри.

```
const playerSections = buildPlayerSectionSummaries(params.input.playerProfiles);
const printablePackage = await this.pdfPackageService.build({
  generationId: params.generationId,
  input: params.input,
  playerSections,
  skeleton: params.skeleton,
  theme: params.theme,
  rules: params.rules,
  presentation: params.presentation,
  cards: params.cards,
  render: params.render,
});

const exportPayload = this.exportPackService.build({
  input: params.input,
  playerSections,
  skeleton: params.skeleton,
  theme: params.theme,
  rules: params.rules,
  presentation: params.presentation,
  assets: params.assets,
  cards: params.cards,
  render: params.render,
  printablePackage,
});
const archivePackage = await this.zipPackageService.build({
  generationId: params.generationId,
  exportPayload,
  render: params.render,
});

exportPayload.archivePackage = archivePackage;
if (archivePackage.zipRelativePath) {
  exportPayload.downloadable.archiveZipRelativePath =
    archivePackage.zipRelativePath;
  exportPayload.manifest.filesPlanned.push(archivePackage.zipRelativePath);
}
```

Рисунок 4.30 – Формування друкованого пакета та ZIP-архіву гри

Після генерації гра вже має готовий набір матеріалів. До нього входять правила, опис механіки, картки, поле, планшети гравців і зображення. Для друку PdfPackageService формує окремий пакет. Усі створені файли також збираються в ZIP-архів. Його можна завантажити й використовувати далі. Тому результатом роботи є не просто запис про гру, а матеріали для проведення партії.

Збереження готової гри та завершення worker-задачі

Проміжні матеріали worker зберігає ще під час виконання генерації. Якщо гра вже повністю зібрана і не виникало помилок, у БД створюється запис Game. Тоді ж система підтверджує використання доступу до генерації і переводить Generation у стан COMPLETED. Запис гри заповнюється вже готовими даними. Назва береться з теми, а опис використовує презентаційний блок. Також зберігаються правила, введені користувачем параметри і повний результат генерації (рис. 4.31).

```
444   const createdGame = await tx.game.create({
445     data: {
446       userId,
447       title: result.theme.title,
448       description: result.presentation.shortPitch,
449       genre: result.input.genre,
450       theme: result.theme.world,
451       tone: result.input.tone,
452       mode: result.input.mode,
453       playersMin: result.input.playersCount,
454       playersMax: result.input.playersCount,
455       durationMinutes: result.input.durationMinutes,
456       rulesMarkdown: result.exportPayload.downloadable.rulesMarkdown,
457       setupMarkdown: result.rules.setupText,
458       storyMarkdown: result.presentation.gameBoxDescription,
459       configJson: result.input as unknown as object,
460       resultJson: result as unknown as object,
461     },
462   });
```

Рисунок 4.31 – Збереження сформованої гри у базі даних

Фінальні файли додаються до запису гри, у які входять поле, картки, планшети гравців, PDF-пакет та ZIP-архів. Одразу після цих процедур worker споживає зарезервований доступ та змінює статус генерації на COMPLETED (рис. 4.32), тобто гру успішно згенеровано та вона готова до використання.

```
547     await consumeReservedGenerationAccessTx(tx, generationId);
548
549     const generationUpdate = await tx.generation.updateMany({
550       where: {
551         id: generationId,
552         status: {
553           not: "CANCELED",
554         },
555       },
556       data: {
557         gameId: createdGame.id,
558         status: "COMPLETED",
559         progress: 100,
560         finishedAt: new Date(),
561         outputJson: result as unknown as object,
562         errorMessage: null,
563       },
564     });
565
566     if (generationUpdate.count !== 1) {
567       throw new GenerationCanceledError();
568     }
569
570     return createdGame;
571   });
572
573   return game.id;
574 }
```

Рисунок 4.32 – Завершення генерації та прив’язка експортних матеріалів

Якщо під час виконання виникає помилка, worker переводить генерацію у стан FAILED, фіксує причини збою та звільняє доступ, який був зарезервований. Якщо задача була скасована, подальше збереження результату не виконується.

Отже, можна зробити висновок, що worker-модуль бере на себе всю реалізацію повного циклу створення персоналізованої настільної гри. Особливістю реалізації, яка забезпечує важливий первинний контроль, є послідовна побудова генераційного pipeline. Кожний наступний етап використовує вже сформований результат попереднього етапу. Деякі фрагменти результату генерації у форматі PDF та JSON наведено в додатку А на рисунках А.1–А.8.

4.5 Інтерфейс користувача та адміністратора

Інтерфейс побудовано таким чином, щоб користувач міг швидко перейти до створення гри, де він заповнює параметри гри, яку бажає згенерувати. Першим важливим екраном є сторінка створення гри (рис. 4.33). Вона реалізована як покроковий майстер, де зліва розташовані етапи заповнення, у центрі розміщені активні кроки, а справа короткий підсумок параметрів. Є можливість скинути параметри через кнопку reset draft.

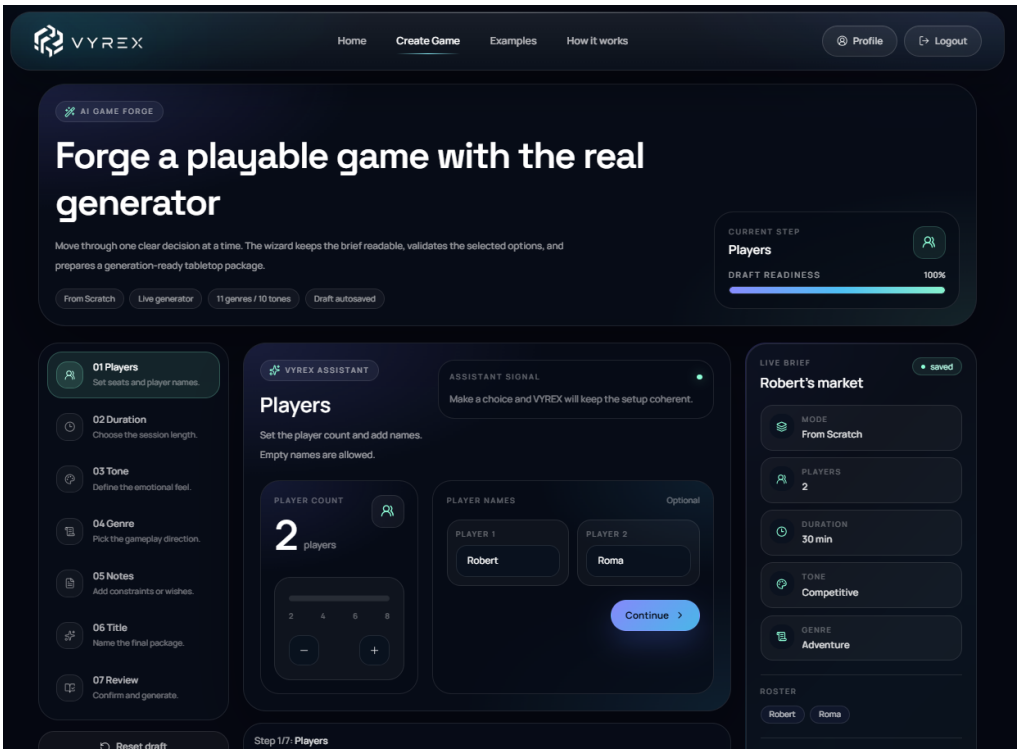

Рисунок 4.33 – Сторінка створення гри

У вебсервісі реалізовано сторінку, яка пояснює користувачу принцип формування персоналізованої гри (рис. 4.34). Вона демонструє послідовні етапи генерації та принцип роботи pipeline. Це показує внутрішні дії у зручному форматі.

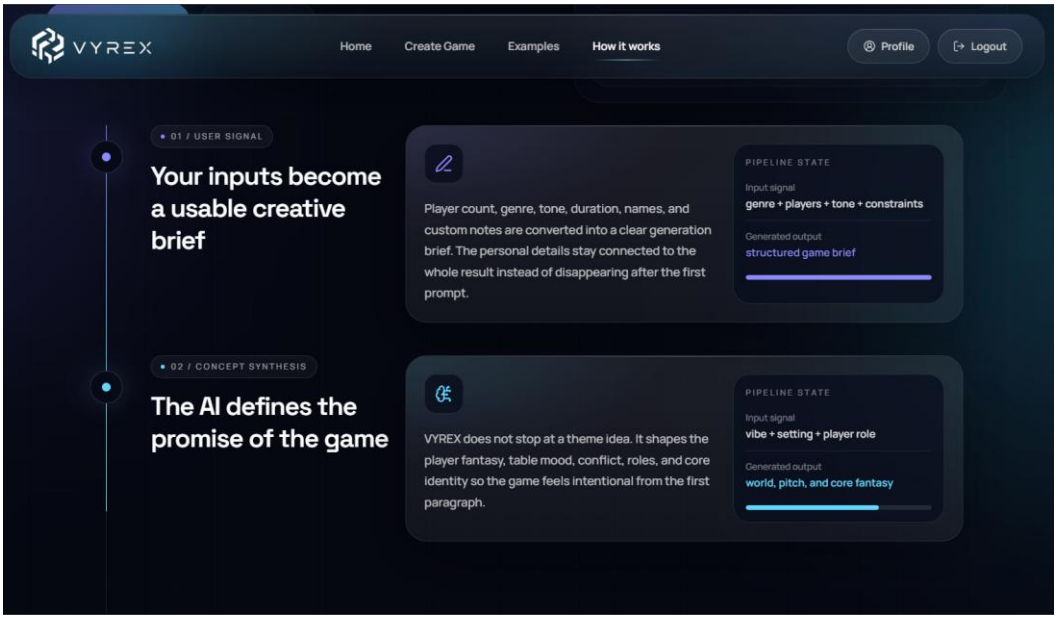

Рисунок 4.34 – Ознайомлення з етапами генерації

Також реалізовано інформаційний блок, який пояснює користувачу принцип персоналізації гри (рис. 4.35). У ньому видно, що введені параметри впливають не лише на початковий запит, а й на тему, ігровий процес та матеріали.

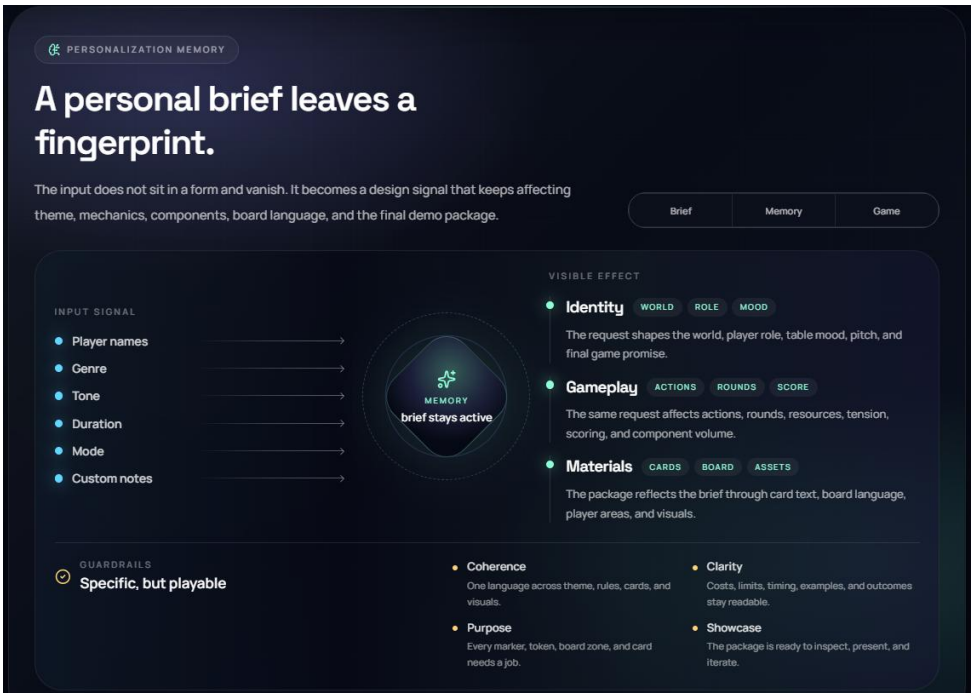

Рисунок 4.35 – Представлення впливу параметрів користувача на формування гри

Окремою сторінкою є перегляд статусу генерації (рис. 4.36). Вона потрібна, бо створення гри виконується не одразу. Користувач бачить загальний прогрес, активний етап та карту pipeline. Це зменшує невизначеність під час очікування.

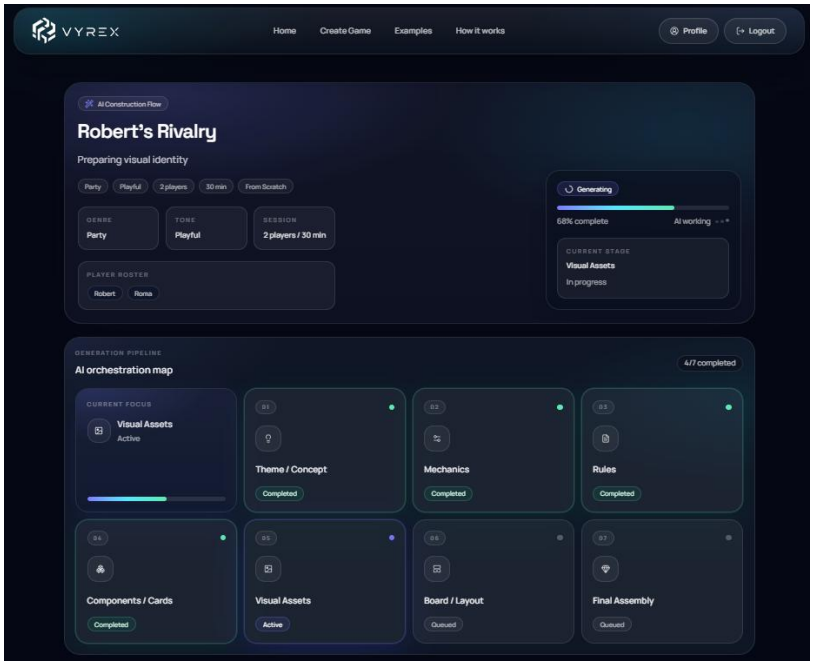

Рисунок 4.36 – Фрагмент зі сторінки відстеження ходу генерації

Одразу, як тільки генерація завершена, користувач переходить до сторінки результату, де показано назву гри, короткий опис, основні параметри, склад гравців, попередній перегляд поля та приклад картки (рис. 4.37). Також присутні дії для відкриття PDF-пакета, перегляду ігрового поля, друку та інше.

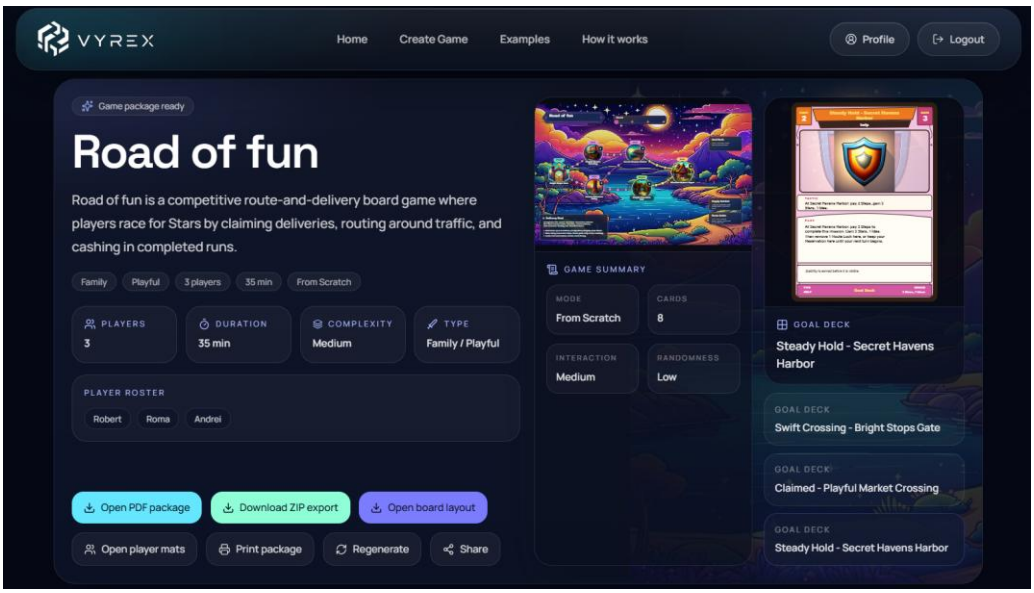

Рисунок 4.37 – Фрагмент зі сторінки результату згенерованої гри

Для користувача є профіль та бібліотека ігор, яка потрібна для зручного повторного доступу до створених ігор. Тут відображаються збережені ігрові пакети, функціонал для зміни пароля або імені, та інші дії керування (рис. 4.38).

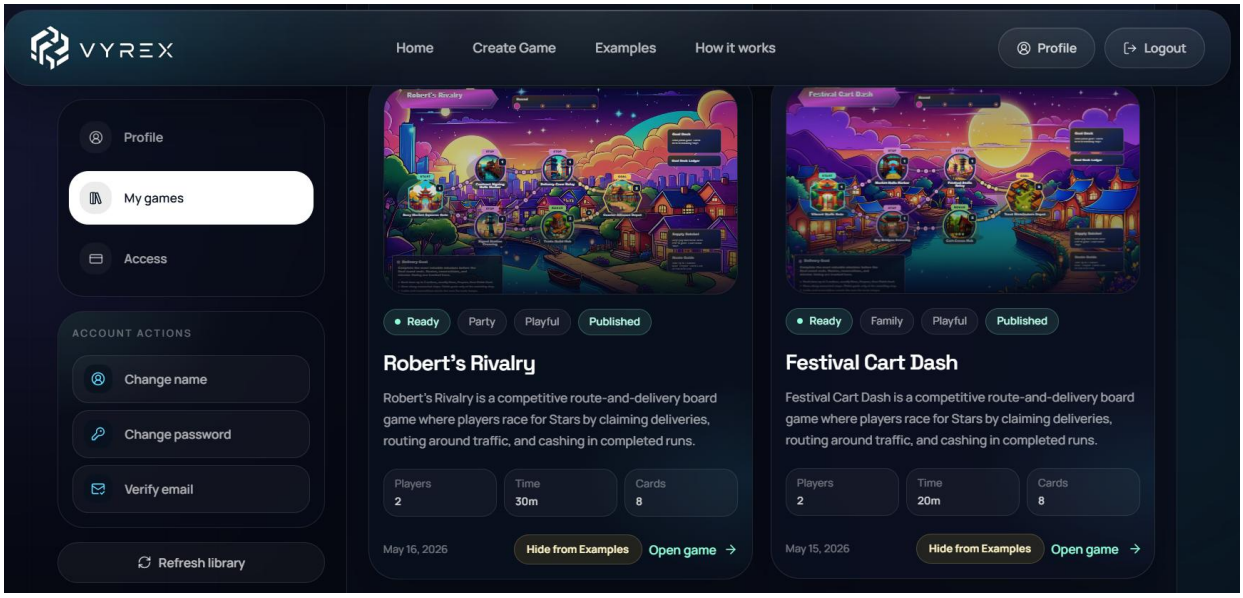

Рисунок 4.38 – Бібліотека збережених ігор користувача у профілі

Адміністратор має ті ж функції, що і користувач, але для нього є окрема сторінка управління. Він може зручно бачити кількість користувачів, активних процесів, згенерованих записів та пакетів (рис. 4.39).

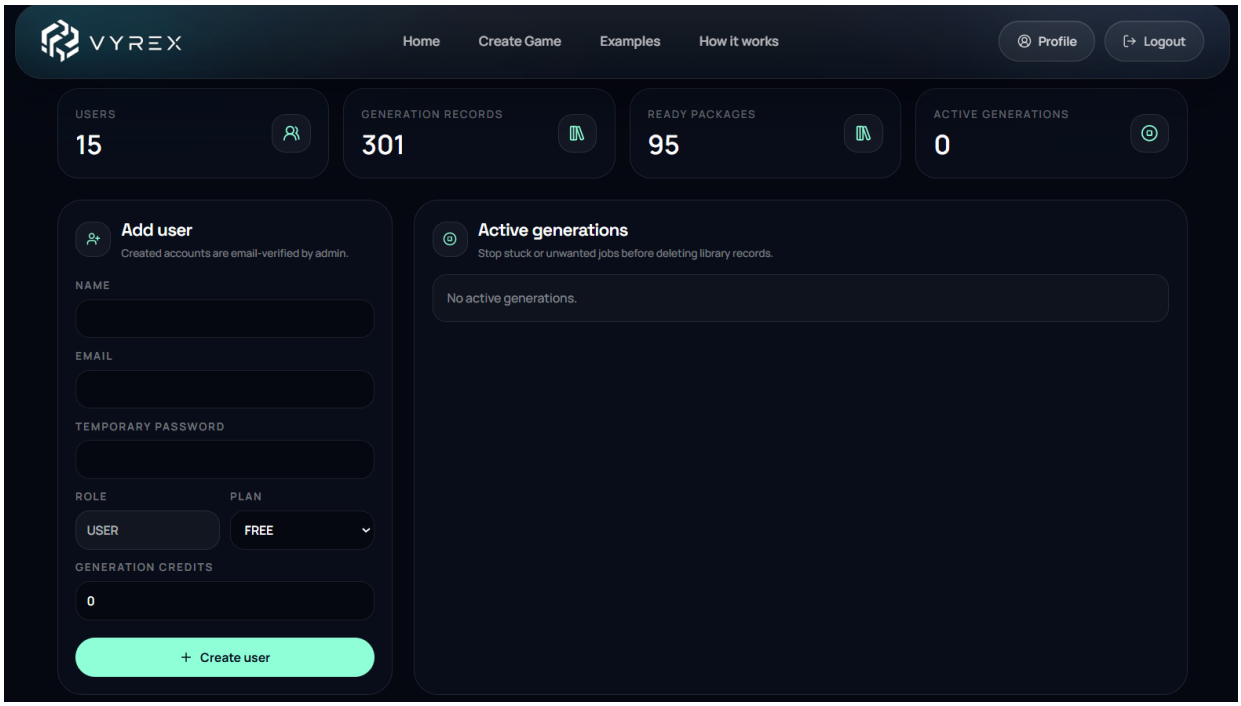

Рисунок 4.39 – Захищена сторінка адміністратора

Присутні функції додавання та видалення користувачів (рис. 4.40). Адміністратор може бачити план кожного користувача, кількість його ігор та дату створення облікового запису. Сам адміністратор також відображається у списку, оскільки він має обліковий запис у вебсервісі.

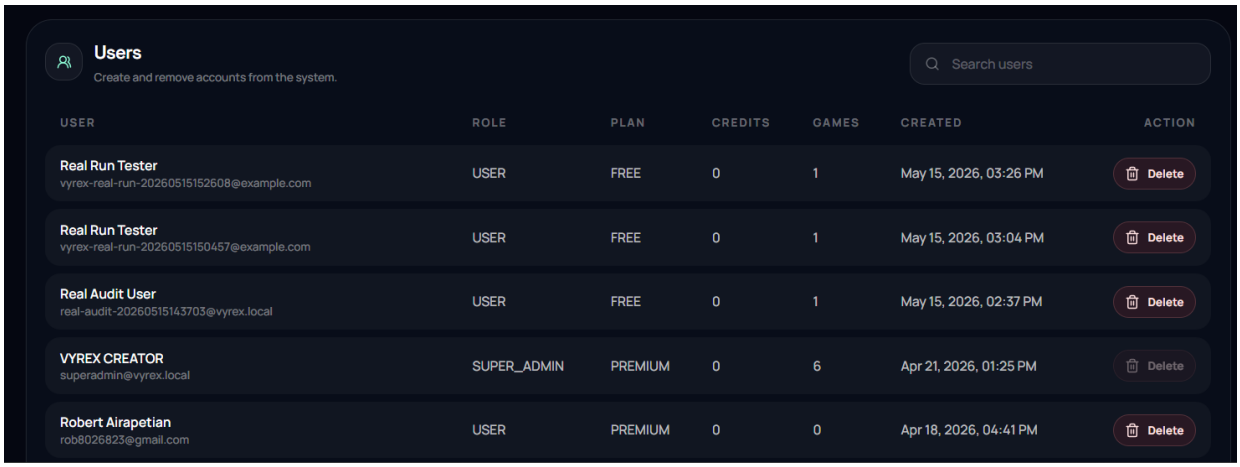

Рисунок 4.40 – Управління користувачами

Окремо адміністратор бачить архів, який можна відсортувати за статусом гри (рис. 4.41). Він може побачити автора гри, дату та час створення та жанр гри. Деталі не розкриваються адміністратору через приватність.

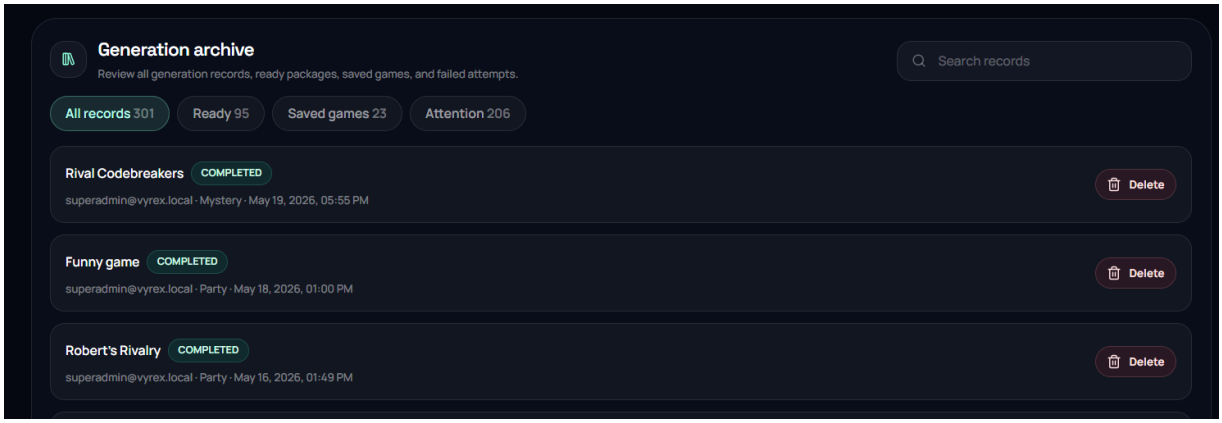

Рисунок 4.41 – Управління архівом ігор

Отже, інтерфейс користувача та адміністратора реалізовано зручним, адаптивним та зрозумілим для роботи з ним. Враховані всі вимоги системи та підходи до проєктування вебсервісу. Інтерфейс дає зрозумілі підказки та відокремлює все на логічні частини, які допомагають взаємодіяти користувачу з системою.

4.6 Тестування роботи вебсервісу

Unit-тестування серверної частини стало способом перевірки критичних сценаріїв роботи вебсервісу (рис. 4.42). Акцент зроблено лише на тих модулях, які зосереджуються на користувацьких даних, запуску генерації, роботі з чергою, повторній генерації та контролі доступу. Для запуску тестів використано Jest, який налаштований у backend-частині проєкту. Під час тестування перевірялись не всі файли проєкту, а лише ті, що є критичними частинами серверної логіки. Наприклад, доступність API, безпека пароля та інші сценарії.

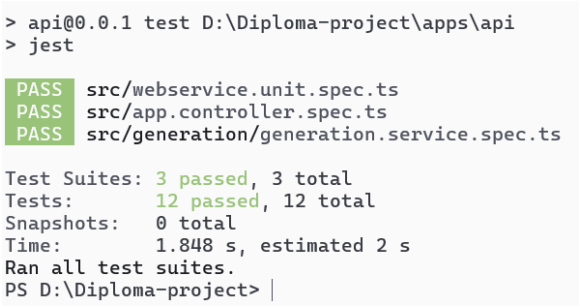

Рисунок 4.42 – Результат запуску unit-тестів серверної частини

У результаті запуску було виконано 3 набори тестів. Усі вони завершились успішно. Загалом пройдено 12 тестових сценаріїв без помилок. Узагальнення виконаних перевірок наведено в таблиці 4.1.

Таблиця 4.1 – Результати unit-тестування серверної частини

Група перевірки	**Що перевірено**	**Результат**
Доступність API	API-метод перевірки стану повертає коректну відповідь сервера	Успішно
Валідація користувача	Реєстраційні дані нормалізуються, некоректний логін відхиляється	Успішно
Безпека пароля	Пароль хешується та не зберігається у відкритому доступі	Успішно
Валідація генерації	Коректні параметри приймаються, некоректні відхиляються	Успішно
Платіжні параметри	Приймає тільки підтримувані типи покупки	Успішно
Створення генерації	Створюється запис генерації та задача передається в чергу	Успішно

Кінець таблиці 4.1

Помилка черги	При помилці постановки задачі доступ звільняється, а генерація стає FAILED	Успішно
Повторна генерація	Повторна генерація отримує збережений inputJson	Успішно
Публікація гри	Незавершена генерація не публікується	Успішно
Адміністрування	Користувач не має доступу до функцій адміністратора	Успішно

Сценарій створення генерації потребував особливої уваги. Тест підтвердив те, що сервер правильно зберігає userId у inputJson, а також передає задачу до черги. Це набуває особливої ваги через те, що worker-модуль надалі бере параметри саме з БД і виконує генерацію асинхронно.

Перевірена і поведінка системи при помилці черги, а саме, якщо задачу не вдалося передати, то сервер одразу звільняє доступ, який був зарезервований та переводить генерацію у помилковий стан FAILED. Це важливо, тому що користувач не втрачає доступ, який може бути як безкоштовним, так і платним.

Важливим аспектом є те, що забезпечується розмежування доступу у вебсервісі. Зокрема, у тестах було зроблено перевірку того, що звичайний користувач не може отримати доступ до сценаріїв адміністратора. Також незавершену генерацію неможливо опублікувати в галерею. Додатково науково-практичну цінність вебсервісу підтверджено апробацією та впровадженням результатів кваліфікаційної роботи (додаток Б).

Висновки до розділу 4

У четвертому розділі було розглянуто програмну реалізацію вебсервісу. Описано структуру проєкту, реалізацію клієнтської та серверної частини, а також роботу worker-модуля, який виконує генерацію настільної гри в асинхронному режимі. Розглянуто деякі фрагменти інтерфейсу користувача.

Показано, що система побудована як окремі взаємопов’язані модулі. Frontend відповідає за взаємодію з користувачем. Backend координує запити, доступ і збереження даних. Worker виконує генераційний pipeline. Також було проведено unit-тестування серверної частини. Усі тестові сценарії завершились успішно.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено інтелектуальний вебсервіс для генерації персоналізованих настільних ігор. Призначення вебсервісу полягає у тому, щоб користувач задав параметри гри, яку він хоче створити, та отримав готові цифрові матеріали, придатні для друку та використання групою гравців.

Під час аналізу предметної області було визначено основні складові персоналізованої настільної гри. До них належать тема, механіка, правила, картки, ігрове поле та візуальні матеріали. Встановлено, що якість результату залежить не лише від наявності цих частин, а й від їх узгодженості між собою. Тому генерація гри у вебсервісі розглядається як послідовний процес, у якому кожний наступний матеріал формується на основі вже підготовленої структури.

У рамках роботи було досліджено існуючі програмні рішення та підходи до автоматизації створення ігрового контенту. Через те, що аналіз сучасних програмних рішень показав зосередження цих сервісів на окремих етапах роботи з настільними іграми, було доцільним створити вебсервіс, який об’єднує все в одному процесі. Важливим було забезпечити відповідність між цими матеріалами через те, що результати не мають бути розрізненими для збереження структури гри. На основі цього для вебсервісу було обрано гібридний підхід до генерації. Мовна модель використовується для тематичного наповнення, дифузійна модель — для візуальних матеріалів, а програмна логіка й механічні каркаси забезпечують керованість структури гри.

Було сформовано функціональні та нефункціональні вимоги до системи. У вебсервісі реалізовано реєстрацію, авторизацію та введення параметрів майбутньої гри. Користувач може запустити генерацію і стежити за її ходом. Після завершення він переглядає готову гру, завантажує сформовані матеріали або за потреби запускає нову генерацію з тими самими параметрами. Окрему увагу приділено вимогам до безпеки даних, контролю доступу, стабільності роботи, асинхронного виконання тривалих задач і базової якості результату генерації.

Проєктування вебсервісу визначило сценарії роботи користувача, алгоритм генерації, інформаційні потоки, архітектуру вебсервісу та модель даних. Система побудована у форматі пов’язаних модулів. Клієнтська частина виконує роботу з користувачами. Серверна частина перевіряє запити, керує доступом і створює задачі генерації. Окремий worker-модуль виконує генераційний pipeline у фоновому режимі. Для збереження даних передбачено сутності користувачів, генерацій, етапів виконання, сформованих ігор та пов’язаних матеріалів.

Робочу версію вебсервісу було реалізовано відповідно до спроєктованої структури. Користувач може підготувати параметри гри, запустити генерацію, переглядати хід роботи та відкрити завершений результат у бібліотеці. Основна обробка виконується у worker-модулі, який формує механічну основу, тематичне наповнення, правила, картки та інші елементи гри. Результат зберігається у системі, а користувач отримує PDF-пакет і ZIP-архів матеріалів гри.

Первинна перевірка узгодженості сформованого контенту стала важливим етапом розроблення. Система забезпечує формування правил гри на основі вже визначеної механіки й теми. Картки та поле спираються на ті самі структуровані дані. Для ігрових зображень карток реалізовано перевірку типових проблем та повторну генерацію, коли система отримує некоректний результат. Це дозволяє зменшити кількість очевидних суперечностей, але для повної перевірки гри на практиці потрібне плейтестування з гравцями.

Роботу основної серверної логіки перевірено unit-тестами. Усі дванадцять тестових сценаріїв успішно завершилися і показали керованість вебсервісу. Перевірено валідацію даних, захист пароля, створення генерації, передавання задачі до черги, обробку помилки черги, повторний запуск, публікацію завершеної гри та обмеження адміністративного доступу.

Отже, поставлену мету досягнуто, а визначені завдання виконано у повному обсязі. Розроблений вебсервіс забезпечує повний процес створення персоналізованої настільної гри. Практична цінність результату полягає у тому, що генерація не обмежується текстовою ідеєю, а завершується правилами, картками, ігровим полем і файлами, які можна надалі використовувати для проведення гри.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sengar S. S., Hasan A. B., Kumar S., Carroll F. Generative artificial intelligence: a systematic review and applications. Multimedia Tools and Applications. 2025. Vol. 84. P. 23661–23700. DOI: 10.1007/s11042-024-20016-1.

2. The AI Index 2026 Annual Report. Stanford University, Stanford Institute for Human-Centered Artificial Intelligence. URL: https://hai.stanford.edu/ai-index/2026-ai-index-report (Accessed: 01.05.2026).

3. The AI Index 2025 Annual Report. Stanford University, Stanford Institute for Human-Centered Artificial Intelligence. URL: https://hai.stanford.edu/ai-index/2025-ai-index-report (Accessed: 01.05.2026).

4. Swacha J., Gracel M. Supporting Serious Game Development with Generative Artificial Intelligence: Mapping Solutions to Lifecycle Stages. Applied Sciences. 2025. Vol. 15. Article 11606. DOI: 10.3390/app152111606.

5. Alharthi S. A. Generative AI in Game Design: Enhancing Creativity or Constraining Innovation? Journal of Intelligence. 2025. Vol. 13. Article 60. DOI: 10.3390/jintelligence13060060.

6. Maxim R. I., Arnedo-Moreno J. Identifying Key Principles and Commonalities in Digital Serious Game Design Frameworks: Scoping Review. JMIR Serious Games. 2025. Vol. 13. Article e54075. DOI: 10.2196/54075.

7. Wu Z., Chen Z., Zhu D., Mousas C., Kao D. A Systematic Review of Generative AI on Game Character Creation: Applications, Challenges, and Future Trends. IEEE Transactions on Games. 2025. P. 1–14. DOI: 10.1109/TG.2025.3564869.

8. Chen Y.-C., Jhala A. GameTileNet: A Semantic Dataset for Low-Resolution Game Art in Procedural Content Generation. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. 2025. Vol. 21, № 1. P. 12–21.

9. Component.Studio. URL: https://component.studio/ (Accessed: 01.05.2026).

10. The Game Crafter. URL: https://www.thegamecrafter.com/ (Accessed: 01.05.2026).

11. nanDECK. URL: https://nandeck.com/ (Accessed: 01.05.2026).

12. Tabletop Simulator. URL: https://www.tabletopsimulator.com/ (Accessed: 01.05.2026).

13. pnpm Workspaces. URL: https://pnpm.io/workspaces (Accessed: 01.05.2026).

14. Turborepo Documentation. URL: https://turborepo.dev/docs (Accessed: 01.05.2026).

15. Next.js App Router. URL: https://nextjs.org/docs/app (Accessed: 01.05.2026).

16. React: Describing the UI. URL: https://react.dev/learn/describing-the-ui (Accessed: 01.05.2026).

17. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (Accessed: 01.05.2026).

18. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs/installation/using-postcss (Accessed: 01.05.2026).

19. NestJS Documentation. URL: https://docs.nestjs.com/ (Accessed: 01.05.2026).

20. Stępień K., Skublewska-Paszkowska M. Performance evaluation of REST and GraphQL API approaches in data retrieval scenarios using NestJS. Journal of Computer Sciences Institute. 2025. Vol. 36. P. 350–356.

21. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (Accessed: 01.05.2026).

22. Prisma ORM Documentation. URL: https://www.prisma.io/docs/orm (Accessed: 01.05.2026).

23. Kim S. H., Oh J. Migrating Monolithic Web Applications to Microservice Architectures Considering Dependencies on Databases and Views. Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing. 2025. P. 1702–1711. DOI: 10.1145/3672608.3707939.

24. Redis Documentation. URL: https://redis.io/docs/latest/ (Accessed: 01.05.2026).

25. BullMQ Documentation. URL: https://docs.bullmq.io/ (Accessed: 01.05.2026).

26. Raja M. S. Architecting Data Pipelines for Scalable and Resilient Data Processing Workflows. International Journal of Emerging Research in Engineering and Technology. 2025. Vol. 6, № 1. P. 1–9. DOI: 10.63282/3050-922X/IJERET-V6I1P101.

27. Gurung N., Shrestha S., Chulyadyo R. Scalability in Microservices: A systematic literature review. Journal of Computer Science & Technology. 2025. Vol. 25, № 2. P. 128–143, DOI: 10.24215/16666038.25.e11.

28. Henríquez C., Ramón Valencia J. D., Sánchez-Torres G. Architectural Evolution from Monolithic to Microservices in Scalable Systems: A Case Study of Netflix. Prospectiva. 2025. Vol. 23, № 1. DOI: 10.15665/rp.v23i1.3683.

29. Ollama Documentation. URL: https://docs.ollama.com/ (Accessed: 01.05.2026).

30. ComfyUI Workflow. URL: https://docs.comfy.org/development/core-concepts/workflow (Accessed: 01.05.2026).

31. Docker Documentation. URL: https://docs.docker.com/ (Accessed: 01.05.2026).

32. Khalifa A., Gallotta R., Barthet M., Liapis A., Togelius J., Yannakakis G. N. The Procedural Content Generation Benchmark: An Open-source Testbed for Generative Challenges in Games. FDG ’25 : Proceedings of the International Conference on the Foundations of Digital Games, Graz, Austria, April 15–18, 2025. New York : ACM, 2025. 12 p. DOI: 10.1145/3723498.3723794.

33. Mao X., Yu W., Okawara Y., Zhan X., Yamada K. D., Zielewski M. R. Procedural Content Generation via Generative Artificial Intelligence. Interdisciplinary Information Sciences. 2026. Advance View. P. 1–11. DOI: 10.4036/iis.2026.R.01.

ДОДАТОК А

Фрагменти результату генерації гри у форматах PDF та JSON


```
"debug": {
  "durationBucket": "short",
  "selectedBecause": [
    "genre matched Mystery",
    "players fit 4",
    "duration fit 35",
    "mechanic profile matched hidden"
  ],
  "selectedTemplate": "hidden_pressure",
  "usedFallbackRules": false,
  "usedFallbackTheme": false,
  "usedFallbackPresentation": false
},
"input": {
  "mode": "FROM_SCRATCH",
  "tone": "Lighthearted",
  "genre": "Mystery",
  "notes": "Fast turns and clear examples;",
  "title": "Rival Codebreakers",
  "userId": "cmo8jen1o0000ulc4ma3uch66",
  "templateId": null,
  "playersCount": 4,
  "playerProfiles": [
    {
      "id": "player-1",
      "name": "Robert"
    },
    {
      "id": "player-2",
      "name": "Roma"
    },
    {
      "id": "player-3",
      "name": "Oleg"
    },
    {
      "id": "player-4",
      "name": "Andrei"
    }
  ],
  "durationMinutes": 35
},
```


Рисунок А.1 – Вхідні параметри та службові дані результату генерації


```
"theme": {
  "title": "Rival Codebreakers",
  "world": "A clandestine network of codebreakers competing to decode encrypted messages before their rivals, where every
  "turnName": "Turn",
  "cardStyle": "Foil-backed with embossed symbols representing encrypted data",
  "deckNames": {
    "role-deck": "Role Deck"
  },
  "iconStyle": "Minimalist with subtle shading to indicate hidden information",
  "roundName": "Round",
  "scoreName": "Leverage",
  "tableMood": "Tense, clever, and psychologically charged with the thrill of deception and anticipation.",
  "zoneNames": {
    "forum-1": "Secure Data Halls Forum",
    "public-1": "Signal Relay Station Chamber",
    "public-2": "Decryption Chamber",
    "public-3": "Cipher Syndicate Chamber",
    "public-4": "Signal Cartel Chamber"
  },
  "boardStyle": "Layered with translucent panels to reveal hidden signals and markers",
  "playerRole": "Secretive rivals investing in plans that become visible at the perfect moment to outmaneuver opponents.",
  "templateId": "hidden_pressure",
  "trackNames": {
    "round": "Round"
  },
  "actionNames": {
    "reveal": "Reveal",
    "scheme": "Scheme",
    "signal": "Signal",
    "interrogate": "Interrogate"
  },
  "coreFantasy": "Winning through hidden preparation, timing, and well-timed reveals of concealed strategies.",
  "visualStyle": "Stylized secretive board game art with layered symbolism, masks, signals, and hidden markers.",
  "cardTypeNames": {
    "mask": "mask",
    "ambition": "ambition",
    "judgment": "judgment"
  },
  "resourceNames": {
    "secrecy": "Secrecy",
    "prestige": "Leverage",
    "influence": "Influence"
  },
```


Рисунок А.2 – Фрагмент тематичного опису згенерованої гри

```
"cards": [
  {
    "id": "role-deck-1",
    "title": "Stable Barrier – Secure Data Halls Forum",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-1-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-1-final.svg"
  },
  {
    "id": "role-deck-2",
    "title": "Open Marker – Signal Relay Station Chamber",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-2-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-2-final.svg"
  },
  {
    "id": "role-deck-3",
    "title": "Fixed Directive – Decryption Chamber",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-3-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-3-final.svg"
  },
  {
    "id": "role-deck-4",
    "title": "Bound Ruling – Cipher Syndicate Chamber",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-4-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-4-final.svg"
  },
  {
    "id": "role-deck-5",
    "title": "Silent Signal – Signal Cartel Chamber",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-5-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-5-final.svg"
  },
  {
    "id": "role-deck-6",
    "title": "Ordered Mandate – Secure Data Halls Forum",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-6-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-6-final.svg"
  },
  {
    "id": "role-deck-7",
    "title": "Silent Authority – Signal Relay Station Chamber",
    "deckKey": "role-deck",
    "svgPath": "D:\\Diploma-project\\apps\\worker\\src\\generated\\cards\\1779206152986\\role-deck-7-final.svg",
    "svgRelativePath": "cards/1779206152986/role-deck-7-final.svg"
  },
```


Рисунок А.3 – Фрагмент JSON із шляхами до згенерованих матеріалів гри


```
"skeleton": {
  "rounds": 4,
  "tracks": [
    {
      "key": "round",
      "max": 4,
      "min": 1,
      "baseName": "Round",
      "startsAt": 1,
      "description": "Session pacing track."
    }
  ],
  "actions": [
    {
      "key": "signal",
      "baseName": "Signal",
      "description": "Send a visible move that may be honest or deceptive.",
      "strategicUse": "Shapes table reads and baits reactions."
    },
    {
      "key": "scheme",
      "baseName": "Scheme",
      "description": "Invest in a hidden or delayed tactical payoff.",
      "strategicUse": "Sets up strong reveals later in the game."
    },
    {
      "key": "reveal",
      "baseName": "Reveal",
      "description": "Cash in a hidden plan for direct board impact or scoring.",
      "strategicUse": "Timing is everything; too early is readable, too late may be useless."
    },
    {
      "key": "interrogate",
      "baseName": "Interrogate",
      "description": "Pressure another player's visible position or force a partial disclosure.",
      "strategicUse": "Breaks passive bluff loops."
    }
  ],
  "loseModel": "round_limit",
  "mechanics": {
    "cardDecks": [
      {
        "key": "role-deck",
        "count": 8,
        "label": "Role Deck",
        "purpose": "Provides hidden role identity and endgame tension without replacing the public pressure game.",
        "cardTypes": [
          "ambition",
          "mask",
          "judgment"
        ]
      }
```


Рисунок А.4 – Фрагмент механічного каркасу гри

На рисунку А.5 наведено приклад згенерованого ігрового поля, у якому поєднано маршрутну структуру, тематичні локації, спеціальні STAR-зони та

короткі ігрові підказки для гравців. Поле не має окремої фінішної клітинки, оскільки за правилами гра завершується після четвертого раунду підрахунком набраних зірок. Такий формат демонструє, що результат генерації містить не лише зображення, а й логічно пов’язану ігрову структуру.

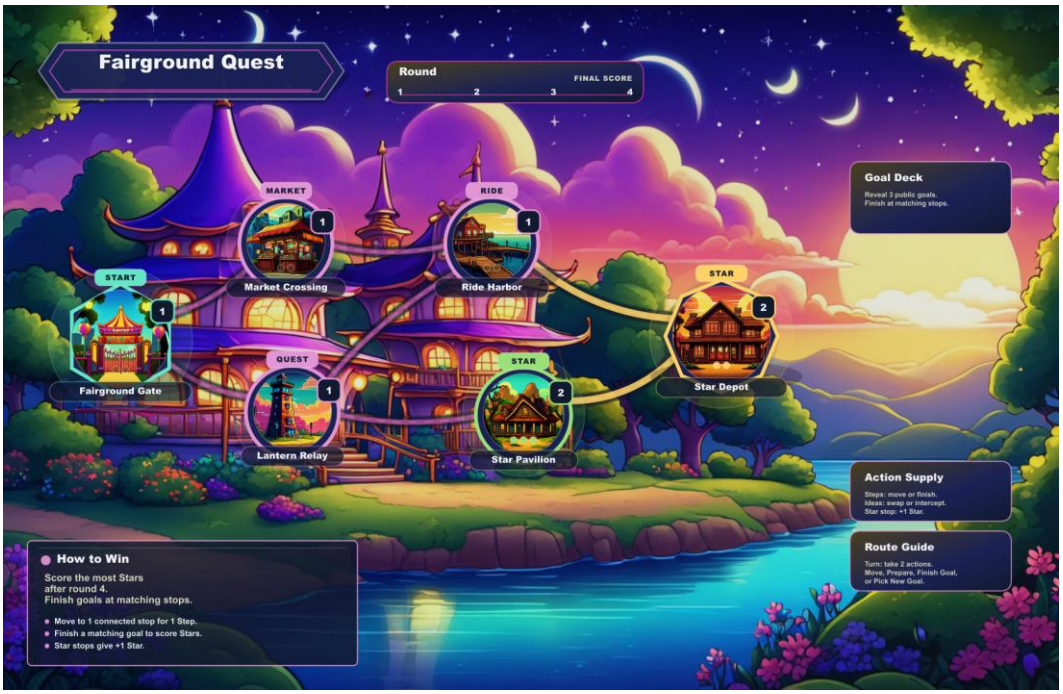

Рисунок А.5 – Приклад сформованого ігрового поля

Рисунок А.6 – Приклад набору ігрових карт

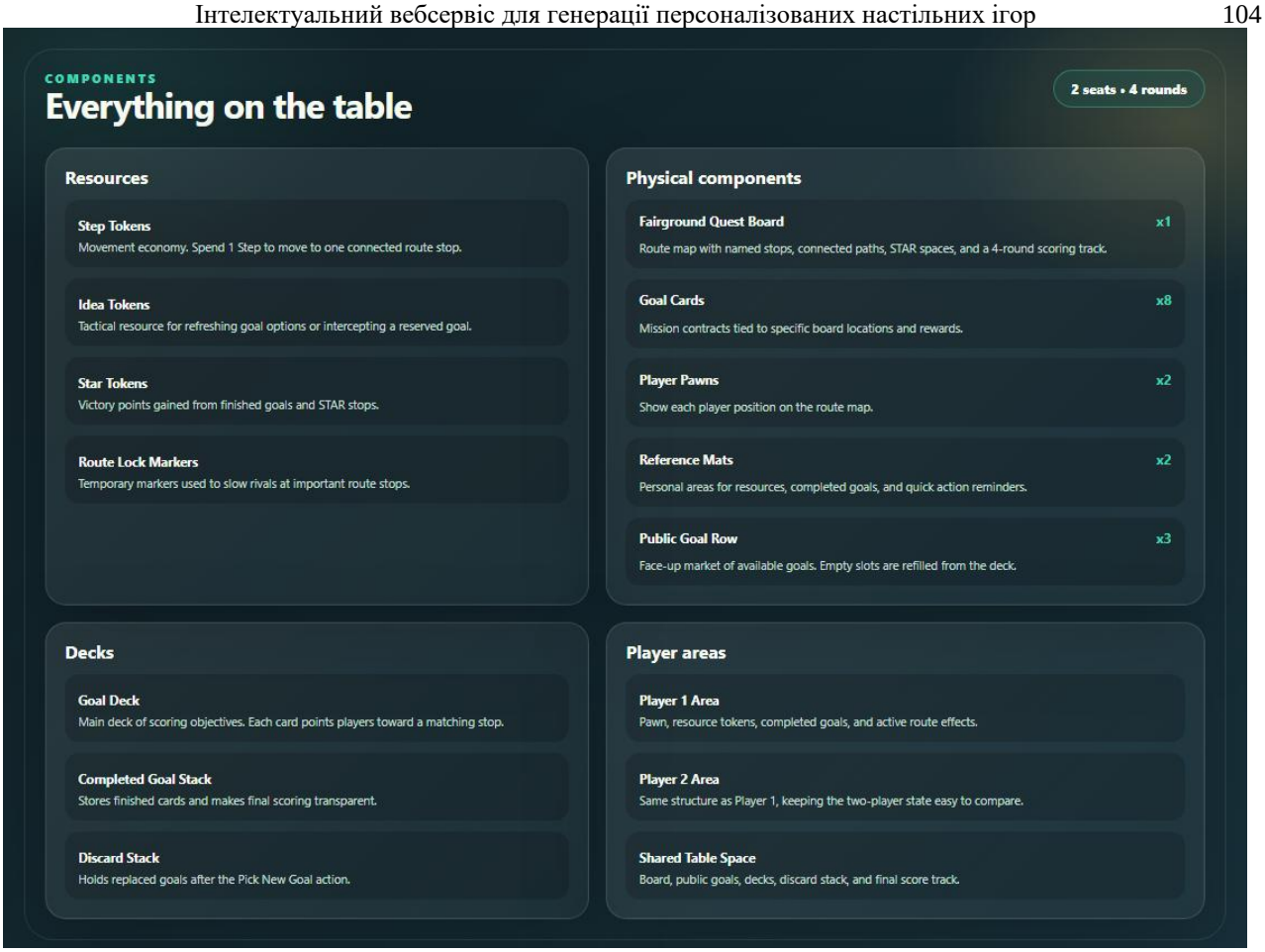

Рисунок А.7 – Фрагмент PDF-пакета зі списком ресурсів, компонентів, колод і зон

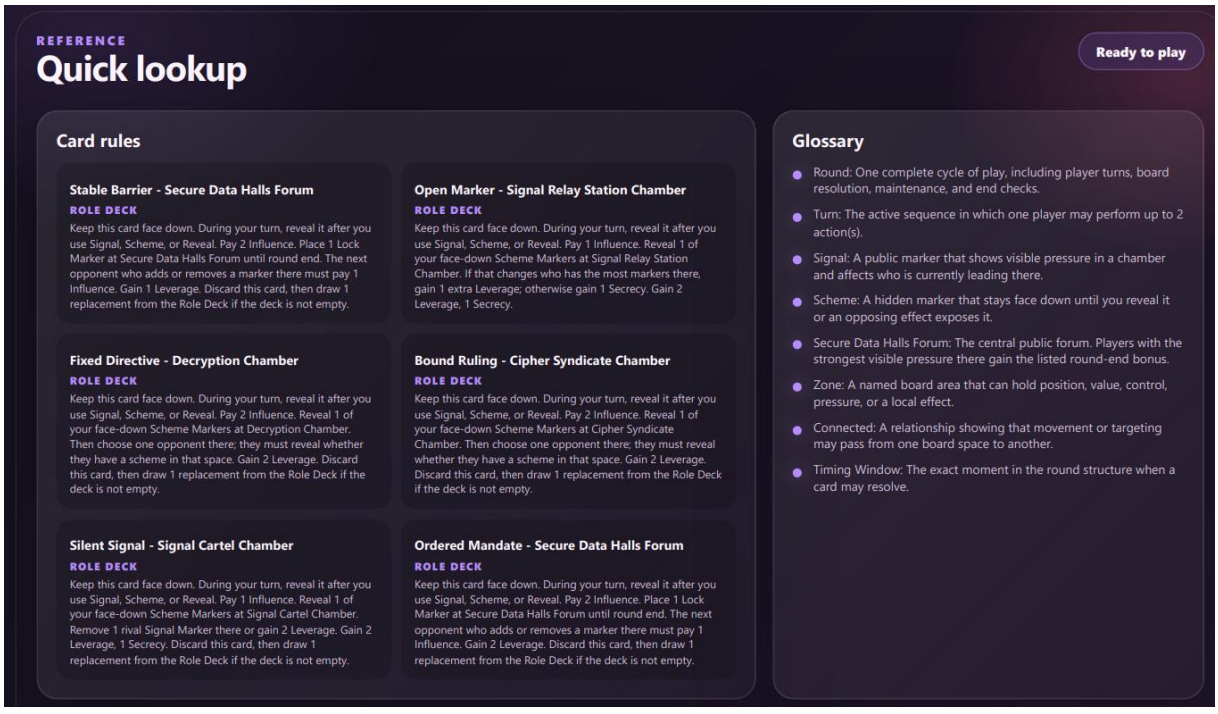

Рисунок А.8 – Приклад сформованих правил карток і довідкових понять гри

ДОДАТОК Б

Впровадження та апробація результатів кваліфікаційної роботи

ДОВІДКА

про впровадження результатів кваліфікаційної роботи

Видана Айрапетяну Роберту Арташесовичу у тому, що результати кваліфікаційної роботи на тему: «Інтелектуальний вебсервіс для генерації персоналізованих настільних ігор» апробовано та впроваджено у діяльність ФОП «Бойчук Р. М.», що здійснює оптове постачання кондитерської продукції.

У процесі апробації розроблений вебсервіс було використано для генерації інтерактивного ігрового контенту, який застосовувався в межах клієнтського сервісу та внутрішніх заходів.

За результатами апробації встановлено, що розроблене програмне забезпечення є функціонально придатним до використання, забезпечує автоматизацію процесу створення інтерактивного ігрового контенту та має практичну цінність.

Довідку видано для подання за місцем вимоги.

ФОП Бойчук Роман

М.П.

Дата «15» травня 2026 р.

Тези доповіді

1. Айрапетян Р., Швед А. Гібридний підхід до генерації ігрового контенту для персоналізованих настільних ігор. Ольвійський форум – 2026: стратегії країн Причорноморського регіону в геополітичному просторі : XXIII Міжнар. наук. конф. 29 черв. – 4 лип. 2026 р., м. Миколаїв : тези / ЧНУ ім. Петра Могили. Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2026.