

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В.о. завідувача кафедри
Артем ВОЛОКИТА

(підпис)

“ ” _____ 2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

**за освітньо-професійною програмою “Інженерія програмного забезпечення
комп’ютерних систем”**

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Інтелектуальна система для генерації користувацьких рекомендацій на
основі аналізу історичних даних

Виконав (-ла): студент (-ка) 4 курсу, групи ІМ-21
(шифр групи)

Чуб Максим Андрійович

(прізвище, ім’я, по батькові)

(підпис)

Керівник асистент, Туганських О. А.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Консультант (нормоконтроль) ст. викл., док. філос. Гончаренко О.О.

(назва розділу)

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

Рецензент доц. кафедри СПСКС, к.т.н., доц. Тарасенко-Клятченко О.В.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Засвідчую, що у цьому дипломному проєкті
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____

(підпис)

Київ – 2026 р.

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

Артем ВОЛОКИТА

(підпис)

“ _ ” _____ 2026 р.

ЗАВДАННЯ

на бакалаврський дипломний проєкт студента

Чуба Максима Андрійовича

1. Тема проєкту Інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних

керівник проєкту Туганських Олександр Антонович, асистент, к.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 3 червня 2026 року № 2055-с

2. Термін здачі студентом закінченого проєкту 2 червня 2026 р.

3. Вихідні дані до проєкту технічна документація. теоретичні дані

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз існуючих підходів до генерації користувацьких рекомендацій на основі аналізу історичних даних

Розділ 2. Розробка інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних

Розділ 3. Прикладні аспекти застосування інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, діаграма класів, алгоритм дій програмного забезпечення
6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Гончаренко О. О.		

7. Дата видачі завдання «19» січня 2026 року

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	<i>Затвердження теми проєкту</i>	<i>10.12.2025-15.12.2025</i>	
2.	<i>Вивчення та аналіз завдання</i>	<i>15.12.2025-15.03.2026</i>	
3.	<i>Розробка архітектури та загальної структури системи</i>	<i>15.03.2026-25.03.2026</i>	
4.	<i>Розробка структур окремих підсистем</i>	<i>25.03.2026-5.04.2026</i>	
5.	<i>Програмна реалізація системи</i>	<i>5.04.2026-15.04.2026</i>	
6.	<i>Оформлення пояснювальної записки</i>	<i>15.04.2026-20.05.2026</i>	
7.	<i>Захист програмного продукту</i>	<i>25.04.2026</i>	
8.	<i>Передзахист</i>	<i>23.05.2026</i>	
9.	<i>Захист</i>	<i>16.06.2026</i>	

Студент-дипломник Максим ЧУБ
(підпис)

Керівник проєкту Олександр ТУГАНСЬКИХ
(підпис)

АНОТАЦІЯ

Дипломний проєкт присвячено створенню інтелектуальної системи генерації користувацьких рекомендацій на основі історичних даних. Метою роботи є підвищення якості та оперативності формування рекомендацій. Об'єктом роботи є процес генерації користувацьких рекомендацій, предметом — методи та програмні засоби їх формування. У роботі проаналізовано підходи до обробки історичних даних, генерації рекомендацій та оцінювання їх якості. Реалізовано прототип системи, що збирає події користувачів, зберігає історію взаємодій, генерує рекомендації та відображає статистику.

Ключові слова: рекомендаційна система, персоналізація, історичні дані, машинне навчання, AWS, Amazon Personalize, DynamoDB, Amazon S3, API, користувацькі рекомендації.

ANNOTATION

The diploma project is devoted to the development of an intelligent system for generating user recommendations based on historical data. The aim of the work is to improve the quality and speed of recommendation generation. The object of the work is the process of generating user recommendations, and the subject is the methods and software tools used for this purpose. The project analyzes approaches to processing historical data, generating recommendations, and evaluating their quality. A system prototype was implemented to collect user events, store interaction history, generate recommendations, and display statistics.

Keywords: recommendation system, personalization, historical data, machine learning, AWS, Amazon Personalize, DynamoDB, Amazon S3, API, user recommendation.

справки	Формат	Значення	Найменування	Кіл. листів	№ екземпляра	Додаток
			Документація загальна			
			Знову розроблена			
	A4	ІАЛЦ.467200.003 ТЗ	Інтелектуальна система для	5		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Технічне завдання			
	A4	ІАЛЦ.467200.003 ПЗ	Інтелектуальна система для	116		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Пояснювальна записка			
	A4	ІАЛЦ.467200.003 Д1	Інтелектуальна система для	1		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Структурна схема			
			системи			
	A4	ІАЛЦ.467200.003 Д2	Інтелектуальна система для	1		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Функціональна схема			
			(діаграма класів)			
	A4	ІАЛЦ.467200.003 Д3	Інтелектуальна система для	1		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Алгоритм дій програмного			
			забезпечення			
	A4	ІАЛЦ.467200.003 Д4	Інтелектуальна система для	42		
			генерації користувацьких рекомендацій			
			на основі аналізу історичних даних			
			Текст програмного коду			

					<i>ІАЛЦ.467200.003 ОА</i>								
<i>Зм</i>	<i>Арк</i>	<i>№ докум.</i>	<i>Підп</i>	<i>Дата</i>									
<i>Розроб.</i>		Чуб М.А.			Інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних Опис альбому			Літ.		Аркуш	Аркушів		
<i>Перев.</i>		Туганських О.А.									1	1	
<i>Реценз.</i>		Тарасенко-Клятченко О.В.						<i>НТУУ «КПІ», ФІОТ, ІМ-21</i>					
<i>Н. Контр.</i>		Гончаренко О. О.											
<i>Затверд.</i>		Волокита А.М.											

ТЕХНІЧНЕ ЗАВДАННЯ ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних»

Київ – 2026

ЗМІСТ

НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ.....	2
ПІДСТАВИ ДЛЯ РОЗРОБКИ.....	2
МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ.....	2
ДЖЕРЕЛА РОЗРОБКИ.....	3
ТЕХНІЧНІ ВИМОГИ.....	3
Вимоги до розробленого продукту.....	3
Вимоги до програмного забезпечення.....	4
Вимоги до апаратної частини.....	4
ЕТАПИ РОЗРОБКИ.....	5

					ІАЛЦ.467200.003 ТЗ			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив	Чуб М. А.				Інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних Технічне завдання	Літ.	Аркуш	Аркушів
Перевірив	Туганських О. А.						1	5
Реценз.	Тарасенко-Кляченко О. В.					НТУУ «КПІ», ФІОТ, ІМ-21		
Н. Контр.	Гончаренко О. О.							
Затвердив	Волокита А. М.							

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних, а також на подальшу підтримку, розширення та вдосконалення розробленої системи.

Областю застосування цієї системи є цифрові сервіси, у яких наявні користувачі, каталог об'єктів та історія взаємодій. Система може використовуватися в інтернет-магазинах, маркетплейсах, освітніх платформах, медіасервісах, сервісах бронювання та інших інформаційних системах, де потрібно формувати персоналізовані рекомендації на основі пошуків, переглядів, кліків, додавання до кошика, покупок та інших дій користувачів.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», яке було затверджене факультетом інформатики та обчислювальної техніки кафедри обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних. Система повинна забезпечувати прийом і зберігання каталогу об'єктів, накопичення поведінкових подій користувачів, формування рекомендацій через API, підтримку резервної логіки рекомендацій та можливість інтеграції з хмарним ML-сервісом Amazon Personalize.

					ІАЛЦ.467200.003 ТЗ	Арк.
						2
Зм.	Арк.	№ докум.	Підпис	Дата		

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки даного дипломного проєкту є офіційна технічна документація AWS, Amazon Personalize, DynamoDB, Amazon S3, AWS Lambda, API Gateway, AWS CDK, документація використаних програмних засобів, науково-технічна література, публікації та статті в мережі Інтернет з тематики рекомендаційних систем, обробки історичних даних та хмарної serverless-архітектури.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Надавати API для завантаження та оновлення каталогу об'єктів.
- Приймати поведінкові події користувачів: пошук, перегляд, клік, додавання до кошика та покупку.
- Зберігати каталог об'єктів та історію взаємодій користувачів в оперативному сховищі.
- Архівувати сирі події користувачів для подальшої обробки та підготовки датасетів.
- Формувати top-N рекомендації для конкретного користувача на основі його історичних даних.
- Підтримувати інтеграцію з Amazon Personalize для генерації персоналізованих рекомендацій.
- Використовувати fallback-рекомендатор у разі недостатньої кількості даних або недоступності ML-компонента.
- Надавати статистику щодо каталогу, подій, користувачів і рекомендаційної видачі.

					ІАЛЦ.467200.003 ТЗ	Арк.
						3
Зм.	Арк.	№ докум.	Підпис	Дата		

- Мати демонстраційний інтерфейс для перегляду каталогу, користувачів, статистики та рекомендацій.
- Мати зрозумілу програмну документацію для розгортання, налаштування та використання системи.

5.2. Вимоги до програмного забезпечення

ОС Windows, macOS або Linux.

- Visual Studio Code або інше середовище розробки.
- Node.js.
- npm.
- AWS CLI.
- AWS CDK.
- Обліковий запис AWS з доступом до необхідних сервісів.
- Веббраузер для роботи з демонстраційним React dashboard.

5.3. Вимоги до апаратної частини

ЦП не менше ніж Intel® Core™ i3 або аналогічний.

- ROM не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.
- Стабільне підключення до мережі Інтернет для розгортання та роботи з AWS-сервісами.

					ІАЛЦ.467200.003 ТЗ	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	10.12.2025-15.12.2025
Вивчення та аналіз завдання	15.12.2025-15.03.2026
Розробка архітектури та загальної структури системи	15.03.2026-25.03.2026
Розробка структур окремих частин системи	25.03.2026-5.04.2026
Програмна реалізація системи	5.04.2026-15.04.2026
Виправлення помилок	15.04.2026-20.05.2026
Оформлення пояснювальної записки	25.04.2026

					ІАЛЦ.467200.003 ТЗ	Арк.
						5
Зм.	Арк.	№ докум.	Підпис	Дата		

ПОЯСНЮВАЛЬНА ЗАПИСКА ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних»

ЗМІСТ

ВСТУП.....	5
РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ.....	10
1.1. Аналіз існуючих технологічних рішень з пошуку та зберігання історичних даних.....	10
1.2. Аналіз існуючих методологій до генерації та оцінки користувацьких рекомендацій	15
1.3. Формалізація задачі та постановка часткових задач проектування.	27
ВИСНОВОК ДО РОЗДІЛУ	37
РОЗДІЛ 2. РОЗРОБКА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ДЛЯ ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ.....	39
2.1. Розробка програмної компоненти інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних	39
2.1.1. Архітектура інтеграції системи генерації користувацьких рекомендацій.....	39
2.1.2. Інженерія вимог	50
2.2. Архітектурне проектування програмної компоненти.....	67
2.2.1. Розробка програмної компоненти інтелектуальної системи для генерації користувацьких рекомендацій	67
ВИСНОВОК ДО РОЗДІЛУ	82

					ІАЛЦ.467200.003 ПЗ		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив	Чуб М. А.				Інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних Пояснювальна записка	Літ.	Аркуш
Перевірив	Туганських О. А.						1
Реценз.	Тарасенко-Клятенко О. В.						116
Н. Контр.	Гончаренко О. О.					НТУУ «КПІ», ФІОТ, ІМ-21	
Затвердив	Волокита А. М.						

РОЗДІЛ 3. ПРИКЛАДНІ АСПЕКТИ ЗАСТОСУВАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ДЛЯ ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ	85
3.1. Демонстрація практичних можливостей розробленої системи	85
3.2. Результати тестування розробленої системи	92
3.3. Програмна документація на програмне забезпечення	99
ВИСНОВОК ДО РОЗДІЛУ	105
ВИСНОВКИ	108
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	112

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface, програмний інтерфейс застосунку

ARN – Amazon Resource Name, унікальна назва ресурсу в AWS

AWS – Amazon Web Services, платформа хмарних сервісів Amazon

AWS CDK – AWS Cloud Development Kit, інструмент опису хмарної інфраструктури у вигляді програмного коду

AWS CLI – AWS Command Line Interface, інтерфейс командного рядка для роботи з AWS

CF – Collaborative Filtering, колаборативна фільтрація

CSV – Comma-Separated Values, формат табличних даних із розділенням значень комами

CTR – Click-Through Rate, показник клікабельності

ETL – Extract, Transform, Load, процес вилучення, перетворення та завантаження даних

HTTP – HyperText Transfer Protocol, протокол передавання гіпертексту

IAM – Identity and Access Management, сервіс керування доступом в AWS

JSON – JavaScript Object Notation, текстовий формат обміну структурованими даними

kNN – k-Nearest Neighbors, метод k найближчих сусідів

MAP – Mean Average Precision, середня точність за всіма релевантними позиціями

MF – Matrix Factorization, матрична факторизація

ML – Machine Learning, машинне навчання

MVP – Minimum Viable Product, мінімально життєздатний продукт

NDCG – Normalized Discounted Cumulative Gain, нормалізована знижена кумулятивна корисність

NoSQL – Not Only SQL, клас нереляційних баз даних

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		3

OLAP – Online Analytical Processing, аналітична обробка даних у реальному часі

REST – Representational State Transfer, архітектурний стиль побудови API

S3 – Simple Storage Service, об'єктне сховище Amazon

SQL – Structured Query Language, мова структурованих запитів

UI – User Interface, користувацький інтерфейс

URL – Uniform Resource Locator, уніфікований покажчик ресурсу

					ІАЛЦ.467200.003 ПЗ	Арк.
						4
Зм.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Актуальність теми

Цифрові сервіси, що працюють із каталогами товарів, контенту, новин, навчальних матеріалів або інших об'єктів, постійно накопичують дані про поведінку користувачів. До таких даних належать пошукові запити, перегляди сторінок, кліки, додавання об'єктів до кошика, покупки, вподобання та інші взаємодії. Самі по собі ці дані мають обмежену цінність, однак після обробки вони можуть бути використані для визначення інтересів користувача, аналізу популярності об'єктів і формування персоналізованих рекомендацій.

Проблема релевантного підбору об'єктів є важливою для багатьох прикладних цифрових систем. Якщо сервіс має великий обсяг товарів або контенту, користувачу складно самотійно швидко знайти потрібний об'єкт. У такій ситуації рекомендаційний механізм дозволяє скоротити час пошуку, підвищити зручність взаємодії із системою та збільшити ймовірність виконання цільової дії. Для інтернет-магазину такою дією може бути покупка або додавання товару до кошика, для медіасервісу — перегляд матеріалу, для освітньої платформи — вибір навчального курсу або події.

Великі технологічні компанії активно використовують власні рекомендаційні системи, оскільки вони безпосередньо впливають на якість користувацького досвіду. Водночас для малих і середніх цифрових сервісів створення повноцінної рекомендаційної інфраструктури з нуля часто є складним завданням. Воно потребує збору та зберігання історичних даних, вибору методології генерації рекомендацій, інтеграції з прикладною системою, оцінювання якості результатів і забезпечення масштабованості. Тому актуальною є задача створення прикладної системи, яка може бути інтегрована з іншим цифровим середовищем і використовувати історичні дані для генерації користувацьких рекомендацій.

					ІАЛЦ.467200.003 ПЗ	Арк.
						5
Зм.	Арк.	№ докум.	Підпис	Дата		

Актуальність теми полягає в необхідності створення доступного, інтегрованого та масштабованого рішення для генерації користувацьких рекомендацій на основі історичних даних. Таке рішення має забезпечувати збір поведінкових подій, їх зберігання, подальший аналіз і формування рекомендацій без потреби розробляти складну рекомендаційну інфраструктуру з нуля для кожного окремого сервісу.

Мета роботи

Підвищення якості та оперативності генерації користувацьких рекомендацій.

Завдання роботи

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Провести аналіз існуючих технологічних підходів до збору, пошуку та зберігання історичних даних користувачів.
2. Провести аналіз існуючих методологій генерації користувацьких рекомендацій та підходів до оцінювання їх якості.
3. Формалізувати задачу генерації користувацьких рекомендацій на основі історичних даних.
4. Розробити підхід до збору, інтерпретації, зберігання та обробки історичних даних користувачів.
5. Скласти структуру програмної системи та спроектувати архітектуру її інтеграції з прикладним цифровим сервісом.
6. Розробити інтелектуальну систему для генерації користувацьких рекомендацій на основі аналізу історичних даних.
7. Провести перевірку працездатності системи, оцінити якість сформованих рекомендацій та представити отриману статистику у графічному форматі.

Об'єкт і предмет дослідження

Об'єктом є процес генерації користувацьких рекомендацій на основі історичних даних.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		6

Предметом дослідження є методи, підходи та програмні засоби генерації користувацьких рекомендацій на основі аналізу історичних даних користувачів.

Сучасний стан проблеми

На сьогодні існує кілька основних підходів до побудови рекомендаційних систем. Найпростішими є рекомендації на основі популярності, коли користувачу пропонуються найбільш переглянуті, придбані або оцінені об'єкти. Такий підхід легко реалізувати, однак він майже не враховує індивідуальні інтереси конкретного користувача.

Більш складним є контентний підхід, за якого рекомендації формуються на основі характеристик об'єктів: категорії, тегів, опису, ціни, бренду або інших властивостей. Він дозволяє рекомендувати користувачу об'єкти, схожі на ті, з якими він уже взаємодівав. Іншим поширеним підходом є колаборативна фільтрація, яка враховує поведінку багатьох користувачів і дозволяє знаходити приховані закономірності у взаємодіях між користувачами та об'єктами.

У практичних системах часто застосовуються гібридні підходи, які поєднують кілька методів рекомендацій. Також поширення отримали хмарні сервіси машинного навчання, які дозволяють будувати рекомендаційні механізми без повної самостійної реалізації складних ML-алгоритмів. Проте навіть за наявності готових сервісів залишається важливою задачею правильної організації збору історичних даних, їх зберігання, підготовки до обробки та інтеграції рекомендаційного модуля з прикладною системою.

Проблема полягає в тому, що для малих і середніх сервісів впровадження рекомендацій часто є складним через нестачу технічних ресурсів, необхідність роботи з даними, потребу в масштабованій інфраструктурі та відсутність універсального механізму інтеграції. Тому актуальним є створення системи, яка може виконувати функцію окремого рекомендаційного сервісу та підключатися до різних backend-застосунків через зрозумілий програмний інтерфейс.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		7

Технічні характеристики об'єкта розробки

У межах дипломної роботи розробляється інтелектуальна система генерації користувацьких рекомендацій, орієнтована на інтеграцію з цифровими сервісами, що мають каталог об'єктів і фіксують дії користувачів. Система повинна підтримувати роботу з історичними подіями, такими як перегляд, клік, пошук, додавання до кошика та покупка.

Основними технічними характеристиками системи є:

- наявність API для передачі подій користувачів із зовнішнього backend-сервісу;
- зберігання історичних даних про взаємодії користувачів із системою;
- можливість зберігання каталогу об'єктів, для яких формуються рекомендації;
- генерація користувацьких рекомендацій на основі накопиченої історії взаємодій;
- підтримка резервної логіки рекомендацій у випадку недостатньої кількості історичних даних;
- можливість подальшої інтеграції з хмарними сервісами машинного навчання;
- орієнтація на масштабовану хмарну архітектуру;
- можливість тестування основних сценаріїв роботи системи.

Така система розглядається не як окремий кінцевий застосунок для користувача, а як програмна компонента, що може бути підключена до інших сервісів і використовуватися для покращення персоналізації.

Очікувані результати

Очікуваним результатом дипломної роботи є спроектована та реалізована інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних. У результаті виконання роботи має бути отримано

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		8

програмне рішення, яке забезпечує збір подій користувачів, зберігання історії взаємодій, обробку даних і надання рекомендацій через програмний інтерфейс.

Практична цінність роботи полягає в тому, що запропонована система може бути використана як основа для впровадження рекомендаційних механізмів у цифрові сервіси, які не мають ресурсів для самостійної розробки складної інфраструктури персоналізації. Розроблене рішення може бути застосоване в інтернет-магазинах, сервісах бронювання, освітніх платформах, медіасервісах та інших системах, де важливо підвищити релевантність запропонованих користувачу об'єктів.

					ІАЛЦ.467200.003 ПЗ	Арк.
						9
Зм.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ

1.1. Аналіз існуючих технологічних рішень з пошуку та зберігання історичних даних

Для побудови системи генерації користувацьких рекомендацій необхідно організувати зберігання та обробку історичних даних. До таких даних належать попередні дії користувачів у цифровій системі: пошукові запити, перегляди об'єктів, кліки, додавання до обраного, додавання до кошика, покупки та інші взаємодії. Саме історія поведінки користувача є одним з основних джерел інформації для подальшого формування персоналізованих рекомендацій.

Історичні дані мають низку особливостей, які потрібно враховувати під час вибору технологічного рішення. Вони постійно накопичуються, можуть мати різну структуру, містять часову прив'язку та часто використовуються не лише для зберігання, а й для подальшого аналізу. Наприклад, одна подія може описувати перегляд товару, інша — пошуковий запит, третя — покупку. Через це система зберігання повинна бути достатньо гнучкою, масштабованою та придатною для подальшої інтеграції з модулями аналітики або машинного навчання.

У загальному вигляді технологічні рішення для зберігання та пошуку історичних даних можна поділити на кілька основних груп: реляційні бази даних, NoSQL-бази даних, об'єктні сховища, пошукові системи та аналітичні сховища. Основні технологічні рішення для пошуку та зберігання історичних даних наведено в таблиці 1.1.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		10

Таблиця 1.1 — Основні технологічні рішення для пошуку та зберігання історичних даних

Група рішень	Приклади технологій	Основне призначення	Переваги	Недоліки
Реляційні бази даних	PostgreSQL, MySQL, Amazon RDS	Зберігання структурованих даних у таблицях	Підтримка SQL, транзакцій, зв'язків між сутностями	Менш зручні для дуже великих потоків подій без додаткової оптимізації
NoSQL-бази даних	Amazon DynamoDB, MongoDB, Cassandra	Зберігання напівструктурованих даних і подій	Гнучка структура, масштабованість, швидкий доступ за ключем	Менш зручні для складних аналітичних запитів
Об'єктні сховища	Amazon S3, Google Cloud Storage, Azure Blob Storage	Довготривале зберігання файлів, логів, експортів, сирих даних	Низька вартість зберігання, масштабованість, зручність для ML-датасетів	Не призначені для швидкого пошуку окремих записів як база даних
Пошукові системи	Elasticsearch, OpenSearch	Повнотекстовий пошук, фільтрація, пошук за індексами	Швидкий пошук за текстом, тегами, категоріями та параметрами	Потребують окремого індексування та додаткової інфраструктури
Аналітичні сховища	ClickHouse, Amazon Redshift, BigQuery	Обробка великих обсягів історичних даних для звітів та аналітики	Висока швидкість агрегованих запитів	Надмірні для невеликих систем і складніші в адмініструванні

Реляційні бази даних є одним із найпоширеніших способів зберігання структурованих даних. Вони добре підходять для систем, де важливо підтримувати зв'язки між сутностями, наприклад між користувачами, товарами, замовленнями та категоріями. PostgreSQL, зокрема, підтримує не лише класичну табличну модель, а й роботу з JSON-даними через тип jsonb, для якого можуть використовуватися GIN-індекси. Це дозволяє частково працювати з напівструктурованими даними в межах реляційної бази. [1]

Перевагою реляційних баз даних є зрозуміла модель даних, підтримка SQL, транзакційність і зручність побудови складних запитів. Водночас для зберігання великої кількості подій користувачів вони можуть потребувати додаткової оптимізації: індексів, партиціювання, архівування старих записів і налаштування

продуктивності. Тому реляційні бази доцільно використовувати для основних структурованих сутностей системи, але не завжди як єдине сховище великого потоку історичних подій.

NoSQL-бази даних орієнтовані на гнучкішу модель зберігання. Вони можуть працювати з key-value, document або wide-column моделями даних. Наприклад, Amazon DynamoDB є NoSQL-базою даних, яка підтримує key-value та document data models і призначена для високопродуктивних serverless-застосунків, що можуть масштабуватися зі зростанням навантаження. [2]

Для задачі зберігання історичних подій NoSQL-підхід є досить зручним, оскільки різні типи дій користувача можуть мати різний набір полів. Наприклад, подія перегляду може містити ідентифікатор об'єкта, подія пошуку — текст запиту, а подія покупки — ціну або кількість одиниць. У таких випадках гнучка структура запису дозволяє не ускладнювати схему бази даних. Основним недоліком NoSQL-рішень є те, що вони менш зручні для складних аналітичних запитів, характерних для класичних SQL-систем.

Об'єктні сховища використовуються для зберігання великих обсягів даних у вигляді файлів або об'єктів. Прикладом такого рішення є Amazon S3. У контексті data lake архітектури Amazon S3 розглядається як базове сховище, що забезпечує практично необмежене масштабування, високу довговічність даних і можливість зберігати інформацію від гігабайтів до петабайтів. [3]

Для рекомендаційної системи об'єктне сховище є корисним тоді, коли потрібно зберігати сирі історичні дані, екпорти, набори даних для машинного навчання або резервні копії. Наприклад, події користувачів можуть накопичуватися у вигляді JSON або CSV-файлів, після чого використовуватися для навчання рекомендаційної моделі. Основний недолік такого підходу полягає в тому, що об'єктне сховище не є повноцінною базою даних для швидкого пошуку окремих записів. Тому його доцільно поєднувати з іншими рішеннями, які забезпечують оперативний доступ до даних.

Пошукові системи, такі як Elasticsearch або OpenSearch, використовуються у випадках, коли важливим є швидкий пошук за текстом, категоріями, тегами або іншими параметрами. Amazon OpenSearch Service, зокрема, підтримує не лише традиційний пошук, а й vector search, який дозволяє шукати семантично схожий контент на основі embedding-представлень. [4]

Пошукові системи є доцільними для каталогів товарів, статей, документів або інших об'єктів, де користувач активно взаємодіє з пошуком і фільтрами. У рекомендаційній системі вони можуть використовуватися як додатковий компонент для пошуку об'єктів або для побудови схожості між елементами. Проте як основне сховище історичних подій вони не завжди є оптимальним вибором, оскільки потребують окремого індексування та підтримки інфраструктури.

Аналітичні сховища застосовуються для швидкої обробки великих масивів історичних даних. Наприклад, ClickHouse є колонковою SQL-базою даних, орієнтованою на OLAP-сценарії, тобто на аналітичну обробку великих наборів даних. Такі системи добре підходять для побудови звітів, обчислення агрегованих показників, аналізу активності користувачів і дослідження поведінкових закономірностей.

Аналітичні сховища мають високу продуктивність для запитів на кшталт підрахунку переглядів, кліків, конверсій, популярності об'єктів або активності користувачів за періодами. Однак для невеликої або середньої системи на початковому етапі таке рішення може бути надмірним. Воно збільшує складність архітектури, потребує додаткового налаштування та окремої логіки завантаження даних. Порівняльна оцінка придатності технологічних рішень для задачі генерації рекомендацій наведена в таблиці 1.2.

Таблиця 1.2 — Оцінка придатності технологічних рішень для задачі генерації рекомендацій

Група рішень	Придатність для зберігання історії дій	Придатність для пошуку	Придатність для ML-датасетів	Загальна оцінка для задачі дипломної роботи
Реляційні бази даних	Середня	Середня	Середня	Доцільні для структурованих сутностей, але не найкращі для великого потоку подій
NoSQL-бази даних	Висока	Середня	Середня	Добре підходять для оперативного зберігання історичних подій
Об'єктні сховища	Висока	Низька	Висока	Добре підходять для архіву історичних даних і підготовки даних для ML
Пошукові системи	Середня	Висока	Середня	Корисні для розвиненого пошуку, але не обов'язкові як базовий компонент
Аналітичні сховища	Висока	Середня	Висока	Доцільні для великих обсягів аналітики, але можуть бути надмірними для MVP

Найбільш придатними є два підходи: NoSQL-бази даних та об'єктні сховища. NoSQL-база даних дозволяє оперативно зберігати історичні події користувачів і швидко отримувати записи за ключами, наприклад за ідентифікатором користувача або об'єкта. Об'єктне сховище, своєю чергою, підходить для накопичення повної історії подій у вигляді файлів, подальшої обробки цих даних і підготовки наборів для моделей машинного навчання.

Реляційні бази даних також можуть застосовуватися в рекомендаційних системах, особливо для зберігання основних сутностей і зв'язків між ними. Проте для потоку історичних подій, який може швидко зростати, NoSQL-підхід часто є простішим і гнучкішим. Пошукові системи доцільно додавати тоді, коли основним сценарієм є складний текстовий або семантичний пошук. Аналітичні сховища стають актуальними на етапі, коли система накопичує великі обсяги даних і потребує складної звітності.

Отже, аналіз існуючих технологічних рішень показує, що для задачі генерації користувацьких рекомендацій на основі історичних даних найбільш доцільним є поєднання оперативного сховища для швидкого запису подій та довготривалого сховища для накопичення сирих даних. Такий підхід дозволяє зберігати поведінкову історію користувачів, використовувати її для побудови рекомендацій і залишати можливість подальшого масштабування системи без надмірного ускладнення архітектури на початковому етапі.

1.2. Аналіз існуючих методологій до генерації та оцінки користувацьких рекомендацій

Сучасні рекомендаційні системи охоплюють широкий спектр методологій: від простих неперсоналізованих правил на основі популярності до латентно-факторних, графових, сесійних і глибинних моделей. Для задачі генерації персоналізованих рекомендацій на основі історичних даних вирішальним є не «найсучасніший» клас моделі сам по собі, а відповідність методу типу доступних сигналів, обсягу даних, вимогам до затримки відповіді та коректній схемі оцінювання. Узагальнений стан проблематики систематизовано насамперед у працях Recommender Systems Handbook [5] та Recommender Systems: The Textbook [6].

Базова класифікація рекомендаційних методів зазвичай розрізняє неперсоналізовані підходи, контентно-орієнтовані методи, колаборативну фільтрацію, гібридні схеми та новіші моделі, що враховують графову структуру зв'язків або послідовність дій користувача. У практичному сенсі це не просто «список алгоритмів», а різні способи формалізації користувацького інтересу: через властивості об'єкта, через колективну поведінку інших користувачів, через латентні представлення, або через порядок подій у часі. [5]

Найпростішою методологією є рекомендація на основі популярності. Вона не моделює індивідуальні вподобання, а повертає найпопулярніші, трендові або найчастіше взаємодіємі об'єкти. Її переваги — майже нульовий cold-start для нових

користувачів, стабільність і мінімальна затримка відповіді; головний недолік — відсутність персоналізації. Показово, що навіть в індустріальних системах такий сценарій виділяють окремо: в документації Amazon Personalize для цього існує окремий use-case popular items. [7]

Контентно-орієнтовані методи формують рекомендацію на основі ознак самих об'єктів: категорій, тегів, текстових описів, бренду, жанру тощо. Їхня сильна сторона полягає в тому, що вони здатні працювати навіть тоді, коли колективної поведінки ще мало, отже є природним засобом зменшення cold-start для нових елементів каталогу. Водночас вони схильні до ефекту надмірної спеціалізації: система продовжує показувати «схоже на вже відоме», але гірше відкриває неочевидні й серендипні об'єкти. [8]

Колаборативна фільтрація спирається не на опис об'єкта, а на структуру взаємодій «користувач–об'єкт». У memory-based варіанті це зазвичай user-based або item-based kNN: прогноз ґрунтується на найближчих сусідах у просторі користувачів чи об'єктів. Огляд neighborhood methods підкреслює їхню простоту, інтуїтивність, пояснюваність і можливість попереднього обчислення сусідств, що забезпечує низьку затримку при видачі рекомендацій. Разом з тим саме ці методи чутливі до розрідженості матриці взаємодій і до cold-start, коли для нового користувача або нового об'єкта нема достатнього «контексту сусідства». [9]

У model-based колаборативній фільтрації центральне місце посідають латентно-факторні моделі, насамперед матрична факторизація. Канонічна праця Matrix Factorization Techniques for Recommender Systems [10] показала, що такі моделі ефективно описують як користувачів, так і об'єкти через щільні латентні вектори, а також легко розширюються на неявний зворотний зв'язок, часову динаміку та впевненість у спостереженнях. Саме тому факторизаційні підходи вважаються одним з базових стандартів для персоналізації на історичних взаємодіях. [11]

Для сценаріїв, де поряд із взаємодіями доступні численні розріджені ознаки користувача, об'єкта або контексту, важливу роль відіграють factorization machines.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		16

У праці Factorization Machines [12] цей клас подано як загальний предиктор для дуже розріджених даних, який поєднує переваги моделей взаємодій та зберігає лінійну обчислювальну складність. Практично це означає, що FM є доречними тоді, коли потрібно одночасно використовувати ID об'єктів, категоріальні атрибути, текстові або контекстні сигнали, не зводячи задачу лише до матриці user-item. [13]

Окремий напрям становлять graph-based методи. У графовій постановці користувачі, об'єкти, атрибути та навіть сесії подаються як вузли багатовідношеневого графа, а рекомендація виникає як наслідок поширення інформації по явних і неявних зв'язках. Огляд Graph Learning based Recommender Systems: A Review [14] підкреслює, що такі моделі здатні краще використовувати високорівневі зв'язки між сутностями та можуть підвищувати точність, надійність і потенційну пояснюваність рекомендацій, але ціною вищої складності побудови й навчання. [14]

Якщо поведінка користувача має виражену часову структуру, особливої ваги набувають session-based і sequence-aware моделі. Для випадків, коли довга персональна історія відсутня або нестабільна, але є короткі сесії, показовою є праця Session-based Recommendations with Recurrent Neural Networks [15], де RNN/GRU-моделі демонструють перевагу над класичними baseline для короткочасного контексту. Коли ж важливий порядок довших послідовностей, застосовують self-attention і transformer-підходи: SASRec [16] балансує між економністю марковських моделей і виразністю RNN, а BERT4Rec [17] використовує двобічне self-attention-представлення історії поведінки. Архітектурним підґрунтям цього напрямку є Transformer з праці Attention Is All You Need [18].

До глибинних методів, що моделюють нелінійні залежності, належать автоенкодера та neural collaborative filtering. У праці AutoRec: Autoencoders Meet Collaborative Filtering [19] автоенкодер безпосередньо реконструює частково спостережувані переваги; у Neural Collaborative Filtering [20] замість скалярного добутку латентних факторів використано нейронну мережу, яка вчиться більш гнучкій функції взаємодії користувача та об'єкта. Ембедінги в цих моделях є не

окремою методологією, а універсальним способом щільного кодування користувачів, об'єктів і контексту; саме вони становлять спільне репрезентаційне ядро для матричної факторизації, FM, NCF і сучасних послідовнісних моделей.[21]

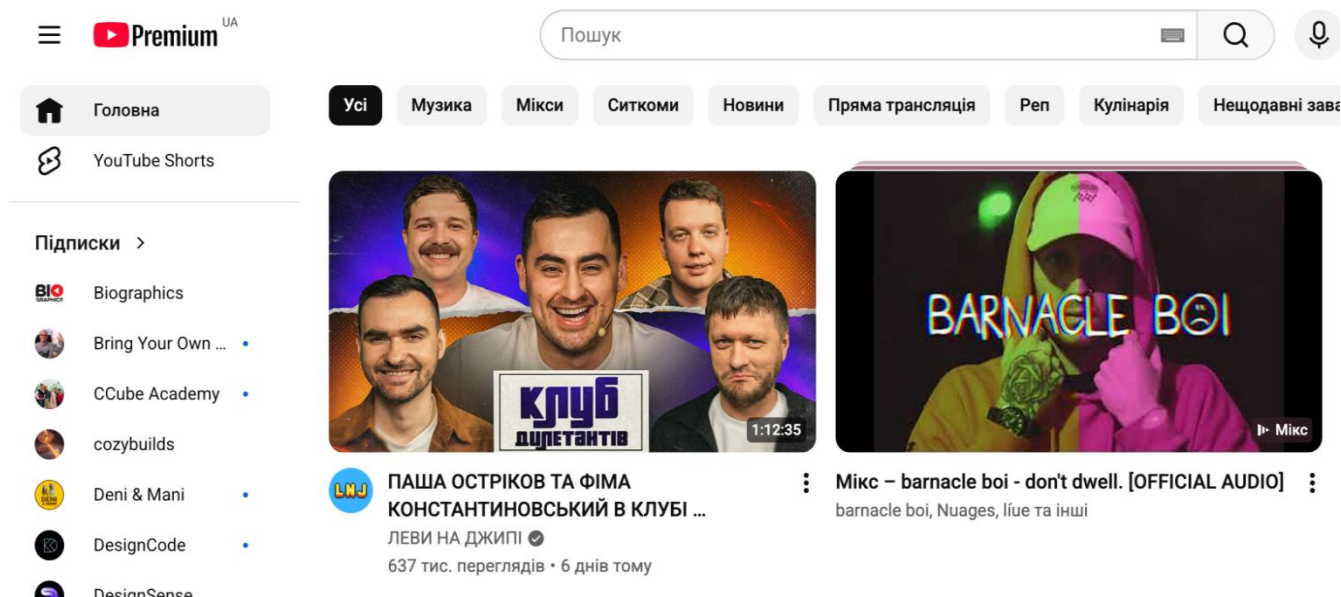


Рисунок 1.1 – Скріншот системи рекомендацій YouTube

На прикладі YouTube видно використання рекомендаційної системи в медіасервісі. Платформа аналізує перегляди, пошукові запити, кліки, тривалість взаємодії з відео та інші поведінкові сигнали, після чого формує персоналізований список відео для користувача. Такий підхід дозволяє скоротити час пошуку релевантного контенту та підвищити залученість користувача.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		18

Рекомендації на основі ваших переглядів



Ігрова приставка Sony PlayStation 5 Slim...

★★★★★ 142

27 999 ₴

Безкоштовна доставка

SMART



Ігрова приставка Sony PlayStation 5 Slim...

★★★★★ 142

31 999 ₴

Безкоштовна доставка

SMART



Смарт-кільце INSPIRE HSR02 розмір 9 Gold...

★★★★★ 5

2-099 ₴

1 629 ₴

Безкоштовна доставка

SMART



Кукурудза для попкорну Popcorn Life...

★★★★★ 20

159 ₴

134 ₴

Безкоштовна доставка

SMART

Рисунок 1.2 – Скріншот системи рекомендацій Rozetka

На прикладі Rozetka рекомендаційний механізм застосовується в інтернет-магазині. Система може враховувати переглянуті товари, категорії, пошукові запити, додавання до кошика, покупки, популярність товарів та схожість між об’єктами каталогу. Рекомендації допомагають користувачу швидше знаходити потрібні товари, а сервісу — підвищувати ймовірність цільової дії, зокрема покупки.

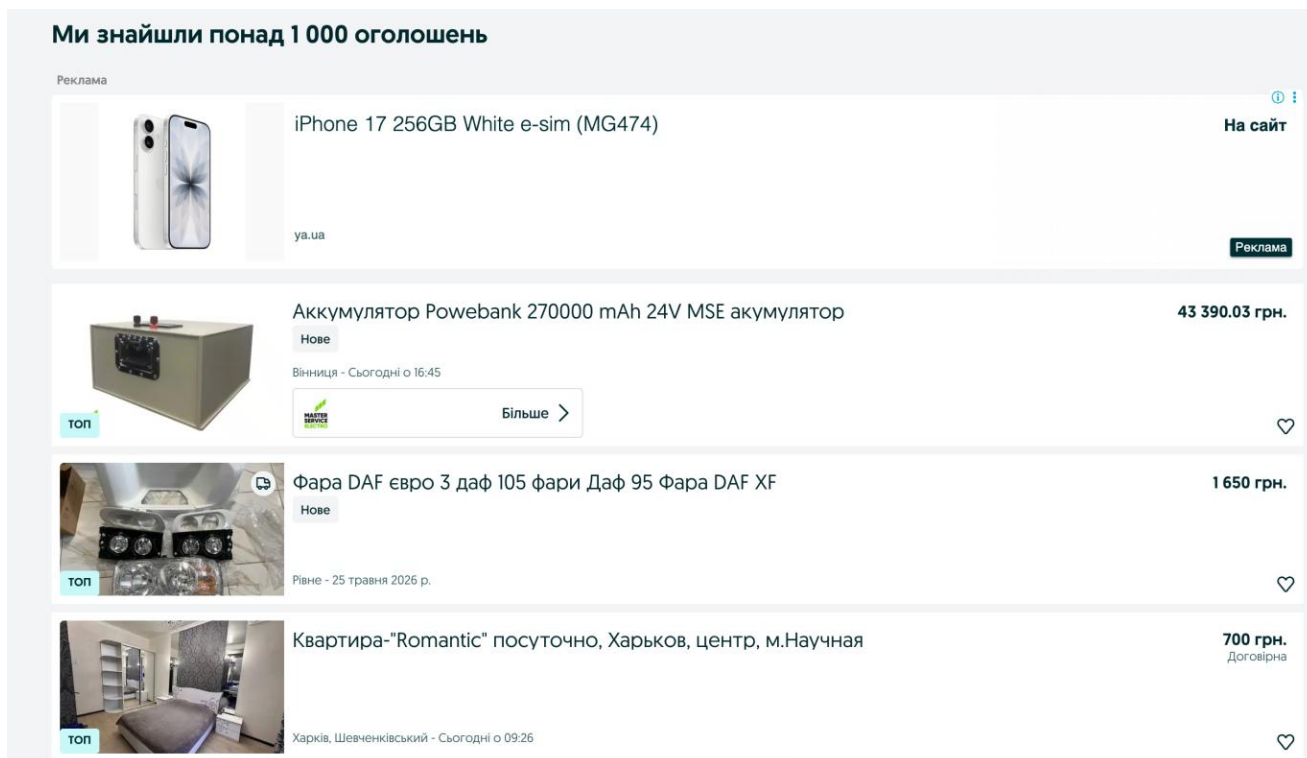


Рисунок 1.3 – Скріншот системи рекомендацій OLX

На прикладі OLX рекомендації використовуються в маркетплейсі оголошень. Система може пропонувати схожі або релевантні оголошення на основі попередніх переглядів, пошукових запитів, категорій, місцезнаходження та інших параметрів. Це особливо важливо для сервісів з великим обсягом різномірних об'єктів, де користувачу складно самостійно переглянути всі доступні варіанти.

Усі наведені приклади об'єднують проблеми, розглянуті в дипломній роботі: необхідність збору історичних даних, збереження поведінкових подій, обробки неявного зворотного зв'язку, формування top-N рекомендацій, роботи з великим каталогом об'єктів, подолання cold-start, забезпечення швидкої відповіді API та масштабованості системи. Незалежно від предметної області — відеоконтент, товари інтернет-магазину або оголошення маркетплейсу — рекомендаційний механізм виконує спільну функцію: аналізує попередні дії користувача й допомагає підібрати найбільш релевантні об'єкти.

Порівняльне узагальнення основних підходів наведено в таблиці; воно синтезує висновки класичних підручників, оглядів neighborhood/graph methods і

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		20

оригінальних праць про MF, FM, RNN, NCF та transformer-рекомендацію. [5]
Порівняльну характеристику основних методологій генерації користувацьких рекомендацій наведено в таблиці 1.3.

Таблиця 1.3 – Порівняльна характеристика методологій генерації користувацьких рекомендацій

Методологія	Сильні сторони	Основні обмеження	Типові сценарії застосування
На основі популярності	Дуже швидка, стійка до відсутності історії, сильний baseline	Немає персоналізації, схильність до концентрації на «хітах»	Нові користувачі, старт системи, fallback
Content-based	Добре працює з багатими метаданими, зменшує cold-start для нових items	Overspecialization, слабше використовує колективний сигнал	Каталоги з добре описаними товарами/контентом
Memory-based CF	Проста, пояснювана, низька онлайн-затримка за precompute	Сильна чутливість до sparsity і cold-start	Середні каталоги, потреба в поясненнях
Model-based CF / MF	Висока якість персоналізації, добре працює з implicit feedback	Менша інтерпретованість, потребує навчання моделі	Зріла історія взаємодій, top-N персоналізація
Factorization Machines	Поєднує interactions і side/context features, добра робота з розрідженими ознаками	Потребує якісного feature design і tuning	Рітейл, реклама, контекстна персоналізація
Graph-based	Моделює багатовідношеневі зв'язки, корисний для складних залежностей	Вища складність побудови, більші обчислювальні витрати	Соціальні, knowledge-graph, multi-relational сценарії
Session-based / sequential	Враховує порядок дій, корисний без стабільного профілю	Чутливий до якості часових послідовностей	Навігаційні сесії, next-item recommendation
Deep learning	Моделює нелінійність і складні взаємодії, придатний для мультимодальних сигналів	Високі вимоги до даних, навчання і відтворюваності	Великі дані, багаті фічі, складна персоналізація

Як видно з таблиці, зі зростанням виразності моделі, як правило, зростають і вимоги до історії взаємодій, обчислювальних ресурсів та дисципліни оцінювання, тоді як пояснюваність часто зменшується. Тому методологічний вибір у дипломній роботі повинен виходити не з максимальної алгоритмічної складності, а з

властивостей доступних історичних даних і цільового режиму використання рекомендацій. [22]

Гібридизація виникла як відповідь на те, що жоден окремий клас методів не усуває одночасно sparsity, cold-start, latency та explainability. Класичний огляд Hybrid Recommender Systems: Survey and Experiments [23] показує, що поєднання content-based, collaborative та knowledge-based технік часто дає практично кращий компроміс, ніж «чисті» методи. Burke виокремлює принаймні такі типи гібридизації, як weighted, switching, mixed, cascade, feature augmentation і meta-level; особливо часто в прикладних системах використовують switching або cascade, коли одна модель працює як основна, а друга — як fallback чи reranker.[24]

Методологічно гібридні підходи є найбільш виправданими у трьох ситуаціях. По-перше, коли є багаті item-метадані, але ще недостатньо interaction-історії: тоді content-based або similar-items компенсує слабкість чистого CF. По-друге, коли потрібно одночасно розв'язувати warm-start і cold-start: популярність або content-based працюють як резерв, а collaborative або sequence-aware модель — як основний персоналізатор після накопичення історії. По-третє, коли вимоги до latency жорсткі: простий candidate generation по популярності, схожості або kNN можна поєднати з точнішим reranking-компонентом. Така логіка відображена й в індустріальних документах Amazon Personalize, де окремо виділено user personalization, similar items і popular items як різні recommendation use-cases. [7, 25, 26]

Нижче подано авторську узагальнену схему вибору методології за основними чинниками: типом міток, розміром історії, наявністю cold-start, вимогами до latency та значущістю порядку подій. Вона не замінює емпіричного порівняння, але добре відбиває консенсус літератури щодо первинного відбору кандидатних класів моделей. Узагальнену схему вибору методології генерації користувацьких рекомендацій наведено на рисунку 1.4.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		22

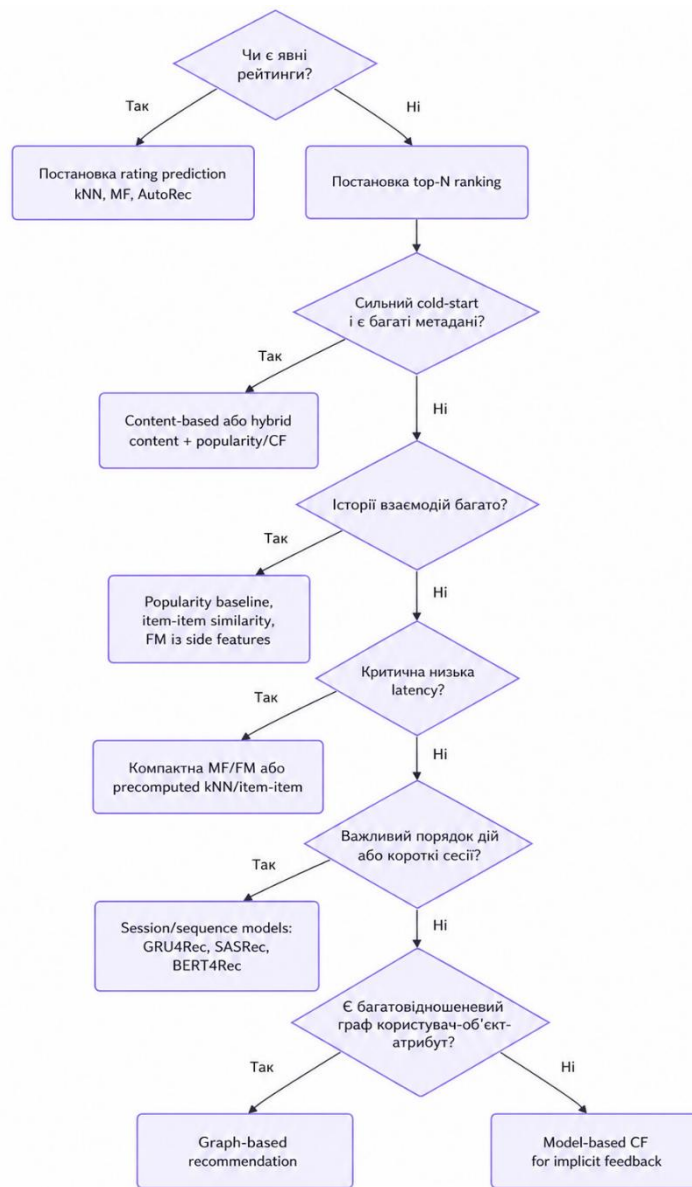


Рисунок 1.4 – Узагальнена схема вибору методології генерації користувацьких рекомендацій

На рисунку наведено узагальнений підхід до вибору методології рекомендаційної системи залежно від типу даних, обсягу історії взаємодій, наявності cold-start та вимог до затримки відповіді.

У прикладних системах центральними обмеженнями є cold-start, вимоги до даних, масштабованість, затримка відповіді та пояснюваність. Для cold-start user/item memory-based CF і матрична факторизація працюють слабше, оскільки їм бракує сигналу взаємодій; натомість content-based, popularity та гібриди дають

кращий старт. Це прямо підкреслюється в огляді neighborhood methods, де cold-start і sparsity названо типовими труднощами, а також в офіційній документації Amazon Personalize, де User-Personalization використовує exploration для нових об'єктів, а Similar-Items може повертати popular items для холодних або невідомих елементів каталогу. [22]

Другий блок чинників пов'язаний з природою міток. Якщо доступні лише події типу перегляд, клік, додавання до кошика чи покупка, то мова йде про implicit feedback, а не про явні рейтинги. Для таких даних відсутність взаємодії не еквівалентна негативній оцінці; тому методології, спеціально пристосовані до implicit feedback, мають перевагу над підходами, що оптимізують прогноз числового рейтингу. Класичною в цьому контексті є праця Collaborative Filtering for Implicit Feedback Datasets [27], де implicit signals трактуються як позитивні спостереження з різними рівнями впевненості. [27]

Масштабованість і latency також зміщують вибір методології. Neighborhood-based системи вигідні тим, що не потребують важкого перенавчання при кожній новій події, а частину обчислень можна винести в offline precompute; тому вони зручні для сервісів із жорсткими вимогами до часу відповіді. Натомість глибинні sequence-моделі та transformer-підходи, хоча часто дають вищу якість на достатньо великих даних, є дорогими щодо навчання і чутливішими до конфігурації. Показово, що систематичне дослідження відтворюваності BERT4Rec зафіксувало істотну залежність результатів від реалізації і тривалості навчання. [22]

Пояснюваність найкраще підтримують popularity-, content-based- та neighborhood-методи, де можна вказати або схожі ознаки об'єкта, або «сусідні» елементи/користувачів, які вплинули на рекомендацію. Латентно-факторні, графові та особливо глибинні нейронні моделі зазвичай менш прозорі, навіть якщо можуть краще вловлювати складні нелінійні залежності. Для прикладної системи це означає, що виграш у якості потрібно співвідносити з вимогами до інтерпретованості та підтримки бізнес-рішень. [22]

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		24

Оцінювання рекомендаційних систем має починатися з правильної постановки задачі. Класичний chapter Evaluating Recommender Systems [28, 29] розрізняє щонайменше три класи задач: прогноз рейтингу, прогноз використання та ранжування елементів. Якщо система має відгадати конкретну числову оцінку, доречні RMSE або MAE; якщо ж її мета — сформувати top-N список із неявних історичних взаємодій, то пріоритет слід надавати ranking-метрикам, а не rating-prediction метрикам. [29]

Для top-N рекомендації найуживанішими є Precision@K, Recall@K, MAP, MRR і NDCG@K. У скороченому вигляді їх можна трактувати так: Precision@K = частка релевантних об'єктів серед перших K; Recall@K = частка всіх релевантних об'єктів користувача, яка потрапила в top-K; MAP усереднює average precision по користувачах; MRR акцентує позицію першого релевантного об'єкта; NDCG@K логарифмічно штрафує релевантні позиції, зсунуті вниз списку. У літературі з оцінювання прямо наголошується, що precision і recall самі по собі не враховують якість внутрішнього порядку серед релевантних об'єктів, тоді як MAP і NDCG є rank-aware. Документація Amazon Personalize додатково дає індустріальне тлумачення MRR, NDCG і Precision@K для рекомендаційних списків обмеженої довжини. [30]

Окремо слід підкреслити, що схема train/test split не є нейтральною технічною деталлю. Дослідження Exploring Data Splitting Strategies for the Evaluation of Recommendation Models [31] показує, що саме стратегія розбиття може істотно змінювати відносний рейтинг алгоритмів. Ще жорсткіше це формулює праця про leakage в offline evaluation: якщо train/test не поважає глобальну часову вісь, модель фактично «підглядає в майбутнє». Тому для історичних рекомендацій випадковий k-fold CV або наївний random split методично слабші за часово-коректні схеми: chronological holdout, leave-last-interaction, leave-last-basket/session чи rolling-window / cross-validation through time. Показово, що й Amazon Personalize в industrial evaluation відділяє пізніші взаємодії для тестування, порівнюючи рекомендації з newest 10% подій у test subset. [32]

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		25

Offline-метрики потрібні, але не достатні. Остаточний критерій якості рекомендації формується в онлайн-середовищі, де стандартом залишається A/B testing. Для швидших і чутливіших pairwise-порівнянь ranking-систем дедалі частіше використовують interleaving: користувачу показують змішаний список результатів двох систем, а дії користувача атрибутуються назад порівнюваним моделям. Amazon Science прямо вказує, що interleaving може бути на порядки чутливішим за класичний A/B test у задачах retrieval/ranking. Разом з тим вибір онлайн-метрики має бути пов'язаний з бізнес-ціллю, а не лише з offline accuracy; counterfactual/off-policy дослідження також показують корисність зв'язку між offline-оцінками та реальним revenue uplift. [33]

Звідси випливає, що поряд з offline ranking-метриками необхідно відстежувати бізнесові KPI: CTR, add-to-cart rate, conversion to purchase, revenue per session або інший доменно релевантний показник. Для систем, де ризик «залипання» на вузькому наборі популярних об'єктів неприйнятний, доцільно також контролювати coverage каталогу, оскільки висока точність за рахунок повторення тих самих елементів не завжди є корисною з погляду товарного відкриття та асортиментної справедливості. [34]

Отже, коректнішою є постановка top-N ranking, а не прогноз числового рейтингу. Через це основними методологіями-кандидатами слід вважати не rating-prediction моделі, а підходи, що працюють з implicit feedback: popularity як обов'язковий baseline; content-based або similar-items як засіб зменшення cold-start; model-based collaborative filtering для персоналізації на накопиченій історії; session/sequence-aware моделі — лише тоді, коли порядок подій є інформативним і дані справді мають часову глибину. За обмеженого обсягу даних або трафіку надмірно складні graph/transformer-моделі не слід вважати методологічно «кращими за замовчуванням» — їхня доречність потрібно доводити емпірично. [27]

Як основні offline-метрики найбільш аргументованими є Recall@10 і NDCG@10, оскільки вони добре відповідають задачі топових рекомендацій на implicit feedback; Precision@10 варто використовувати як додаткову

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		26

характеристику «чистоти» списку, MRR — як індикатор того, наскільки високо піднімається перший релевантний об’єкт, а coverage — як захист від деградації до повторюваних popular-items. Якщо інтерфейс реально показує лише 5 позицій, логічно дублювати вимірювання і на $K=5$; якщо список довший карусельного формату — на $K=10$. Онлайн-перевірка, якщо вона доступна, має спиратися на A/B testing за CTR, add-to-cart rate і purchase conversion, а для швидкого pairwise-відбору близьких моделей може доповнюватися interleaving. [34]

У підсумку, методологічно найбільш обґрунтованим є не пошук «універсально найточнішого» алгоритму, а послідовне порівняння кількох класів методів у єдиній, часово коректній і rank-oriented схемі оцінювання. Саме такий підхід дозволяє коректно пов’язати академічну точність моделі з реальною прикладною цінністю рекомендаційної системи. [29]

1.3. Формалізація задачі та постановка часткових задач проєктування.

Задача побудови інтелектуальної системи рекомендацій формалізується не як абстрактна задача «передбачити будь-яку корисність», а як прикладна задача персоналізованого ранжування об’єктів за історією неявних взаємодій користувача з каталогом. Такий вибір відповідає природі наявних даних у проєкті: система оперує подіями типу search, view, click, add_to_cart, purchase, тобто переважно неявним зворотним зв’язком, де основною ціллю є формування списку top-N рекомендацій у межах обмеженого часу відповіді API, а не прогнозування явної числової оцінки. Класична постановка рекомендаційної задачі через функцію корисності залишається коректною, однак для цього проєкту її доцільно конкретизувати як задачу ранжування. [35]

З огляду на використання хмарної інфраструктури Amazon Web Services, подієвої serverless-архітектури та поєднання оперативного і довготривалого зберігання, формалізація повинна охоплювати не лише математичну модель рекомендацій, а й структуру вхідних даних, критерії якості, нефункціональні вимоги та декомпозицію на інженерні підзадачі. Для цього проєкту ключовими є:

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		27

низька затримка відповіді, стійка робота за розріджених даних, підтримка cold-start сценаріїв, можливість архівування подій у S3 та підготовки датасетів для керованого ML-компонента, зокрема Amazon Personalize як опційного зовнішнього модуля рекомендацій.

У класичній постановці рекомендаційна задача задається множиною користувачів (U), множиною об'єктів (I) та функцією корисності, де для кожного користувача потрібно знайти об'єкт або множину об'єктів з найбільшою оціненою корисністю.

$$u: U \times I \rightarrow \mathbb{R} \quad (1.1)$$

де U — множина користувачів;

I — множина об'єктів;

u — функція корисності.

У загальному вигляді це відповідає вибору

$$i^*(u) = \arg \max u(u, i) \quad (1.2)$$

де $i^*(u)$ — об'єкт з найбільшою оціненою корисністю для користувача;

i — об'єкт із множини об'єктів I ;

$u(u, i)$ — оцінена корисність об'єкта i для користувача u .

Для систем із каталожними об'єктами така постановка є базовою і надалі конкретизується залежно від типу зворотного зв'язку: явного рейтингового або неявного поведінкового. [35]

Для розглядуваного проєкту більш адекватною є постановка через неявний зворотний зв'язок. Нехай (E) — множина подій взаємодії, де кожна подія

$$e = (u, i, a, t, s, q, m) \quad (1.3)$$

де e — подія взаємодії;

u — ідентифікатор користувача;

i — ідентифікатор об'єкта;

a — тип дії;

t — часова мітка;

s — ідентифікатор сесії;

					ІАЛЦ.467200.003 ПЗ	Арк.
						28
Зм.	Арк.	№ докум.	Підпис	Дата		

q — необов'язковий текст запиту;

m — контекстні атрибути.

містить ідентифікатор користувача $u \in U$, об'єкта $i \in I$, тип дії (a), часову мітку (t), ідентифікатор сесії (s), необов'язковий текст запиту (q) та контекстні атрибути (m).

Тоді для пари $((u, i))$ може бути визначений агрегований сигнал інтересу

$$r_{ui}(T) = \sum_{e \in E_{ui} \leq T} w(a_e) \cdot \gamma(T - t_e) \quad (1.4)$$

де $r_{ui}(T)$ — агрегований сигнал інтересу для пари користувач–об'єкт;

$E_{ui} \leq T$ — множина подій між користувачем u та об'єктом i до моменту T ;

$w(a_e)$ — вага типу події;

γ — функція часової деконцентрації;

$T - t_e$ — часова відстань між моментом оцінювання та моментом події.

Практична задача полягає у побудові ранжувальної функції

$$f(u, i | H_u, M_i, C_t) \quad (1.5)$$

де $f(u, i | H_u, M_i, C_t)$ — ранжувальна функція;

H_u — історія користувача;

M_i — метадані об'єкта;

C_t — поточний контекст.

$$\pi N(u, T) = \text{Top}N f(u, i | H_u, M_i, C_t) \quad (1.6)$$

$$i \in I \setminus I_{u \text{ seen}}$$

де $\pi N(u, T)$ — список top- N рекомендацій для користувача u у момент T ;

$\text{Top}N$ — оператор вибору N найрелевантніших об'єктів;

$I_{u \text{ seen}}$ — множина об'єктів, з якими користувач уже взаємодіяв.

Саме така постановка відповідає API рекомендацій у проєкті, де на вхід подається `userId`, а на вихід повертається упорядкований список об'єктів обмеженої довжини.

Для прикладної системи додатково вводяться обмеження. Перше — обмеження затримки:

$$T_{resp}(u, N) \leq L_{max} \quad (1.7)$$

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		29

де $Tresp(u, N)$ — час відповіді сервісу рекомендацій;

L_{max} — гранично допустимий час відповіді.

Друге — cold-start для користувача та об'єкта:

$$|E_u| < \tau_u \text{ або } |E_i| < \tau_i \quad (1.8)$$

де E_u — множина подій користувача;

E_i — множина подій, пов'язаних з об'єктом;

τ_u — мінімальний поріг історії користувача;

τ_i — мінімальний поріг історії об'єкта.

де недостатній обсяг історії не дозволяє стабільно оцінити інтереси або релевантність. Третє — розрідженість даних, яку зручно описувати щільністю матриці спостережень

$$\rho = |\{(u, i): E_{ui} \neq \emptyset\}| / (|U| \cdot |I|) \quad (1.9)$$

де ρ — щільність матриці спостережень;

E_{ui} — множина подій для пари користувач–об'єкт;

U — множина користувачів;

I — множина об'єктів.

причому для реальних рекомендаційних задач зазвичай для реальних рекомендаційних задач зазвичай $\rho \ll 1$. Саме cold-start і sparsity традиційно вважаються одними з головних обмежень рекомендаційних систем, а в рамках цього проєкту вони прямо впливають на необхідність fallback-логіки на основі популярності та метаданих. [35]

Вхідні дані системи доцільно поділити на дві групи: каталог об'єктів та історію подій.

Каталог об'єктів:

- `itemId` — ідентифікатор об'єкта;
- `title` — назва об'єкта;
- `category` — категорія;
- `brand` — бренд;
- `price` — ціна;

					ІАЛЦ.467200.003 ПЗ	Арк.
						30
Зм.	Арк.	№ докум.	Підпис	Дата		

- tags — теги;
- popularity — показник популярності;
- active — ознака активності;
- metadata — додаткові метадані.

Історія подій взаємодії користувача:

- eventId — ідентифікатор події;
- userId — ідентифікатор користувача;
- sessionId — ідентифікатор сесії;
- itemId — ідентифікатор об'єкта;
- eventType — тип події;
- timestamp — часова мітка;
- query — пошуковий запит;
- recommendationId — ідентифікатор рекомендації;
- impression — дані про показ рекомендації;
- properties — додаткові властивості події.

Така структура безпосередньо відповідає подієвій моделі сервісу: для search визначальним є поле query, для view/click/add_to_cart/purchase — itemId, а timestamp задає часовий порядок, необхідний як для персоналізації, так і для коректного офлайнового розбиття на train/test. Для подальшої інтеграції з Amazon Personalize особливо важливими є userId, itemId, timestamp, eventType, sessionId, recommendationId та impression. Офіційна документація Amazon Personalize прямо вимагає для interactions-датасету поля USER_ID, ITEM_ID і TIMESTAMP; EVENT_TYPE та IMPRESSION підтримуються як додаткові службові поля. Крім того, для реєстрації подій у режимі реального часу використовується sessionId, який пов'язує анонімну поведінку з подальшою ідентифікованою історією користувача. [36]

У контексті реалізованого проєкту ці дані зберігаються у двох формах. Оперативний шар представлено таблицями DynamoDB: каталог — за ключем itemId, а події — у вигляді логічного журналу з ключами pk = USER#{userId} та sk

					ІАЛЦ.467200.003 ПЗ	Арк.
						31
Зм.	Арк.	№ докум.	Підпис	Дата		

= $TS\{\text{timestamp}\}\#\{\text{eventId}\}$, що дає змогу швидко отримувати останні дії конкретного користувача. Архівний шар представлено S3, де сирі JSON-події зберігаються за датою та користувачем, а для навчання або завантаження в керований сервіс додатково формуються CSV-набори `items.csv` та `interactions.csv`. Така подвійна форма подання є виправданою: DynamoDB відповідає за низьколатентний операційний доступ, а S3 — за довготривале зберігання, повторну обробку та експорт датасетів.

Для зовнішнього ML-компонента формати слід вважати частиною формальної моделі інтерфейсу системи. Внутрішня подія може містити додаткові поля, але перед навчанням вона має бути нормалізована до мінімально необхідного представлення для `interactions`-датасету. Це також пояснює, чому в поточному проєкті часові мітки зберігаються у подіях у машинозручному форматі, а під час експорту перетворюються до схеми, сумісної з сервісом навчання. Реальний час важливий і на етапі інференсу: офіційно Amazon Personalize підтримує отримання рекомендацій у реальному часі, а нові події для підтримуваних рецептів можуть впливати на релевантність рекомендацій протягом секунд після імпорту. [37]

Оскільки цільова задача має форму `top-N` ранжування, основними метриками якості є ранжувальні, а не регресійні. Нехай G_u — множина релевантних об'єктів для користувача в тестовому вікні, а $\pi_K(u)$ — перші K рекомендацій. Тоді $\text{Precision}@K$ відображає частку релевантних об'єктів у верхній частині списку; $\text{Recall}@K$ — частку покритих релевантних об'єктів; Average Precision усереднює Precision по позиціях, де трапляються релевантні об'єкти; MAP є середнім Average Precision по множині тестових користувачів; $\text{NDCG}@K$ нормалізує накопичений дисконтований виграш і тому чутлива до порядку об'єктів у верхній частині ранжування; MRR фіксує позицію першого релевантного об'єкта; $\text{HitRate}@K$ дорівнює 1, якщо у `top-K` є хоча б один релевантний елемент, і 0 — інакше; AUC інтерпретується як ймовірність того, що модель призначить вищий `score` позитивному прикладу, ніж негативному. Вибір саме цих метрик узгоджується з

класичними роботами з оцінювання рекомендаційних систем та з усталеними метриками інформаційного пошуку. [38]

Ключовими слід вважати Precision@K, Recall@K, NDCG@K, HitRate@K і MRR, оскільки API повертає короткий список рекомендацій, а практична цінність визначається якістю саме верхніх позицій. MAP доцільно використовувати як агреговану офлайнову метрику, коли для кожного користувача доступна множина релевантних майбутніх взаємодій. AUC є корисною допоміжною метрикою для моделей, що навчаються на бінарному або парному сигналі, але для інтерфейсу top-N вона менш інтерпретована, ніж NDCG або HitRate. [38]

Оцінювання системи слід розділяти на offline та online. Офлайнове оцінювання виконується на історичних даних, зазвичай з часовим розбиттям, коли модель навчається на минулих подіях, а тестується на наступному відрізку часу; саме такий підхід є природним для подієвої моделі з часовими мітками. Онлайнове оцінювання проводиться після інтеграції в прикладний сервіс як контрольований експеримент, де порівнюються бізнесово значущі поведінкові показники різних варіантів ранжування. У літературі ці типи оцінювання розглядаються як різні й не взаємозамінні: офлайнові дані дозволяють швидко відсіяти слабкі підходи, а польові А/В-випробування перевіряють реальний вплив на поведінку користувачів. [30]

Для розглядуваної системи нефункціональні вимоги не є другорядними, оскільки саме вони визначають придатність рекомендаційного компонента до використання як окремого сервісу. По-перше, потрібна масштабованість за кількістю користувачів, подій та об'єктів каталогу. По-друге, необхідна низька затримка відповіді на запит рекомендацій. Для обраної архітектурної лінії це обґрунтовується властивостями DynamoDB як операційного сховища: сервіс спроектований для serverless-навантажень, підтримує consistent single-digit millisecond performance та масштабування до таблиць практично довільного розміру. Це робить його придатним для шляху запису подій і читання короткої історії користувача перед побудовою рекомендацій. [39]

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		33

По-третє, система повинна мати дешевий та надійний архівний шар. Тут S3 є доречним не як механізм оперативного пошуку, а як об'єктне сховище для сирих подій, експортованих датасетів та повторної обробки. Офіційно S3 Standard проєктується на 99.999999999% durability і 99.99% availability, що робить його природним вибором для довготривалої історії подій та тренувальних наборів. У формалізованих вимогах це означає розділення системи на online-path і offline-path: перший оптимізується за latency, другий — за durability, replay та вартістю зберігання. [40]

По-четверте, система має дотримуватися вимог безпеки і захисту даних. Оскільки історичні події можуть містити ідентифікатори користувача, сесії, патерни поведінки та контекст, формальна постановка повинна включати мінімізацію даних, цільове обмеження обробки, обмеження строків зберігання, цілісність і конфіденційність. Офіційні роз'яснення Європейська комісія та Європейська рада із захисту даних щодо GDPR прямо вимагають purpose limitation, data minimisation, storage limitation та належних технічних і організаційних заходів безпеки. Для цього проєкту це означає: відокремлення операційного та архівного контурів, шифрування даних під час передавання і зберігання, контроль доступу до таблиць і бакетів, а також політику анонімізації або видалення застарілих подій. [41]

З формальної постановки задачі впливають часткові задачі проєктування, які охоплюють повний життєвий цикл роботи рекомендаційної системи: приймання та валідацію подій, оперативне зберігання каталогу й історії взаємодій, архівування даних, підготовку наборів для навчання, генерацію рекомендацій, резервну логіку та моніторинг якості. Для поточного проєкту доцільно виділити сім таких задач, систематизованих у таблиці 1.4.

Таблиця 1.4 – Часткові задачі проєктування інтелектуальної системи генерації користувацьких рекомендацій

Часткова задача	Відповідальний компонент	Вхід	Вихід	Критерій прийняття
Збір і валідація подій	HTTP API / Lambda обробник подій	JSON-подія від backend-сервісу	Нормалізована подія з eventId, timestamp, валідними обов'язковими полями	подія проходить schema-validation; для search є query, для item-подій — itemId
Оперативне зберігання каталогу та історії	DynamoDB ItemsTable, EventsTable	Каталогові записи і нормалізовані події	Доступні для низьколатентного читання дані	можливе читання каталогу й останніх подій користувача в межах online-path
Архівування та реплієсирих даних	S3 RawEventsBucket	Події у внутрішньому JSON-форматі	Історичний архів подій за датою та користувачем	події не втрачаються після прийому API; можливий подальший експорт і повторна обробка
Підготовка тренувальних датасетів	Export job / batch-скрипти	Дані з DynamoDB і S3	items.csv, interactions.csv у сумісному форматі	файли містять лише коректні записи та відповідають схемі зовнішнього ML-сервісу
Побудова ознак і вибір моделі	Fallback ranker / опційно Amazon Personalize	Історія подій, метадані каталогу, популярність	Навчений або конфігурований ранжувальник	модель або евристика формує коректний top-N для users with history і fallback для cold-start
Сервіс рекомендацій	Lambda обробник /recommendations	userId, limit, контекст запиту	Упорядкований список рекомендацій з score та source	API повертає структуровану top-N відповідь і не блокує основний сервіс
Моніторинг і оцінювання	Логи, debug-метрики, dashboard, A/B-контур	Результати інференсу, історія кліків, офлайнні тестові розбиття	Метрики якості, тривоги, порівняння експериментів	доступні offline-метрики та дані для подальшого online-порівняння ранжувальників

Логіка такої декомпозиції прямо узгоджується з архітектурою проєкту. Збір подій і видача рекомендацій є онлайн-задачами, тому вони тяжіють до serverless-компонентів і DynamoDB. Архівування, підготовка CSV-наборів та можливе тренування зовнішнього керованого модуля — офлайн-задачі, тому вони природно виносяться в S3 і пакетну обробку. Вибір моделі також розщеплюється на два режими: базовий fallback-рекомендатор для cold-start і розрідженого сценарію та

керований ранжувальник Amazon Personalize як опція для накопиченої історії й реального часу. Такий поділ не перевантажує MVP-рішення, але залишає чітку траєкторію масштабування.

Проект можна вважати успішним, якщо формалізована постановка задачі коректно відображається в архітектурі та інтерфейсах системи. Це означає, що сервіс повинен: приймати події взаємодії у стандартизованому форматі; зберігати їх одночасно в оперативному та архівному контурах; повертати персоналізований top-N список для користувача з історією; застосовувати зрозумілу fallback-стратегію за відсутності достатніх даних; підтримувати формування тренувальних наборів для подальшого ML-ранжування; надавати можливість обчислення офлайн-метрик і підготовки до онлайн-експериментів. У межах цього підпункту важливо, що всі ці результати описуються як критерії проектування, а не як уже отримані експериментальні показники.

Очікуваним результатом формалізації є цілісна модель системи, в якій математична постановка рекомендаційної задачі узгоджується з реальною інженерною реалізацією. Для цього проекту такою моделлю є система, де цільова функція задається через персоналізоване top-N ранжування за історією подій, дані організовані у формі каталогу та потоку взаємодій, якість оцінюється передусім ранжувальними метриками, а технічна архітектура розділяє online- та offline-контури. Саме така постановка створює коректну основу для наступного розділу, у якому вже буде розглянуто архітектурне та програмне проектування системи.

ВИСНОВОК ДО РОЗДІЛУ

У першому розділі проведено аналіз існуючих підходів до побудови систем генерації користувацьких рекомендацій на основі історичних даних. У межах розділу розглянуто технологічні рішення для зберігання та пошуку історичних даних, основні методології формування рекомендацій, підходи до їх оцінювання, а також виконано формалізацію задачі та постановку часткових задач проектування. Проведений аналіз підтвердив, що ключовою основою рекомендаційних систем є історія взаємодій користувача з об'єктами системи, а отже, ефективність рекомендацій безпосередньо залежить від повноти, якості та придатності цих даних до подальшої обробки.

У результаті аналізу встановлено, що сучасні рекомендаційні системи найчастіше ґрунтуються на трьох базових класах методів: контентно-орієнтованих, колаборативних та гібридних. Саме така класифікація є загальноприйнятою в профільній науковій літературі, а гібридний підхід розглядається як практично найбільш стійкий, оскільки дозволяє частково компенсувати обмеження окремих методів, зокрема проблему розрідженості даних і холодного старту. [42]

Також з'ясовано, що в прикладних цифрових сервісах доцільним є поетапний підхід, коли базова логіка рекомендацій може працювати на простіших евристичках або правилах, а зі зростанням обсягу накопичених взаємодій система переходить до персоналізованих моделей, зокрема на основі керованих хмарних ML-сервісів, які підтримують реальний час, перенавчання та масштабування (Google Developers Recommendation Systems; Amazon Personalize Documentation). [43]

Окремий висновок розділу стосується оцінювання якості рекомендацій. Аналіз показав, що оцінка рекомендаційної системи не може обмежуватися одним показником точності, оскільки для ранжованих списків важливими є як релевантність, так і порядок подання об'єктів користувачу. Саме тому для офлайн-оцінювання доцільно використовувати метрики класу Precision@K, Recall@K, MAP, MRR та NDCG, а для практичної перевірки корисності системи — також поведінкові та прикладні показники, зокрема CTR, конверсію та залученість

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		37

користувачів (Herlocker et al., 2004; Recommender Systems Evaluation, Springer; Amazon Personalize evaluation guides). [44]

Це означає, що система рекомендацій повинна проектуватися не лише як алгоритм видачі релевантних об'єктів, а як повноцінна програмна компонента, для якої заздалегідь визначено критерії коректності й ефективності.

На основі проведеного аналізу було формалізовано задачу дипломного проєкту як задачу побудови окремої інтелектуальної програмної компоненти, що забезпечує збір історичних подій, зберігання каталогу та історії взаємодій, генерацію персоналізованих рекомендацій, підтримку резервної логіки для випадків недостатності даних, а також можливість оцінювання результатів роботи. Головний висновок першого розділу полягає в тому, що для поставленої теми найбільш доцільним є проектування масштабованої інтегрованої системи, орієнтованої на аналіз неявних поведінкових даних, API-взаємодію із зовнішніми сервісами та можливість поступового розвитку від базових механізмів рекомендацій до більш інтелектуальних моделей. Отримані результати є підставою для переходу до другого розділу, у якому буде розглянуто розробку архітектури системи, інженерію вимог та програмну реалізацію основних компонентів.

РОЗДІЛ 2. РОЗРОБКА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ДЛЯ ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ

2.1. Розробка програмної компоненти інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних

2.1.1. Архітектура інтеграції системи генерації користувацьких рекомендацій

Після аналізу існуючих підходів до зберігання історичних даних, генерації рекомендацій та формалізації задачі необхідно перейти до проєктування архітектури інтеграції системи. У межах даного проєкту система рекомендацій розглядається не як окремий користувацький застосунок, а як самостійна програмна компонента, яку можна підключити до іншого цифрового сервісу. Таким сервісом може бути інтернет-магазин, маркетплейс, освітня платформа, медіасервіс, сервіс бронювання або будь-яка інша система, у якій є користувачі, каталог об'єктів і історія взаємодій.

Вибір інтеграційної архітектури зумовлений тим, що для поставленої задачі важливо не лише реалізувати алгоритм генерації рекомендацій, а й забезпечити створення універсального механізму, здатного функціонувати разом з уже наявним backend-застосунком. У більшості прикладних сервісів уже існує власна бізнес-логіка: реєстрація користувачів, каталог товарів або контенту, замовлення, кошик, платежі, адміністративна панель. Тому рекомендаційний модуль не повинен дублювати ці функції або вимагати повної перебудови основного продукту. Його доцільно винести в окремий сервіс, який отримує від основної системи події користувачів, накопичує історичні дані, формує рекомендації та повертає їх через API.

Основна ідея архітектури полягає в тому, що прикладний backend-сервіс залишається центральною частиною бізнес-системи, а розроблювана інтелектуальна система виконує роль зовнішнього рекомендаційного шару. Такий

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		39

підхід дозволяє розділити відповідальність між компонентами. Основний backend відповідає за користувачів, авторизацію, каталог, замовлення та бізнес-процеси, а система рекомендацій відповідає за прийом поведінкових подій, їх зберігання, підготовку даних для моделі, отримання рекомендацій і надання результату назад у прикладний сервіс.

Узагальнено взаємодію компонентів можна описати так:

Зовнішній сервіс → API рекомендаційної системи → обробники подій і каталогу → оперативне сховище та архів історичних даних → рекомендаційний модуль → API видачі рекомендацій → зовнішній сервіс.

У такій схемі зовнішній сервіс не повинен знати деталі реалізації рекомендаційного алгоритму. Для нього система рекомендацій є окремим API, до якого можна звернутися у двох основних випадках: коли потрібно передати нову подію користувача та коли потрібно отримати список рекомендованих об'єктів для конкретного користувача. Це робить архітектуру придатною для повторного використання в різних доменах.

Для реалізації такої архітектури доцільно використати хмарну serverless-модель. Вона дозволяє не підтримувати постійно активні сервери, а запускати обчислювальні компоненти лише під час надходження запиту. Це важливо для рекомендаційної системи, яка може мати нерівномірне навантаження: у певні періоди кількість подій може бути невеликою, а під час активності користувачів — швидко зростати. Serverless-підхід дозволяє масштабувати окремі обробники запитів незалежно один від одного та зменшує витрати на підтримку інфраструктури.

Основні компоненти архітектури інтеграції рекомендаційної системи наведено в таблиці 2.1.

					ІАЛЦ.467200.003 ПЗ	Арк.
						40
Зм.	Арк.	№ докум.	Підпис	Дата		

Таблиця 2.1 – Основні компоненти архітектури інтеграції рекомендаційної системи

Компонент	Призначення в системі	Причина вибору
HTTP API Gateway	Прийом зовнішніх HTTP-запитів від backend-сервісу або демонстраційного інтерфейсу	Дає єдину точку входу до системи та спрощує інтеграцію через REST API
AWS Lambda	Обробка запитів до каталогу, подій, рекомендацій, статистики та перевірки працездатності	Дозволяє реалізувати незалежні serverless-обробники без підтримки окремого сервера
DynamoDB ItemsTable	Оперативне зберігання каталогу об'єктів	Забезпечує швидкий доступ до даних об'єктів за itemId
DynamoDB EventsTable	Оперативне зберігання історії подій користувачів	Дає змогу швидко отримувати останні дії конкретного користувача
Amazon S3 RawEventsBucket	Архівне зберігання сирих подій і CSV-наборів для навчання	Підходить для довготривалого зберігання, повторної обробки та підготовки ML-датасетів
Amazon Personalize	Керований ML-компонент для генерації персоналізованих рекомендацій	Дозволяє використати готову хмарну інфраструктуру рекомендацій без самостійної реалізації складних ML-моделей
Fallback-рекомендатор	Резервна логіка рекомендацій на основі популярності, категорій, брендів і пошукових слів	Забезпечує роботу API навіть до накопичення достатньої історії або за відсутності ML-компонента
Інфраструктура CDK	Опис та розгортання хмарних ресурсів як коду	Забезпечує повторюваність розгортання і контроль структури системи
Демонстраційний React-інтерфейс	Візуальна перевірка каталогу, користувачів, рекомендацій і метрик	Дає змогу показати практичну роботу системи в межах дипломного проєкту

Першим рівнем архітектури є інтеграційний API. Він повинен забезпечувати набір кінцевих точок, через які зовнішня система може передавати дані та отримувати результат.

Основними endpoint-ами є:

- POST /catalog/items — додавання або оновлення об'єктів каталогу;
- POST /events — передавання події користувача;
- GET /recommendations — отримання рекомендацій для користувача;
- GET /health — перевірка працездатності сервісу;
- GET /catalog/items, GET /catalog/stats, GET /events/stats, GET /users, GET /users/{userId}/summary — допоміжні endpoint-и для демонстрації, аналізу та налагодження.

Наявність окремих endpoint-ів для каталогу, подій і рекомендацій дозволяє розділити життєвий цикл даних. Каталог описує об'єкти, які потенційно можуть бути рекомендовані. Події описують поведінку користувачів. Рекомендаційний endpoint використовує ці дві групи даних для формування персоналізованої відповіді.

Другим рівнем є шар обробки запитів. Кожна група запитів обробляється окремою Lambda-функцією. Такий поділ є доцільним, оскільки різні сценарії мають різне навантаження та різну логіку. Наприклад, обробник подій повинен швидко прийняти дію користувача, перевірити її структуру, записати в оперативне сховище та архів. Обробник рекомендацій, навпаки, повинен отримати історію користувача, звернутися до рекомендаційного модуля або fallback-логіки, доповнити результати даними каталогу та повернути відповідь. Обробник каталогу відповідає за завантаження або оновлення об'єктів, а обробник статистики — за підготовку даних для аналізу та демонстраційної панелі.

Я обрав поділ на окремі Lambda-функції, оскільки він зменшує зв'язаність компонентів. Якщо в майбутньому потрібно буде змінити логіку рекомендацій, це не повинно впливати на прийом подій або оновлення каталогу. Якщо потрібно буде додати нові типи подій, зміни стосуватимуться переважно обробника подій і схеми валідації. Якщо з'явиться потреба в окремій адміністративній аналітиці, її можна буде розширити через окремий обробник статистики.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		42

Третім рівнем є шар зберігання даних. Для нього обрано поєднання оперативного NoSQL-сховища та об'єктного архівного сховища. Оперативний шар представлено двома таблицями DynamoDB. Перша таблиця зберігає каталог об'єктів. Її основним ключем є `itemId`, оскільки саме за цим ідентифікатором система повинна швидко отримувати інформацію про рекомендований об'єкт: назву, категорію, бренд, ціну, теги, показник популярності та статус активності. Друга таблиця зберігає історію подій користувачів. Для неї доцільно використати складений ключ, де `partition key` має вигляд `USER#{userId}`, а `sort key` має вигляд `TS#{timestamp}#{eventId}`. Така структура дозволяє швидко отримувати останні події конкретного користувача у зворотному хронологічному порядку.

Вибір саме такої структури `EventsTable` пояснюється характером запиту рекомендацій. Коли зовнішній сервіс звертається до `endpoint-a /recommendations`, системі часто потрібно отримати не всю історію користувача, а останні взаємодії, які найкраще відображають його поточний інтерес. Наприклад, якщо користувач нещодавно переглядав смартфони певного бренду або шукав конкретну категорію товарів, ці дані мають вплинути на рекомендації. Завдяки ключу `USER#{userId}` система може швидко знайти події саме цього користувача, а завдяки часовому `sort key` — відібрати найновіші записи.

Паралельно з оперативним зберіганням події мають записуватися в Amazon S3 у вигляді сирих JSON-файлів. Це необхідно з кількох причин. По-перше, S3 виконує роль довготривалого архіву історичних даних. По-друге, сирі події можуть бути використані для повторної обробки, якщо в майбутньому зміниться логіка підготовки датасетів. По-третє, саме об'єктне сховище зручно використовувати як проміжний шар для формування CSV-файлів, які можуть бути імпортовані в Amazon Personalize. Отже, DynamoDB відповідає за швидкий online-доступ, а S3 — за offline-контур, архівування та підготовку даних для машинного навчання.

Четвертим рівнем є рекомендаційний модуль. Його доцільно спроектувати у двох режимах роботи: основному та резервному. Основний режим передбачає використання Amazon Personalize. У цьому випадку система передає до керованого

ML-сервісу підготовлені набори даних про об'єкти та взаємодії користувачів, створює dataset group, схеми, датасети, імпортує дані, створює recommender і event tracker. Після цього endpoint /recommendations може звертатися до Amazon Personalize Runtime та отримувати персоналізований список itemId для конкретного userId.

Резервний режим потрібний для випадків, коли ML-компонент ще не налаштований, у системі недостатньо історичних даних або запит до зовнішнього рекомендаційного сервісу завершується помилкою. У такому разі рекомендації формуються за евристичною fallback-логікою. Вона враховує популярність об'єкта, збіг категорії з останніми переглянутими товарами, збіг бренду та частковий збіг із пошуковими словами користувача. Також доцільно виключати з рекомендацій ті об'єкти, з якими користувач уже нещодавно взаємодіяв. Такий підхід не є повноцінною ML-персоналізацією, але він забезпечує працездатність системи на ранньому етапі та вирішує проблему холодного старту.

Особливу увагу потрібно приділити життєвому циклу події. Коли користувач у зовнішньому сервісі виконує дію, наприклад переглядає товар, натискає на рекомендацію, додає товар до кошика або здійснює покупку, backend цього сервісу повинен надіслати подію до endpoint-а POST /events. Подія повинна містити userId, sessionId, eventType, itemId для товарних взаємодій, timestamp, а також додаткові властивості. Для події типу search замість itemId обов'язковим є query, тобто текст пошукового запиту. Така структура дозволяє системі працювати як із діями над конкретними об'єктами, так і з пошуковими намірами користувача.

Після отримання події система виконує її валідацію. Це важливо, оскільки рекомендаційний модуль залежить від якості історичних даних. Якщо в систему потраплятимуть некоректні події, наприклад подія покупки без itemId або пошукова подія без query, це може погіршити якість рекомендацій і ускладнити подальшу підготовку датасетів. Тому на рівні API потрібно перевіряти обов'язкові поля, допустимі типи подій і формат часової мітки. Якщо eventId або timestamp не передано, система може згенерувати їх самостійно.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		44

Після нормалізації подія записується у два місця. В DynamoDB вона зберігається як оперативний запис, доступний для швидкого читання під час побудови рекомендацій. У S3 вона зберігається як сирий JSON-об'єкт, наприклад у структурі за датою та користувачем. Такий подвійний запис підвищує гнучкість системи: online-шлях залишається швидким, а offline-шлях зберігає повну історію для подальшої аналітики та навчання.

Якщо в системі вже налаштований Amazon Personalize Event Tracker, товарні події можуть додатково передаватися в Amazon Personalize через механізм PutEvents. При цьому події типу search доцільно зберігати локально, але не передавати напрямку в Personalize як товарну взаємодію, оскільки вони не завжди мають конкретний itemId. Для item-подій, таких як view, click, add_to_cart і purchase, потрібно виконувати перетворення внутрішніх типів подій у формат, зрозумілий для Personalize: View, Click, AddToCart, Purchase. Якщо користувач взаємодіє саме з елементом, який був показаний у рекомендаційному списку, разом із подією доцільно передавати recommendationId та impression. Це дозволяє точніше пов'язати подальшу дію користувача з конкретною видачею рекомендацій.

Окремим процесом є підготовка даних для навчання рекомендаційної моделі. Для цього система повинна мати batch-скрипти або окремі службові процеси, які зчитують каталог і події з DynamoDB, фільтрують некоректні записи, перетворюють часові мітки у Unix timestamp і формують два основні CSV-файли: items.csv та interactions.csv. Файл items.csv містить інформацію про об'єкти каталогу, зокрема ITEM_ID, PRICE, CATEGORY_L1 і BRAND. Файл interactions.csv містить взаємодії користувачів із полями USER_ID, ITEM_ID, TIMESTAMP і EVENT_TYPE. Після формування ці файли завантажуються в S3, звідки можуть бути імпортовані в Amazon Personalize.

Важливо, що підготовка датасетів відокремлена від основного API. Користувацькі запити не повинні чекати, поки система сформує CSV або імпортує дані в ML-сервіс. Тому архітектура розділяється на online-контур та offline-контур. Online-контур обробляє запити в реальному часі: приймає події, зберігає їх,

повертає рекомендації. Offline-контур виконує важчі операції: експорт даних, імпорт у Personalize, створення recommender-a, оновлення конфігурації та аналіз метрик. Такий поділ робить систему стабільнішою, оскільки довготривалі ML-операції не блокують основний користувацький сценарій.

Процес отримання рекомендацій також складається з кількох етапів. Коли зовнішній backend викликає GET /recommendations із параметрами userId та limit, система спочатку перевіряє коректність запиту. Далі, якщо доступний Amazon Personalize, виконується запит до рекомендаційного сервісу. У відповідь система отримує список itemId та score. Однак для зовнішнього сервісу одного itemId недостатньо, тому результати потрібно доповнити даними з каталогу: назвою, категорією, брендом і ціною. Після цього API повертає структуровану відповідь із полями itemId, score, source, reason, title, category, brand і price. Якщо використовується Personalize, у відповіді також доцільно повертати recommendationId.

Якщо Amazon Personalize недоступний або ще не налаштований, система переходить до fallback-режиму. У цьому випадку вона отримує останні події користувача з EventsTable, завантажує пов'язані з ними об'єкти з ItemsTable, сканує активні об'єкти каталогу та обчислює евристичний score для кожного кандидата. Перевагу отримують об'єкти з високим popularityScore, об'єкти з категорій, які користувач нещодавно переглядав, об'єкти знайомих брендів і товари, назва або теги яких частково відповідають останнім пошуковим словам. Після сортування система повертає перші N об'єктів. Таким чином, навіть без повноцінного ML-компонента користувач отримує не випадковий список, а базово персоналізовану видачу.

У структурі відповіді важливо вказувати джерело рекомендації. Для цього використовується поле source, яке може мати значення personalize або fallback. Такий підхід корисний для налагодження, тестування та подальшого аналізу якості. Якщо рекомендація отримана від Personalize, її можна порівнювати з fallback-видачею. Якщо система часто переходить у fallback-режим, це сигналізує про

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		46

проблеми з ML-компонентом або його конфігурацією. Якщо fallback-рекомендації показують прийнятну якість на малих даних, їх можна залишати як резервну логіку для cold-start сценаріїв.

Для контролю якості та демонстрації роботи системи доцільно передбачити окремий аналітичний шар. Він не є обов'язковим для інтеграції з зовнішнім сервісом, але важливий у межах дипломного проєкту, оскільки дозволяє показати, що система не лише повертає список рекомендацій, а й дає можливість аналізувати дані. До такого шару належать endpoint-и для отримання статистики каталогу, статистики подій, списку користувачів і короткого профілю користувача. Наприклад, можна визначити кількість активних об'єктів, розподіл за категоріями та брендами, кількість подій, кількість унікальних користувачів, останні взаємодії конкретного користувача та категорії, з якими він найчастіше взаємодівав.

Також доцільно передбачити debug-режим для endpoint-а рекомендацій. У цьому режимі API може повертати не лише сам список рекомендацій, а й допоміжну інформацію: кількість останніх подій користувача, розподіл його останніх дій за типами, категорії та бренди нещодавно переглянутих об'єктів, статистику рекомендованих елементів, збіг рекомендованих категорій із останніми інтересами користувача та метрики якості для заданого набору релевантних itemId. Це допомагає перевірити логіку рекомендацій ще до повноцінного онлайн-тестування.

У межах архітектури варто також передбачити демонстраційний інтерфейс. Його завданням є не заміна основного сервісу, а візуалізація роботи рекомендаційної системи. Через такий інтерфейс можна показати каталог товарів, обрати користувача, отримати рекомендації, переглянути стратегію видачі, побачити recommendationId, оцінити метрики та переглянути статистику подій. Це особливо корисно для дипломного проєкту, оскільки дозволяє продемонструвати практичну працездатність backend-архітектури без необхідності інтегрувати її в реальний комерційний продукт.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		47

З погляду безпеки, базова MVP-архітектура може спочатку працювати без складної авторизації, але для продукційного використання потрібно передбачити механізм доступу до API. Оскільки система приймає поведінкові дані користувачів, зовнішній backend повинен мати право передавати події лише від свого застосунку. Для цього в подальшому можна використати API-ключі, JWT-авторизацію, IAM-політики або окремий gateway-рівень перевірки. Також важливо не передавати до рекомендаційної системи надлишкові персональні дані. Для генерації рекомендацій достатньо технічного userId або псевдонімізованого ідентифікатора, sessionId, itemId, типу події та часової мітки. Такий підхід зменшує ризики обробки чутливої інформації.

Інфраструктуру системи доцільно описувати як код за допомогою AWS CDK. Це дозволяє створювати всі основні ресурси однаково при кожному розгортанні: API Gateway, Lambda-функції, DynamoDB-таблиці, S3-бакет, IAM-дозволи, CloudWatch-логи та, за потреби, ресурси Amazon Personalize. Такий підхід зменшує кількість ручних дій, спрощує повторне розгортання системи в іншому середовищі та робить архітектуру прозорою. Для дипломного проєкту це також важливо, оскільки інфраструктура є частиною програмної реалізації, а не зовнішньою неформалізованою конфігурацією.

Узагальнену архітектуру інтеграції системи можна подати як послідовність взаємодій:

1. Адміністратор або зовнішній backend передає об'єкти каталогу через POST /catalog/items.
2. Система перевіряє структуру об'єктів і записує їх до ItemsTable.
3. Користувач у зовнішньому сервісі виконує дію: пошук, перегляд, клік, додавання до кошика або покупку.
4. Backend зовнішнього сервісу надсилає подію до POST /events.
5. Система валідує подію, нормалізує її, записує до EventsTable і додатково архівує у S3.

					ІАЛЦ.467200.003 ПЗ	Арк.
						48
Зм.	Арк.	№ докум.	Підпис	Дата		

6. Якщо налаштовано Amazon Personalize Event Tracker, item-подія додатково передається до Personalize.
7. Offline-процес формує items.csv та interactions.csv і завантажує їх до S3.
8. Amazon Personalize імпортує датасети, створює або оновлює recommender.
9. Зовнішній backend викликає GET /recommendations із userId та limit.
10. Система отримує рекомендації з Personalize або формує fallback-видачу.
11. Результати доповнюються даними каталогу та повертаються у зовнішній сервіс.
12. Зовнішній сервіс показує рекомендації користувачу, а подальші кліки або покупки знову надсилаються як події.

Така архітектура підтримує замкнений цикл персоналізації. Чим більше користувач взаємодіє із системою, тим більше подій накопичується. Чим якісніші події накопичуються, тим точніше можна формувати рекомендації. Якщо користувач взаємодіє з рекомендованими об'єктами, ці дії знову повертаються в систему як нові сигнали. У результаті рекомендаційний модуль поступово покращує розуміння інтересів користувачів.

Перевагою обраної архітектури є її модульність. Кожен компонент має чітко визначену відповідальність. API Gateway приймає запити. Lambda-функції виконують прикладну логіку. DynamoDB зберігає оперативні дані. S3 зберігає архів і датасети. Amazon Personalize відповідає за ML-рекомендації. Fallback-рекомендатор гарантує працездатність у випадках, коли ML ще недоступний. CDK описує інфраструктуру. Демонстраційний інтерфейс дозволяє перевірити роботу системи.

Ще однією перевагою є масштабованість. Якщо кількість подій збільшується, можна масштабувати обробник подій і DynamoDB-таблицю. Якщо збільшується кількість запитів рекомендацій, окремо масштабується Lambda-функція рекомендацій. Якщо накопичується великий історичний архів, він залишається в S3 і не перевантажує оперативну базу. Якщо базової fallback-логіки стає недостатньо, система може перейти до повноцінного ML-рекомендера без зміни

зовнішнього API. Це означає, що зовнішній сервіс може продовжувати використовувати ті самі endpoint-и, навіть якщо внутрішній алгоритм рекомендацій буде змінено.

Обрана структура також зручна для поступової розробки. На першому етапі можна реалізувати прийом каталогу, прийом подій і fallback-рекомендації. Це вже дасть працездатний MVP, який повертає рекомендації на основі популярності, категорій, брендів і пошукових сигналів. На другому етапі можна додати архівування в S3 і формування CSV-датасетів. На третьому етапі — підключити Amazon Personalize, імпорт даних, event tracker і персоналізовані рекомендації. На четвертому етапі — додати аналітику, debug-режим, метрики якості та демонстраційний інтерфейс. Такий поетапний підхід зменшує ризик розробки і дозволяє перевіряти систему після кожного етапу.

Отже, архітектура інтеграції системи генерації користувацьких рекомендацій повинна поєднувати API-рівень, serverless-обробку запитів, оперативне NoSQL-сховище, архівний шар, ML-компонент і fallback-логіку. Я обрав саме таку структуру, оскільки вона відповідає меті дипломного проєкту: створити не ізольований алгоритм, а масштабовану інтелектуальну програмну компоненту, яку можна підключати до різних backend-сервісів. Вона дозволяє збирати історичні дані, зберігати їх у придатній для подальшої обробки формі, формувати рекомендації через API, підтримувати роботу за недостатньої кількості даних і поступово переходити від простих евристик до керованої хмарної ML-моделі.

2.1.2. Інженерія вимог

Після визначення архітектури інтеграції системи необхідно сформулювати вимоги до програмної компоненти. Інженерія вимог у межах цього проєкту потрібна для того, щоб чітко визначити, які функції повинна виконувати система, з якими даними вона має працювати, які обмеження потрібно врахувати під час розробки та за якими критеріями можна буде перевірити коректність реалізації.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		50

Розроблювана система не є класичним кінцевим застосунком, у якому користувач напряму виконує всі дії через графічний інтерфейс. Основним призначенням системи є робота як окремого рекомендаційного сервісу, що інтегрується з іншими backend-застосунками. Тому вимоги до неї мають охоплювати не лише користувацькі сценарії, а й вимоги до API, структури подій, зберігання даних, інтеграції з хмарними сервісами, резервної логіки, масштабованості, безпеки та можливості подальшого розвитку.

Для систематизації вимог у межах проєкту було використано підхід, за якого вони поділяються на кілька категорій: бізнес-вимоги, вимоги зацікавлених сторін, функціональні та нефункціональні вимоги, вимоги до структури даних, інтеграційні вимоги та критерії приймання. Такий поділ дозволяє не змішувати різні рівні опису системи. Наприклад, бізнес-вимога пояснює, навіщо система потрібна, функціональна вимога описує конкретну поведінку програмного забезпечення, а нефункціональна вимога визначає обмеження щодо швидкодії, масштабованості або безпеки.

Головна бізнес-вимога до системи полягає в тому, що вона повинна дати можливість цифровим сервісам отримувати персоналізовані рекомендації без необхідності самостійно розробляти повну ML-інфраструктуру. Для малого або середнього сервісу створення власного рекомендаційного модуля з нуля може бути складним через потребу в збиранні даних, побудові сховища, підготовці датасетів, розгортанні моделей і підтримці інфраструктури. Тому система повинна надати готовий інтеграційний механізм: зовнішній сервіс передає каталог і поведінкові події, а у відповідь отримує рекомендації через API.

Основними зацікавленими сторонами системи є власник або команда зовнішнього цифрового сервісу, backend-розробник, адміністратор системи, кінцевий користувач зовнішнього сервісу та розробник рекомендаційної компоненти. Кожна з цих сторін має різні очікування. Власнику сервісу важливо підвищити релевантність пропозицій і залученість користувачів. Backend-розробнику важливо мати простий і передбачуваний API для інтеграції.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		51

Адміністратору важливо бачити статистику роботи системи. Кінцевому користувачу важливо отримувати доречні об'єкти, які відповідають його інтересам. Розробнику системи важливо мати зрозумілу структуру компонентів, яку можна тестувати, підтримувати й розширювати. Основні зацікавлені сторони системи та їхні очікування наведено в таблиці 2.2.

Таблиця 2.2 – Зацікавлені сторони та їхні очікування від системи

Зацікавлена сторона	Основний інтерес	Очікування від системи
Власник зовнішнього сервісу	Підвищення ефективності взаємодії з користувачами	Рекомендації мають підвищувати релевантність запропонованих об'єктів
Backend-розробник	Швидка інтеграція з наявною системою	API має бути зрозумілим, стабільним і не вимагати зміни основної бізнес-логіки
Адміністратор системи	Контроль стану системи та даних	Потрібні endpoint-и для статистики, перевірки працездатності та аналізу подій
Кінцевий користувач	Отримання корисних рекомендацій	Система має пропонувати об'єкти, пов'язані з його інтересами та діями
Розробник рекомендаційної компоненти	Підтримка і розвиток системи	Архітектура має бути модульною, тестованою і придатною до розширення

Першою групою є функціональні вимоги. Вони описують, які саме дії повинна виконувати система. У межах проєкту основними функціональними можливостями є прийом і зберігання каталогу об'єктів, прийом подій користувачів, архівування історичних даних, генерація рекомендацій, використання fallback-логіки, підготовка даних для Amazon Personalize, отримання статистики та підтримка демонстраційного інтерфейсу.

Каталог об'єктів є базовою сутністю системи, оскільки рекомендації не можуть формуватися без інформації про об'єкти, які потенційно можуть бути запропоновані користувачу. Система повинна дозволяти зовнішньому backend-сервісу передавати об'єкти каталогу через API. Для кожного об'єкта потрібно зберігати itemId, назву, категорію, бренд, ціну, теги, показник популярності, статус

активності та додаткові метадані. Поле `itemId` повинно бути обов'язковим, оскільки воно використовується як основний ідентифікатор об'єкта в каталозі, історичних подіях і рекомендаційних відповідях.

Другою важливою функцією є прийом подій користувачів. Система повинна підтримувати події типу `search`, `view`, `click`, `add_to_cart` і `purchase`. Саме ці події відображають базову поведінку користувача в цифровому сервісі: пошук, перегляд, взаємодію з елементом, намір купівлі та завершену покупку. Для подій, пов'язаних із конкретним об'єктом, обов'язковим має бути `itemId`. Для пошукової події обов'язковим має бути `query`, тобто текст пошукового запиту. Такий поділ потрібний для того, щоб система могла коректно розрізняти об'єктні взаємодії та пошукові наміри.

Кожна подія повинна містити `userId`, `eventType` і `timestamp`. Якщо `timestamp` не передано, система повинна автоматично сформувати часову мітку на момент отримання події. Якщо `eventId` не передано, система повинна згенерувати його самостійно. Такий підхід зменшує вимоги до зовнішнього сервісу й одночасно забезпечує унікальність записів у журналі подій. Додатково подія може містити `sessionId`, `recommendationId`, `impression`, `filters` і `properties`. Ці поля не є обов'язковими для базової роботи, але вони потрібні для точнішої аналітики та подальшого зв'язування рекомендацій із поведінкою користувача.

Після прийому події система повинна виконати її валідацію. Це одна з важливих вимог, оскільки якість рекомендацій прямо залежить від якості вхідних даних. Якщо система буде приймати неповні або некоректні події, у подальшому це може призвести до помилок у підготовці датасетів, некоректного профілю користувача та зниження релевантності рекомендацій. Тому API має повертати зрозумілу помилку, якщо подія не відповідає очікуваній структурі: наприклад, якщо для `search` відсутнє поле `query`, якщо для `view` або `purchase` відсутнє поле `itemId`, якщо `eventType` має недопустиме значення або якщо `timestamp` передано в некоректному форматі.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		53

Третя функціональна вимога стосується зберігання подій. Система повинна записувати події в оперативне сховище, придатне для швидкого доступу під час формування рекомендацій. Крім того, кожна подія повинна зберігатися в архівному сховищі у сирому JSON-форматі. Я обрав таку вимогу, оскільки оперативне сховище потрібне для швидкої роботи API, а архівне сховище — для повторної обробки, аналітики та підготовки навчальних наборів. Таким чином, система не втрачає історію навіть тоді, коли змінюється логіка подальшої обробки.

Четверта функціональна вимога — генерація рекомендацій. Система повинна надавати endpoint, через який зовнішній backend може отримати top-N рекомендацій для конкретного userId. На вхід передається userId і бажана кількість рекомендацій limit. На вихід система повинна повернути список об'єктів з itemId, score, source, reason, title, category, brand і price. Така відповідь є зручною для інтеграції, оскільки зовнішній сервіс отримує не лише ідентифікатори, а й основні дані, потрібні для відображення рекомендацій користувачу.

Важливою вимогою є підтримка двох стратегій рекомендацій: personalize і fallback. Стратегія personalize використовується тоді, коли налаштовано Amazon Personalize і система може отримати рекомендації від керованого ML-сервісу. Стратегія fallback використовується тоді, коли Personalize ще не налаштований, недоступний або не має достатньо даних. У fallback-режимі рекомендації мають формуватися на основі популярності об'єктів, категорій і брендів, з якими користувач нещодавно взаємодіяв, а також на основі пошукових слів. Така вимога є критичною для MVP, оскільки система повинна залишатися працездатною навіть на ранніх етапах, коли історичних даних ще недостатньо.

П'ята функціональна вимога стосується підготовки даних для машинного навчання. Система повинна мати можливість формувати датасети у форматі, придатному для імпорту в Amazon Personalize. Для цього потрібно перетворювати внутрішні записи каталогу та подій у файли items.csv та interactions.csv. У interactions.csv повинні потрапляти лише ті записи, які містять userId, itemId, timestamp і eventType. Пошукові події можуть зберігатися локально, але не завжди

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		54

можуть бути напряду використані як взаємодії з об'єктами, якщо в них немає itemId. Така вимога дозволяє відокремити збір сирих даних від підготовки структурованого ML-набору.

Шоста функціональна вимога — підтримка статистики та діагностики. Система повинна надавати endpoint-и для отримання кількості об'єктів каталогу, кількості подій, розподілу за типами подій, списку користувачів і короткого профілю активності конкретного користувача. Такі endpoint-и потрібні не тільки для демонстрації в дипломному проєкті, а й для практичного контролю стану даних. Наприклад, якщо в системі мало подій або певний тип події взагалі не надходить, це можна швидко побачити через статистику.

Окремо потрібно передбачити debug-режим для рекомендацій. У цьому режимі система повинна повертати додаткову інформацію про те, на основі яких даних було сформовано рекомендації: кількість останніх подій користувача, популярні категорії та бренди в його історії, статистику рекомендованих об'єктів і, за наявності ground truth, метрики якості. Це полегшує тестування та пояснення роботи системи, особливо на етапі розробки. Основні функціональні вимоги до системи наведено в таблиці 2.3.

Таблиця 2.3 – Функціональні вимоги до системи

Код вимоги	Назва вимоги	Опис вимоги	Пріоритет
FR-01	Прийом каталогу об'єктів	Система повинна приймати через API один або кілька об'єктів каталогу та зберігати їх в оперативному сховищі	Високий
FR-02	Валідація каталогу	Система повинна перевіряти наявність обов'язкових полів itemId, title і category	Високий

Продовження таблиці 2.3

Код вимоги	Назва вимоги	Опис вимоги	Пріоритет
FR-03	Прийом подій користувача	Система повинна приймати події search, view, click, add_to_cart і purchase	Високий
FR-04	Валідація подій	Система повинна перевіряти коректність eventType, userId, query для search і itemId для об'єктних подій	Високий
FR-05	Генерація eventId і timestamp	Якщо eventId або timestamp не передано, система повинна сформувати їх автоматично	Середній
FR-06	Оперативне зберігання подій	Система повинна записувати нормалізовані події в оперативне сховище	Високий
FR-07	Архівування сирих подій	Система повинна зберігати копію події в об'єктному сховищі у JSON-форматі	Високий
FR-08	Отримання рекомендацій	Система повинна повертати top-N рекомендацій за userId і limit	Високий
FR-09	Підтримка fallback-рекомендацій	Якщо ML-компонент недоступний, система повинна сформувати рекомендації за евристичною логікою	Високий
FR-10	Інтеграція з Amazon Personalize	Система повинна підтримувати отримання рекомендацій і передавання item-подій до Amazon Personalize після налаштування відповідних параметрів	Середній
FR-11	Підготовка CSV-датасетів	Система повинна формувати items.csv та interactions.csv для подальшого імпорту в ML-сервіс	Середній
FR-12	Статистика каталогу	Система повинна надавати дані про кількість об'єктів, активні об'єкти, категорії, бренди та цінові характеристики	Середній

Кінець таблиці 2.3

Код вимоги	Назва вимоги	Опис вимоги	Пріоритет
FR-13	Статистика подій	Система повинна надавати кількість подій, користувачів, сесій, об'єктів з взаємодіями та розподіл за типами подій	Середній
FR-14	Профіль користувача	Система повинна надавати коротке зведення останньої активності користувача	Середній
FR-15	Debug-режим рекомендацій	Система повинна повертати додаткові дані для аналізу рекомендацій і розрахунку метрик	Низький
FR-16	Перевірка працездатності	Система повинна мати endpoint для health-check	Високий

Наступною групою є нефункціональні вимоги. Вони визначають не те, які функції виконує система, а те, якою має бути якість її роботи. Для рекомендаційного сервісу нефункціональні вимоги є особливо важливими, оскільки навіть правильна функціональність може бути непридатною для використання, якщо API працює повільно, нестабільно або погано масштабується.

Першою нефункціональною вимогою є масштабованість. Система повинна бути спроектована так, щоб збільшення кількості користувачів, об'єктів каталогу та подій не вимагало повної зміни архітектури. Саме тому в архітектурі обрано serverless-компоненти та NoSQL-сховище. Обробники подій, рекомендацій і статистики можуть масштабуватися незалежно один від одного. Це важливо, оскільки кількість подій зазвичай зростає швидше, ніж кількість запитів рекомендацій або оновлень каталогу.

Другою вимогою є низька затримка відповіді. Endpoint рекомендацій повинен повертати результат за прийнятний для інтеграції час, оскільки зовнішній сервіс може використовувати його під час завантаження сторінки, формування блоку “Рекомендовано для вас” або оновлення персональної каруселі товарів. Система не повинна виконувати важкі batch-операції під час запиту рекомендацій. Саме тому підготовка датасетів, імпорт у Personalize та інші offline-операції мають бути винесені за межі основного request-response сценарію.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		57

Третьою вимогою є відмовостійкість. Якщо Amazon Personalize недоступний або не налаштований, система не повинна повертати критичну помилку для базового сценарію рекомендацій. Замість цього вона повинна перейти до fallback-режиму. Такий підхід дозволяє зберегти працездатність API навіть у випадку проблем із ML-компонентом. Для зовнішнього сервісу це означає, що блок рекомендацій може продовжувати працювати, хоча якість персоналізації може бути нижчою.

Четверта вимога — спостережуваність. Система повинна логувати помилки обробки запитів, помилки синхронізації з Amazon Personalize, некоректні вхідні дані та інші важливі події. Також потрібні endpoint-и статистики, які дають змогу швидко перевірити, чи надходять події, чи є активні об'єкти каталогу та які користувачі мають історію взаємодій. Без цього складно діагностувати проблеми інтеграції.

П'ята вимога — безпека та мінімізація даних. Оскільки система працює з поведінковими подіями користувачів, вона не повинна вимагати надлишкової персональної інформації. Для роботи рекомендацій достатньо технічного userId або псевдонімізованого ідентифікатора, sessionId, itemId, типу події, часової мітки та додаткових контекстних властивостей. У продукційному середовищі API має бути захищений механізмом авторизації, наприклад API-ключем, JWT або іншим способом обмеження доступу. У межах MVP може бути передбачена спрощена модель доступу, але архітектура повинна дозволяти додати авторизацію без повної переробки системи.

Шоста вимога — підтримуваність. Код системи має бути розділений за відповідальністю: окремі обробники для каталогу, подій, рекомендацій, статистики та перевірки працездатності; окремі сервіси для роботи зі сховищем, валідації, fallback-логіки, Personalize та метрик. Такий поділ спрощує тестування та майбутні зміни. Наприклад, зміну fallback-алгоритму можна виконати без зміни API Gateway або логіки прийому подій. Основні нефункціональні вимоги до системи наведено в таблиці 2.4.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		58

Таблиця 2.4 – Нефункціональні вимоги до системи

Код вимоги	Назва вимоги	Опис вимоги	Критерій перевірки
NFR-01	Масштабованість	Система повинна підтримувати зростання кількості користувачів, подій і об'єктів без зміни базової архітектури	Компоненти можуть масштабуватися незалежно
NFR-02	Низька затримка відповіді	Запити до рекомендаційного API не повинні блокуватися довготривалими ML-операціями	Offline-процеси винесені окремо від API
NFR-03	Відмовостійкість	При недоступності Personalize система повинна переходити до fallback-рекомендацій	API повертає рекомендації зі strategy = fallback
NFR-04	Спостережуваність	Система повинна мати логи та endpoint-и статистики	Можна перевірити кількість подій, користувачів і об'єктів
NFR-05	Безпека даних	Система не повинна вимагати надлишкових персональних даних	Для роботи достатньо технічних ідентифікаторів
NFR-06	Підтримуваність	Логіка системи повинна бути розділена на окремі модулі	Зміна одного модуля не потребує зміни всієї системи
NFR-07	Розширюваність	Система повинна дозволяти додавання нових типів подій, метрик або рекомендаційних стратегій	Нові компоненти можна додати без зміни зовнішнього API
NFR-08	Повторюваність розгортання	Інфраструктура повинна описуватися як код	Ресурси можна повторно створити через CDK

Кінець таблиці 2.4

Код вимоги	Назва вимоги	Опис вимоги	Критерій перевірки
NFR-09	Тестованість	Основні сценарії мають бути перевірені через API-запити та демонстраційний інтерфейс	Можна перевірити health, catalog, events і recommendations
NFR-10	Узгодженість форматів	Дані мають зберігатися у форматах, придатних для API, архіву й ML-експорту	Події можна перетворити в interactions.csv

Наведені нефункціональні вимоги визначають експлуатаційні характеристики системи та забезпечують її придатність до використання як окремого інтеграційного сервісу. Особливу увагу приділено масштабованості, підтримуваності, відмовостійкості та можливості повторюваного розгортання інфраструктури.

Окремо необхідно сформулювати вимоги до даних. Дані в рекомендаційній системі є не допоміжним ресурсом, а основою роботи всієї системи. Якщо каталог неповний, рекомендації не матимуть достатньої інформації для відображення. Якщо події неякісні, система неправильно відображатиме інтереси користувача. Якщо часові мітки некоректні, неможливо буде правильно відновити послідовність взаємодій.

Каталоговий об'єкт повинен мати мінімальний набір обов'язкових полів: itemId, title і category. Поле itemId використовується як технічний ідентифікатор, title — для відображення та пошуку, category — для групування та fallback-логіки. Додаткові поля brand, price, tags, popularityScore, isActive і metadata підвищують якість рекомендацій, але не всі є обов'язковими. Наприклад, якщо brand відсутній, fallback-рекомендатор не зможе використати збіг бренду, але все ще зможе врахувати категорію та популярність.

Подія користувача повинна мати userId і eventType. Для об'єктних подій також потрібен itemId, а для пошукових — query. Поле timestamp потрібне для правильного впорядкування історії, визначення останніх інтересів користувача та

підготовки часово коректних train/test-наборів. Поле sessionId дозволяє пов'язувати події в межах однієї сесії. Поля recommendationId та impression потрібні для аналізу того, як користувач взаємодіяв саме з рекомендованими об'єктами. Основні вимоги до структури даних системи наведено в таблиці 2.5.

Таблиця 2.5 — Вимоги до структури даних

Сутність	Поле	Обов'язковість	Призначення
CatalogItem	itemId	Обов'язкове	Унікальний ідентифікатор об'єкта
CatalogItem	title	Обов'язкове	Назва об'єкта для відображення та пошуку
CatalogItem	category	Обов'язкове	Категорія для фільтрації та fallback-логіки
CatalogItem	brand	Необов'язкове	Додатковий сигнал для персоналізації
CatalogItem	price	Необов'язкове	Інформація для відображення та аналітики
CatalogItem	tags	Необов'язкове	Слова для пошукового та контентного збігу
CatalogItem	popularityScore	Необов'язкове	Базовий сигнал популярності
CatalogItem	isActive	Необов'язкове	Ознака доступності об'єкта для рекомендацій
UserEvent	eventId	Необов'язкове	Унікальний ідентифікатор події, може генеруватися системою
UserEvent	userId	Обов'язкове	Ідентифікатор користувача
UserEvent	sessionId	Необов'язкове	Ідентифікатор сесії
UserEvent	itemId	Обов'язкове для item-подій	Об'єкт, з яким взаємодіяв користувач
UserEvent	eventType	Обов'язкове	Тип дії користувача
UserEvent	timestamp	Необов'язкове	Час події, може генеруватися системою
UserEvent	query	Обов'язкове для search	Текст пошукового запиту
UserEvent	recommendationId	Необов'язкове	Ідентифікатор рекомендаційної видачі
UserEvent	impression	Необов'язкове	Список показаних користувачу itemId
UserEvent	properties	Необов'язкове	Додаткові контекстні властивості

Інтеграційні вимоги визначають, як зовнішній сервіс повинен взаємодіяти з рекомендаційною системою. Важливо, щоб зовнішня інтеграція була простою і не

залежала від внутрішньої реалізації. Тому система повинна надавати HTTP API з JSON-форматом запитів і відповідей. Це дозволяє підключити рекомендаційний модуль до backend-застосунків, написаних різними мовами програмування.

Для інтеграції зовнішній сервіс повинен виконувати три основні дії. По-перше, передати або оновити каталог об'єктів. По-друге, надсилати події користувачів після важливих дій у системі. По-третє, запитувати рекомендації для конкретного користувача. Такий сценарій не вимагає від зовнішнього сервісу знання про DynamoDB, S3, Amazon Personalize або fallback-алгоритм. Усі ці деталі приховані всередині рекомендаційної системи. Основні інтеграційні вимоги до API наведено в таблиці 2.6.

Таблиця 2.6 — Інтеграційні вимоги до API

Endpoint	Метод	Призначення	Основні вхідні дані	Очікуваний результат
/health	GET	Перевірка працездатності системи	—	Статус роботи сервісу
/catalog/items	POST	Додавання або оновлення каталогу	Масив items	Кількість збережених об'єктів
/catalog/items	GET	Отримання об'єктів каталогу для демонстрації	limit, search, activeOnly, category, brand	Список об'єктів
/events	POST	Передавання події користувача	userId, eventType, itemId або query	eventId і статус синхронізації
/recommendations	GET	Отримання рекомендацій	userId, limit, debug	Список рекомендованих об'єктів
/catalog/stats	GET	Статистика каталогу	—	Розподіл за категоріями, брендами та цінами

Кінець таблиці 2.6

Endpoint	Метод	Призначення	Основні вхідні дані	Очікуваний результат
/events/stats	GET	Статистика подій	—	Кількість подій, користувачів, сесій і типів подій
/users	GET	Отримання списку користувачів	limit, search	Список користувачів з короткою статистикою
/users/{userId}/summary	GET	Отримання профілю активності користувача	userId	Останні події та зведення інтересів

Оскільки система повинна підтримувати не лише базові рекомендації, а й можливість оцінювання їх якості, потрібно сформулювати вимоги до метрик. У межах debug-режиму система повинна мати можливість розраховувати Precision@K, Recall@K, Average Precision, MAP, NDCG і HitRate для переданого або автоматично сформованого набору релевантних itemId. Ці метрики не обов'язково мають використовуватися в кожному продукційному запиті, але вони важливі для тестування та демонстрації коректності системи.

Також потрібно визначити обмеження системи. Перше обмеження полягає в тому, що якість персоналізації залежить від обсягу історичних даних. Якщо користувач новий або має дуже мало подій, система не може побудувати глибокий персональний профіль. У такому випадку вона повинна використовувати fallback-логіку. Друге обмеження — рекомендаційний сервіс не відповідає за бізнес-правила зовнішнього застосунку, наприклад за наявність товару на складі, знижки, права доступу або регіональні обмеження. Ці правила залишаються відповідальністю зовнішнього backend-сервісу. Третє обмеження — система не повинна зберігати надлишкову персональну інформацію, якщо вона не потрібна для рекомендацій.

Для формалізації пріоритетів вимог доцільно використати три рівні: обов'язкові, бажані та додаткові. Обов'язкові вимоги потрібні для мінімально працездатної системи. До них належать прийом каталогу, прийом подій, зберігання

даних, отримання рекомендацій, fallback-режим і health-check. Бажані вимоги покращують якість і повноту системи: інтеграція з Amazon Personalize, архівування в S3, статистика, debug-режим і підготовка CSV-датасетів. Додаткові вимоги можуть бути реалізовані в подальшому розвитку: авторизація для кожного клієнта, мультиорендність, А/В-тестування, розвинена аналітична панель і автоматичне перенавчання моделей. Пріоритезацію основних вимог до системи наведено в таблиці 2.7.

Таблиця 2.7 — Пріоритезація вимог

Рівень пріоритету	Вимоги
Обов'язкові	Прийом каталогу, прийом подій, валідація даних, оперативне зберігання, отримання рекомендацій, fallback-режим, health-check
Бажані	Архівування сирих подій, інтеграція з Amazon Personalize, підготовка CSV-датасетів, статистика каталогу та подій, профіль користувача, debug-режим
Додаткові	Повноцінна авторизація клієнтів, мультиорендність, А/В-тестування, автоматичне перенавчання, розширена панель моніторингу

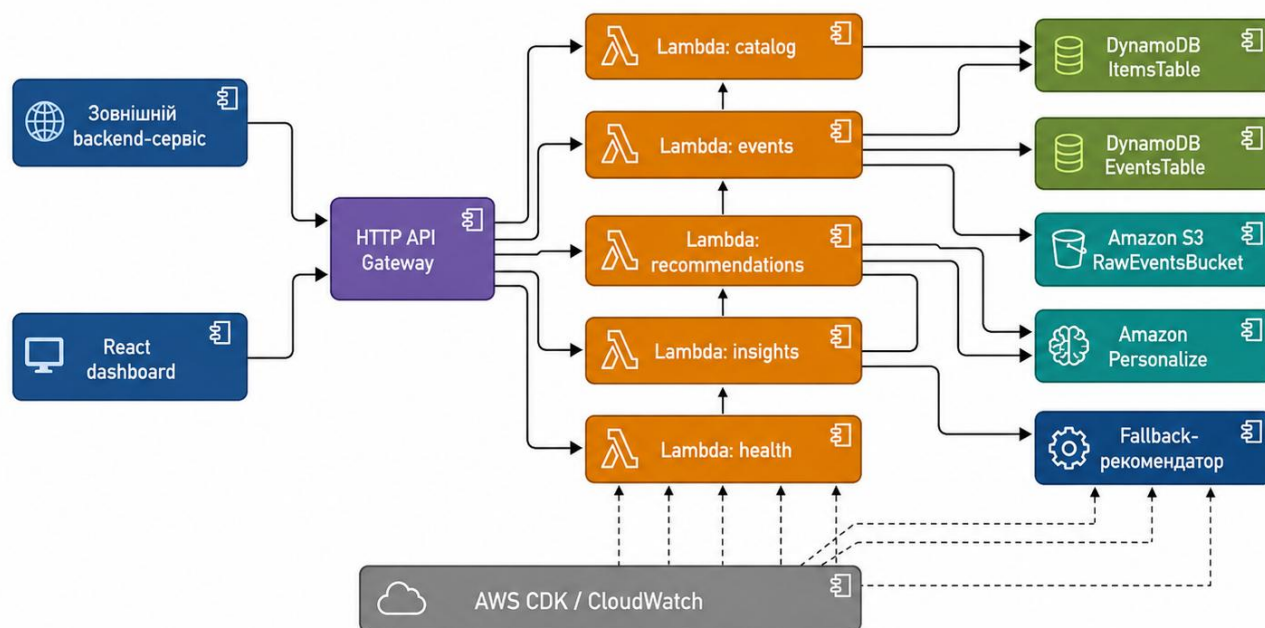
Критерії приймання потрібні для того, щоб перевірити, чи відповідає реалізація сформованим вимогам. Для кожної ключової функції потрібно мати зрозумілий результат перевірки. Наприклад, якщо на endpoint /events передати коректну подію view з userId і itemId, система повинна повернути ok: true, eventId і зберегти подію в оперативному та архівному сховищах. Якщо передати подію search без query, система повинна повернути помилку валідації. Якщо викликати /recommendations для користувача з історією, система повинна повернути список рекомендацій. Якщо Personalize недоступний, система повинна повернути рекомендації зі strategy = fallback. Основні критерії приймання системи наведено в таблиці 2.8.

Таблиця 2.8 — Критерії приймання основних сценаріїв

Сценарій	Умова перевірки	Очікуваний результат
Перевірка працездатності	Виконати GET /health	API повертає ok: true і часову мітку
Додавання каталогу	Передати коректний масив items	Система зберігає об'єкти та повертає кількість upserted
Некоректний каталог	Передати item без itemId або title	Система повертає помилку валідації
Додавання події view	Передати userId, itemId і eventType = view	Система генерує eventId, зберігає подію та повертає ok: true
Некоректна search-подія	Передати eventType = search без query	Система повертає помилку валідації
Отримання рекомендацій	Передати userId і limit	Система повертає список не більше limit об'єктів
Fallback-режим	Вимкнути або не налаштувати Personalize	Система повертає strategy = fallback
Personalize-режим	Налаштувати PERSONALIZE_RECOMMENDER_ARN	Система повертає strategy = personalize за умови успішного запиту
Debug-режим	Виконати /recommendations?debug=true	Система повертає додатковий debug-блок
Статистика подій	Виконати GET /events/stats	Система повертає кількість подій і розподіл за eventType

Важливою частиною інженерії вимог є визначення меж системи. Рекомендаційна система не повинна виконувати функції основного backend-застосунку. Вона не відповідає за реєстрацію користувачів, автентифікацію кінцевого користувача, оформлення замовлення, оплату, управління залишками товарів або повний пошук по каталогу. Її зона відповідальності — прийом даних про поведінку, збереження історії, підготовка даних для рекомендаційної моделі, формування рекомендацій і надання допоміжної статистики. Архітектуру взаємодії

основних компонентів рекомендаційної системи наведено на рисунку 2.1.



Опис схеми:

- Зовнішній backend-сервіс: передає каталог, події користувачів і запитує рекомендації.
- React dashboard: відображає каталог, статистику, користувачів і рекомендаційну видачу через API.
- HTTP API Gateway: єдина точка входу до системи.
- AWS Lambda: обробка endpoint-ів /catalog, /events, /recommendations, /insights та /health.
- DynamoDB: оперативне зберігання каталогу об'єктів і історії подій.
- Amazon S3 та Amazon Personalize: архівування даних, підготовка датасетів і персоналізовані рекомендації.
- Fallback-рекомендатор: резервна логіка рекомендацій у разі нестачі даних або недоступності ML-компонента.

Рисунок 2.1 – Діаграма компонентів системи рекомендацій

На рисунку 2.1 наведено діаграму компонентів розробленої системи рекомендацій. Вона відображає основні частини програмної компоненти та зв'язки між ними: зовнішній backend-сервіс і демонстраційний React dashboard взаємодіють із системою через HTTP API Gateway, який передає запити до відповідних AWS Lambda-функцій. Окремі Lambda-обробники відповідають за роботу з каталогом, прийом подій користувачів, формування рекомендацій, отримання статистики та перевірку працездатності системи.

Дані каталогу зберігаються в DynamoDB ItemsTable, а історія взаємодій користувачів — у DynamoDB EventsTable. Для архівування сирих подій і підготовки наборів даних використовується Amazon S3. Компонент Amazon Personalize забезпечує формування персоналізованих рекомендацій на основі накопиченої історії, а fallback-рекомендатор використовується як резервний

механізм у випадку нестачі даних або недоступності ML-компонента. AWS CDK і CloudWatch забезпечують розгортання інфраструктури, логування та моніторинг роботи системи. Така структура демонструє модульність реалізації та чіткий розподіл відповідальності між API-рівнем, обробниками, сховищами даних і рекомендаційними компонентами.

У результаті інженерії вимог було визначено, що система повинна бути API-орієнтованою, подієвою, масштабованою, придатною до інтеграції з ML-сервісом і стійкою до нестачі історичних даних. Основними об'єктами системи є каталоговий item, користувацька подія та рекомендаційна відповідь. Основними сценаріями є завантаження каталогу, передавання подій, отримання рекомендацій, аналіз статистики та перевірка якості. Вимоги сформовано так, щоб на першому етапі можна було створити працездатний MVP із fallback-рекомендаціями, а надалі розширити його за рахунок Amazon Personalize, аналітики та більш складних механізмів оцінювання.

Отже, інженерія вимог у межах цього проєкту дозволяє перейти від загальної ідеї рекомендаційної системи до конкретного набору перевірюваних характеристик. Сформовані вимоги визначають, які дані повинні надходити до системи, як вони мають перевірятися і зберігатися, яким має бути API, як повинні формуватися рекомендації та які критерії дозволяють оцінити коректність реалізації. Це створює основу для подальшого архітектурного проєктування програмної компоненти та її практичної реалізації.

2.2. Архітектурне проектування програмної компоненти

2.2.1. Розробка програмної компоненти інтелектуальної системи для генерації користувацьких рекомендацій

На основі сформованих вимог та обраної архітектури інтеграції розробка програмної компоненти інтелектуальної системи повинна бути спрямована на створення окремого backend-сервісу, який приймає історичні дані користувачів, зберігає каталог об'єктів, накопичує події взаємодії, формує рекомендації та надає

					ІАЛЦ.467200.003 ПЗ	Арк.
						67
Зм.	Арк.	№ докум.	Підпис	Дата		

результати через API. Така компонента не повинна замінювати основний backend зовнішнього застосунку, а має виконувати роль спеціалізованого рекомендаційного шару, що підключається до нього через програмний інтерфейс. Це відповідає попередньо визначеним технічним характеристикам системи: наявність API для передачі подій, зберігання історичних взаємодій, підтримка каталогу, генерація рекомендацій, резервна логіка та орієнтація на масштабовану хмарну архітектуру.

Вибір модульної структури програмної компоненти зумовлений тим, що рекомендаційна система включає кілька функціонально незалежних підсистем: прийом каталогу, прийом подій, роботу зі сховищами даних, генерацію рекомендацій, інтеграцію з ML-сервісом, розрахунок метрик, аналітику та інфраструктурне розгортання. Якщо реалізовувати всю логіку в одному монолітному обробнику, система швидко стане складною для підтримки, тестування й подальшого розширення. Тому програмну компоненту доцільно розділити на окремі модулі, кожен із яких відповідає за одну групу задач. Основні складові програмної компоненти рекомендаційної системи наведено в таблиці 2.9.

У загальному вигляді програмна компонента складається з таких частин:

Таблиця 2.9 — Структура програмної компоненти рекомендаційної системи

Частина системи	Призначення	Основна відповідальність
API-рівень	Прийом HTTP-запитів	Надання endpoint-ів для каталогу, подій, рекомендацій, статистики та перевірки працездатності
Lambda-обробники	Виконання прикладної логіки	Обробка запитів, валідація, виклик сервісів, формування відповіді
Сервісний шар	Повторно використовується бізнес-логіка	Робота з DynamoDB, Amazon S3, Amazon Personalize, fallback-алгоритмом, метриками
Шар моделей даних	Опис основних сутностей	CatalogItem, UserEvent, RecommendationResult

Кінець таблиці 2.9

Частина системи	Призначення	Основна відповідальність
Інфраструктурний шар	Опис AWS-ресурсів	API Gateway, Lambda, DynamoDB, S3, IAM, CloudWatch, Personalize-ресурси
Batch-скрипти	Підготовка даних для ML	Генерація demo-даних, експорт CSV, імпорт у Personalize, створення recommender-a
Демонстраційний інтерфейс	Перевірка роботи системи	Перегляд каталогу, користувачів, рекомендацій, debug-метрик і статистики

Такий поділ дозволяє розглядати програмну компоненту як набір взаємопов'язаних, але логічно незалежних модулів. API-рівень відповідає лише за маршрутизацію запитів. Lambda-обробники виконують сценарії використання. Сервісний шар містить логіку, яку можна перевикористовувати в різних обробниках. Інфраструктурний шар забезпечує створення ресурсів у хмарному середовищі. Batch-скрипти не впливають на основний online-сценарій, але забезпечують підготовку даних для машинного навчання. Демонстраційний інтерфейс використовується для перевірки та представлення результатів роботи системи.

Основними сутностями програмної компоненти є об'єкт каталогу, подія користувача та результат рекомендації. Об'єкт каталогу описує одиницю, яку система потенційно може рекомендувати користувачу. Це може бути товар, курс, стаття, медіаконтент або інший елемент зовнішнього сервісу. Подія користувача описує факт взаємодії користувача із системою: пошук, перегляд, клік, додавання до кошика або покупку. Результат рекомендації описує об'єкт, який система пропонує користувачу, разом із оцінкою, джерелом рекомендації та додатковою інформацією для відображення. Основні моделі даних програмної компоненти наведено в таблиці 2.10.

Таблиця 2.10 — Основні моделі даних програмної компоненти

Модель	Основні поля	Призначення
CatalogItem	itemId, title, category, brand, price, tags, popularityScore, isActive, metadata	Описує об'єкт каталогу, який може бути рекомендований користувачу
UserEvent	eventId, userId, sessionId, itemId, recommendationId, impression, eventType, timestamp, query, filters, properties	Описує історичну дію користувача в зовнішній системі
RecommendationResult	itemId, score, source, reason, title, category, brand, price	Описує один елемент у списку рекомендацій

Модель CatalogItem повинна містити мінімальний набір полів, необхідний для ідентифікації та відображення об'єкта. Поле itemId є основним ключем, за яким об'єкт зберігається в каталозі та зв'язується з подіями користувачів. Поле title потрібне для відображення у клієнтському інтерфейсі. Поле category використовується як один із базових сигналів для fallback-рекомендацій. Поля brand, price і tags не завжди є обов'язковими, але вони підвищують якість ранжування та дають більше контексту для аналізу. Поле popularityScore використовується як базовий сигнал популярності, а isActive дозволяє виключати з рекомендацій неактуальні об'єкти.

Модель UserEvent є центральною для всієї рекомендаційної системи, оскільки саме історія подій формує основу для персоналізації. Для подій типу view, click, add_to_cart і purchase обов'язковим є itemId, оскільки ці події пов'язані з конкретним об'єктом каталогу. Для події типу search обов'язковим є query, оскільки вона описує не взаємодію з конкретним об'єктом, а пошуковий намір користувача. Поля recommendationId та impression потрібні для зв'язування подальших дій користувача з попередньою рекомендаційною видачею. Це дає змогу в майбутньому аналізувати, які рекомендації справді призводять до кліків, додавання до кошика або покупок.

Модель RecommendationResult потрібна для уніфікованої відповіді API. Незалежно від того, чи рекомендація отримана від Amazon Personalize, чи

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		70

сформована fallback-алгоритмом, зовнішній backend повинен отримувати однакову структуру відповіді. Саме тому в результаті передбачається поле source, яке може мати значення personalize або fallback. Це дозволяє не змінювати зовнішню інтеграцію при зміні внутрішньої стратегії рекомендацій.

Програмна компонента реалізується за принципом поділу на online- та offline-контур. Online-контур обробляє запити, які надходять у реальному часі: додавання каталогу, прийом подій, отримання рекомендацій, перегляд статистики та перевірка працездатності. Offline-контур відповідає за підготовку даних для навчання моделі: генерацію або збирання історії, формування CSV-файлів, завантаження даних у S3, імпорт у Amazon Personalize та створення рекомендаційного ресурсу. Такий поділ потрібний для того, щоб довготривалі операції машинного навчання не блокували основний API.

Основним вхідним модулем системи є API Gateway. Через нього зовнішній сервіс отримує доступ до endpoint-ів рекомендаційної системи. У межах програмної компоненти передбачаються такі основні групи endpoint-ів:

- endpoint-и каталогу;
- endpoint-и подій;
- endpoint рекомендацій;
- endpoint-и статистики;
- endpoint перевірки працездатності;
- endpoint-и профілю користувача для демонстрації та аналізу.

Endpoint POST /catalog/items призначений для додавання або оновлення об'єктів каталогу. Його логіка реалізується окремим обробником каталогу. На вхід цей обробник отримує масив об'єктів, перевіряє їхню структуру та записує до таблиці ItemsTable. Для оптимізації запису об'єкти обробляються пакетами. Такий підхід зручний, оскільки зовнішній сервіс може передати одразу кілька товарів або елементів каталогу, а система виконає їх масове оновлення.

Перед записом каталогу обов'язково виконується валідація даних. Вона перевіряє наявність полів itemId, title і category, а також коректність необов'язкових

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		71

полів: price, tags, popularityScore, isActive і metadata. Використання окремого модуля валідації зумовлене тим, що правила перевірки даних не повинні дублюватися в різних Lambda-обробниках. У такому разі при зміні структури item або event достатньо оновити одну схему валідації, а не змінювати всю бізнес-логіку системи.

Endpoint POST /events є одним із ключових у системі, оскільки він приймає поведінкові події користувачів із зовнішнього backend-сервісу. Після отримання запиту обробник подій виконує кілька послідовних етапів. Спочатку здійснюється розбір JSON-тіла запиту. Далі виконується перевірка даних відповідно до схеми UserEvent. Якщо eventId не передано, система генерує його автоматично. Якщо timestamp відсутній, система встановлює поточну часову мітку. Після цього формується нормалізована подія, яка має єдину внутрішню структуру незалежно від того, які необов'язкові поля були передані у вхідному запиті.

Після нормалізації подія записується в EventsTable. Для зберігання використовується складений ключ: pk = USER#{userId}, sk = TS#{timestamp}#{eventId}. Така структура була обрана тому, що основним сценарієм читання є отримання останніх подій конкретного користувача. Partition key дозволяє швидко знаходити всі події одного користувача, а sort key забезпечує їх упорядкування за часом і отримання найновіших взаємодій. Це особливо важливо для endpoint-а рекомендацій, який може використовувати останні дії користувача для побудови fallback-видачі або debug-профілю.

Паралельно з записом у DynamoDB подія зберігається в Amazon S3 у сирому JSON-форматі. Для цього формується шлях, який містить дату події, userId та eventId. Наприклад, подія може бути збережена за логікою events/date=YYYY-MM-DD/{userId}/{eventId}.json. Така структура зручна для подальшої пакетної обробки, оскільки дозволяє відбирати події за датою або користувачем. DynamoDB при цьому виконує роль оперативного сховища, а S3 — роль архіву та джерела для майбутнього формування навчальних наборів.

					ІАЛЦ.467200.003 ПЗ	Арк.
						72
Зм.	Арк.	№ докум.	Підпис	Дата		

Якщо в середовищі налаштований ідентифікатор Amazon Personalize Event Tracker, обробник подій додатково намагається передати item-подію до Amazon Personalize. Для цього подія внутрішнього типу view, click, add_to_cart або purchase перетворюється у формат, придатний для Personalize: View, Click, AddToCart або Purchase. Подія типу search не передається як item-взаємодія, оскільки вона не завжди містить itemId. Така поведінка дозволяє не втрачати пошукові сигнали локально, але водночас не передавати до ML-сервісу неповні події.

Важливо, що помилка передавання події до Amazon Personalize не повинна зупиняти основний сценарій прийому події. Якщо локальний запис у DynamoDB і S3 виконано успішно, подія вважається прийнятою системою, навіть якщо синхронізація з Personalize тимчасово не відбулася. У відповіді API для цього використовується ознака personalizeSynced. Такий підхід підвищує відмовостійкість системи, оскільки зовнішній backend не залежить повністю від доступності ML-компонента.

Endpoint GET /recommendations реалізує основну функцію системи — формування списку рекомендованих об'єктів. На вхід він отримує userId, limit і додаткові параметри debug, relevantItemIds, groundTruthMode, groundTruthWindow і kValues. Параметр userId є обов'язковим, оскільки рекомендація формується для конкретного користувача. Параметр limit обмежує кількість елементів у відповіді та повинен мати допустиме значення, щоб уникнути надто великих відповідей і зайвого навантаження.

Логіка endpoint-а рекомендацій реалізується за принципом вибору стратегії. Якщо в системі налаштовано Amazon Personalize, обробник спочатку намагається отримати рекомендації з цього сервісу. Для цього викликається сервісний модуль Personalize, який приховує деталі роботи з AWS SDK. У відповідь система отримує список itemId, score і recommendationId. Далі виконується збагачення рекомендацій даними з каталогу: для кожного itemId завантажується відповідний CatalogItem з ItemsTable, після чого до результату додаються title, category, brand і price. Це

потрібно для того, щоб зовнішній сервіс міг одразу відобразити рекомендації без додаткових запитів до каталогу.

Якщо Amazon Personalize не налаштований або запит до нього завершується помилкою, система переходить до fallback-режиму. У цьому режимі рекомендації формуються локально. Спочатку система отримує останні події користувача з EventsTable. Потім за itemId цих подій завантажуються нещодавно переглянуті або використані об'єкти. Далі система сканує активні об'єкти каталогу та для кожного кандидата розраховує евристичну оцінку.

Fallback-оцінка складається з кількох сигналів. Базою є popularityScore об'єкта. Якщо категорія кандидата збігається з категоріями об'єктів, з якими користувач нещодавно взаємодівав, score збільшується. Якщо бренд кандидата збігається з брендами з історії користувача, score також збільшується. Якщо в історії є пошукові запити, система виділяє слова з query і перевіряє, чи трапляються вони в назві або тегах кандидата. Також із рекомендацій виключаються об'єкти, з якими користувач уже нещодавно взаємодівав. Після цього кандидати сортуються за score, і в результат потрапляють перші N об'єктів.

Узагальнено fallback-алгоритм можна подати так:

1. Отримати userId і limit із запиту.
2. Завантажити останні події користувача.
3. Визначити itemId об'єктів, з якими користувач уже взаємодівав.
4. Завантажити нещодавні об'єкти користувача з каталогу.
5. Виділити множину останніх категорій, брендів і пошукових слів.
6. Просканувати активні об'єкти каталогу.
7. Виключити вже переглянуті або використані об'єкти.
8. Для кожного кандидата розрахувати score.
9. Відсортувати кандидатів за score у спадному порядку.
10. Повернути перші limit рекомендацій.

Основні сигнали fallback-рекомендатора наведено в таблиці 2.11.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		74

Таблиця 2.11 — Сигнали fallback-рекомендатора

Сигнал	Джерело даних	Вплив на рекомендацію
Популярність об'єкта	popularityScore з CatalogItem	Формує базовий рейтинг кандидата
Збіг категорії	Останні item-події користувача	Підвищує оцінку об'єктів зі знайомих користувачу категорій
Збіг бренду	Останні item-події користувача	Підвищує оцінку об'єктів брендів, з якими користувач уже взаємодіяв
Збіг із пошуковими словами	Події типу search	Підвищує оцінку об'єктів, назва або теги яких відповідають пошуковим запитам
Виключення переглянутих об'єктів	Історія item-подій	Запобігає повторному показу вже відомих користувачу об'єктів

Саме така fallback-логіка була обрана тому, що вона залишається простою, пояснюваною та працездатною навіть за невеликого обсягу історичних даних. Вона не потребує попереднього навчання моделі, але водночас дозволяє враховувати індивідуальний контекст користувача. Це особливо важливо на початковому етапі розробки, коли Amazon Personalize ще може бути не налаштований або коли система ще не містить достатнього обсягу взаємодій для якісного ML-ранжування.

Окремим модулем програмної компоненти є сервіс інтеграції з Amazon Personalize. Його завдання полягає в тому, щоб ізолювати решту системи від деталей роботи з AWS SDK. У межах цього модуля визначаються функції перевірки доступності Personalize, отримання рекомендацій, передавання подій і перетворення типів подій у формат Amazon Personalize. Такий поділ є важливим, тому що в майбутньому можна буде замінити Amazon Personalize на інший рекомендаційний механізм або додати власну ML-модель, не змінюючи всі Lambda-обробники.

Програмна компонента також містить модуль роботи з DynamoDB. Він відповідає за запис подій, отримання останніх подій користувача, масове оновлення каталогу, отримання об'єктів за itemId, сканування каталогу та сканування подій для статистики. Такий сервісний шар потрібний для того, щоб

обробники не працювали напряму з низькорівневими командами бази даних. Наприклад, обробник рекомендацій не повинен знати, як саме формується QueryCommand для отримання останніх подій. Він повинен викликати функцію getRecentEvents, а деталі ключів і запиту мають залишатися всередині сервісу доступу до даних.

Модуль статистики та аналітики реалізує допоміжні endpoint-и. Вони потрібні для перегляду стану каталогу, кількості подій, списку користувачів і короткого профілю активності користувача. Наприклад, endpoint статистики каталогу може повертати кількість активних та неактивних об'єктів, розподіл за категоріями, розподіл за брендами та цінову статистику. Endpoint статистики подій може повертати загальну кількість подій, кількість унікальних користувачів, кількість сесій, кількість об'єктів із взаємодіями та розподіл за eventType. Це дозволяє швидко перевірити, чи справді система накопичує дані та чи є вони придатними для рекомендацій.

Для аналізу конкретного користувача передбачається endpoint профілю. Він отримує останні події користувача, пов'язані з ними об'єкти каталогу та формує коротке зведення: кількість останніх подій, кількість унікальних об'єктів, розподіл типів подій, категорії, бренди, цінові характеристики, приклади останніх подій і приклади останніх об'єктів. Такий профіль не є повноцінним користувацьким кабінетом, але він корисний для демонстрації того, як історія користувача перетворюється на основу для персоналізації.

Ще одним важливим модулем є debug-модуль рекомендацій. Він використовується тоді, коли в запиті до /recommendations передано debug=true. У цьому випадку система повертає не лише список рекомендацій, а й додатковий блок із профілем користувача, статистикою рекомендованих елементів і метриками якості. Такий механізм потрібний для перевірки логіки рекомендацій на етапі розробки та підготовки матеріалів для тестування.

Модуль метрик реалізує розрахунок Precision@K, Recall@K, Average Precision@K, MAP@K, NDCG@K і HitRate@K. Для цього він приймає список

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		76

рекомендованих `itemId`, список релевантних `itemId` і значення `K`. Якщо в запиті явно передано `relevantItemIds`, система використовує їх як `ground truth`. Якщо обрано `groundTruthMode=recent`, система може використати останні унікальні `itemId` з історії користувача як умовно релевантні. Такий режим не замінює повноцінне офлайн-оцінювання на відкладеній вибірці, але дозволяє показати механізм розрахунку метрик без окремого експериментального стенду. Основні модулі сервісного шару програмної компоненти наведено в таблиці 2.12.

Таблиця 2.12 — Модулі сервісного шару програмної компоненти

Модуль	Основні функції	Причина виділення в окремий модуль
validation	Перевірка структури каталогу та подій	Централізація правил валідації
dynamo	Запис і читання даних із DynamoDB	Ізоляція доступу до сховища
personalize	Отримання рекомендацій і передавання подій до Amazon Personalize	Ізоляція зовнішнього ML-сервісу
fallback	Формування локальних рекомендацій	Підтримка роботи без ML-компонента
metrics	Розрахунок ranking-метрик	Повторне використання метрик у debug-режимі
recommendation-debug	Побудова debug-відповіді	Пояснення та перевірка рекомендацій
env	Отримання змінних середовища	Контроль конфігурації Lambda-функцій

Окрема увага приділяється інфраструктурному шару. Він описує хмарні ресурси, які потрібні для роботи програмної компоненти. До них належать DynamoDB-таблиці `ItemsTable` та `EventsTable`, S3-бакет `RawEventsBucket`, Lambda-функції, HTTP API Gateway, IAM-дозволи, CloudWatch-логи та опційні ресурси Amazon Personalize. Я обрав Infrastructure as Code, оскільки ручне створення таких ресурсів у хмарному середовищі є менш надійним і складніше відтворюється. Завдяки CDK можна описати всю інфраструктуру в коді, повторно розгорнути її в іншому середовищі та контролювати, які саме ресурси входять до системи.

У межах інфраструктурного шару створюються окремі Lambda-функції для подій, каталогу, рекомендацій, health-check і статистики. Для кожної функції задаються змінні середовища: назви таблиць, назва S3-бакета, регіон AWS, ARN recommender-a, campaign ARN, filter ARN і tracking ID. Це дозволяє не зашивати конфігурацію в код, а передавати її через середовище виконання. Такий підхід спрощує розгортання в різних середовищах, наприклад dev або prod.

Для доступу до ресурсів використовуються IAM-дозволи за принципом мінімально необхідних прав. Обробник каталогу отримує право запису до ItemsTable. Обробник подій отримує право запису до EventsTable і право запису об'єктів у S3. Обробник рекомендацій отримує право читання з ItemsTable і EventsTable, а також дозвіл на виклик Amazon Personalize Runtime. Обробник статистики отримує право читання з таблиць. Такий розподіл прав зменшує ризики і робить архітектуру безпечнішою.

Batch-частина системи реалізується у вигляді скриптів, які запускаються окремо від API. Один скрипт може заповнювати систему демонстраційними об'єктами та подіями. Інший — генерувати тренувальні дані. Окремий скрипт експортує items.csv та interactions.csv у локальну директорію і завантажує їх до S3. Далі скрипти можуть створювати роль імпорту для Personalize, створювати dataset group, схеми, датасети, запускати імпорт, створювати recommender і event tracker. Такий підхід дозволяє виконувати ML-підготовку контрольовано, не змішуючи її з runtime-логікою API.

У програмній компоненті також передбачається демонстраційний React-інтерфейс. Він не є обов'язковою частиною рекомендаційного backend-a, але потрібний для перевірки та презентації роботи системи. Інтерфейс звертається до API, отримує каталог, список користувачів, рекомендації, статистику та debug-дані. Завдяки цьому можна візуально показати, як змінюється рекомендаційна видача для різних користувачів і як історія взаємодій впливає на результат. Для дипломного проєкту це важливо, оскільки система має бути не лише описана, а й продемонстрована в практичному сценарії.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		78

Загальний сценарій роботи програмної компоненти можна описати як послідовність етапів. Спочатку каталог об'єктів передається через API і зберігається в ItemsTable. Потім зовнішній сервіс починає надсилати події користувачів до endpoint-а /events. Кожна подія проходить валідацію, нормалізується, записується в EventsTable і архівується в S3. Якщо налаштовано Personalize Event Tracker, item-подія також передається до Amazon Personalize. Паралельно offline-скрипти можуть формувати CSV-датасети та оновлювати ML-рекомендатор. Коли зовнішній сервіс запитує рекомендації, система або отримує їх від Personalize, або формує fallback-видачу. Після цього рекомендації збагачуються даними каталогу і повертаються клієнту.

У вигляді алгоритму основний runtime-сценарій роботи системи має такий вигляд:

1. Зовнішній backend надсилає об'єкти каталогу до POST /catalog/items.
2. Обробник каталогу перевіряє структуру item-ів.
3. Коректні об'єкти записуються до ItemsTable.
4. Користувач виконує дію в зовнішньому сервісі.
5. Зовнішній backend надсилає подію до POST /events.
6. Обробник подій перевіряє структуру події.
7. Система генерує eventId і timestamp, якщо їх не передано.
8. Подія записується в EventsTable.
9. Сирий JSON події архівується в RawEventsBucket.
10. За наявності tracking ID item-подія передається до Amazon Personalize.
11. Зовнішній backend запитує GET /recommendations для userId.
12. Система обирає personalize- або fallback-стратегію.
13. Рекомендації збагачуються даними каталогу.
14. API повертає список рекомендацій зовнішньому сервісу.
15. Подальші кліки або покупки за рекомендаціями знову надсилаються як події.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		79

Відповідність між основними вимогами системи та модулями реалізації наведено в таблиці 2.13.

Таблиця 2.13 — Відповідність вимог і модулів реалізації

Вимога	Модуль реалізації	Спосіб виконання
Прийом каталогу	catalog handler, validation, dynamo	Валідація item-ів і пакетний запис до ItemsTable
Прийом подій	events handler, validation, dynamo, S3	Нормалізація подій, запис до EventsTable і архівування в S3
Генерація рекомендацій	recommendations handler, personalize, fallback	Вибір personalize або fallback-стратегії
Резервна логіка	fallback	Евристичне ранжування за популярністю, категоріями, брендами та search-сигналами
ML-інтеграція	personalize, scripts/personalize	Отримання рекомендацій, PutEvents, імпорт датасетів
Статистика	insights handler, dynamo	Агрегація даних каталогу, подій і користувачів
Метрики якості	metrics, recommendation-debug	Розрахунок Precision@K, Recall@K, MAP@K, NDCG@K, HitRate@K
Розгортання	CDK stack	Створення AWS-ресурсів як коду
Демонстрація	React UI	Візуальна перевірка каталогу, користувачів, рекомендацій і debug-даних

Наведена відповідність між вимогами та модулями реалізації демонструє, що архітектура програмної компоненти забезпечує ізоляцію окремих груп відповідальності та дозволяє реалізовувати функціональні можливості системи незалежними компонентами.

Під час розробки програмної компоненти важливо не обмежуватися лише роботою endpoint-а рекомендацій. Рекомендаційна система є корисною лише тоді, коли весь ланцюг даних працює коректно: каталог завантажується, події надходять, дані зберігаються, рекомендації повертаються, а результати можна перевірити. Саме тому в проєкті передбачено не тільки /recommendations, а й допоміжні endpoint-и /catalog/items, /events, /catalog/stats, /events/stats, /users і

/users/{userId}/summary. Вони створюють повний контур спостереження за системою.

Окремою перевагою такої програмної структури є можливість поступового розвитку. На першому етапі можна використовувати лише каталог, події та fallback-рекомендатор. Це дає мінімально працездатну систему, яку вже можна інтегрувати із зовнішнім сервісом. На другому етапі можна додати архівування в S3 і формування CSV-датасетів. На третьому етапі — підключити Amazon Personalize. На четвертому — додати розширені метрики, A/B-тестування, авторизацію клієнтів, мультиорендність і автоматичне перенавчання моделей. Важливо, що зовнішній API при цьому може залишатися майже незмінним.

З погляду якості програмної реалізації, обрана структура має кілька переваг. По-перше, вона зменшує зв'язаність компонентів: кожен Lambda-обробник відповідає за окремий сценарій. По-друге, вона підвищує тестованість: можна окремо перевіряти валідацію, fallback-алгоритм, метрики, роботу з DynamoDB і інтеграцію з Personalize. По-третє, вона забезпечує гнучкість: у майбутньому можна змінити алгоритм рекомендацій, не змінюючи формат подій і структуру зовнішнього API. По-четверте, вона добре відповідає хмарній архітектурі, оскільки кожен компонент може масштабуватися відповідно до свого навантаження.

Отже, розробка програмної компоненти інтелектуальної системи для генерації користувацьких рекомендацій передбачає створення модульного serverless backend-сервісу, який поєднує API-рівень, обробники запитів, сервісний шар, оперативне та архівне зберігання даних, fallback-рекомендатор, інтеграцію з Amazon Personalize, модулі статистики й оцінювання, а також інфраструктуру як код. Така структура дозволяє реалізувати повний цикл роботи рекомендаційної системи: від прийому каталогу та подій до формування персоналізованої рекомендаційної відповіді. Обрана програмна архітектура є достатньо простою для MVP, але водночас залишає можливість подальшого масштабування, підключення складніших моделей і використання системи як незалежної рекомендаційної компоненти для різних backend-застосунків.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		81

ВИСНОВОК ДО РОЗДІЛУ

У другому розділі було виконано проєктування інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних. Основну увагу приділено не лише вибору загальної архітектури, а й визначенню способу інтеграції рекомендаційної компоненти з іншими backend-сервісами, формуванню вимог до системи та розробці структури програмної реалізації. У результаті було обґрунтовано, що рекомендаційна система має розроблятися як окремий програмний сервіс, який підключається до зовнішніх цифрових платформ через API та не дублює їхню основну бізнес-логіку.

У межах архітектури інтеграції було визначено, що система повинна працювати за подієвою моделлю: зовнішній сервіс передає до неї дані про каталог об'єктів і поведінкові події користувачів, а система накопичує ці дані, обробляє їх і повертає рекомендації у вигляді структурованої API-відповіді. Такий підхід дозволяє використовувати рекомендаційну компоненту в різних предметних областях, зокрема в інтернет-магазинах, освітніх платформах, медіасервісах, сервісах бронювання та інших інформаційних системах, де є каталог об'єктів і історія взаємодій користувачів.

Було обґрунтовано доцільність використання хмарної serverless-архітектури, яка поєднує API Gateway, Lambda-функції, DynamoDB, Amazon S3 та опційну інтеграцію з Amazon Personalize. Така структура дає змогу розділити систему на online- та offline-контури. Online-контур відповідає за прийом подій, оперативне зберігання даних і видачу рекомендацій, а offline-контур — за архівування історії, підготовку датасетів і подальшу інтеграцію з ML-компонентом. Завдяки цьому довготривалі операції з підготовки або оновлення моделі не блокують основні API-запити.

У процесі інженерії вимог було сформовано функціональні та нефункціональні вимоги до системи. До основних функціональних вимог належать прийом каталогу, валідація об'єктів і подій, зберігання історичних взаємодій, формування top-N рекомендацій, підтримка fallback-логіки, підготовка CSV-

					ІАЛЦ.467200.003 ПЗ	Арк.
						82
Зм.	Арк.	№ докум.	Підпис	Дата		

дасетів, статистика та debug-режим. Нефункціональні вимоги охоплюють масштабованість, низьку затримку відповіді, відмовостійкість, спостережуваність, безпеку даних, підтримуваність і повторюваність розгортання. Такий набір вимог дозволяє розглядати систему не як окремий алгоритм, а як повноцінну програмну компоненту, придатну до інтеграції та подальшого розвитку.

Окремо було визначено вимоги до структури даних. Основними сутностями системи є об'єкт каталогу, подія користувача та результат рекомендації. Для каталогу важливими є ідентифікатор об'єкта, назва, категорія, бренд, ціна, теги, показник популярності та статус активності. Для подій користувача важливими є `userId`, `eventType`, `timestamp`, `itemId` або `query` залежно від типу події. Саме ці дані формують основу для персоналізації, аналізу поведінки користувачів і підготовки навчальних наборів для рекомендаційної моделі.

У межах архітектурного проектування програмної компоненти було запропоновано модульну структуру реалізації. Система поділяється на API-рівень, Lambda-обробники, сервісний шар, шар моделей даних, інфраструктурний шар, batch-скрипти та демонстраційний інтерфейс. Такий поділ зменшує зв'язаність компонентів, спрощує тестування та дозволяє окремо розвивати логіку каталогу, подій, рекомендацій, статистики, інтеграції з Amazon Personalize і fallback-рекомендатора.

Особливе значення в розділі має fallback-рекомендатор, який забезпечує працездатність системи в умовах недостатньої кількості історичних даних або недоступності ML-компонента. Його логіка базується на популярності об'єктів, збігу категорій і брендів з останніми діями користувача, аналізі пошукових запитів і виключенні вже переглянутих об'єктів. Це дозволяє системі залишатися корисною навіть на початковому етапі використання, коли персоналізована модель ще не має достатнього обсягу даних для якісного навчання.

У результаті виконаного проектування було сформовано цілісну архітектурну основу системи, яка підтримує повний цикл роботи з рекомендаціями: завантаження каталогу, прийом користувацьких подій, оперативне й архівне

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		83

зберігання історичних даних, підготовку датасетів, інтеграцію з керованим ML-сервісом, формування рекомендацій, fallback-режим, статистику та оцінювання якості. Це створює основу для практичної реалізації системи, її тестування та подальшої демонстрації прикладних можливостей у наступному розділі дипломної роботи.

РОЗДІЛ 3. ПРИКЛАДНІ АСПЕКТИ ЗАСТОСУВАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ДЛЯ ГЕНЕРАЦІЇ КОРИСТУВАЦЬКИХ РЕКОМЕНДАЦІЙ НА ОСНОВІ АНАЛІЗУ ІСТОРИЧНИХ ДАНИХ

3.1. Демонстрація практичних можливостей розробленої системи

У цьому підрозділі наведено демонстрацію практичних можливостей розробленої інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних. Основна увага приділяється не тестуванню окремих помилкових або граничних сценаріїв, а показу того, що система розгортається у хмарному середовищі, приймає каталог об'єктів, накопичує поведінкові події, зберігає їх у різних контурах, формує рекомендації та надає аналітичну інформацію для подальшого контролю якості.

Демонстрація виконувалася на розгорнутій версії системи в AWS. У практичному сценарії були задіяні HTTP API Gateway, Lambda-функції, таблиці DynamoDB для оперативного зберігання каталогу та подій, Amazon S3 для архівування сирих JSON-подій і зберігання CSV-наборів, Amazon Personalize як керований ML-компонент рекомендацій, а також React dashboard для візуального відображення стану системи.

Початковою перевіркою доступності сервісу був виклик endpoint-a /health. API повернув ok = true, назву сервісу recommendation-engine та часову мітку 2026-05-12T18:27:55.159Z. Це підтвердило, що API-рівень доступний, а запит коректно маршрутизується до backend-компоненти. Загальну характеристику демонстраційного запуску системи наведено в таблиці 3.1.

Таблиця 3.1 — Загальна характеристика демонстраційного запуску системи

Показник	Значення
Статус health-check	ok = true
Назва сервісу	recommendation-engine
Час перевірки API	2026-05-12T18:27:55.159Z
Кількість об'єктів каталогу	68

Кінець таблиці 3.1

Показник	Значення
Кількість активних об'єктів	68
Кількість історичних подій	7603
Кількість користувачів з історією	95
Кількість сесій	946
Кількість об'єктів із взаємодіями	64
Активна стратегія рекомендацій	personalize

Наведені показники демонструють, що система працює не лише як ізольований endpoint рекомендацій, а як повний контур збору та використання історичних даних. У демонстраційному середовищі каталог містив 68 активних об'єктів, а поведінкова історія — 7603 події для 95 користувачів. Цього набору достатньо для показу основних прикладних можливостей системи: зберігання каталогу, накопичення подій, формування профілю користувача, генерації рекомендацій і перегляду статистики.

Каталог об'єктів зберігався в DynamoDB ItemsTable. Кожен запис містив itemId, категорію, бренд, ознаку активності, службові metadata-поля та показник популярності. Наявність каталогу в оперативному сховищі потрібна не лише для генерації рекомендацій, а й для збагачення відповіді API: зовнішній сервіс отримує не тільки ідентифікатор рекомендованого об'єкта, а й дані, необхідні для його відображення користувачу. Приклад збереження каталогу об'єктів у DynamoDB ItemsTable наведено на рисунку 3.1.

Table: RecommendationEngineStack-ItemsTable5AAC2C46-4I21PQCZHGGQ - Items returned (50)

Actions

Create item

Scan started on May 12, 2026, 21:33:47

< 1 ... >

☐	itemId (String)	brand	category	isActive	metadata	popularity
☐	sku-4	Sony	headphones	true		75
☐	ml-sku-42	Microsoft	gaming	true	{ "seed": { "...	43
☐	sku-6	Dell	laptops	true		70
☐	sku-8	Samsung	wearables	true		58
☐	ml-sku-22	Dell	laptops	true	{ "seed": { "...	43
☐	ml-sku-35	Bose	audio	true	{ "seed": { "...	51
☐	ml-sku-19	Lenovo	laptops	true	{ "seed": { "...	51
☐	ml-sku-31	Apple	audio	true	{ "seed": { "...	61
☐	ml-sku-47	Nintendo	gaming	true	{ "seed": { "...	51
☐	ml-sku-20	Lenovo	laptops	true	{ "seed": { "...	59
☐	ml-sku-18	Lenovo	laptops	true	{ "seed": { "...	43
☐	ml-sku-24	Dell	laptops	true	{ "seed": { "...	59

Рисунок 3.1 — Збереження каталогу об’єктів у DynamoDB ItemsTable

Статистика каталогу показує, що всі 68 об’єктів мають активний статус. Найбільшими категоріями є laptops і wearables — по 14 об’єктів у кожній. Категорії audio, gaming і phones містять по 12 об’єктів. Найменші категорії — headphones і smartphones, у кожній з яких збережено по 2 об’єкти. Розподіл об’єктів за категоріями наведено на рис. 3.2.



Рисунок 3.2 — Розподіл об’єктів каталогу за категоріями

Цінові характеристики каталогу також використовувалися для аналізу даних і відображення рекомендацій. Мінімальна ціна становила 184, максимальна — 1919, середня — 755,47. Найбільшу кількість об’єктів мав бренд Apple — 20 позицій, що становить 29,41 % демонстраційного каталогу. Наявність категорій, брендів, цін і показників популярності дає системі достатній набір контекстних ознак для fallback-ранжування та для збагачення рекомендаційної відповіді.

Поведінкова історія користувачів зберігалася в DynamoDB EventsTable. Для подій використовувалася структура з partition key за користувачем та sort key за часовою міткою. Це дозволяє швидко отримати останні дії конкретного користувача, що важливо для побудови профілю інтересів і для локальної fallback-логіки. Приклад збереження історичних подій користувачів у DynamoDB EventsTable наведено на рисунку 3.3.

Table: RecommendationEngineStack-EventsTableD24865E5-12XN9MQTWNP5L - Items returned (50)

Actions

Create item

Scan started on May 12, 2026, 21:34:16

< 1 ... > ⚙

☐	pk (String)	sk (String)	eventId	eventType	itemId
☐	USER#manual-test-2...	TS#2026-05-12T18:2...	f0baa0a6-1...	search	
☐	USER#manual-test-2...	TS#2026-05-12T18:2...	474...	view	ml-sku-1
☐	USER#manual-test-2...	TS#2026-05-12T18:2...	5d238986-...	click	ml-sku-2
☐	USER#manual-test-2...	TS#2026-05-12T18:2...	55aa02c5-4...	add_to_cart	ml-sku-3
☐	USER#ml-user-1	TS#2026-04-04T01:0...	ba23dba1-...	view	ml-sku-31
☐	USER#ml-user-1	TS#2026-04-04T01:0...	2bd44942-...	click	ml-sku-31
☐	USER#ml-user-1	TS#2026-04-04T01:0...	051da945-...	view	ml-sku-29
☐	USER#ml-user-1	TS#2026-04-04T01:0...	b00efead-6...	click	ml-sku-29
☐	USER#ml-user-1	TS#2026-04-04T01:0...	2c8014b9-...	add_to_cart	ml-sku-29
☐	USER#ml-user-1	TS#2026-04-04T01:0...	a18fc9b8-f...	purchase	ml-sku-29
☐	USER#ml-user-1	TS#2026-04-04T07:1...	3b55c4cc-6...	view	ml-sku-10
☐	USER#ml-user-1	TS#2026-04-04T07:1...	a01f6517-3...	click	ml-sku-10

Рисунок 3.3 — Збереження історичних подій користувача в DynamoDB EventsTable

Крім оперативного запису в DynamoDB, сири події архівувалися в Amazon S3 у JSON-форматі. Такий підхід розділяє online- та offline-контури системи: DynamoDB використовується для швидкого читання останньої історії користувача, а S3 — для довготривалого зберігання подій, повторної обробки та підготовки CSV-датасетів для Amazon Personalize. Приклад архівування сирих подій користувачів в Amazon S3 наведено на рисунку 3.4.

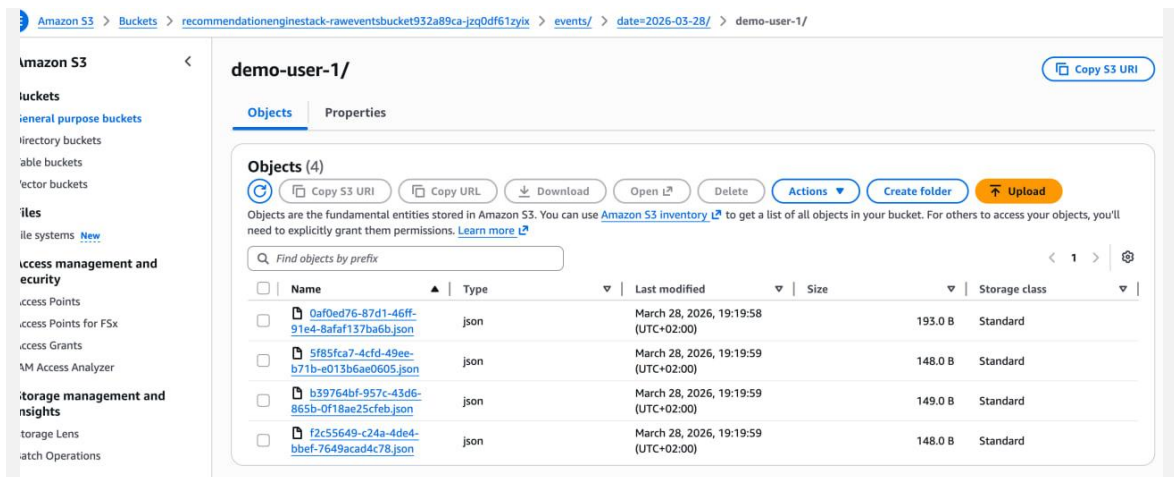


Рисунок 3.4 — Архівування сирих подій користувачів в Amazon S3

За результатами виклику endpoint-а /events/stats система містила 7603 історичні події. Найчастіше зустрічалася подія view — 3164 записи, або 41,62 % від загальної кількості подій. Другою за частотою була подія click — 2656 записів, або 34,93 %. Події add_to_cart становили 1118 записів, purchase — 344 записи, а search — 321 запис. Такий розподіл відповідає типовій логіці поведінкового журналу, у якому перегляди та кліки трапляються частіше, ніж покупки.

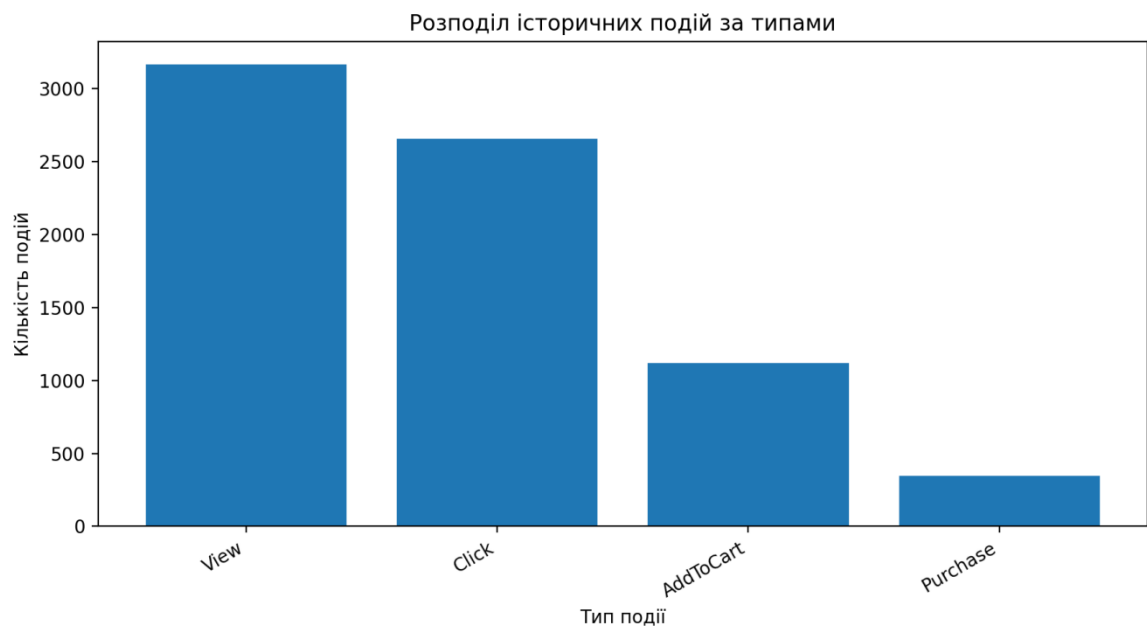


Рисунок 3.5 — Розподіл історичних подій за типами

Окремо було проаналізовано активність користувачів за кількістю interaction-подій. Більшість користувачів у демонстраційному наборі мають від 50 до 109 подій, що дозволяє формувати для них не випадкову, а поведінково обґрунтовану рекомендаційну видачу. Розподіл користувачів за кількістю подій наведено на рис. 3.6.



Рисунок 3.6 — Розподіл користувачів за кількістю подій взаємодії

Для зручної демонстрації результатів було використано React dashboard. Інтерфейс показує статус API, активну стратегію рекомендацій, поточного користувача, загальну кількість об'єктів каталогу, кількість користувачів, кількість подій і картки рекомендованих об'єктів. У демонстраційному запуску dashboard відобразив 68 об'єктів каталогу, понад 7,6 тис. подій та персоналізовану видачу, сформовану стратегією personalize. Приклад відображення рекомендацій у демонстраційному інтерфейсі системи наведено на рисунку 3.7.

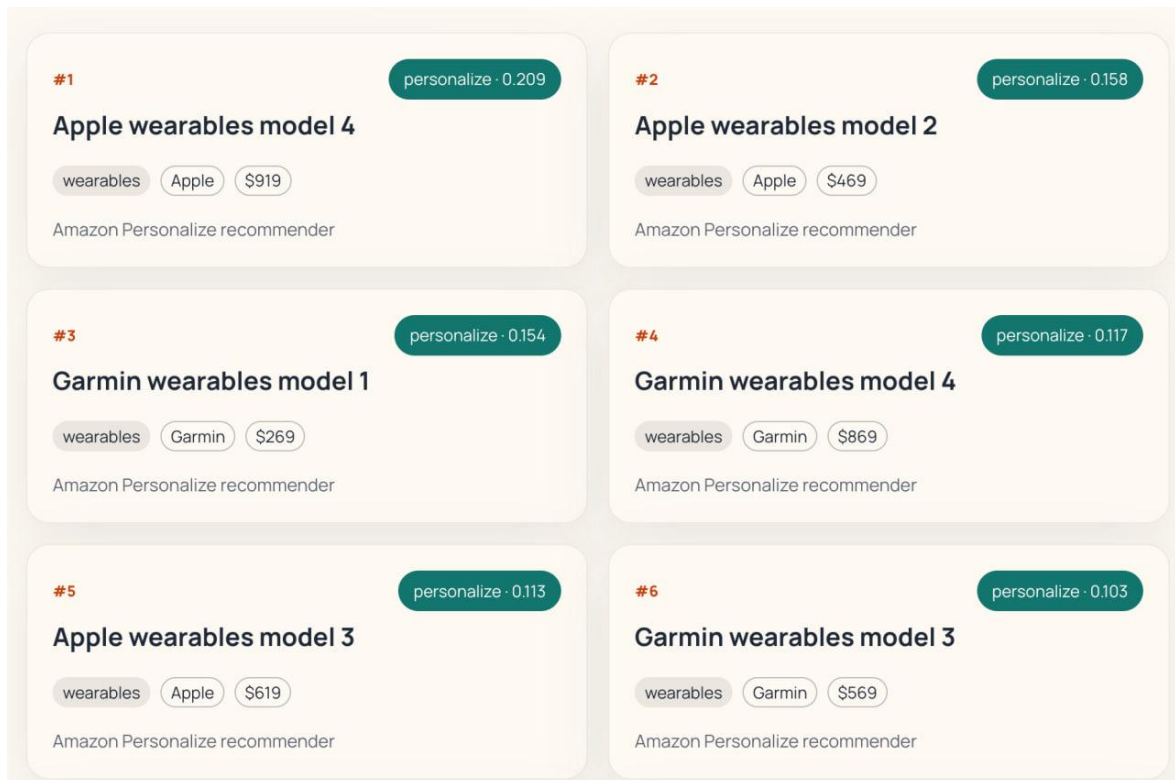


Рисунок 3.7 — Відображення рекомендацій у демонстраційному інтерфейсі системи

У межах демонстрації було підтверджено роботу інтеграції з Amazon Personalize. У відповідях рекомендаційного endpoint-а значення strategy дорівнювало personalize, а кожен рекомендований об'єкт мав reason = Amazon Personalize recommender. Це підтверджує, що система використовувала керований ML-компонент, а не лише локальну fallback-логіку. Водночас fallback-рекомендатор залишається важливою частиною архітектури, оскільки забезпечує роботу API у випадках, коли ML-компонент ще не налаштований або тимчасово недоступний.

Таким чином, практична демонстрація показала, що реалізована програмна компонента має завершений цикл роботи з історичними даними: від завантаження каталогу та прийому подій до формування рекомендацій і відображення результату в dashboard. Система може використовуватися як окремий рекомендаційний сервіс для зовнішнього backend-застосунку, оскільки приховує внутрішні деталі DynamoDB, S3 та Amazon Personalize за єдиним API.

3.2. Результати тестування розробленої системи

Тестування виконувалося через HTTP API та демонстраційний інтерфейс. Для перевірки використовувалися JSON-відповіді endpoint-ів /health, /catalog/stats, /events/stats, /users, /users/{userId}/summary, /recommendations та /events. Основною метою було підтвердити, що ключові сценарії виконуються коректно й утворюють цілісний end-to-end потік даних. Перелік основних перевірених сценаріїв наведено в таблиці 3.2.

Таблиця 3.2 — Перелік перевірених сценаріїв

Сценарій	Очікуваний результат	Фактичний результат
GET /health	API повертає ok = true	Виконано: ok = true, service = recommendation-engine
GET /catalog/stats	Повертається статистика каталогу	Виконано: 68 об'єктів, 68 активних
GET /events/stats	Повертається статистика подій	Виконано: 7603 події, 95 користувачів
GET /users	Повертається список користувачів з історією	Виконано: totalUsers = 95
GET /users/ml-user-1/summary	Повертається короткий профіль користувача	Виконано: 20 останніх подій, 9 унікальних об'єктів

Кінець таблиці 3.2

Сценарій	Очікуваний результат	Фактичний результат
GET /recommendations	Повертається top-N рекомендацій	Виконано: strategy = personalize
POST /events	Нова подія зберігається в системі	Виконано: події search, view, click, add_to_cart прийнято
Повторний GET /recommendations	Рекомендації доступні після оновлення історії	Виконано: отримано нову рекомендаційну відповідь

Для перевірки побудови користувацького профілю було обрано користувача ml-user-1. Endpoint /users/ml-user-1/summary повернув 20 останніх подій, пов'язаних з 9 унікальними об'єктами. У недавній історії цього користувача переважали події view і click, а серед категорій найчастіше траплялися wearables, phones, audio та laptops. Зведення останньої активності користувача ml-user-1 наведено в таблиці 3.3.

Таблиця 3.3 — Зведення останньої активності користувача ml-user-1

Показник	Значення
Кількість останніх подій	20
Кількість унікальних об'єктів	9
Розподіл типів подій	view — 10; click — 8; add_to_cart — 1; search — 1
Основні категорії	wearables — 5; phones — 2; audio — 1; laptops — 1
Основні бренди	Apple — 5; Samsung — 3; Dell — 1
Ціновий діапазон останніх об'єктів	min = 239, max = 919, avg = 617,89

Основним тестом рекомендаційного сценарію був запит до /recommendations для користувача ml-user-1 з параметрами limit = 10 і debug = true. У відповіді система повернула strategy = personalize, recommendationId RID-d0-4f17-9310-7bc100026f7b-CID-f74e74 та 10 рекомендованих об'єктів. Усі елементи відповіді були збагачені даними каталогу: назвою, категорією, брендом, ціною та

поясненням джерела рекомендації. Рекомендаційну видачу для користувача ml-user-1 наведено в таблиці 3.4.

Таблиця 3.4 — Рекомендаційна видача для користувача ml-user-1

№	Item ID	Назва	Категорія	Бренд	Ціна	Score
1	ml-sku-52	Apple wearables model 4	wearables	Apple	919	0.2042
2	ml-sku-50	Apple wearables model 2	wearables	Apple	469	0.1636
3	ml-sku-57	Garmin wearables model 1	wearables	Garmin	269	0.1510
4	ml-sku-60	Garmin wearables model 4	wearables	Garmin	869	0.1222
5	ml-sku-51	Apple wearables model 3	wearables	Apple	619	0.1146
6	ml-sku-59	Garmin wearables model 3	wearables	Garmin	569	0.1011
7	ml-sku-53	Samsung wearables model 1	wearables	Samsung	239	0.0641
8	ml-sku-56	Samsung wearables model 4	wearables	Samsung	839	0.0459
9	ml-sku-43	Microsoft gaming model 3	gaming	Microsoft	709	0.0032
10	ml-sku-45	Nintendo gaming model 1	gaming	Nintendo	314	0.0027

Результати показують, що перші позиції рекомендаційної видачі належать переважно до категорії wearables. З 10 рекомендованих об'єктів 8 належать до цієї категорії, а ще 2 — до категорії gaming. Найбільше у видачі представлено бренди Apple і Garmin — по 3 об'єкти кожного. Score-статистика для top-10 мала такі значення: мінімум — 0,0027, максимум — 0,2042, середнє — 0,0973.

У тестовому запуску debug-блок рекомендацій повернув evaluation-структуру з профільними метриками Precision@K, Recall@K, NDCG@K і MAP@K. Для оцінювання використовувався режим profile_catalog, у якому релевантність визначається на основі відповідності рекомендацій профілю користувача та пов'язаним з ним категоріям каталогу. У поточному прикладі система визначила 37 релевантних об'єктів, а найактуальнішою категорією користувача стала laptops.

Отримані результати показують, що перші позиції рекомендаційної видачі добре узгоджуються з профілем користувача: для K=1 і K=3 значення Precision, NDCG та MAP дорівнюють 1.000, тобто всі перші рекомендації є релевантними та

розташовані у верхній частині списку. Для $K=5$ Precision становить 0.800, а для $K=10$ — 0.875, що свідчить про збереження високої частки релевантних об'єктів навіть при розширенні рекомендаційного списку. Значення $\text{Recall}@K$ зростає від 0.027 при $K=1$ до 0.189 при $K=10$, що є очікуваним, оскільки зі збільшенням кількості рекомендацій система охоплює більшу частину релевантного каталогу.

Додатково debug-блок зафіксував $\text{Category match} = 1.000$, тобто рекомендовані об'єкти повністю відповідають основній категорії інтересу користувача. Середній score рекомендацій становив 0.077. Отримані показники підтверджують, що механізм профільної оцінки працює коректно та може використовуватися для демонстраційного аналізу релевантності рекомендаційної видачі. Водночас ці метрики слід розглядати саме як debug-оцінювання на основі профілю користувача, а не як фінальну offline-оцінку моделі на відкладеній тестовій вибірці. Поточний стан debug-метрик рекомендаційної видачі наведено на рисунку 3.8.

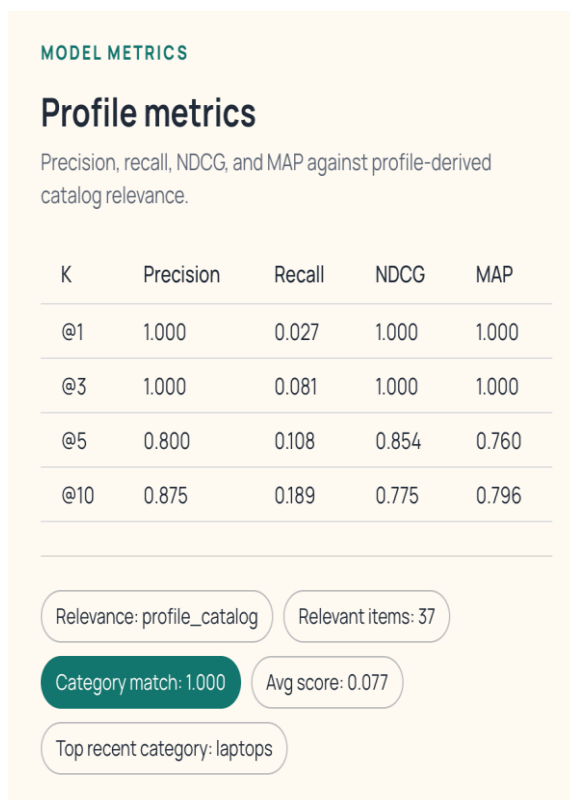


Рисунок 3.8 — Поточний стан debug-метрик рекомендаційної видачі

Для перевірки прийому нових подій було створено тестового користувача manual-test-20260512-212839. Спочатку для нього було отримано базовий список рекомендацій, після чого в систему були передані події search, view, click і add_to_cart. Результати відповідей endpoint-a /events наведено в табл. 3.5.

Таблиця 3.5 — Результати передавання ручних подій тестового користувача

Тип події	Event ID	personalizeSynced	Пояснення
search	f0baa0a6-1f7c-465e-8772-c70fdd3885d2	false	Подія прийнята, але не синхронізується з Personalize як item-взаємодія, оскільки не містить itemId
view	474cb565-62e1-4a87-9ad5-81ed89b7ee1f	true	Подія прийнята й синхронізована з Amazon Personalize
click	5d238986-3755-4139-8505-7e113f25f2ff	true	Подія прийнята й синхронізована з Amazon Personalize
add_to_cart	55aa02c5-4aab-4255-85cf-7c9f8ec2596d	true	Подія прийнята й синхронізована з Amazon Personalize

Важливо, що для події search значення personalizeSynced дорівнює false. Це не є помилкою обробки події: пошуковий запит зберігається локально в історії користувача, але не передається до Amazon Personalize як item-взаємодія, оскільки для такої синхронізації потрібен конкретний itemId. Натомість події view, click і add_to_cart були синхронізовані з Personalize. Порівняння рекомендацій до та після появи нових подій наведено в таблиці 3.6.

Таблиця 3.6 — Порівняння рекомендацій до та після появи нових подій

Етап	Стратегія	Recommendation ID	Топ-5 рекомендацій
До передавання подій	personalize	RID-f9...CID-f74e74	ml-sku-17 (laptops, 0.0399); ml-sku-14 (laptops, 0.0304); ml-sku-50 (wearables, 0.0295); ml-sku-18 (laptops, 0.0293); ml-sku-13 (laptops, 0.0290)
Після передавання подій	personalize	RID-22...CID-f74e74	ml-sku-49 (wearables, 0.1415); ml-sku-52 (wearables, 0.0960); ml-sku-50 (wearables, 0.0896); ml-sku-58 (wearables, 0.0879); ml-sku-57 (wearables, 0.0834)

Порівняння рекомендацій до та після передачі нових подій показує зміну складу топ-5 видачі. Початковий список для нового користувача містив об'єкти з категорій laptops і wearables, тоді як після появи ручних подій система повернула п'ять об'єктів з категорії wearables. Такий результат не слід трактувати як повноцінну статистичну оцінку якості моделі, однак він підтверджує, що система приймає нові події, зберігає їх у профілі користувача та продовжує формувати рекомендаційну відповідь через активну стратегію personalize. Розподіл оцінок рекомендацій після накопичення нових подій користувача наведено на рисунку 3.9.

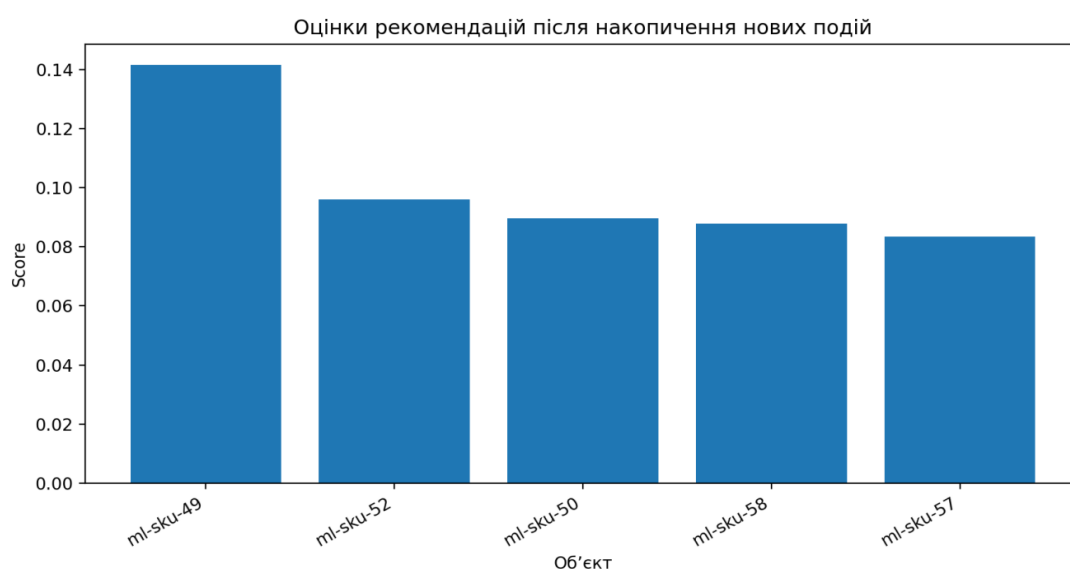


Рисунок 3.9 — Оцінки рекомендацій після накопичення нових подій користувача

Порівняння категорій рекомендацій до та після передавання нових подій користувача наведено на рисунку 3.10.

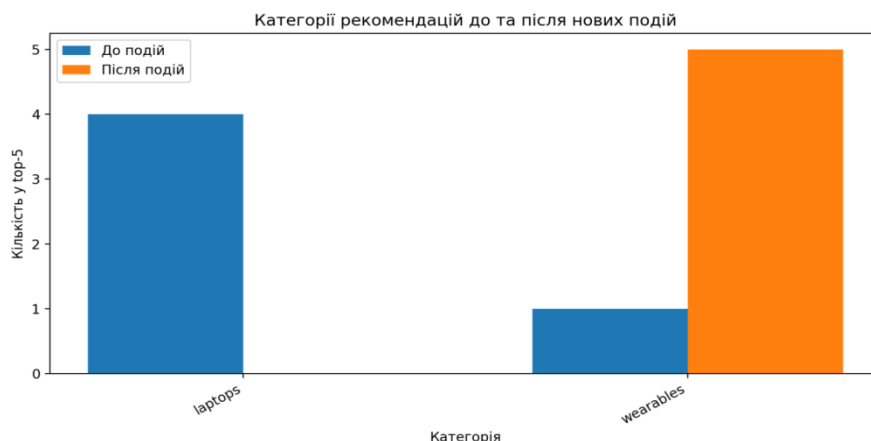


Рисунок 3.10 — Категорії рекомендацій до та після передавання нових подій користувача

Після накопичення нових подій повторний debug-запит для користувача manual-test-20260512-212839 зафіксував 4 останні події, 3 унікальні об'єкти, категорію phones як основну категорію останніх взаємодій і бренд Apple як основний бренд у профілі. У рекомендаційній відповіді всі п'ять об'єктів належали до категорії wearables, а серед рекомендованих брендів були Apple та Garmin. Score-статистика для top-5 після подій становила: мінімум — 0,0834, максимум — 0,1415, середнє — 0,0997.

За результатами прикладного тестування підтверджено, що основні сценарії системи виконуються коректно: API доступний, каталог і події зберігаються, статистика формується, профіль користувача будується на основі останніх взаємодій, рекомендаційний endpoint повертає структуровану відповідь, а item-події можуть синхронізуватися з Amazon Personalize. Поточне обмеження стосується debug-метрик: їхня структура вже присутня у відповіді API, однак фактичні значення потребують повторної перевірки після виправлення механізму розрахунку або формування релевантного ground truth.

Отже, тестування показало, що розроблена система придатна до демонстраційного використання як рекомендаційна backend-компонента. Вона

підтримує повний шлях даних від події користувача до рекомендаційної відповіді та має достатню кількість службових endpoint-ів для контролю стану даних.

3.3. Програмна документація на програмне забезпечення

Розроблена інтелектуальна система рекомендацій реалізована як окрема програмна компонента, що інтегрується з прикладними backend-сервісами через HTTP API. Її призначення полягає у прийомі каталогу об'єктів, накопиченні історичних подій користувачів, зберіганні даних в оперативному та архівному контурах, формуванні рекомендацій і наданні статистичної інформації для аналізу роботи системи.

Програмне забезпечення побудоване на базі хмарної інфраструктури AWS. Для опису ресурсів використовується AWS CDK, основна backend-логіка реалізована у вигляді AWS Lambda-функцій, зовнішній доступ забезпечується через API Gateway. Для оперативного зберігання каталогу та подій використовується DynamoDB, для архівування сирих подій і CSV-датасетів — Amazon S3. Персоналізовані рекомендації формуються за допомогою Amazon Personalize, а у випадку недоступності ML-компонента система переходить до fallback-рекомендатора. Основні компоненти програмного забезпечення наведено в таблиці 3.7.

Таблиця 3.7 — Основні компоненти програмного забезпечення

Компонент	Призначення
API Gateway	Прийом зовнішніх HTTP-запитів до рекомендаційної системи
AWS Lambda	Обробка запитів до каталогу, подій, рекомендацій і статистики
DynamoDB ItemsTable	Оперативне зберігання об'єктів каталогу
DynamoDB EventsTable	Зберігання історичних подій користувачів
Amazon S3	Архівування сирих подій і зберігання CSV-датасетів

Кінець таблиці 3.7

Компонент	Призначення
Amazon Personalize	Формування персоналізованих рекомендацій
Fallback-рекомендатор	Резервна логіка рекомендацій без ML-компонента
React dashboard	Візуальна демонстрація роботи системи

Для запуску проєкту необхідно мати встановлені Node.js, npm, AWS CLI та AWS CDK. Також потрібен налаштований AWS-профіль з правами на створення Lambda-функцій, API Gateway, DynamoDB-таблиць, S3-бакетів, IAM-ролей і ресурсів Amazon Personalize.

Початкова підготовка проєкту виконується через встановлення залежностей, перевірку збірки та синтез CDK-шаблону. Після цього виконується розгортання інфраструктури у вибраному AWS-акаунті та регіоні. Приклад команд розгортання інфраструктури наведено нижче:

```
npm install
npm run ui:install
npm run build
npm run cdk:synth
npm run cdk:deploy -- --outputs-file cdk-outputs.json
```

Після успішного розгортання у файлі cdk-outputs.json зберігаються основні вихідні параметри системи: URL API, назви DynamoDB-таблиць і назва S3-бакета. Ці значення переносяться у файл .env, який використовується для запуску службових скриптів, генерації демонстраційних даних та інтеграції з Amazon Personalize. Основні змінні середовища системи наведено в таблиці 3.8.

Таблиця 3.8 — Основні змінні середовища

Змінна	Призначення
AWS_REGION	Регіон AWS, у якому розгортається система
API_URL	URL розгорнутого API Gateway
ITEMS_TABLE_NAME	Назва таблиці каталогу

Кінець таблиці 3.8

Змінна	Призначення
EVENTS_TABLE_NAME	Назва таблиці подій
RAW_EVENTS_BUCKET_NAME	Назва S3-бакета для сирих подій
PERSONALIZE_IMPORT_ROLE_ARN	IAM-роль для імпорту даних в Amazon Personalize
PERSONALIZE_RECOMMENDER_ARN	ARN рекомендаційного ресурсу Amazon Personalize
PERSONALIZE_TRACKING_ID	Ідентифікатор event tracker для передавання подій

Для наповнення системи демонстраційними даними використовується окремий seed-скрипт. Після його виконання працездатність API можна перевірити через endpoint /health. Приклад запуску seed-скрипта та перевірки працездатності API наведено нижче:

```
API_URL="$API_URL" npm run seed:demo
```

```
curl "$API_URL/health"
```

Розроблена система має набір endpoint-ів, які забезпечують повний цикл роботи з рекомендаціями: завантаження каталогу, прийом подій, отримання рекомендацій, перегляд статистики та аналіз активності користувачів. Основні API endpoint-и системи наведено в таблиці 3.9.

Таблиця 3.9 — Основні API endpoint-и системи

Endpoint	Метод	Призначення
/health	GET	Перевірка працездатності API
/catalog/items	POST	Додавання або оновлення об'єктів каталогу
/catalog/items	GET	Отримання об'єктів каталогу
/events	POST	Передавання події користувача
/recommendations	GET	Отримання рекомендацій для користувача
/catalog/stats	GET	Отримання статистики каталогу

Кінець таблиці 3.9

Endpoint	Метод	Призначення
/events/stats	GET	Отримання статистики подій
/users	GET	Отримання списку користувачів
/users/{userId}/summary	GET	Отримання короткого профілю активності користувача

Для додавання події користувача зовнішній сервіс надсилає POST-запит до /events. Подія повинна містити userId, eventType, а також itemId для об'єктних подій або query для пошукової події. Якщо eventId або timestamp не передані, система формує їх автоматично. Приклад передавання події користувача через endpoint /events наведено нижче:

```
curl -X POST "$API_URL/events" \
  -H "content-type: application/json" \
  -d '{
    "userId": "demo-user-1",
    "sessionId": "session-1",
    "eventType": "view",
    "itemId": "ml-sku-1"
  }'
```

Після прийому події система виконує її валідацію, зберігає нормалізований запис у DynamoDB і додатково архівує сирий JSON-об'єкт в Amazon S3. Якщо налаштований Amazon Personalize Event Tracker, item-подія також може бути передана до Amazon Personalize.

Для отримання рекомендацій використовується endpoint /recommendations. Основним параметром є userId, а параметр limit визначає кількість рекомендованих об'єктів у відповіді. Приклад отримання рекомендацій через endpoint /recommendations наведено нижче:

```
"$API_URL/recommendations?userId=demo-user-1&limit=10"
```

У відповідь система повертає список об'єктів із полями itemId, score, source, reason, title, category, brand і price. Поле source показує, якою стратегією

сформовано рекомендацію: `personalize` або `fallback`. Це дає змогу відстежувати, чи використовується Amazon Personalize, чи система працює в резервному режимі.

Для аналізу роботи рекомендаційного механізму передбачено `debug`-режим. У цьому режимі API повертає не лише рекомендації, а й додаткову інформацію про профіль користувача, останні події, рекомендовані категорії, середній score і метрики релевантності. Приклад виклику `debug`-режиму рекомендаційного API наведено нижче:

```
curl "$API_URL/recommendations?userId=demo-user-1&limit=10&debug=true"
```

Для підготовки даних до Amazon Personalize у проєкті передбачені службові `npm`-скрипти. Вони генерують або експортують CSV-файли `items.csv` та `interactions.csv`, завантажують їх в Amazon S3 і запускають процес імпорту в Amazon Personalize. Основні `npm`-скрипти проєкту наведено в таблиці 3.10.

Таблиця 3.10 — Основні `npm`-скрипти проєкту

Команда	Призначення
<code>npm run build</code>	Компіляція TypeScript-коду
<code>npm run cdk:synth</code>	Перевірка CDK-шаблону
<code>npm run cdk:deploy</code>	Розгортання AWS-інфраструктури
<code>npm run seed:demo</code>	Завантаження демонстраційних даних
<code>npm run personalize:generate</code>	Генерація даних для Amazon Personalize
<code>npm run personalize:export</code>	Експорт CSV-датасетів
<code>npm run personalize:import</code>	Імпорт датасетів в Amazon Personalize
<code>npm run ui:dev</code>	Запуск демонстраційного React-інтерфейсу
<code>npm run ui:build</code>	Збірка демонстраційного інтерфейсу

Окремо в системі реалізовано демонстраційний React-інтерфейс. Він використовується для візуальної перевірки `backend`-компоненти: перегляду кількості об'єктів каталогу, кількості історичних подій, списку користувачів,

поточного користувача, рекомендаційної видачі, профілю активності та debug-метрик. Запуск інтерфейсу виконується командою `npm run ui:dev`.

У процесі експлуатації система працює за таким сценарієм. Спочатку зовнішній backend або адміністратор передає каталог об'єктів через API. Далі під час взаємодії користувача із зовнішнім сервісом backend надсилає події до endpoint-a `/events`. Система накопичує ці події в DynamoDB і S3. Після цього зовнішній сервіс звертається до endpoint-a `/recommendations` і отримує top-N список об'єктів для конкретного користувача. Якщо Amazon Personalize доступний, рекомендації формуються ML-компонентом, а якщо він недоступний, використовується fallback-рекомендатор.

Поточна реалізація має кілька обмежень. Якість персоналізації залежить від обсягу накопичених історичних даних, тому для нових користувачів або користувачів із малою кількістю подій система використовує fallback-логіку. Також у межах поточного MVP не реалізовано повноцінний production-рівень авторизації API, тому для комерційного використання доцільно додати API-ключі, JWT або інший механізм контролю доступу. Крім того, A/B-тестування та автоматичне перенавчання моделі можуть бути реалізовані як наступні етапи розвитку.

Отже, програмна документація визначає порядок підготовки, запуску, розгортання та використання розробленої рекомендаційної системи. Описані команди, змінні середовища, API endpoint-и та сценарії експлуатації дозволяють відтворити роботу системи, перевірити її основні функції та інтегрувати рекомендаційну компоненту із зовнішнім backend-сервісом.

					ІАЛЦ.467200.003 ПЗ	Арк.
						104
Зм.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВОК ДО РОЗДІЛУ

У третьому розділі було розглянуто прикладні аспекти застосування розробленої інтелектуальної системи для генерації користувацьких рекомендацій на основі аналізу історичних даних. Основну увагу приділено практичній демонстрації роботи системи, перевірці її основних сценаріїв, аналізу результатів тестування та підготовці програмної документації, необхідної для розгортання, налаштування й подальшого використання програмного забезпечення.

У межах демонстрації практичних можливостей було підтверджено, що система успішно розгортається у хмарному середовищі AWS та працює як окрема рекомендаційна backend-компонента. Під час демонстраційного запуску було перевірено доступність API через endpoint /health, завантаження та зберігання каталогу об'єктів, накопичення історичних подій користувачів, архівування сирих даних в Amazon S3, використання DynamoDB для оперативного зберігання, а також формування рекомендацій через активну стратегію personalize. Отримані результати показали, що реалізована система підтримує повний цикл роботи з історичними даними: від прийому подій до формування рекомендаційної відповіді.

Окремо було проаналізовано структуру демонстраційного набору даних. Каталог містив 68 активних об'єктів, а історія взаємодій — 7603 події для 95 користувачів. Розподіл подій за типами показав, що найбільшу частку становлять перегляди та кліки, що є типовим для поведінкових журналів цифрових сервісів. Наявність подій різних типів, зокрема view, click, add_to_cart, purchase і search, дала змогу перевірити, що система здатна працювати з різними формами неявного зворотного зв'язку та використовувати їх для побудови користувацького профілю.

У процесі тестування було перевірено ключові endpoint-и системи: /health, /catalog/stats, /events/stats, /users, /users/{userId}/summary, /recommendations та /events. Результати підтвердили, що API коректно повертає статистику каталогу й подій, формує короткий профіль користувача, генерує top-N рекомендації та приймає нові події. Для користувача ml-user-1 система повернула персоналізовану рекомендаційну видачу з активною стратегією personalize, а всі рекомендовані

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		105

об'єкти були збагачені даними каталогу: назвою, категорією, брендом, ціною та оцінкою релевантності.

Важливим результатом тестування стала перевірка роботи системи після надходження нових подій. Для тестового користувача було передано події `search`, `view`, `click` і `add_to_cart`, після чого виконано повторний запит рекомендацій. Порівняння результатів до та після появи нових взаємодій показало зміну складу рекомендаційної видачі, що підтверджує працездатність `end-to-end` сценарію: система приймає події, зберігає їх, оновлює історію користувача та продовжує формувати рекомендації через активний рекомендаційний механізм.

У межах `debug`-режиму було перевірено можливість аналізу рекомендаційної відповіді за допомогою додаткових службових даних і метрик. Система повертає інформацію про профіль користувача, рекомендовані категорії, джерело рекомендацій та показники оцінювання. Водночас зазначено, що `debug`-метрики слід розглядати як допоміжний інструмент для демонстраційного аналізу, а не як фінальну `offline`-оцінку моделі на відкладеній тестовій вибірці. Це є важливим обмеженням, оскільки повноцінне оцінювання якості рекомендацій потребує окремого експериментального сценарію з коректним формуванням `ground truth`.

У підрозділі програмної документації було описано призначення системи, її архітектурну структуру, основні компоненти, порядок підготовки середовища, розгортання, запуску `backend`-частини, роботи з демонстраційним інтерфейсом та використання `API`. Документація підтверджує, що розроблена система має чітко визначений порядок використання й може бути повторно розгорнута завдяки застосуванню `AWS CDK` та конфігурації через змінні середовища. Це підвищує практичну цінність програмного забезпечення, оскільки система може бути перенесена в інше середовище або адаптована до іншого прикладного `backend`-сервісу.

У результаті виконання третього розділу було підтверджено, що розроблена інтелектуальна система не обмежується теоретичною моделлю або окремим алгоритмом рекомендацій, а є працездатною програмною компонентою з `API`,

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		106

сховищами даних, ML-інтеграцією, fallback-логікою, статистикою, debug-режимом і демонстраційним інтерфейсом. Система здатна приймати каталог і поведінкові події, накопичувати історичні дані, використовувати їх для формування рекомендацій, а також надавати службову інформацію для аналізу й контролю її роботи.

Отже, результати третього розділу підтверджують досягнення практичної мети дипломного проєкту. Розроблена система може бути використана як основа для впровадження рекомендаційного механізму в цифрові сервіси, що мають каталог об'єктів і накопичують історію взаємодій користувачів. Вона забезпечує інтеграцію через HTTP API, підтримує зберігання даних в оперативному та архівному контурах, використовує Amazon Personalize для персоналізованої видачі та зберігає працездатність завдяки fallback-рекомендатору. Отримані результати створюють підстави для загальних висновків дипломної роботи щодо доцільності та практичної придатності запропонованого підходу до побудови інтелектуальної системи користувацьких рекомендацій.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		107

ВИСНОВКИ

У дипломному проєкті розроблено інтелектуальну систему для генерації користувацьких рекомендацій на основі аналізу історичних даних. У межах роботи було досліджено теоретичні основи побудови рекомендаційних систем, проаналізовано підходи до зберігання та обробки поведінкових даних, спроектовано архітектуру програмної компоненти, реалізовано рекомендаційний backend-сервіс, проведено демонстрацію його практичних можливостей і виконано тестування основних сценаріїв роботи.

У першому розділі було проаналізовано існуючі технологічні рішення для зберігання, пошуку та подальшої обробки історичних даних користувачів. Розглянуто реляційні бази даних, NoSQL-рішення, об'єктні сховища, пошукові системи та аналітичні сховища. За результатами аналізу встановлено, що для поставленої задачі доцільним є поєднання оперативного сховища для швидкого доступу до подій та архівного сховища для довготривалого зберігання сирих даних і підготовки наборів для машинного навчання. Такий підхід дозволяє одночасно забезпечити швидку роботу API та зберегти можливість подальшої аналітики, повторної обробки й навчання рекомендаційних моделей.

Також у роботі було розглянуто основні методології формування користувацьких рекомендацій. Проаналізовано підходи на основі популярності, контентної схожості, колаборативної фільтрації, гібридних моделей, session-based та deep learning підходів. У результаті встановлено, що вибір рекомендаційного методу повинен залежати не лише від потенційної точності моделі, а й від обсягу доступних історичних даних, типу взаємодій, наявності проблеми холодного старту, вимог до затримки відповіді та можливості практичного впровадження. Для задачі дипломного проєкту найбільш доцільною є постановка top-N ранжування на основі неявного зворотного зв'язку, оскільки система працює з подіями перегляду, кліку, пошуку, додавання до кошика та покупки.

У другому розділі було спроектовано архітектуру інтелектуальної системи рекомендацій. Систему визначено як окрему програмну компоненту, що може

					ІАЛЦ.467200.003 ПЗ	Арк.
						108
Зм.	Арк.	№ докум.	Підпис	Дата		

інтегруватися з різними прикладними backend-сервісами через HTTP API. Такий підхід дозволяє не змінювати основну бізнес-логіку зовнішнього застосунку, а винести функції збору подій, зберігання історії, підготовки даних і генерації рекомендацій в окремий рекомендаційний сервіс.

Для реалізації системи було обрано хмарну serverless-архітектуру на базі AWS. Основними компонентами стали API Gateway, AWS Lambda, DynamoDB, Amazon S3, Amazon Personalize, fallback-рекомендатор, AWS CDK та демонстраційний React dashboard. API Gateway забезпечує зовнішній доступ до системи, Lambda-функції виконують прикладну логіку, DynamoDB використовується для оперативного зберігання каталогу та подій, Amazon S3 — для архівування сирих даних і CSV-наборів, а Amazon Personalize — для формування персоналізованих рекомендацій. Fallback-рекомендатор забезпечує працездатність системи у випадках, коли ML-компонент недоступний або історичних даних недостатньо.

У межах інженерії вимог було сформовано функціональні та нефункціональні вимоги до системи. До основних функціональних вимог належать прийом каталогу, прийом і валідація подій користувачів, зберігання історичних даних, формування top-N рекомендацій, підтримка fallback-логіки, підготовка датасетів для Amazon Personalize, отримання статистики та робота debug-режиму. Нефункціональні вимоги охоплюють масштабованість, низьку затримку відповіді, відмовостійкість, спостережуваність, підтримуваність, розширюваність і повторюваність розгортання. Це дало змогу розглядати систему не як ізольований алгоритм, а як повноцінне програмне забезпечення, придатне до практичного використання та подальшого розвитку.

Програмну компоненту було спроектовано за модульним принципом. У системі виділено API-рівень, Lambda-обробники, сервісний шар, моделі даних, інфраструктурний шар, batch-скрипти та демонстраційний інтерфейс. Такий поділ зменшує зв'язаність компонентів і спрощує подальшу підтримку системи. Окремі модулі відповідають за валідацію даних, роботу з DynamoDB, інтеграцію з Amazon

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		109

Personalize, fallback-рекомендації, розрахунок метрик, статистику та debug-аналіз. Завдяки цьому систему можна розширювати без повної зміни її зовнішнього API.

У третьому розділі було продемонстровано практичну роботу реалізованої системи. Під час демонстраційного запуску підтверджено доступність API, коректність зберігання каталогу, накопичення історичних подій, архівування сирих даних в Amazon S3, використання DynamoDB для оперативного доступу та роботу рекомендаційного endpoint-a. У тестовому середовищі каталог містив 68 активних об'єктів, а історія взаємодій — 7603 події для 95 користувачів. Це підтвердило, що система працює з повним циклом даних: від прийому каталогу та подій до формування рекомендацій і відображення результатів у dashboard.

Під час тестування було перевірено основні endpoint-и системи: /health, /catalog/stats, /events/stats, /users, /users/{userId}/summary, /recommendations та /events. Результати тестування показали, що система коректно повертає статистику каталогу й подій, формує профіль користувача, генерує рекомендаційну відповідь, приймає нові події та продовжує працювати після оновлення історії користувача. Для рекомендаційного сценарію підтверджено роботу активної стратегії personalize, а item-події типу view, click і add_to_cart були синхронізовані з Amazon Personalize.

Окремо було перевірено зміну рекомендаційної видачі після появи нових подій користувача. Порівняння результатів до та після передавання подій показало, що система здатна приймати нові поведінкові сигнали, зберігати їх у профілі користувача та формувати оновлену рекомендаційну відповідь. Це підтвердило працездатність end-to-end сценарію, у якому зовнішній сервіс передає події до рекомендаційної системи, а система використовує накопичену історію для персоналізованої видачі.

У межах програмної документації було описано призначення системи, її архітектуру, основні компоненти, структуру API, порядок підготовки середовища, розгортання інфраструктури, запуску backend-частини та використання демонстраційного інтерфейсу. Наявність такої документації підвищує практичну

					ІАЛЦ.467200.003 ПЗ	Арк.
						110
Зм.	Арк.	№ докум.	Підпис	Дата		

цінність розробки, оскільки дає змогу повторно розгорнути систему, перевірити її функції та інтегрувати рекомендаційну компоненту з іншим backend-сервісом.

Поточна реалізація має певні обмеження, які визначають напрями подальшого розвитку. Якість персоналізації залежить від кількості накопичених історичних даних, тому для нових користувачів або користувачів із малою кількістю подій важливим залишається fallback-рекомендатор. Debug-метрики можуть використовуватися для демонстраційного аналізу, однак для повноцінної оцінки якості моделі доцільно реалізувати окрему offline-оцінку на відкладеній тестовій вибірці. Для продукційного використання також доцільно додати повноцінну авторизацію API, A/B-тестування, автоматичне перенавчання моделі та розширений моніторинг.

Отже, мету дипломного проєкту досягнуто. У результаті роботи спроектовано та реалізовано інтелектуальну систему генерації користувацьких рекомендацій на основі аналізу історичних даних, яка забезпечує прийом поведінкових подій, зберігання каталогу й історії взаємодій, формування персоналізованих рекомендацій, підтримку fallback-логіки, інтеграцію з Amazon Personalize та можливість підключення до зовнішніх backend-сервісів через API. Практична цінність розробленого рішення полягає в тому, що воно може бути використане як основа для впровадження рекомендаційних механізмів у цифрові сервіси, які потребують персоналізації, але не мають власної складної ML-інфраструктури.

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		111

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація PostgreSQL: JSON Types // PostgreSQL Documentation. URL: <https://www.postgresql.org/docs/current/datatype-json.html> (дата звернення: 14.05.2026).
2. Amazon DynamoDB Documentation // Amazon Web Services. URL: <https://aws.amazon.com/ru/documentation-overview/dynamodb/> (дата звернення: 16.05.2026).
3. Amazon S3 as a Data Lake Storage Platform // Amazon Web Services. URL: https://docs.aws.amazon.com/pt_br/whitepapers/latest/building-data-lakes/amazon-s3-data-lake-storage-platform.html (дата звернення: 11.05.2026).
4. Vector Search // Amazon OpenSearch Service Developer Guide. URL: <https://docs.aws.amazon.com/opensearch-service/latest/developerguide/vector-search.html> (дата звернення: 09.05.2026).
5. Recommender Systems Handbook // Springer. URL: <https://link.springer.com/book/10.1007/978-1-4899-7637-6> (дата звернення: 18.05.2026).
6. Ricci F., Rokach L., Shapira B. Recommender Systems: The Textbook // Springer. URL: <https://link.springer.com/book/10.1007/978-3-319-29659-3> (дата звернення: 20.05.2026).
7. Working with Predefined Recipes // Amazon Personalize Developer Guide. URL: <https://docs.aws.amazon.com/personalize/latest/dg/working-with-predefined-recipes.html> (дата звернення: 08.05.2026).
8. Recommender Systems: The Textbook // Academia.edu. URL: https://www.academia.edu/42933732/Recommender_Systems_The_Textbook (дата звернення: 15.05.2026).

9. A Comprehensive Survey of Neighborhood-Based Recommendation Methods // Elsevier Pure. URL: <https://ohiostate.elsevierpure.com/en/publications/a-comprehensive-survey-of-neighborhood-based-recommendation-metho/> (дата звернення: 13.05.2026).
10. Adomavicius G., Tuzhilin A. Toward the Next Generation of Recommender Systems // IEEE Computer Society. URL: <https://doi.org/10.1109/MC.2009.263> (дата звернення: 21.05.2026).
11. Recommender Systems [Netflix] // DataJobs. URL: <https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf> (дата звернення: 10.05.2026).
12. Factorization Machines // IEEE International Conference on Data Mining. URL: <https://doi.org/10.1109/ICDM.2010.127> (дата звернення: 17.05.2026).
13. Rendle S. Factorization Machines // ISMLL. URL: <https://www.ismll.uni-hildesheim.de/pub/pdfs/Rendle2010FM.pdf> (дата звернення: 14.05.2026).
14. Session-Based Recommendations with Transformer Models // IJCAI Proceedings. URL: <https://www.ijcai.org/proceedings/2021/630> (дата звернення: 22.05.2026).
15. Wide & Deep Learning for Recommender Systems // arXiv. URL: <https://arxiv.org/abs/1511.06939> (дата звернення: 06.05.2026).
16. Neural Collaborative Filtering // IEEE International Conference on Data Mining. URL: <https://doi.org/10.1109/ICDM.2018.00035> (дата звернення: 18.05.2026).
17. BERT4Rec: Sequential Recommendation with Bidirectional Encoder Representations // ACM Digital Library. URL: <https://doi.org/10.1145/3357384.3357895> (дата звернення: 07.05.2026).
18. Attention Is All You Need // arXiv. URL: <https://arxiv.org/abs/1706.03762> (дата звернення: 12.05.2026).
19. Session-Based Recommendations with Recurrent Neural Networks // ACM Digital Library. URL: <https://doi.org/10.1145/2740908.2742726> (дата звернення: 05.05.2026).

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		113

20. Neural Attentive Session-Based Recommendation // ACM Digital Library. URL: <https://doi.org/10.1145/3038912.3052569> (дата звернення: 09.05.2026).
21. Session-Based Recommendations with Recurrent Neural Networks // ACM Digital Library. URL: <https://dl.acm.org/doi/10.1145/2740908.2742726> (дата звернення: 16.05.2026).
22. A Comprehensive Survey of Neighborhood-Based Recommendation Methods // McAuley Lab. URL: https://mcauleylab.ucsd.edu/public_datasets/pml_backup/PML%20book/references/A_Comprehensive_Survey_of_Neighborhood-Based_Recom.pdf (дата звернення: 20.05.2026).
23. Linden G., Smith B., York J. Amazon.com Recommendations: Item-to-Item Collaborative Filtering // Springer. URL: <https://doi.org/10.1023/A:1021240730564> (дата звернення: 11.05.2026).
24. Amazon.com Recommendations: Item-to-Item Collaborative Filtering // Springer. URL: <https://link.springer.com/article/10.1023/A:1021240730564> (дата звернення: 08.05.2026).
25. USER_PERSONALIZATION Recipe // Amazon Personalize Developer Guide. URL: https://docs.aws.amazon.com/personalize/latest/dg/native-recipe-new-item-USER_PERSONALIZATION.html (дата звернення: 19.05.2026).
26. Similar-Items Recipe // Amazon Personalize Developer Guide. URL: <https://docs.aws.amazon.com/personalize/latest/dg/native-recipe-similar-items.html> (дата звернення: 04.05.2026).
27. Collaborative Filtering for Implicit Feedback Datasets // Yifan Hu. URL: <https://yifanhu.net/PUB/cf.pdf> (дата звернення: 13.05.2026).
28. Evaluating Collaborative Filtering Recommender Systems // Springer. URL: https://doi.org/10.1007/978-1-4899-7637-6_8 (дата звернення: 06.05.2026).
29. Evaluation Metrics for Recommender Systems // Ben-Gurion University of the Negev. URL:

<https://tzin.bgu.ac.il/~shanigu/Publications/EvaluationMetrics.17.pdf>

(дата

звернення: 15.05.2026).

30. Evaluating Collaborative Filtering Recommender Systems // GroupLens Research.

URL: <https://grouplens.org/site-content/uploads/evaluating-TOIS-20041.pdf>

(дата звернення: 22.05.2026).

31. A Survey on Bias and Fairness in Recommender Systems // arXiv. URL:

<https://arxiv.org/abs/2007.13237> (дата звернення: 12.05.2026).

32. A Survey on Bias and Fairness in Recommender Systems // arXiv. URL:

<https://arxiv.org/pdf/2007.13237> (дата звернення: 05.05.2026).

33. Recommendations with Negative Feedback via Pairwise Deep Reinforcement

Learning // ACM Digital Library. URL:

<https://dl.acm.org/doi/10.1145/3159652.3159687> (дата звернення: 17.05.2026).

34. Working with Training Metrics // Amazon Personalize Developer Guide. URL:

<https://docs.aws.amazon.com/personalize/latest/dg/working-with-training-metrics.html> (дата звернення: 21.05.2026).

35. Recommender Systems Survey // Universidade Federal de Minas Gerais. URL:

<https://homepages.dcc.ufmg.br/~nivio/cursos/ri13/sources/recommender-systems-survey-2005.pdf> (дата звернення: 18.05.2026).

36. Interactions Dataset Requirements // Amazon Personalize Developer Guide. URL:

<https://docs.aws.amazon.com/personalize/latest/dg/interactions-dataset-requirements.html> (дата звернення: 10.05.2026).

37. Recommendations // Amazon Personalize Developer Guide. URL:

<https://docs.aws.amazon.com/personalize/latest/dg/recommendations.html> (дата звернення: 07.05.2026).

38. Gunawardana A., Shani G. A Survey of Accuracy Evaluation Metrics of Recommendation Tasks // Journal of Machine Learning Research. URL:

<https://jmlr.org/papers/v10/gunawardana09a.html> (дата звернення: 19.05.2026).

					ІАЛЦ.467200.003 ПЗ	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		115

39. Introduction to Amazon DynamoDB // Amazon Web Services. URL: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html> (дата звернення: 04.05.2026).
40. Data Durability in Amazon S3 // Amazon Simple Storage Service User Guide. URL: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html> (дата звернення: 14.05.2026).
41. Data Protection in the European Union // European Commission. URL: https://commission.europa.eu/law/law-topic/data-protection/data-protection-explained_en (дата звернення: 09.05.2026).
42. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions // New York University. URL: <https://pages.stern.nyu.edu/~atuzhili/pdf/TKDE-Paper-as-Printed.pdf> (дата звернення: 20.05.2026).
43. Types of Recommendation Systems // Google Developers. URL: <https://developers.google.com/machine-learning/recommendation/overview/types> (дата звернення: 11.05.2026).
44. Amazon.com Recommendations: Item-to-Item Collaborative Filtering // ACM Digital Library. URL: <https://dl.acm.org/doi/10.1145/963770.963772> (дата звернення: 16.05.2026).

ДОДАТОК А

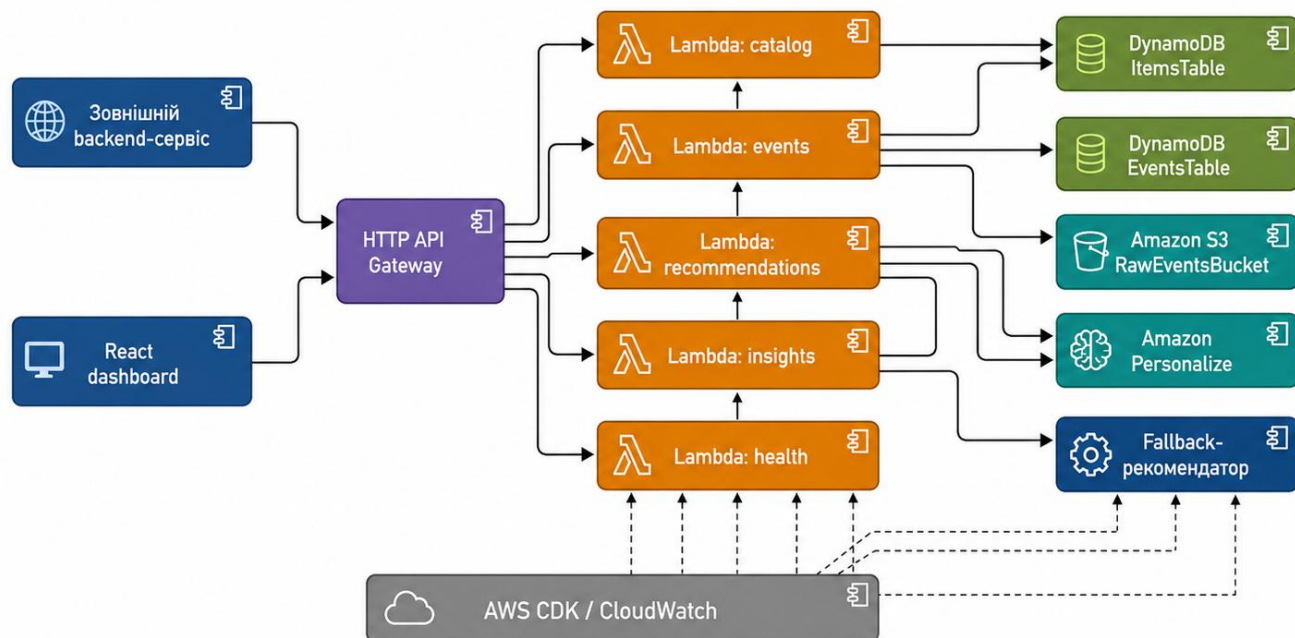
Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних

Структурна схема системи

ІАЛЦ.467200.003 Д1

Аркушів 1

Київ 2026 р



Опис схеми:

- Зовнішній backend-сервіс: передає каталог, події користувачів і запитує рекомендації.
- React dashboard: відображає каталог, статистику, користувачів і рекомендаційну видачу через API.
- HTTP API Gateway: єдина точка входу до системи.
- AWS Lambda: обробка endpoint-ів /catalog, /events, /recommendations, /insights та /health.
- DynamoDB: оперативне зберігання каталогу об'єктів і історії подій.
- Amazon S3 та Amazon Personalize: архівування даних, підготовка датасетів і персоналізовані рекомендації.
- Fallback-рекомендатор: резервна логіка рекомендацій у разі нестачі даних або недоступності ML-компонента.

					ІАЛЦ.467200.003 Д1						
					Інтелектуальна система для генерації користувачьких рекомендацій на основі аналізу історичних даних Структурна схема системи	Літ.		Маса		Масш.	
Зм.	Арк.	№ докум.	Підпис	Дата							
Розробив		Чуб М.А.									
Перевірив		Туганських О.А.									
Реценз.		Тарасенко-Клятенко О. В.									
Н. Контр.		Гончаренко О. О.				Аркуш		Аркушів			
Затвердив		Волокита А. М.				НТУУ КПІ, ФІОТ, ІМ-21					

ДОДАТОК Б

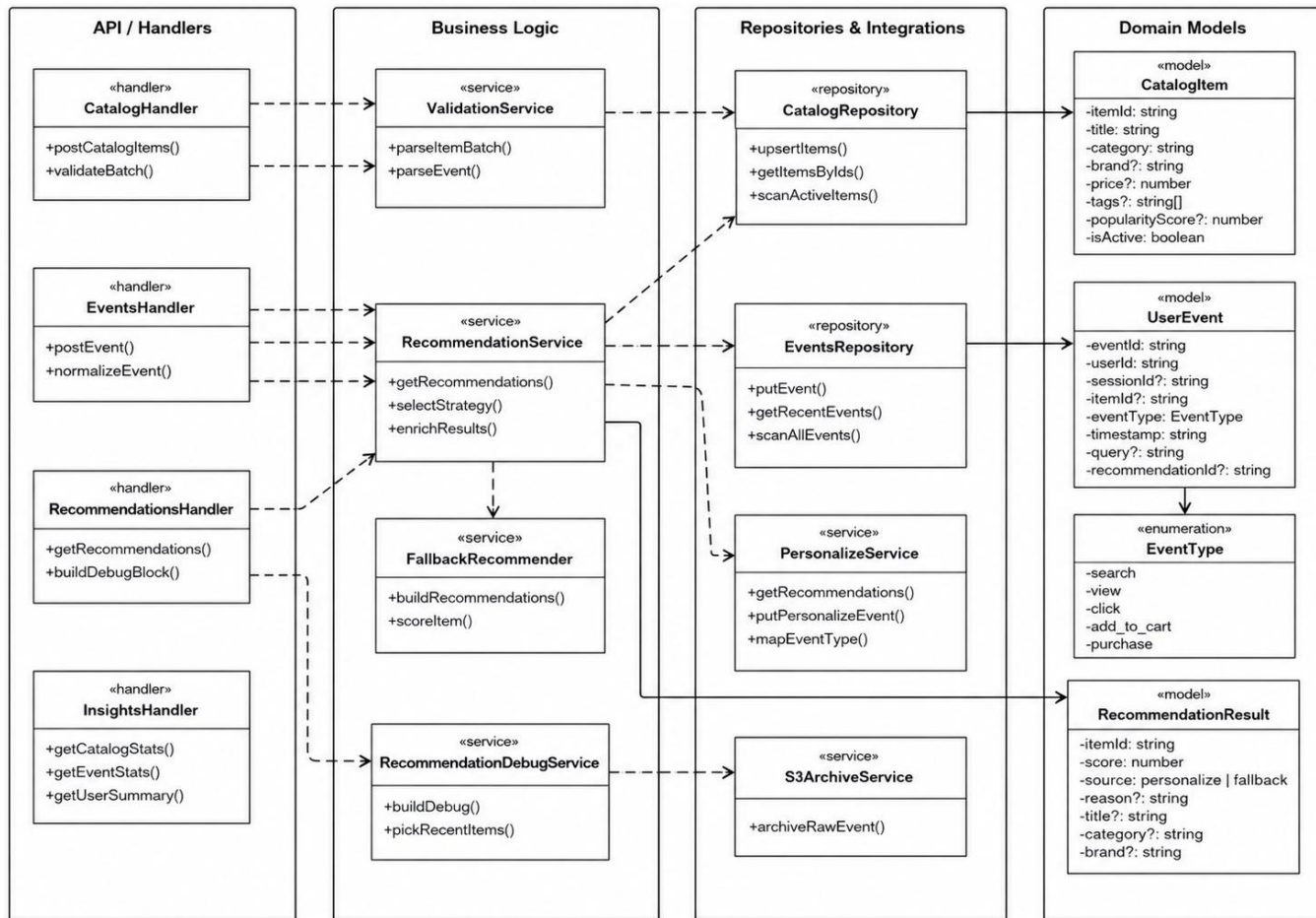
Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних

Функціональна схема (діаграма класів)

ІАЛЦ.467200.003 Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.003 Д2								
					Інтелектуальна система для генерації користувачьких рекомендацій на основі аналізу історичних даних Функціональна схема (діаграма класів)			Літ.		Маса		Масш.	
Зм.	Арк.	№ докум.	Підпис	Дата									
Розробив		Чуб М.А.											
Перевірів		Туганських О.А.											
Реценз.		Тарасенко-Клятченко О. В.					Аркуш			Аркушів			
Н. Контр.		Гончаренко О. О.					НТУУ КПІ, ФІОТ, ІМ-21						
Затвердив		Волокита А. М.											

ДОДАТОК В

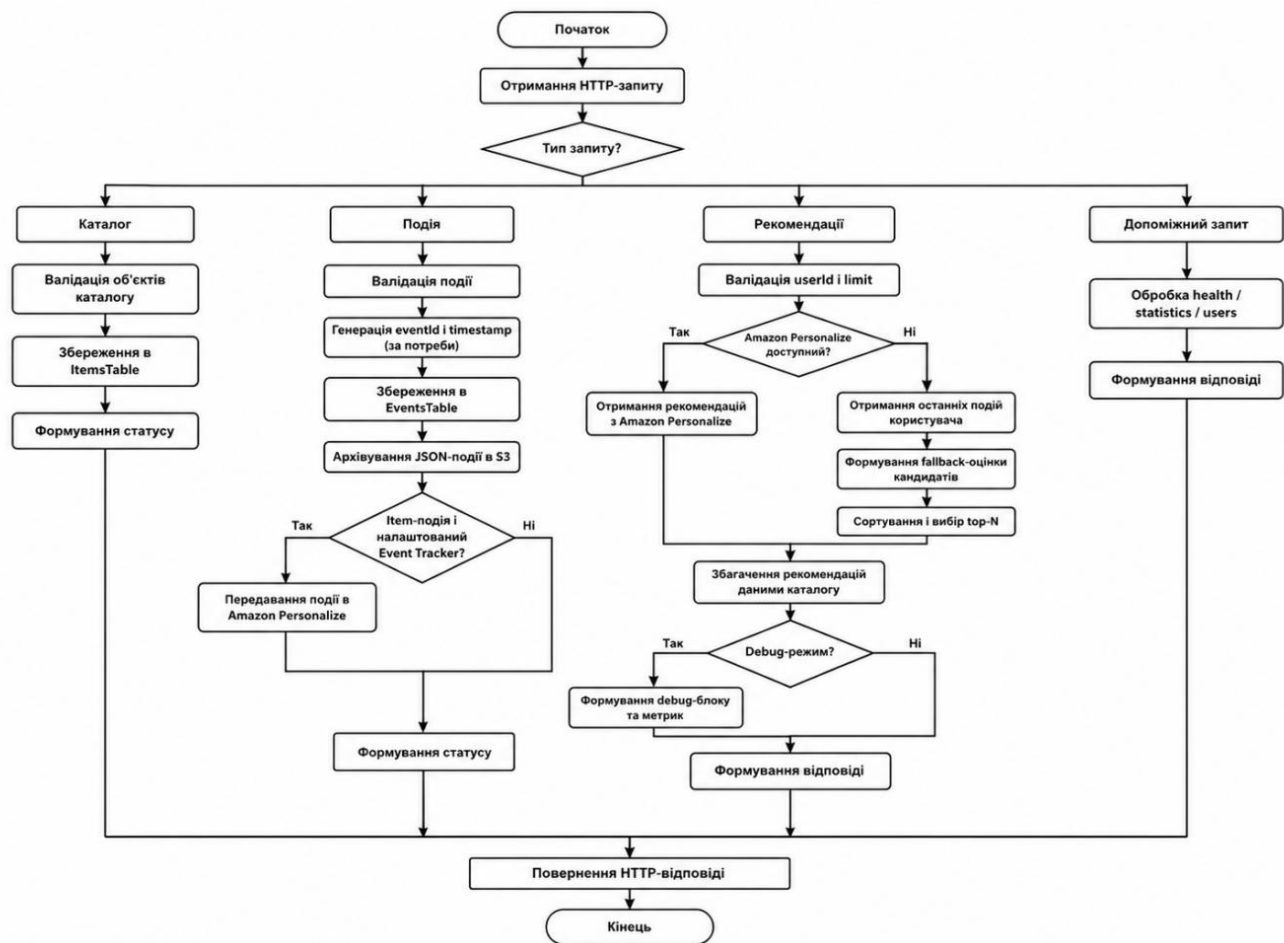
Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.003 ДЗ

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.003 ДЗ			
					Інтелектуальна система для генерації користувацьких рекомендацій на основі аналізу історичних даних Алгоритм дій програмного забезпечення	Літ.	Маса	Масш.
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив	Чуб М.А.							
Перевірив	Туганських О.А.					Аркуш		Аркушів
Реценз.	Тарасенко-Клятченко О. В.					НТУУ КПІ, ФІОТ, ІМ-21		
Н. Контр.	Гончаренко О. О.							
Затвердив	Волокита А. М.							

ДОДАТОК Г

Інтелектуальна система для генерації користувацьких
рекомендацій на основі аналізу історичних даних

Текст програмного коду

ІАЛЦ.467200.003 Д4

Аркушів 42

Київ 2026 р

RecommendationService:

Файл: `src/Lambdas/recommendations.ts`

```
import type { APIGatewayProxyHandlerV2 } from "aws-lambda";
import type { CatalogItem, RecommendationResult } from "../common/types";
import { buildFallbackRecommendations } from "../services/fallback";
import { getItem, getItemByIds, getRecentEvents, scanItems } from "../services/dynamo";
import { buildRecommendationDebug, parseBooleanFlag, parseCsvParam, parseKValues,
pickRecentUniqueItemIds } from "../services/recommendation-debug";
import { getEnv, mustGetEnv } from "../services/env";
import { canUsePersonalizeRecommendations, getPersonalizedRecommendations } from
"../services/personalize";

const eventsTable = mustGetEnv("EVENTS_TABLE_NAME");
const itemsTable = mustGetEnv("ITEMS_TABLE_NAME");
const personalizeFilterArn = getEnv("PERSONALIZE_FILTER_ARN");

function enrichRecommendations(recommendations: RecommendationResult[], items: CatalogItem[]):
RecommendationResult[] {
  const itemById = new Map(items.map((item) => [item.itemId, item]));

  return recommendations.map((recommendation) => {
    const item = itemById.get(recommendation.itemId);
    if (!item) return recommendation;

    return {
      ...recommendation,
      title: item.title,
      category: item.category,
      brand: item.brand,
      price: item.price
    };
  });
}

export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  try {
    const userId = event.queryStringParameters?.userId?.trim();
    const limit = Number(event.queryStringParameters?.limit ?? 10);
    const debugEnabled = parseBooleanFlag(event.queryStringParameters?.debug);
    const kValues = parseKValues(event.queryStringParameters?.kValues);
    const requestedRelevantItemIds = parseCsvParam(event.queryStringParameters?.relevantItemIds);
    const groundTruthMode = event.queryStringParameters?.groundTruthMode?.trim().toLowerCase();
    const groundTruthWindow = Number(event.queryStringParameters?.groundTruthWindow ?? 3);

    if (!userId) {
      return { statusCode: 400, body: JSON.stringify({ ok: false, error: "userId is required" }) };
    }
    if (!Number.isInteger(limit) || limit < 1 || limit > 50) {
      return {
        statusCode: 400,
        body: JSON.stringify({ ok: false, error: "limit must be an integer between 1 and 50" })
      };
    }
  }
}
```

					ІАЛЦ.467200.003 Д4						
					Інтелектуальна система для генерації користувачьких рекомендацій на основі аналізу історичних даних Текст програмного коду	Літ.		Маса		Масш.	
Зм.	Арк.	№ докум.	Підпис	Дата							
Розробив		Чуб М.А.									
Перевірів		Туганських О.А.				Аркуш			Аркушів		
Реценз.		Тарасенко-Клятченко О. В.									
Н. Контр.		Гончаренко О. О.			НТУУ КПІ, ФІОТ, ІМ-21						
Затвердив		Волокита А. М.									

```

}

const recentEvents = debugEnabled || !canUsePersonalizeRecommendations() ? await
getRecentEvents(eventsTable, userId, 20) : [];
const recentItems = (
  await Promise.all(
    recentEvents
      .map((itemEvent) => itemEvent.itemId)
      .filter((itemId): itemId is string => Boolean(itemId))
      .slice(0, 10)
      .map((itemId) => getItem(itemsTable, itemId))
  )
).filter((item): item is NonNullable<typeof item> => Boolean(item));

const derivedRelevantItemIds =
  groundTruthMode === "recent" ? pickRecentUniqueItemIds(recentEvents,
Number.isInteger(groundTruthWindow) && groundTruthWindow > 0 ? groundTruthWindow : 3) : [];
const relevantItemIds = requestedRelevantItemIds.length > 0 ? requestedRelevantItemIds :
derivedRelevantItemIds;
const relevantItemSource =
  requestedRelevantItemIds.length > 0 ? "query" : derivedRelevantItemIds.length > 0 ?
"recent_user_events" : "none";

if (canUsePersonalizeRecommendations()) {
  try {
    const personalizeResponse = await getPersonalizedRecommendations({
      userId,
      limit,
      filterArn: personalizeFilterArn
    });
    const recommendationItems = await getItemByIds(
      itemsTable,
      personalizeResponse.recommendations.map((recommendation) => recommendation.itemId)
    );
    const recommendations = enrichRecommendations(personalizeResponse.recommendations,
recommendationItems);

    return {
      statusCode: 200,
      body: JSON.stringify({
        ok: true,
        strategy: "personalize",
        recommendationId: personalizeResponse.recommendationId,
        recommendations,
        ...(debugEnabled
          ? {
            debug: buildRecommendationDebug({
              userId,
              limit,
              strategy: "personalize",
              recommendations,
              recentEvents,
              recentItems,
              relevantItemIds,
              relevantItemSource,
              kValues
            })
          } : {})
      })
    };
  } catch (error) {
    // ...
  }
}

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		2

```

        }
        : {})
    })
};
} catch (error) {
    console.error("Personalize request failed, falling back to heuristic engine", error);
}
}

const items = await scanItems(itemsTable, 200);
const recommendations = buildFallbackRecommendations({
    items,
    recentItems,
    recentEvents,
    limit
});

return {
    statusCode: 200,
    body: JSON.stringify({
        ok: true,
        strategy: "fallback",
        recommendations,
        ...(debugEnabled
            ? {
                debug: buildRecommendationDebug({
                    userId,
                    limit,
                    strategy: "fallback",
                    recommendations,
                    recentEvents,
                    recentItems,
                    relevantItemIds,
                    relevantItemSource,
                    kValues
                })
            }
            : {})
    })
};
} catch (error) {
    return {
        statusCode: 500,

body: JSON.stringify({ ok: false, error: error instanceof Error ? error.message : "Unknown error"
    })
    };
}
};

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		3

EventIngestionService:

Файл: *src/lambda/events.ts*

```
import { randomUUID } from "node:crypto";
import { PutObjectCommand, S3Client } from "@aws-sdk/client-s3";
import type { APIGatewayProxyHandlerV2 } from "aws-lambda";
import { eventSchema } from "../services/validation";
import { mustGetEnv, getEnv } from "../services/env";
import { putEvent } from "../services/dynamo";
import { canStreamToPersonalize, putPersonalizeEvent } from "../services/personalize";

const s3 = new S3Client({});
const eventsTable = mustGetEnv("EVENTS_TABLE_NAME");
const rawEventsBucket = mustGetEnv("RAW_EVENTS_BUCKET_NAME");
const trackingId = getEnv("PERSONALIZE_TRACKING_ID");

export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  try {
    const payload = event.body ? JSON.parse(event.body) : {};
    const parsed = eventSchema.parse(payload);

    const normalized = {
      eventId: parsed.eventId ?? randomUUID(),
      userId: parsed.userId,
      sessionId: parsed.sessionId,
      itemId: parsed.itemId,
      recommendationId: parsed.recommendationId,
      impression: parsed.impression,
      eventType: parsed.eventType,
      timestamp: parsed.timestamp ?? new Date().toISOString(),
      query: parsed.query,
      filters: parsed.filters,
      properties: parsed.properties
    };

    await putEvent(eventsTable, normalized);

    const date = normalized.timestamp.slice(0, 10);
    await s3.send(
      new PutObjectCommand({
        Bucket: rawEventsBucket,
        Key: `events/date=${date}/${normalized.userId}/${normalized.eventId}.json`,
        Body: JSON.stringify(normalized),
        ContentType: "application/json"
      })
    );

    let personalizeSynced = false;
    if (canStreamToPersonalize() && trackingId && normalized.itemId && normalized.eventType !==
"search") {
      try {
        await putPersonalizeEvent({
          trackingId,

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		4

```

    userId: normalized.userId,
    sessionId: normalized.sessionId || normalized.userId,
    itemId: normalized.itemId,
    eventType: normalized.eventType,
    recommendationId: normalized.recommendationId,
    impression: normalized.impression
  });
  personalizeSynced = true;
} catch (error) {
  console.error("Failed to mirror event to Amazon Personalize", error);
}
}

return {
  statusCode: 201,
  body: JSON.stringify({ ok: true, eventId: normalized.eventId, personalizeSynced })
};
} catch (error) {
  return {
    statusCode: 400,
    body: JSON.stringify({ ok: false, error: error instanceof Error ? error.message : "Unknown
error" })
  };
}
};

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		5

CatalogService:*Файл: src/lambda/catalog.ts*

```
import type { APIGatewayProxyHandlerV2 } from "aws-lambda";
import { batchUpsertItems } from "../services/dynamo";
import { mustGetEnv } from "../services/env";
import { itemBatchSchema } from "../services/validation";

const itemsTable = mustGetEnv("ITEMS_TABLE_NAME");

export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  try {
    const payload = event.body ? JSON.parse(event.body) : {};
    const parsed = itemBatchSchema.parse(payload);
    await batchUpsertItems(itemsTable, parsed.items);

    return {
      statusCode: 200,
      body: JSON.stringify({ ok: true, upserted: parsed.items.length })
    };
  } catch (error) {
    return {
      statusCode: 400,
      body: JSON.stringify({ ok: false, error: error instanceof Error ? error.message : "Unknown
error" })
    };
  }
};
```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		6

FallbackRecommendationService:*Файл: src/services/fallback.ts*

```
import type { CatalogItem, RecommendationResult, UserEvent } from "../common/types";

function scoreItem(candidate: CatalogItem, recentItems: CatalogItem[], recentEvents: UserEvent[]): number {
  let score = candidate.popularityScore ?? 0;
  const categories = new Set(recentItems.map((item) => item.category));
  const brands = new Set(recentItems.map((item) => item.brand).filter(Boolean));

  if (categories.has(candidate.category)) score += 10;
  if (candidate.brand && brands.has(candidate.brand)) score += 4;

  const recentSearchWords = recentEvents
    .filter((e) => e.eventType === "search" && e.query)
    .flatMap((e) => e.query!.toLowerCase().split(/\s+/));

  if (recentSearchWords.some((word) => candidate.title.toLowerCase().includes(word))) score += 3;
  if (recentSearchWords.some((word) => candidate.tags?.some((tag) => tag.toLowerCase().includes(word)))) score += 2;

  return score;
}

export function buildFallbackRecommendations(params: {
  items: CatalogItem[];
  recentItems: CatalogItem[];
  recentEvents: UserEvent[];
  limit: number;
}): RecommendationResult[] {
  const viewedIds = new Set(params.recentEvents.map((e) => e.itemId).filter(Boolean));

  return params.items
    .filter((item) => item.isActive !== false)
    .filter((item) => !viewedIds.has(item.itemId))
    .map((item) => ({
      itemId: item.itemId,
      score: scoreItem(item, params.recentItems, params.recentEvents),
      source: "fallback" as const,
      reason: "Category/brand/popularity heuristic",
      title: item.title,
      category: item.category,
      brand: item.brand,
      price: item.price
    })))
    .sort((a, b) => b.score - a.score)
    .slice(0, params.limit);
}
```

					ІАЛЦ.467200.003 Д4	Арк.
						7
Зм.	Арк.	№ докум.	Підпис	Дата		

MetricsService:*Файл: src/services/metrics.ts*

```
function toRelevantSet(relevantItemIds: string[]): Set<string> {
  return new Set(relevantItemIds.filter(Boolean));
}

export function precisionAtK(recommendedItemIds: string[], relevantItemIds: string[], k: number):
number {
  const limit = Math.max(0, Math.min(k, recommendedItemIds.length));
  if (limit === 0) return 0;

  const relevant = toRelevantSet(relevantItemIds);
  const hits = recommendedItemIds.slice(0, limit).filter((itemId) => relevant.has(itemId)).length;
  return hits / limit;
}

export function recallAtK(recommendedItemIds: string[], relevantItemIds: string[], k: number):
number {
  const relevant = toRelevantSet(relevantItemIds);
  if (relevant.size === 0) return 0;

  const hits = recommendedItemIds
    .slice(0, Math.max(0, Math.min(k, recommendedItemIds.length)))
    .filter((itemId) => relevant.has(itemId)).length;

  return hits / relevant.size;
}

export function averagePrecisionAtK(recommendedItemIds: string[], relevantItemIds: string[], k:
number): number {
  const relevant = toRelevantSet(relevantItemIds);
  if (relevant.size === 0) return 0;

  let hits = 0;
  let precisionSum = 0;
  const limit = Math.max(0, Math.min(k, recommendedItemIds.length));

  for (let index = 0; index < limit; index += 1) {
    if (!relevant.has(recommendedItemIds[index])) continue;

    hits += 1;
    precisionSum += hits / (index + 1);
  }

  return precisionSum / Math.min(relevant.size, Math.max(limit, 1));
}

function discountedCumulativeGain(recommendedItemIds: string[], relevantItemIds: string[], k:
number): number {
  const relevant = toRelevantSet(relevantItemIds);
  const limit = Math.max(0, Math.min(k, recommendedItemIds.length));
  let dcg = 0;
```

					ІАЛЦ.467200.003 Д4	Арк.
						8
Зм.	Арк.	№ докум.	Підпис	Дата		

```

for (let index = 0; index < limit; index += 1) {
  if (!relevant.has(recommendedItemIds[index])) continue;
  dcg += 1 / Math.log2(index + 2);
}

return dcg;
}

export function ndcgAtK(recommendedItemIds: string[], relevantItemIds: string[], k: number):
number {
  const relevant = toRelevantSet(relevantItemIds);
  if (relevant.size === 0) return 0;

  const dcg = discountedCumulativeGain(recommendedItemIds, [...relevant], k);
  const ideallist = Array.from(relevant);
  const idcg = discountedCumulativeGain(ideallist, ideallist, Math.min(k, ideallist.length));

  if (idcg === 0) return 0;
  return dcg / idcg;
}

export function hitRateAtK(recommendedItemIds: string[], relevantItemIds: string[], k: number):
number {
  const relevant = toRelevantSet(relevantItemIds);
  if (relevant.size === 0) return 0;

  const hit = recommendedItemIds
    .slice(0, Math.max(0, Math.min(k, recommendedItemIds.length)))
    .some((itemId) => relevant.has(itemId));

  return hit ? 1 : 0;
}

export function metricsAtK(recommendedItemIds: string[], relevantItemIds: string[], k: number) {
  const averagePrecision = averagePrecisionAtK(recommendedItemIds, relevantItemIds, k);

  return {
    precision: precisionAtK(recommendedItemIds, relevantItemIds, k),
    recall: recallAtK(recommendedItemIds, relevantItemIds, k),
    averagePrecision,
    map: averagePrecision,
    ndcg: ndcgAtK(recommendedItemIds, relevantItemIds, k),
    hitRate: hitRateAtK(recommendedItemIds, relevantItemIds, k)
  };
}

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		9

RecommendationDebugService:*Файл: src/services/recommendation-debug.ts*

```
import type { CatalogItem, RecommendationResult, UserEvent } from "../common/types";
import { metricsAtK } from "../metrics";

export function parseBooleanFlag(value?: string): boolean {
  if (!value) return false;
  return ["1", "true", "yes", "on"].includes(value.trim().toLowerCase());
}

export function parseCsvParam(value?: string): string[] {
  if (!value) return [];

  return [...new Set(value.split(",").map((part) => part.trim()).filter(Boolean))];
}

export function parseKValues(value: string | undefined, fallback: number[] = [1, 3, 5, 10]):
number[] {
  const parsed = parseCsvParam(value)
    .map((part) => Number(part))
    .filter((candidate) => Number.isInteger(candidate) && candidate > 0 && candidate <= 100);

  return parsed.length > 0 ? [...new Set(parsed)].sort((a, b) => a - b) : fallback;
}

export function pickRecentUniqueItemIds(events: UserEvent[], window: number): string[] {
  const picked: string[] = [];
  const seen = new Set<string>();

  for (const event of events) {
    if (!event.itemId || seen.has(event.itemId)) continue;

    seen.add(event.itemId);
    picked.push(event.itemId);

    if (picked.length >= window) break;
  }

  return picked;
}

function buildCounts(values: Array<string | undefined>): Array<{ value: string; count: number }> {
  const counts = new Map<string, number>();

  for (const value of values) {
    if (!value) continue;
    counts.set(value, (counts.get(value) ?? 0) + 1);
  }

  return [...counts.entries()]
    .map(([value, count]) => ({ value, count }))
    .sort((a, b) => b.count - a.count || a.value.localeCompare(b.value));
}
```

					ІАЛЦ.467200.003 Д4	Арк.
						10
Зм.	Арк.	№ докум.	Підпис	Дата		

```

function numericStats(values: Array<number | undefined>) {
  const numbers = values.filter((value): value is number => typeof value === "number" &&
    Number.isFinite(value));
  if (numbers.length === 0) {
    return { count: 0, min: null, max: null, avg: null };
  }

  const sum = numbers.reduce((acc, value) => acc + value, 0);
  return {
    count: numbers.length,
    min: Math.min(...numbers),
    max: Math.max(...numbers),
    avg: Number((sum / numbers.length).toFixed(4))
  };
}

export function buildRecommendationDebug(params: {
  userId: string;
  limit: number;
  strategy: "personalize" | "fallback";
  recommendations: RecommendationResult[];
  recentEvents: UserEvent[];
  recentItems: CatalogItem[];
  relevantItemIds: string[];
  relevantItemSource: "query" | "recent_user_events" | "none";
  kValues: number[];
}) {
  const recommendedItemIds = params.recommendations.map((recommendation) =>
    recommendation.itemId);

  const metrics =
    params.relevantItemIds.length > 0
      ? Object.fromEntries(params.kValues.map((k) => [String(k), metricsAtK(recommendedItemIds,
        params.relevantItemIds, k)]))
      : undefined;

  const topCategory = buildCounts(params.recentItems.map((item) => item.category))[0]?.value ??
    null;
  const matchingTopCategoryCount = topCategory
    ? params.recommendations.filter((recommendation) => recommendation.category ===
      topCategory).length
    : 0;

  return {
    request: {
      userId: params.userId,
      limit: params.limit,
      strategy: params.strategy
    },
    profile: {
      recentEventsCount: params.recentEvents.length,
      recentUniqueItemCount: new Set(params.recentEvents.map((event) =>

```

```

event.itemId).filter(Boolean)).size,
    recentEventTypes: buildCounts(params.recentEvents.map((event) => event.eventType)),
    recentCategories: buildCounts(params.recentItems.map((item) => item.category)),
    recentBrands: buildCounts(params.recentItems.map((item) => item.brand)),
    recentPrices: numericStats(params.recentItems.map((item) => item.price))
  },
  recommendationStats: {
    categories: buildCounts(params.recommendations.map((recommendation) =>
recommendation.category)),
    brands: buildCounts(params.recommendations.map((recommendation) => recommendation.brand)),
    prices: numericStats(params.recommendations.map((recommendation) => recommendation.price)),
    scores: numericStats(params.recommendations.map((recommendation) => recommendation.score)),
    categoryAlignment: {
      topRecentCategory: topCategory,
      matchingRecommendations: matchingTopCategoryCount,
      matchingShare: params.recommendations.length > 0 ? Number((matchingTopCategoryCount /
params.recommendations.length).toFixed(4)) : 0
    }
  },
  evaluation:
    params.relevantItemIds.length > 0
      ? {
          relevantItemIds: params.relevantItemIds,
          relevantItemSource: params.relevantItemSource,
          metricsAtK: metrics,
          note: "For a single recommendation list, MAP@K equals AveragePrecision@K."
        }
      : {
          relevantItemIds: [],
          relevantItemSource: params.relevantItemSource,
          metricsAtK: undefined,
          note: "Pass relevantItemIds=a,b,c or use groundTruthMode=recent to calculate
Precision@K, Recall@K, MAP@K, and NDCG@K."
        }
    };
}

```

PersonalizeIntegrationService:

Файл: *src/services/personalize.ts*

```
import { PersonalizeRuntimeClient, GetRecommendationsCommand } from "@aws-sdk/client-personalize-runtime";
import { PersonalizeEventsClient, PutEventsCommand } from "@aws-sdk/client-personalize-events";
import type { EventType, RecommendationResult } from "../common/types";

const region = process.env.AWS_HOST_REGION || "us-east-1";
const runtimeClient = new PersonalizeRuntimeClient({ region });
const eventsClient = new PersonalizeEventsClient({ region });

function getRecommendationResource(): { recommenderArn?: string; campaignArn?: string } {
  const recommenderArn = process.env.PERSONALIZE_RECOMMENDER_ARN || undefined;
  const campaignArn = process.env.PERSONALIZE_CAMPAIGN_ARN || undefined;
  return { recommenderArn, campaignArn };
}

export function canUsePersonalizeRecommendations(): boolean {
  const { recommenderArn, campaignArn } = getRecommendationResource();
  return Boolean(recommenderArn || campaignArn);
}

export function canStreamToPersonalize(): boolean {
  return Boolean(process.env.PERSONALIZE_TRACKING_ID);
}

export interface PersonalizedRecommendationResponse {
  recommendationId?: string;
  recommendations: RecommendationResult[];
}

export function mapEventTypeToPersonalize(eventType: EventType): string | null {
  switch (eventType) {
    case "view":
      return "View";
    case "click":
      return "Click";
    case "add_to_cart":
      return "AddToCart";
    case "purchase":
      return "Purchase";
    default:
      return null;
  }
}

export async function getPersonalizedRecommendations(params: {
  userId: string;
  limit: number;
  filterArn?: string;
  filterValues?: Record<string, string>;
}): Promise<PersonalizedRecommendationResponse> {
  const { recommenderArn, campaignArn } = getRecommendationResource();

  if (!recommenderArn && !campaignArn) {
    throw new Error("Neither PERSONALIZE_RECOMMENDER_ARN nor PERSONALIZE_CAMPAIGN_ARN is configured");
  }
}
```

					ІАЛЦ.467200.003 Д4	Арк.
						13
Зм.	Арк.	№ докум.	Підпис	Дата		

```

const response = await runtimeClient.send(
  new GetRecommendationsCommand({
    ...(recommenderArn ? { recommenderArn } : { campaignArn }),
    userId: params.userId,
    numResults: params.limit,
    ...(params.filterArn ? { filterArn: params.filterArn } : {}),
    ...(params.filterValues ? { filterValues: params.filterValues } : {})
  })
);

return {
  recommendationId: response.recommendationId,
  recommendations: (response.itemList ?? []).map((item, index) => ({
    itemId: item.itemId ?? `unknown-${index}`,
    score: item.score ?? 0,
    source: "personalize" as const,
    reason: recommenderArn ? "Amazon Personalize recommender" : "Amazon Personalize campaign"
  }))
};
}

export async function putPersonalizeEvent(params: {
  trackingId: string;
  sessionId: string;
  itemId: string;
  eventType: EventType;
  userId?: string;
  recommendationId?: string;
  impression?: string[];
}): Promise<void> {
  const mappedType = mapEventTypeToPersonalize(params.eventType);
  if (!mappedType) return;

  await eventsClient.send(
    new PutEventsCommand({
      trackingId: params.trackingId,
      sessionId: params.sessionId,
      userId: params.userId,
      eventList: [
        {
          eventType: mappedType,
          itemId: params.itemId,
          sentAt: new Date(),
          ...(params.recommendationId ? { recommendationId: params.recommendationId } : {}),
          ...(params.impression && params.impression.length > 0 ? { impression: params.impression
} : {}))
        ]
      ]
    })
  );
}

```

DynamoStorageService:

Файл: *src/services/dynamo.ts*

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  BatchGetCommand,
  DynamoDBDocumentClient,
  PutCommand,
  QueryCommand,
  ScanCommand,
  GetCommand
} from "@aws-sdk/lib-dynamodb";
import type { CatalogItem, UserEvent } from "../common/types";

const client = new DynamoDBClient({});
export const ddb = DynamoDBDocumentClient.from(client, {
  marshallOptions: { removeUndefinedValues: true }
});

export async function putEvent(tableName: string, event: UserEvent): Promise<void> {
  await ddb.send(
    new PutCommand({
      TableName: tableName,
      Item: {
        pk: `USER#${event.userId}`,
        sk: `TS#${event.timestamp}#${event.eventId}`,
        ...event
      }
    })
  );
}

export async function getRecentEvents(tableName: string, userId: string, limit = 20):
Promise<UserEvent[]> {
  const response = await ddb.send(
    new QueryCommand({
      TableName: tableName,
      KeyConditionExpression: "pk = :pk AND begins_with(sk, :sk)",
      ExpressionAttributeValues: {
        ":pk": `USER#${userId}`,
        ":sk": "TS#"
      },
      ScanIndexForward: false,
      Limit: limit
    })
  );
  return (response.Items ?? []).map((item) => ({
    eventId: item.eventId,
    userId: item.userId,
    sessionId: item.sessionId,
    itemId: item.itemId,
    recommendationId: item.recommendationId,
  }));
}
```

					ІАЛЦ.467200.003 Д4	Арк.
						15
Зм.	Арк.	№ докум.	Підпис	Дата		

```

impression: item.impression,
  eventType: item.eventType,
  timestamp: item.timestamp,
  query: item.query,
  filters: item.filters,
  properties: item.properties
})) as UserEvent[];
}

export async function batchUpsertItems(tableName: string, items: CatalogItem[]): Promise<void> {
  const chunks: CatalogItem[][] = [];
  for (let i = 0; i < items.length; i += 25) chunks.push(items.slice(i, i + 25));

  for (const chunk of chunks) {
    let pending = chunk.map((item) => ({
      PutRequest: { Item: { ...item, isActive: item.isActive ?? true } }
    }));
    let attempt = 0;

    while (pending.length > 0) {
      const response = await ddb.send(
        new BatchWriteCommand({
          RequestItems: {
            [tableName]: pending
          }
        })
      );

      pending = (response.UnprocessedItems?.[tableName] ?? []) as typeof pending;
      if (pending.length === 0) break;

      attempt += 1;
      if (attempt >= 10) {
        throw new Error(`Failed to persist ${pending.length} catalog items after ${attempt}
retries`);
      }

      await new Promise((resolve) => setTimeout(resolve, Math.min(250 * 2 ** (attempt - 1),
2_000)));
    }
  }
}

export async function getItem(tableName: string, itemId: string): Promise<CatalogItem | undefined>
{
  const response = await ddb.send(
    new GetCommand({
      TableName: tableName,
      Key: { itemId }
    })
  );
  return response.Item as CatalogItem | undefined;
}

```

```

export async function getItemByIds(tableName: string, itemIds: string[]): Promise<CatalogItem[]>
{
    const uniqueIds = [...new Set(itemIds)];
    if (uniqueIds.length === 0) return [];

    const chunks: string[][] = [];
    for (let i = 0; i < uniqueIds.length; i += 100) chunks.push(uniqueIds.slice(i, i + 100));

    const items: CatalogItem[] = [];
    for (const chunk of chunks) {
        const response = await ddb.send(
            new BatchGetCommand({
                RequestItems: {
                    [tableName]: {
                        Keys: chunk.map((itemId) => ({ itemId })))
                    }
                }
            })
        );

        items.push(...((response.Responses?.[tableName] ?? []) as CatalogItem[]));
    }

    return items;
}

export async function scanItems(tableName: string, limit = 200): Promise<CatalogItem[]> {
    const response = await ddb.send(
        new ScanCommand({
            TableName: tableName,
            Limit: limit
        })
    );
    return (response.Items ?? []) as CatalogItem[];
}

async function scanAll<T>(tableName: string): Promise<T[]> {
    const items: T[] = [];
    let exclusiveStartKey: Record<string, unknown> | undefined;

    do {
        const response = await ddb.send(
            new ScanCommand({
                TableName: tableName,
                ExclusiveStartKey: exclusiveStartKey
            })
        );

        items.push(...((response.Items ?? []) as T[]));
        exclusiveStartKey = response.LastEvaluatedKey as Record<string, unknown> | undefined;
    } while (exclusiveStartKey);

    return items;
}

```

```

}

export async function scanAllItems(tableName: string): Promise<CatalogItem[]> {
  return scanAll<CatalogItem>(tableName);
}

export async function scanAllEvents(tableName: string): Promise<UserEvent[]> {
  const items = await scanAll<Record<string, unknown>>(tableName);

  return items.map((item) => ({
    eventId: item.eventId as string,
    userId: item.userId as string,
    sessionId: item.sessionId as string | undefined,
    itemId: item.itemId as string | undefined,
    recommendationId: item.recommendationId as string | undefined,
    impression: item.impression as string[] | undefined,
    eventType: item.eventType as UserEvent["eventType"],
    timestamp: item.timestamp as string,
    query: item.query as string | undefined,
    filters: item.filters as UserEvent["filters"],
    properties: item.properties as UserEvent["properties"]
  })));
}

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		18

ValidationService:*Файл: src/services/validation.ts*

```
import { z } from "zod";

export const eventSchema = z
  .object({
    eventId: z.string().uuid().optional(),
    userId: z.string().min(1),
    sessionId: z.string().min(1).optional(),
    itemId: z.string().min(1).optional(),
    recommendationId: z.string().min(1).optional(),
    impression: z.array(z.string().min(1).max(50).optional()),
    eventType: z.enum(["search", "view", "click", "add_to_cart", "purchase"]),
    timestamp: z.string().datetime().optional(),
    query: z.string().trim().min(1).optional(),
    filters: z.record(z.union([z.string(), z.number(), z.boolean(), z.null()])).optional(),
    properties: z.record(z.union([z.string(), z.number(), z.boolean(), z.null()])).optional()
  })
  .superRefine((payload, ctx) => {
    if (payload.eventType === "search" && !payload.query) {
      ctx.addIssue({
        code: z.ZodIssueCode.custom,
        path: ["query"],
        message: "query is required for search events"
      });
    }

    if (payload.eventType !== "search" && !payload.itemId) {
      ctx.addIssue({
        code: z.ZodIssueCode.custom,
        path: ["itemId"],
        message: "itemId is required for item interaction events"
      });
    }
  });

export const itemSchema = z.object({
  itemId: z.string().min(1),
  title: z.string().min(1),
  category: z.string().min(1),
  brand: z.string().optional(),
  price: z.number().nonnegative().optional(),
  tags: z.array(z.string()).optional(),
  popularityScore: z.number().nonnegative().optional(),
  isActive: z.boolean().optional(),
  metadata: z.record(z.union([z.string(), z.number(), z.boolean(), z.null()])).optional()
});

export const itemBatchSchema = z.object({
  items: z.array(itemSchema).min(1).max(25)
});
```

AnalyticsInsightsService:

Файл: *src/lambda/insights.ts*

```
import type { APIGatewayProxyHandlerV2 } from "aws-lambda";
import type { CatalogItem, UserEvent } from "../common/types";
import { getItemsByIds, getRecentEvents, scanAllEvents, scanAllItems } from "../services/dynamo";
import { mustGetEnv } from "../services/env";

const itemsTable = mustGetEnv("ITEMS_TABLE_NAME");
const eventsTable = mustGetEnv("EVENTS_TABLE_NAME");

function buildCounts(values: Array<string | undefined>): Array<{ value: string; count: number }> {
  const counts = new Map<string, number>();

  for (const value of values) {
    if (!value) continue;
    counts.set(value, (counts.get(value) ?? 0) + 1);
  }

  return [...counts.entries()]
    .map(([value, count]) => ({ value, count }))
    .sort((a, b) => b.count - a.count || a.value.localeCompare(b.value));
}

function numericStats(values: Array<number | undefined>) {
  const numbers = values.filter((value): value is number => typeof value === "number" &&
    Number.isFinite(value));
  if (numbers.length === 0) {
    return { count: 0, min: null, max: null, avg: null };
  }

  const sum = numbers.reduce((acc, value) => acc + value, 0);
  return {
    count: numbers.length,
    min: Math.min(...numbers),
    max: Math.max(...numbers),
    avg: Number((sum / numbers.length).toFixed(4))
  };
}

function toJson(body: unknown, statusCode = 200) {
  return {
    statusCode,
    body: JSON.stringify(body)
  };
}

function parseBoundedInteger(value: string | undefined, fallback: number, min: number, max:
number): number {
  const parsed = Number(value);
  if (!Number.isInteger(parsed)) return fallback;
  return Math.min(Math.max(parsed, min), max);
}
```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		20

```

function summarizeRecentItems(recentItems: CatalogItem[]) {
  return {
    categories: buildCounts(recentItems.map((item) => item.category)),
    brands: buildCounts(recentItems.map((item) => item.brand)),
    prices: numericStats(recentItems.map((item) => item.price))
  };
}

async function handleCatalogCount() {
  const items = await scanAllItems(itemsTable);

  return toJson({
    ok: true,
    totalItems: items.length,
    activeItems: items.filter((item) => item.isActive !== false).length,
    inactiveItems: items.filter((item) => item.isActive === false).length
  });
}

async function handleCatalogStats() {
  const items = await scanAllItems(itemsTable);
  const activeItems = items.filter((item) => item.isActive !== false);

  return toJson({
    ok: true,
    totals: {
      items: items.length,
      activeItems: activeItems.length,
      inactiveItems: items.length - activeItems.length
    },
    categories: buildCounts(activeItems.map((item) => item.category)),
    brands: buildCounts(activeItems.map((item) => item.brand)),
    prices: numericStats(activeItems.map((item) => item.price)),
    sampleItems: activeItems.slice(0, 10).map((item) => ({
      itemId: item.itemId,
      title: item.title,
      category: item.category,
      brand: item.brand,
      price: item.price
    })))
  });
}

async function handleCatalogItems(searchParams?: Record<string, string | undefined>) {
  const limit = parseBoundedInteger(searchParams?.limit, 48, 1, 200);
  const search = searchParams?.search?.trim().toLowerCase();
  const activeOnly = searchParams?.activeOnly?.trim().toLowerCase() !== "false";
  const category = searchParams?.category?.trim().toLowerCase();
  const brand = searchParams?.brand?.trim().toLowerCase();

  const items = await scanAllItems(itemsTable);
  const filteredItems = items

```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		21

```

.filter((item) => (activeOnly ? item.isActive !== false : true))
.filter((item) => (category ? item.category.toLowerCase() === category : true))
.filter((item) => (brand ? (item.brand ?? "").toLowerCase() === brand : true))
.filter((item) => {
  if (!search) return true;

  return [item.itemId, item.title, item.category, item.brand, ...(item.tags ?? [])]
    .filter((value): value is string => Boolean(value))
    .some((value) => value.toLowerCase().includes(search));
})
.sort((left, right) => {
  const popularityDelta = (right.popularityScore ?? 0) - (left.popularityScore ?? 0);
  if (popularityDelta !== 0) return popularityDelta;
  return left.title.localeCompare(right.title);
});

return toJson({
  ok: true,
  totalItems: filteredItems.length,
  items: filteredItems.slice(0, limit).map((item) => ({
    itemId: item.itemId,
    title: item.title,
    category: item.category,
    brand: item.brand,
    price: item.price,
    popularityScore: item.popularityScore ?? 0,
    isActive: item.isActive !== false,
    tags: item.tags ?? []
  })))
});
}

async function handleEventsCount() {
  const events = await scanAllEvents(eventsTable);

  return toJson({
    ok: true,
    totalEvents: events.length,
    uniqueUsers: new Set(events.map((event) => event.userId)).size,
    uniqueItems: new Set(events.map((event) => event.itemId).filter(Boolean)).size
  });
}

async function handleEventsStats() {
  const events = await scanAllEvents(eventsTable);
  const timestamps = events.map((event) => event.timestamp).filter(Boolean).sort();

  return toJson({
    ok: true,
    totals: {
      events: events.length,
      users: new Set(events.map((event) => event.userId)).size,
      sessions: new Set(events.map((event) => event.sessionId).filter(Boolean)).size,
      itemsWithEvents: new Set(events.map((event) => event.itemId).filter(Boolean)).size
    }
  });
}

```

```

    },
    eventTypes: buildCounts(events.map((event) => event.eventType)),
    firstEventAt: timestamps[0] ?? null,
    lastEventAt: timestamps[timestamps.length - 1] ?? null
  });
}

async function handleUsers(searchParams?: Record<string, string | undefined>) {
  const limit = parseBoundedInteger(searchParams?.limit, 50, 1, 200);
  const search = searchParams?.search?.trim().toLowerCase();
  const events = (await scanAllEvents(eventsTable)).sort((left, right) =>
right.timestamp.localeCompare(left.timestamp));
  const userMap = new Map<
    string,
    {
      userId: string;
      totalEvents: number;
      uniqueItems: Set<string>;
      recentEventTypes: string[];
      lastEventAt: string | null;
    }
  >();

  for (const event of events) {
    const existing =
      userMap.get(event.userId) ??
      {
        userId: event.userId,
        totalEvents: 0,
        uniqueItems: new Set<string>(),
        recentEventTypes: [],
        lastEventAt: null
      };

    existing.totalEvents += 1;
    if (event.itemId) existing.uniqueItems.add(event.itemId);
    if (existing.recentEventTypes.length < 20) existing.recentEventTypes.push(event.eventType);
    if (!existing.lastEventAt || event.timestamp > existing.lastEventAt) existing.lastEventAt =
event.timestamp;
    userMap.set(event.userId, existing);
  }

  const users = [...userMap.values()]
    .filter((user) => (search ? user.userId.toLowerCase().includes(search) : true))
    .sort((left, right) => {
      const leftTime = left.lastEventAt ?? "";
      const rightTime = right.lastEventAt ?? "";
      return rightTime.localeCompare(leftTime) || right.totalEvents - left.totalEvents ||
left.userId.localeCompare(right.userId);
    });

  return toJson({
    ok: true,
    totalUsers: users.length,
  });
}

```

```

    users: users.slice(0, limit).map((user) => ({
      userId: user.userId,
      totalEvents: user.totalEvents,
      uniqueItems: user.uniqueItems.size,
      lastEventAt: user.lastEventAt,
      recentEventTypes: buildCounts(user.recentEventTypes).slice(0, 3)
    })))
  });
}

async function handleUserSummary(userId: string) {
  const recentEvents = await getRecentEvents(eventsTable, userId, 20);
  const recentItems = await getItemsByIds(
    itemsTable,
    [...new Set(recentEvents.map((event) => event.itemId).filter((itemId): itemId is string =>
Boolean(itemId)))]
  );

  return toJson({
    ok: true,
    userId,
    totals: {
      recentEvents: recentEvents.length,
      uniqueItems: new Set(recentEvents.map((event) => event.itemId).filter(Boolean)).size
    },
    recentEventTypes: buildCounts(recentEvents.map((event) => event.eventType)),
    recentItemSummary: summarizeRecentItems(recentItems),
    recentEvents: recentEvents.slice(0, 10),
    recentItems: recentItems.slice(0, 10).map((item) => ({
      itemId: item.itemId,
      title: item.title,
      category: item.category,
      brand: item.brand,
      price: item.price
    })))
  });
}

export const handler: APIGatewayProxyHandlerV2 = async (event) => {
  try {
    const path = event.rawPath;

    if (path === "/catalog/items") return handleCatalogItems(event.queryStringParameters);
    if (path === "/catalog/count") return handleCatalogCount();
    if (path === "/catalog/stats") return handleCatalogStats();
    if (path === "/events/count") return handleEventsCount();
    if (path === "/events/stats") return handleEventsStats();
    if (path === "/users") return handleUsers(event.queryStringParameters);

    if (event.routeKey === "GET /users/{userId}/summary") {
      const userId = event.pathParameters?.userId?.trim();
      if (!userId) {
        return toJson({ ok: false, error: "userId is required" }, 400);
      }
    }
  }
}

```

```
    }

    return handleUserSummary(userId);
}

return toJson({ ok: false, error: "Route not found" }, 404);
} catch (error) {
    return toJson({ ok: false, error: error instanceof Error ? error.message : "Unknown error" },
500);
}
};
```

					ІАЛЦ.467200.003 Д4	Арк.
						25
Зм.	Арк.	№ докум.	Підпис	Дата		

InfrastructureStack:

Файл: Lib/recommendation-stack.ts

```
import * as path from "path";
import * as cdk from "aws-cdk-lib";
import { Construct } from "constructs";
import * as apigwv2 from "aws-cdk-lib/aws-apigatewayv2";
import * as integrations from "aws-cdk-lib/aws-apigatewayv2-integrations";
import * as dynamodb from "aws-cdk-lib/aws-dynamodb";
import * as iam from "aws-cdk-lib/aws-iam";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as lambdaNodejs from "aws-cdk-lib/aws-lambda-nodejs";
import * as logs from "aws-cdk-lib/aws-logs";
import * as personalize from "aws-cdk-lib/aws-personalize";
import * as s3 from "aws-cdk-lib/aws-s3";

export class RecommendationEngineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const personalizeStackMode = process.env.PERSONALIZE_STACK_MODE ?? "off"; // off | datasets | imports
    const personalizePrefix = process.env.PERSONALIZE_PREFIX ?? "personalize/input";
    const personalizeGroupName = process.env.PERSONALIZE_DATASET_GROUP_NAME ?? "reco-ecommerce-dev";
    const personalizeImportRoleArn = process.env.PERSONALIZE_IMPORT_ROLE_ARN ?? "";

    const itemsTable = new dynamodb.Table(this, "ItemsTable", {
      partitionKey: { name: "itemId", type: dynamodb.AttributeType.STRING },
      billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });

    const eventsTable = new dynamodb.Table(this, "EventsTable", {
      partitionKey: { name: "pk", type: dynamodb.AttributeType.STRING },
      sortKey: { name: "sk", type: dynamodb.AttributeType.STRING },
      billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      timeToLiveAttribute: "ttl"
    });

    const rawEventsBucket = new s3.Bucket(this, "RawEventsBucket", {
      autoDeleteObjects: true,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL,
      encryption: s3.BucketEncryption.S3_MANAGED
    });

    const commonEnv = {
      AWS_HOST_REGION: this.region,
      EVENTS_TABLE_NAME: eventsTable.tableName,
      ITEMS_TABLE_NAME: itemsTable.tableName,
      RAW_EVENTS_BUCKET_NAME: rawEventsBucket.bucketName,
      PERSONALIZE_RECOMMENDER_ARN: process.env.PERSONALIZE_RECOMMENDER_ARN ?? ""
    };
  }
}
```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		26

```

PERSONALIZE_CAMPAIGN_ARN: process.env.PERSONALIZE_CAMPAIGN_ARN ?? "",
PERSONALIZE_FILTER_ARN: process.env.PERSONALIZE_FILTER_ARN ?? "",
PERSONALIZE_TRACKING_ID: process.env.PERSONALIZE_TRACKING_ID ?? ""
};

const createNodeFn = (id: string, entry: string) => {
  const logGroup = new logs.LogGroup(this, `${id}LogGroup`, {
    retention: logs.RetentionDays.ONE_WEEK,
    removalPolicy: cdk.RemovalPolicy.DESTROY
  });

  return new lambdaNode.NodejsFunction(this, id, {
    entry: path.join(__dirname, "..", entry),
    runtime: lambda.Runtime.NODEJS_22_X,
    memorySize: 512,
    timeout: cdk.Duration.seconds(20),
    logGroup,
    bundling: {
      minify: false,
      sourceMap: true,
      target: "node22"
    },
    environment: commonEnv
  });
};

const eventsFn = createNodeFn("EventsFn", "src/lambdaas/events.ts");
const catalogFn = createNodeFn("CatalogFn", "src/lambdaas/catalog.ts");
const recommendationsFn = createNodeFn("RecommendationsFn", "src/lambdaas/recommendations.ts");
const healthFn = createNodeFn("HealthFn", "src/lambdaas/health.ts");
const insightsFn = createNodeFn("InsightsFn", "src/lambdaas/insights.ts");

eventsTable.grantReadWriteData(eventsFn);
itemsTable.grantReadWriteData(catalogFn);
itemsTable.grantReadData(recommendationsFn);
eventsTable.grantReadData(recommendationsFn);
itemsTable.grantReadData(insightsFn);
eventsTable.grantReadData(insightsFn);
rawEventsBucket.grantPut(eventsFn);

recommendationsFn.addToRolePolicy(
  new iam.PolicyStatement({
    actions: ["personalize:GetRecommendations"],
    resources: ["*"]
  })
);

eventsFn.addToRolePolicy(
  new iam.PolicyStatement({
    actions: ["personalize:PutEvents"],
    resources: ["*"]
  })
);

```

```

const api = new apigwv2.HttpApi(this, "RecommendationApi", {
  apiName: "recommendation-api",
  corsPreflight: {
    allowHeaders: ["content-type", "authorization"],
    allowMethods: [apigwv2.CorsHttpMethod.ANY],
    allowOrigins: ["*"]
  }
});

api.addRoutes({
  path: "/health",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("HealthIntegration", healthFn)
});

api.addRoutes({
  path: "/events",
  methods: [apigwv2.HttpMethod.POST],
  integration: new integrations.HttpLambdaIntegration("EventsIntegration", eventsFn)
});

api.addRoutes({
  path: "/catalog/items",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("CatalogItemsReadIntegration",
insightsFn)
});

api.addRoutes({
  path: "/catalog/items",
  methods: [apigwv2.HttpMethod.POST],
  integration: new integrations.HttpLambdaIntegration("CatalogIntegration", catalogFn)
});

api.addRoutes({
  path: "/recommendations",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("RecommendationsIntegration",
recommendationsFn)
});

api.addRoutes({
  path: "/catalog/count",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("CatalogCountIntegration", insightsFn)
});

api.addRoutes({
  path: "/catalog/stats",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("CatalogStatsIntegration", insightsFn)
});

```

```

api.addRoutes({
  path: "/events/count",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("EventsCountIntegration", insightsFn)
});

api.addRoutes({
  path: "/events/stats",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("EventsStatsIntegration", insightsFn)
});

api.addRoutes({
  path: "/users/{userId}/summary",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("UserSummaryIntegration", insightsFn)
});

api.addRoutes({
  path: "/users",
  methods: [apigwv2.HttpMethod.GET],
  integration: new integrations.HttpLambdaIntegration("UsersIntegration", insightsFn)
});

let datasetGroupOutput: string | undefined;
let interactionsDatasetOutput: string | undefined;
let itemsDatasetOutput: string | undefined;

if (personalizeStackMode !== "off") {
  rawEventsBucket.addToResourcePolicy(
    new iam.PolicyStatement({
      sid: "AllowPersonalizeListBucket",
      effect: iam.Effect.ALLOW,
      principals: [new iam.ServicePrincipal("personalize.amazonaws.com")],
      actions: ["s3:ListBucket"],
      resources: [rawEventsBucket.bucketArn]
    })
  );

  rawEventsBucket.addToResourcePolicy(
    new iam.PolicyStatement({
      sid: "AllowPersonalizeReadObjects",
      effect: iam.Effect.ALLOW,
      principals: [new iam.ServicePrincipal("personalize.amazonaws.com")],
      actions: ["s3:GetObject"],
      resources: [rawEventsBucket.arnForObjects(`${personalizePrefix}/*`)]
    })
  );

  const datasetGroup = new personalize.CfnDatasetGroup(this, "PersonalizeDatasetGroup", {
    name: personalizeGroupName,
    domain: "ECOMMERCE"
  });
}

```

					ІАЛЦ.467200.003 Д4	Арк.
						29
Зм.	Арк.	№ докум.	Підпис	Дата		

```

const interactionsSchema = new personalize.CfnSchema(this, "PersonalizeInteractionsSchema",
{
    name: `${personalizeGroupName}-interactions-schema`,
    domain: "ECOMMERCE",
    schema: JSON.stringify({
        type: "record",
        name: "Interactions",
        namespace: "com.amazonaws.personalize.schema",
        fields: [
            { name: "USER_ID", type: "string" },
            { name: "ITEM_ID", type: "string" },
            { name: "TIMESTAMP", type: "long" },
            { name: "EVENT_TYPE", type: "string" }
        ],
        version: "1.0"
    })
});

const itemsSchema = new personalize.CfnSchema(this, "PersonalizeItemsSchema", {
    name: `${personalizeGroupName}-items-schema`,
    domain: "ECOMMERCE",
    schema: JSON.stringify({
        type: "record",
        name: "Items",
        namespace: "com.amazonaws.personalize.schema",
        fields: [
            { name: "ITEM_ID", type: "string" },
            { name: "PRICE", type: "float" },
            { name: "CATEGORY_L1", type: "string", categorical: true },
            { name: "BRAND", type: ["null", "string"], categorical: true },
        ],
        version: "1.0"
    })
});

const importJobsEnabled = personalizeStackMode === "imports";
if (importJobsEnabled && !personalizeImportRoleArn) {
    throw new Error("PERSONALIZE_IMPORT_ROLE_ARN is required when
PERSONALIZE_STACK_MODE=imports");
}

const interactionsDatasetProps: personalize.CfnDatasetProps = {
    datasetGroupArn: datasetGroup.attrDatasetGroupArn,
    datasetType: "Interactions",
    name: process.env.PERSONALIZE_INTERACTIONS_DATASET_NAME ?? "reco-interactions",
    schemaArn: interactionsSchema.attrSchemaArn,
    ...(importJobsEnabled
    ? {
        datasetImportJob: {
            jobName: `${personalizeGroupName}-interactions-import`,
            dataSource: {
                dataLocation:
`s3://${rawEventsBucket.bucketName}/${personalizePrefix}/interactions.csv`

```

```

        },
        roleArn: personalizeImportRoleArn
    }
}
: {}))
};

const itemsDatasetProps: personalize.CfnDatasetProps = {
    datasetGroupArn: datasetGroup.attrDatasetGroupArn,
    datasetType: "Items",
    name: process.env.PERSONALIZE_ITEMS_DATASET_NAME ?? "reco-items",
    schemaArn: itemsSchema.attrSchemaArn,
    ...(importJobsEnabled
    ? {
        datasetImportJob: {
            jobName: `${personalizeGroupName}-items-import`,
            dataSource: {
                dataLocation:
`s3://${rawEventsBucket.bucketName}/${personalizePrefix}/items.csv`,
            },
            roleArn: personalizeImportRoleArn
        }
    }
    : {}))
};

const interactionsDataset = new personalize.CfnDataset(this,
"PersonalizeInteractionsDataset", interactionsDatasetProps);
interactionsDataset.addDependency(datasetGroup);
interactionsDataset.addDependency(interactionsSchema);

const itemsDataset = new personalize.CfnDataset(this, "PersonalizeItemsDataset",
itemsDatasetProps);
itemsDataset.addDependency(datasetGroup);
itemsDataset.addDependency(itemsSchema);

datasetGroupOutput = datasetGroup.attrDatasetGroupArn;
interactionsDatasetOutput = interactionsDataset.attrDatasetArn;
itemsDatasetOutput = itemsDataset.attrDatasetArn;

new cdk.CfnOutput(this, "PersonalizeDatasetGroupArn", { value:
datasetGroup.attrDatasetGroupArn });
new cdk.CfnOutput(this, "PersonalizeInteractionsDatasetArn", { value:
interactionsDataset.attrDatasetArn });
new cdk.CfnOutput(this, "PersonalizeItemsDatasetArn", { value: itemsDataset.attrDatasetArn
});
new cdk.CfnOutput(this, "PersonalizeStackMode", { value: personalizeStackMode });
}

new cdk.CfnOutput(this, "ApiUrl", { value: api.apiEndpoint });
new cdk.CfnOutput(this, "ItemsTableName", { value: itemsTable.tableName });
new cdk.CfnOutput(this, "EventsTableName", { value: eventsTable.tableName });
new cdk.CfnOutput(this, "RawEventsBucketName", { value: rawEventsBucket.bucketName });

```

```

    if (datasetGroupOutput) {
      new cdk.CfnOutput(this, "PersonalizeDatasetGroupArnMirror", { value: datasetGroupOutput });
    }
    if (interactionsDatasetOutput) {
      new cdk.CfnOutput(this, "PersonalizeInteractionsDatasetArnMirror", { value:
interactionsDatasetOutput });
    }
    if (itemsDatasetOutput) {
      new cdk.CfnOutput(this, "PersonalizeItemsDatasetArnMirror", { value: itemsDatasetOutput });
    }
  }
}

```

					ІАЛЦ.467200.003 Д4	Арк.
						32
Зм.	Арк.	№ докум.	Підпис	Дата		

TrainingDataGenerationService:

Файл: scripts/personalize/generate-training-data.ts

```
import { randomUUID } from "node:crypto";
import type { CatalogItem, EventType, UserEvent } from "../../src/common/types";
import { batchUpsertItems, putEvent } from "../../src/services/dynamo";
import { requireEnv } from "../_shared";

const ITEMS_TABLE_NAME = requireEnv("ITEMS_TABLE_NAME");
const EVENTS_TABLE_NAME = requireEnv("EVENTS_TABLE_NAME");

const categories = ["phones", "laptops", "audio", "gaming", "wearables"] as const;
const priceTiers = ["budget", "mid", "premium", "flagship"] as const;

type Category = (typeof categories)[number];
type PriceTier = (typeof priceTiers)[number];

type GeneratedItem = CatalogItem & {
  metadata: {
    seed: true;
    priceTier: PriceTier;
    modelIndex: number;
  };
};

type UserPersona = {
  userId: string;
  preferredCategory: Category;
  secondaryCategory: Category;
  preferredBrand: string;
  secondaryBrand: string;
  priceTier: PriceTier;
  explorationRate: number;
};

const brandsByCategory: Record<Category, string[]> = {
  phones: ["Apple", "Samsung", "Google"],
  laptops: ["Apple", "Lenovo", "Dell"],
  audio: ["Sony", "Apple", "Bose"],
  gaming: ["Sony", "Microsoft", "Nintendo"],
  wearables: ["Apple", "Samsung", "Garmin"]
};

const relatedCategories: Record<Category, Category[]> = {
  phones: ["wearables", "audio"],
  laptops: ["audio", "gaming"],
  audio: ["phones", "gaming"],
  gaming: ["audio", "laptops"],
  wearables: ["phones", "audio"]
};

const priceTemplatesByCategory: Record<Category, number[]> = {
  phones: [499, 699, 899, 1199],
  laptops: [799, 1099, 1399, 1799],
  audio: [149, 249, 379, 549],
};
```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		33

```

    gaming: [299, 449, 649, 899],
    wearables: [199, 349, 499, 799]
  };

const brandPriceOffsets: Record<string, number> = {
  Apple: 120,
  Samsung: 40,
  Google: 20,
  Lenovo: -80,
  Dell: -30,
  Sony: 35,
  Bose: 90,
  Microsoft: 60,
  Nintendo: 15,
  Garmin: 70
};

const defaultUserCount = parsePositiveInt(process.env.PERSONALIZE_SEED_USER_COUNT, 90);
const minSessionsPerUser = parsePositiveInt(process.env.PERSONALIZE_SEED_MIN_SESSIONS, 8);
const maxSessionsPerUser = Math.max(minSessionsPerUser,
  parsePositiveInt(process.env.PERSONALIZE_SEED_MAX_SESSIONS, 12));

function parsePositiveInt(value: string | undefined, fallback: number): number {
  const parsed = Number(value);
  return Number.isInteger(parsed) && parsed > 0 ? parsed : fallback;
}

function pick<T>(list: readonly T[]): T {
  return list[Math.floor(Math.random() * list.length)];
}

function randomInt(min: number, max: number): number {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}

function chance(probability: number): boolean {
  return Math.random() < probability;
}

function priceTierForModel(index: number): PriceTier {
  return priceTiers[Math.min(index, priceTiers.length - 1)];
}

function priceTierDistance(a: PriceTier, b: PriceTier): number {
  return Math.abs(priceTiers.indexOf(a) - priceTiers.indexOf(b));
}

function clamp(value: number, min: number, max: number): number {
  return Math.max(min, Math.min(max, value));
}

function makeItems(): GeneratedItem[] {
  const items: GeneratedItem[] = [];
  let idx = 1;

```

```
for (const category of categories) {
  const basePrices = priceTemplatesByCategory[category];

  for (const brand of brandsByCategory[category]) {
    for (let modelIndex = 0; modelIndex < basePrices.length; modelIndex += 1) {
      const priceTier = priceTierForModel(modelIndex);
      const price = basePrices[modelIndex] + (brandPriceOffsets[brand] ?? 0);
      const popularityScore = 35 + modelIndex * 8 + (brand === "Apple" ? 10 : 0);

      items.push({
        itemId: `ml-sku-${idx}`,
        title: `${brand} ${category} model ${modelIndex + 1}`,
        category,
        brand,
        price,
        tags: [category, brand.toLowerCase(), priceTier, `model-${modelIndex + 1}`],
        popularityScore,
        isActive: true,
        metadata: {
          seed: true,
          priceTier,
          modelIndex: modelIndex + 1
        }
      });

      idx += 1;
    }
  }
}

return items;
}
```

```
function buildPersona(userId: string, index: number): UserPersona {
  const preferredCategory = categories[index % categories.length];
  const secondaryCategory = relatedCategories[preferredCategory][index %
relatedCategories[preferredCategory].length];
  const preferredBrand = brandsByCategory[preferredCategory][index %
brandsByCategory[preferredCategory].length];
  const secondaryBrand = brandsByCategory[secondaryCategory][(index + 1) %
brandsByCategory[secondaryCategory].length];
  const priceTier = priceTiers[index % priceTiers.length];

  return {
    userId,
    preferredCategory,
    secondaryCategory,
    preferredBrand,
    secondaryBrand,
    priceTier,
    explorationRate: 0.08 + (index % 4) * 0.04
  };
}
```

```

function chooseSessionCategory(persona: UserPersona): Category {
  const roll = Math.random();
  if (roll < 0.72) return persona.preferredCategory;
  if (roll < 0.92) return persona.secondaryCategory;
  return pick(categories);
}

function scoreItemForPersona(item: GeneratedItem, persona: UserPersona, focusCategory: Category):
number {
  let score = 1;

  if (item.category === focusCategory) score += 4;
  if (item.category === persona.preferredCategory) score += 4;
  if (item.category === persona.secondaryCategory) score += 1.5;
  if (item.brand === persona.preferredBrand) score += 3.5;
  if (item.brand === persona.secondaryBrand) score += 1;

  const distance = priceTierDistance(item.metadata.priceTier, persona.priceTier);
  if (distance === 0) score += 4;
  if (distance === 1) score += 1.5;

  return score;
}

function weightedPickItem(items: GeneratedItem[], persona: UserPersona, focusCategory: Category,
excludedIds: Set<string>): GeneratedItem {
  const pool = items.filter((item) => !excludedIds.has(item.itemId));
  const totalWeight = pool.reduce((sum, item) => sum + scoreItemForPersona(item, persona,
focusCategory), 0);
  let roll = Math.random() * totalWeight;

  for (const item of pool) {
    roll -= scoreItemForPersona(item, persona, focusCategory);
    if (roll <= 0) return item;
  }

  return pool[pool.length - 1];
}

function pickSessionItems(items: GeneratedItem[], persona: UserPersona, focusCategory: Category):
GeneratedItem[] {
  const desiredCount = randomInt(2, 4);
  const picked: GeneratedItem[] = [];
  const excludedIds = new Set<string>();

  while (picked.length < desiredCount) {
    const item = weightedPickItem(items, persona, focusCategory, excludedIds);
    picked.push(item);
    excludedIds.add(item.itemId);
  }

  return picked;
}

```

```

function buildSearchQuery(persona: UserPersona, focusCategory: Category): string {
  if (focusCategory === persona.preferredCategory) {
    return `${persona.preferredBrand.toLowerCase()} ${focusCategory}`;
  }

  return `${persona.priceTier} ${focusCategory}`;
}

function buildEventTypes(item: GeneratedItem, persona: UserPersona, focusCategory: Category):
EventType[] {
  const affinity =
    (item.category === persona.preferredCategory ? 0.35 : 0) +
    (item.brand === persona.preferredBrand ? 0.25 : 0) +
    (priceTierDistance(item.metadata.priceTier, persona.priceTier) === 0 ? 0.2 : 0) +
    (item.category === focusCategory ? 0.1 : 0);

  const clickProbability = clamp(0.55 + affinity, 0.35, 0.98);
  const cartProbability = clamp(0.1 + affinity * 0.7, 0.05, 0.8);
  const purchaseProbability = clamp(0.03 + affinity * 0.45, 0.02, 0.45);

  const eventTypes: EventType[] = ["view"];
  const clicked = chance(clickProbability);
  const addedToCart = chance(cartProbability);
  const purchased = addedToCart && chance(purchaseProbability);

  if (clicked) eventTypes.push("click");
  if (addedToCart || purchased) eventTypes.push("add_to_cart");
  if (purchased) eventTypes.push("purchase");

  return eventTypes;
}

async function main(): Promise<void> {
  const items = makeItems();
  await batchUpsertItems(ITEMS_TABLE_NAME, items);

  const users = Array.from({ length: defaultUserCount }, (_, index) => `ml-user-${index + 1}`);
  const personas = users.map((userId, index) => buildPersona(userId, index));
  const events: UserEvent[] = [];
  const now = Date.now();

  for (const [index, persona] of personas.entries()) {
    const sessionCount = randomInt(minSessionsPerUser, maxSessionsPerUser);

    for (let sessionIndex = 0; sessionIndex < sessionCount; sessionIndex += 1) {
      const focusCategory = chooseSessionCategory(persona);
      const sessionId = `${persona.userId}-session-${sessionIndex + 1}`;
      const sessionStart =
        now -
        (index * maxSessionsPerUser + sessionIndex) * 6 * 60 * 60 * 1000 -
        randomInt(0, 45) * 60 * 1000;

      let cursor = sessionStart;
    }
  }
}

```

```

if (chance(0.35)) {
  events.push({
    eventId: randomUUID(),
    userId: persona.userId,
    sessionId,
    eventType: "search",
    timestamp: new Date(cursor).toISOString(),
    query: buildSearchQuery(persona, focusCategory),
    properties: { seed: true }
  });
  cursor += randomInt(20, 45) * 1000;
}

const sessionItems = pickSessionItems(items, persona, focusCategory);
for (const item of sessionItems) {
  const eventTypes = buildEventTypes(item, persona, focusCategory);

  for (const eventType of eventTypes) {
    events.push({
      eventId: randomUUID(),
      userId: persona.userId,
      sessionId,
      itemId: item.itemId,
      eventType,
      timestamp: new Date(cursor).toISOString(),
      query: eventType === "view" && chance(0.15) ? `${item.brand?.toLowerCase()}
${item.category}` : undefined,
      properties: {
        seed: true,
        preferredCategory: persona.preferredCategory,
        preferredBrand: persona.preferredBrand,
        priceTier: persona.priceTier
      }
    });

    cursor += randomInt(15, 70) * 1000;
  }
}
}

for (let i = 0; i < events.length; i += 25) {
  await Promise.all(events.slice(i, i + 25).map((event) => putEvent(EVENTS_TABLE_NAME, event)));
}

const exportableEvents = events.filter((event) => event.eventType !== "search");

console.log(
  JSON.stringify(
    {
      ok: true,
      items: items.length,
      users: users.length,

```

```

    sessionsPerUser: {
      min: minSessionsPerUser,
      max: maxSessionsPerUser
    },
    events: {
      total: events.length,
      exportableInteractions: exportableEvents.length
    },
    personas: personas.slice(0, 5)
  },
  null,
  2
)
);
}

main().catch((error) => {
  console.error(error);
  process.exit(1);
});

```

DatasetExportService:

Файл: *scripts/personalize/export-datasets.ts*

```
import fs from 'node:fs/promises';
import path from 'node:path';
import { csvEscape, putTextToS3, requireEnv, requirePersonalizeBucketName, scanAll,
toUnixTimestamp } from './_shared';

type ItemRow = {
  itemId?: string;
  title?: string;
  category?: string;
  brand?: string;
  price?: number | string;
  createdAt?: string | number;
  updatedAt?: string | number;
};

type EventRow = {
  eventId?: string;
  userId?: string;
  sessionId?: string;
  itemId?: string;
  eventType?: string;
  timestamp?: string | number;
  createdAt?: string | number;
};

const EXPORTABLE_EVENT_TYPES = new Set(['view', 'click', 'add_to_cart', 'purchase']);

function normalizeEventType(input: string): string | null {
  const value = input.trim().toLowerCase();
  if (!EXPORTABLE_EVENT_TYPES.has(value)) return null;

  switch (value) {
    case 'view':
      return 'View';
    case 'click':
      return 'Click';
    case 'add_to_cart':
      return 'AddToCart';
    case 'purchase':
      return 'Purchase';
    default:
      return null;
  }
}

async function main(): Promise<void> {
  const itemsTable = requireEnv('ITEMS_TABLE_NAME');
  const eventsTable = requireEnv('EVENTS_TABLE_NAME');
  const bucket = requirePersonalizeBucketName();
  const prefix = process.env.PERSONALIZE_PREFIX || 'personalize/input';
```

					ІАЛЦ.467200.003 Д4	Арк.
Зм.	Арк.	№ докум.	Підпис	Дата		40

```

const outputDir = path.resolve(process.cwd(), 'generated', 'personalize');
await fs.mkdir(outputDir, { recursive: true });

const [items, events] = await Promise.all([
  scanAll<ItemRow>(itemsTable),
  scanAll<EventRow>(eventsTable),
]);

const itemLines: string[] = ['ITEM_ID,PRICE,CATEGORY_L1,BRAND'];
let skippedItems = 0;

for (const item of items) {
  const itemId = item.itemId?.trim();
  const category = item.category?.trim();
  const price = typeof item.price === 'number' ? item.price : Number(item.price);

  if (!itemId || !category || !Number.isFinite(price) || price < 0) {
    skippedItems += 1;
    continue;
  }

  itemLines.push(
    [
      csvEscape(itemId),
      csvEscape(price),
      csvEscape(category),
      csvEscape(item.brand ?? ''),
    ].join(','),
  );
}

const interactionLines: string[] = ['USER_ID,ITEM_ID,TIMESTAMP,EVENT_TYPE'];

for (const event of events) {
  const eventType = event.eventType ? normalizeEventType(event.eventType) : null;
  const itemId = event.itemId?.trim();
  const userId = event.userId?.trim() || event.sessionId?.trim();
  const ts = toUnixTimestamp(event.timestamp ?? event.createdAt);

  if (!eventType || !itemId || !userId || !ts) continue;

  interactionLines.push(
    [csvEscape(userId), csvEscape(itemId), csvEscape(ts), csvEscape(eventType)].join(','),
  );
}

const itemsCsv = `${itemLines.join('\n')}\n`;
const interactionsCsv = `${interactionLines.join('\n')}\n`;

await fs.writeFile(path.join(outputDir, 'items.csv'), itemsCsv, 'utf8');
await fs.writeFile(path.join(outputDir, 'interactions.csv'), interactionsCsv, 'utf8');

```

```

const itemsS3Uri = await putTextToS3(bucket, `${prefix}/items.csv`, itemsCsv);
const interactionsS3Uri = await putTextToS3(bucket, `${prefix}/interactions.csv`,
interactionsCsv);

console.log(
  JSON.stringify(
    {
      ok: true,
      counts: {
        items: Math.max(0, itemLines.length - 1),
        interactions: Math.max(0, interactionLines.length - 1),
      },
      skipped: {
        itemsMissingRequiredFields: skippedItems,
      },
      outputDir,
      s3: {
        items: itemsS3Uri,
        interactions: interactionsS3Uri,
      },
    },
    null,
    2,
  ),
);
}

main().catch((error) => {
  console.error(error);
  process.exit(1);
});

```

					ІАЛЦ.467200.003 Д4	Арк.
						42
Зм.	Арк.	№ докум.	Підпис	Дата		