ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інформаційна система порівняння ORM-фреймворків на основі показників продуктивності виконання операцій»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Владислав СМІРНОВ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62
Владислав СМІРНОВ

Керівник: Максим КУКЛІНСЬКИЙ
канд. техн. наук, доцент

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Смірнову Владиславу Олександровичу

1. Тема кваліфікаційної роботи: «Інформаційна система порівняння ORM-фреймворків на основі показників продуктивності виконання операцій»

керівник кваліфікаційної роботи Максим КУКЛІНСЬКИЙ, канд. техн. наук, доцент

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, параметри оцінювання ORM-фреймворків, класифікація ORM-фреймворків, показники продуктивності виконання операцій ORM-фреймворків.

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

1. Аналіз сучасних ORM-фреймворків та проблеми оцінювання їх продуктивності.

2. Методи та моделі побудови інформаційної системи порівняння ORM-фреймворків.
3. Реалізація та експериментальне дослідження інформаційної системи.

5. Перелік ілюстративного матеріалу:
1. Мета, об’єкт та предмет дослідження
2. ORM як проміжний шар доступу до даних.
3. Класифікація ORM-фреймворків за рівнем абстракції.
4. ORM-фреймворки .NET-екосистеми.
5. Основні ORM-операції.
6. Стратегії завантаження даних: Lazy та Eager Loading.
7. Постановка задачі.
8. Критерії оцінювання ORM-фреймворків.
9. Інтерфейс користувача системи
10. Вхідні параметри сценарію
11. Алгоритм рекомендації
12. Висновки

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10-04.11.25	
2	Вивчення матеріалів для аналізу сучасних ORM-фреймворків	05.11-12.11.25	
3	Дослідження методів побудови інформаційної системи порівняння ORM-фреймворків	13.11-19.11.25	
4	Формування стеку технологій для реалізації проекту	20.11-26.11.25	
5	Реалізація та експериментальне дослідження інформаційної системи	27.11-03.12.25	
6	Оформлення роботи: вступ, висновки, реферат	04.12-10.12.25	
7	Розробка демонстраційних матеріалів	11.12-16.12.25	
8	Попередній захист роботи	17.12-19.12.25	

Здобувач вищої освіти _____ Владислав СМІРНОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Максим КУКЛІНСЬКИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 102 стор., 2 табл., 26 рис., 29 джерел.

Мета роботи – розробка інформаційної системи, що дозволяє аналізувати та порівнювати ефективність використання ORM-фреймворків та підтримувати процес прийняття рішень при виборі ORM-технологій.

Об'єкт дослідження – процес порівняння показників ORM-фреймворків.

Предмет дослідження – показники оцінювання продуктивності виконання операцій ORM-фреймворками.

У роботі використовувалися наступні методи дослідження:

- аналіз і систематизація наукових і технічних джерел щодо ORM-фреймворків;
- порівняльний аналіз показників продуктивності виконання CRUD-операцій;
- експериментальні методи вимірювання часу виконання запитів;
- статистичний аналіз результатів тестування.

Для досягнення поставленої в роботі мети було розв'язано наступні задачі:

- проаналізовано принципи роботи та архітектурні особливості ORM-фреймворків;
- визначено основні показники продуктивності виконання операцій доступу до даних;
- проведено порівняльний аналіз ORM-фреймворків за часом виконання CRUD-операцій;
- розроблено інформаційну систему для збору, зберігання та аналізу результатів тестування;
- реалізовано механізм візуалізації результатів порівняння продуктивності;

- сформульовано рекомендації щодо вибору ORM-фреймворків для різних сценаріїв використання.

Перспективи подальших досліджень можуть бути спрямовані для розширення переліку ORM-фреймворків і показників продуктивності, дослідженні їх поведінки в умовах високих навантажень і розподілених архітектур, а також у застосуванні інтелектуальних методів аналізу для автоматичного формування рекомендацій щодо вибору ORM залежно від типу застосунку та сценаріїв використання.

Результати дослідженя можуть бути застосовані під час проєктування та розробки веб- і корпоративних інформаційних систем, вибору ORM-фреймворків для .NET-застосунків, оптимізації доступу до баз даних, а також у навчальному процесі та подальших наукових дослідженнях з питань продуктивності програмного забезпечення.

КЛЮЧОВІ СЛОВА: ІНФОРМАЦІЙНА СИСТЕМА, ORM-ФРЕЙМВОРК, ПРОДУКТИВНІСТЬ, CRUD-ОПЕРАЦІЇ, ДОСТУП ДО ДАНИХ, ПОРІВНЯЛЬНИЙ АНАЛІЗ, .NET-ТЕХНОЛОГІЇ.

ABSTRACT

Text part of the master's qualification work: 102 pages, 26 pictures, 2 tables, 29 sources.

The purpose of the work is to develop an information system that enables the analysis and comparison of the efficiency of using ORM-frameworks and supports the decision-making process when selecting ORM-technologies.

Object of research – is the process of comparing the performance indicators of ORM-frameworks.

The subject of the research – is the indicators used to evaluate the performance of ORM-frameworks in executing operations.

The following research methods were used in this work:

- analysis and systematization of scientific and technical sources related to ORM-frameworks;
- comparative analysis of the performance indicators of CRUD operations;
- experimental methods for measuring query execution time;
- statistical analysis of testing results.

To achieve the objective of the work, the following tasks were accomplished:

- analysis of the operating principles and architectural features of ORM-frameworks;
- identification of the main performance indicators of data access operations;
- comparative analysis of ORM-frameworks based on CRUD operation execution time;
- development of an information system for collecting, storing, and analyzing test results;
- implementation of a mechanism for visualizing the results of performance comparison;

• formulation of recommendations for selecting ORM-frameworks for various usage scenarios.

Prospects for further research may be aimed at expanding the range of ORM-frameworks and performance indicators, studying their behavior under high-load conditions and in distributed architectures, as well as applying intelligent analysis methods for the automatic generation of recommendations on ORM selection depending on the type of application and usage scenarios.

The research results can be applied in the design and development of web and enterprise information systems, the selection of ORM-frameworks for .NET-applications, optimization of database access, as well as in the educational process and further scientific research in the field of software performance.

KEYWORDS: INFORMATION SYSTEM, ORM-FRAMEWORK, PERFORMANCE, CRUD OPERATIONS, DATA ACCESS, COMPARATIVE ANALYSIS, .NET TECHNOLOGIES.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

.NET – Платформа від Microsoft для розробки, виконання та підтримки кросплатформових програмних застосунків різними мовами програмування (зокрема C#, F#)

ПЗ – Програмне забезпечення

СКБД – Система керування базою даних

CSS – Каскадні таблиці стилів (англ., Cascading Style Sheets)

CRUD – основні операції управління даними (англ., Create, Read, Update, Delete)

DOM – Об’єктна модель документу (англ., Document Object Model)

IDE – Інтегроване середовище розробки (англ., Integrated Development Environment)

HTML – Мова гіпертекстової розмітки (англ., HyperText Markup Language)

LINQ – Запити інтегровані в мову (англ., Language Integrated Query)

ORM – Об’єктно-реляційне відображення (англ., Object-Relational Mapping)

ВСТУП

Сучасні інформаційні системи характеризуються постійним зростанням обсягів даних, ускладненням бізнес-логіки та підвищеними вимогами до продуктивності й масштабованості програмного забезпечення. Однією з ключових складових таких систем є механізм доступу до даних, який безпосередньо впливає на швидкодію виконання операцій, ефективність використання ресурсів і загальну стабільність роботи програмних рішень. У цьому контексті широкого поширення набули ORM-фреймворки, що забезпечують об’єктно-реляційне відображення та спрощують взаємодію прикладного коду з базами даних.

Незважаючи на зручність використання ORM-технологій, їх застосування не є універсальним і може по-різному впливати на продуктивність інформаційних систем. Різні ORM-фреймворки відрізняються за внутрішньою архітектурою, підходами до формування SQL-запитів, підтримкою асинхронних операцій, механізмами кешування та стратегіями завантаження даних. У результаті вибір ORM-рішення без урахування показників продуктивності може призвести до зниження швидкодії, проблем масштабування та ускладнення супроводу програмного забезпечення.

У зв’язку з цим особливої актуальності набуває завдання об’єктивного порівняння ORM-фреймворків на основі кількісних показників продуктивності виконання операцій. Відсутність універсального інструменту для систематизованого аналізу таких показників ускладнює процес прийняття технічних рішень і зумовлює необхідність розробки спеціалізованої інформаційної системи, здатної автоматизувати збір, аналіз і представлення результатів порівняння.

Метою даної роботи є розробка інформаційної системи, що дозволяє аналізувати та порівнювати ефективність використання ORM-фреймворків і підтримувати процес прийняття рішень при виборі ORM-технологій для розробки

програмного забезпечення. Досягнення поставленої мети передбачає комплексний підхід до оцінювання продуктивності виконання операцій доступу до даних у різних сценаріях використання.

Об’єктом дослідження є процес порівняння показників ORM-фреймворків, що відображає взаємозв’язок між архітектурними особливостями ORM-рішень та їх впливом на продуктивність інформаційних систем.

Предметом дослідження є показники оцінювання продуктивності виконання операцій ORM-фреймворками, зокрема час виконання CRUD-операцій, вплив стратегій завантаження даних та здатність до ефективної роботи з великими обсягами інформації.

Розробка інформаційної системи порівняння ORM-фреймворків на основі показників продуктивності виконання операцій є актуальним і практично значущим завданням, результати якого можуть бути використані як у процесі розробки прикладних інформаційних систем, так і для подальших наукових досліджень у галузі оптимізації доступу до даних.

1 АНАЛІЗ СУЧАСНИХ ORM-ФРЕЙМВОРКІВ ТА ПРОБЛЕМИ ОЦІНЮВАННЯ ЇХ ПРОДУКТИВНОСТІ

1.1 Поняття об’єктно-реляційного відображення та роль ORM у сучасних інформаційних системах

У сучасних інформаційних системах, що орієнтовані на обробку та зберігання структурованих даних, ключову роль відіграють реляційні системи управління базами даних. Водночас програмні застосунки переважно розробляються з використанням об’єктно-орієнтованих мов програмування, у межах яких дані подаються у вигляді класів, об’єктів та їхніх взаємозв’язків. Така різниця між логікою представлення даних у програмному коді та способом їх фізичного зберігання в базі даних призводить до виникнення об’єктно-реляційного розриву, що ускладнює процес проєктування та реалізації інформаційних систем.

Об’єктно-реляційне відображення (Object-Relational Mapping, ORM) є концептуальним і технологічним підходом, спрямованим на подолання зазначеного розриву шляхом встановлення відповідності між об’єктною моделлю прикладного програмного забезпечення та реляційною моделлю даних. Основна ідея ORM полягає в автоматизованому перетворенні структур даних: класи предметної області співвідносяться з таблицями бази даних, атрибути класів – зі стовпцями, а екземпляри об’єктів – із записами таблиць. Взаємозв’язки між об’єктами, зокрема асоціації, агрегації та спадкування, реалізуються через зовнішні ключі, допоміжні таблиці або спеціалізовані механізми відображення [1].

З теоретичної точки зору ORM можна розглядати як проміжний абстрактний шар між прикладною логікою та рівнем зберігання даних. Цей шар інкапсулює деталі взаємодії з конкретною системою управління базами даних і надає розробнику уніфікований інтерфейс доступу до даних. Таким чином, робота

з базою даних відбувається на рівні об’єктів і методів мови програмування, що зменшує потребу у прямому використанні мови SQL та знижує залежність програмного коду від конкретної системи керування базою даних (СКБД). Схема взаємодії ОРМ з базою даних наведена на рис. 1.1.

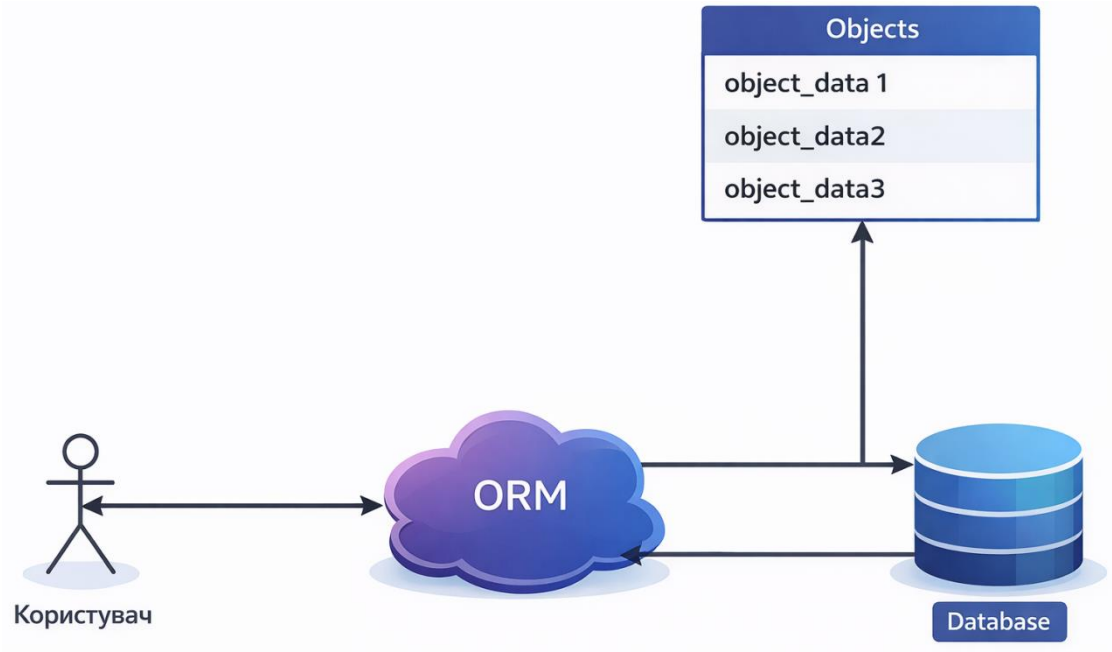


Рис. 1.1 Діаграма взаємодії користувача з базою даних через ORM

Використання ORM-фреймворків у сучасних інформаційних системах суттєво впливає на архітектуру програмного забезпечення. Більшість таких фреймворків забезпечує реалізацію типових задач доступу до даних, зокрема керування транзакціями, синхронізацію стану об’єктів з базою даних, обробку конкурентного доступу та підтримку цілісності даних. Крім того, ORM часто інтегрується з іншими компонентами програмної платформи, що сприяє побудові багаторівневих та модульних архітектур.

Важливою теоретичною особливістю ORM є підтримка різних стратегій завантаження даних і оптимізації доступу до них. Зокрема, застосовуються механізми ледачого та жадібного завантаження пов’язаних сутностей, а також кешування результатів запитів. Такі механізми дозволяють зменшити кількість звернень до бази даних і підвищити ефективність роботи системи, однак водночас вимагають коректного налаштування та глибокого розуміння внутрішніх процесів ORM [2].

Незважаючи на значні переваги, використання об’єктно-реляційного відображення пов’язане з низкою обмежень, насамперед у контексті продуктивності виконання операцій. Додатковий рівень абстракції між прикладним кодом і базою даних спричиняє накладні витрати, пов’язані з генерацією SQL-запитів, керуванням життєвим циклом об’єктів та обробкою метаданих. У випадках некоректного проєктування або неефективного використання ORM це може призводити до зниження швидкодії, збільшення часу відгуку та надмірного навантаження на систему управління базами даних.

У зв’язку з цим у сучасних наукових і прикладних дослідженнях особливу увагу приділяють аналізу та порівнянню ORM-фреймворків за показниками продуктивності виконання базових і складних операцій. Обґрунтований вибір ORM-інструменту з урахуванням специфіки предметної області, обсягу даних і вимог до швидкодії є важливим чинником забезпечення ефективності інформаційних систем. Саме це зумовлює актуальність досліджень, спрямованих на розроблення інформаційних систем для порівняльної оцінки ORM-фреймворків на основі кількісних показників продуктивності.

1.2 Класифікація ORM-фреймворків

Різноманіття сучасних ORM-фреймворків зумовлене відмінностями у вимогах до архітектури інформаційних систем, продуктивності, гнучкості доступу до даних і рівня абстракції. З теоретичної точки зору ORM-інструменти доцільно розглядати не лише як засоби відображення об’єктів у реляційні структури, а як складні програмні компоненти, що реалізують певну філософію взаємодії між прикладною логікою та базою даних. Саме тому їх класифікація ґрунтується на сукупності концептуальних, архітектурних та функціональних ознак [1-2].

Одним із ключових критеріїв класифікації ORM-фреймворків є рівень абстракції, який вони надають розробнику під час роботи з базою даних. За рівнем абстракції доступу до даних ORM-фреймворки можна класифікувати на наступні:

- фреймворки з високим рівнем абстракції повністю приховують реляційну модель від прикладного коду. Взаємодія з даними здійснюється виключно через об’єктну модель, а SQL-запити формуються автоматично. Такий підхід спрощує розробку, проте може ускладнювати контроль над ефективністю виконання запитів.

- ORM з помірним рівнем абстракції поєднують автоматичне відображення об’єктів із можливістю тонкого налаштування запитів. Вони дозволяють використовувати мову запитів високого рівня (наприклад, LINQ), але при цьому залишають розробнику контроль над структурою вибірок.

- мікро-ORM та ORM низького рівня абстракції мінімізують накладні витрати та майже не приховують SQL. Вони зосереджені на швидкому мапінгу результатів запитів у об’єкти, що робить їх привабливими для високонавантажених систем.

З позицій програмної інженерії ORM-фреймворки також доцільно класифікувати за використовуваними патернами доступу до даних:

- Active Record – логіка доступу до даних безпосередньо інтегрована в класи сутностей.

- Data Mapper – об’єкти предметної області відокремлені від механізмів збереження та вибірки даних.

- Repository-орієнтований підхід – доступ до ORM інкапсулюється у спеціалізованих компонентах, що підвищує тестованість та масштабованість системи.

Необхідно відмітити, що ORM-інструменти створюються для конкретних мов програмування чи їхніх екосистем. Існують рішення, що тісно інтегровані з JVM, для .NET – відповідні реалізації під C#, а для Python – пакети, адаптовані до екосистеми цієї мови. Тому у цьому аспекті ORM також класифікуються:

- ORM у середовищі Java.

Hibernate – один із найвідоміших ORM-фреймворків для платформи Java, що реалізує ідеї об’єктно-реляційного відображення та стандарти Jakarta Persistence API (JPA). Він автоматизує створення SQL-запитів, підтримує роботу

зі складними зв’язками між сутностями, механізми кешування та оптимізації вибірок, а також пропонує можливості конфігурації через анотації чи XML-опис.

Іншим прикладом у цій категорії є Apache Cayenne – ORM-фреймворк з відкритим вихідним кодом, який також підтримує управління транзакціями, генерацію SQL та управління зв’язками між об’єктами. Його ключовою особливістю є можливість інтеграції з різними джерелами даних і сервісами.

Підхід iBATIS (MyBatis) відрізняється від традиційних ORM тим, що фокусує увагу на керуванні SQL-запитами; маппінг між об’єктами і запитами визначається й конфігурується вручну через XML-файли, що дає розробнику гнучкість контролю над SQL, але зменшує рівень автоматизації.

- ORM у .NET-екосистемі

У .NET-середовищі одним із найпопулярніших рішень є Entity Framework (EF) – ORM-технологія від Microsoft, що тісно інтегрована з платформою .NET та підтримує LINQ для формування запитів на рівні об’єктної моделі. EF пропонує кілька підходів до роботи з моделями даних, зокрема Code-First та Database-First, що дає гнучкість у проєктуванні структури даних.

NHibernate являє собою порт Hibernate для платформи .NET і підтримує схожу функціональність із мапінгом об’єктів .NET у реляційні структури, включно з кешуванням, lazy loading та розширеними можливостями конфігурації.

- ORM у динамічних мовах програмування

У середовищі Python одним із базових ORM є SQLAlchemy, що поєднує потужний інструментарій для формування запитів із високим ступенем гнучкості та можливостями низькорівневого контролю за SQL-операціями.

Крім того, багато веб-фреймворків містять власні ORM-компоненти: так, Django ORM інтегрований безпосередньо в популярний фреймворк Django і забезпечує зручний API для CRUD-операцій, складних запитів і міграцій бази даних.

У JavaScript-середовищі ORM-рішення, такі як Sequelize, пропонують підтримку різних СУБД (PostgreSQL, MySQL, SQLite і т.д.) та асинхронний API для Node.js-додатків, що робить їх привабливими для сучасних веб-проєктів.

Класифікація ORM-фреймворків показує, що сучасні засоби об’єктно-реляційного відображення суттєво відрізняються за рівнем абстракції, архітектурними підходами та ступенем впливу на продуктивність інформаційних систем. Частина фреймворків орієнтована на максимальну автоматизацію доступу до даних і спрощення розробки, приховуючи деталі взаємодії з реляційною базою даних, тоді як інші надають розробнику більший контроль над SQL-запитами, зменшуючи накладні витрати та підвищуючи передбачуваність швидкодії [1-2].

Застосування різних архітектурних патернів, таких як Active Record або Data Mapper, зумовлює відмінності у структурі програмного коду, рівні його гнучкості та складності супроводу. Крім того, на вибір ORM-фреймворка істотно впливають вимоги до масштабованості, навантаження та специфіка предметної області.

Жоден ORM-фреймворк не є універсальним рішенням для всіх типів задач, а їх доцільність використання визначається компромісом між зручністю розробки та ефективністю виконання операцій. Саме це обґрунтовує необхідність подальшого порівняльного аналізу ORM-фреймворків на основі кількісних показників продуктивності, що є важливим етапом у проєктуванні сучасних інформаційних систем.

1.3 Огляд популярних ORM-фреймворків .NET-екосистеми

Платформа .NET є однією з найбільш поширених технологічних основ для розроблення корпоративних, веб- та сервісноорієнтованих інформаційних систем. Значна частина таких систем передбачає інтенсивну взаємодію з реляційними базами даних, що зумовлює необхідність використання ефективних засобів об’єктно-реляційного відображення. У межах .NET-екосистеми сформувався широкий спектр ORM-фреймворків, які відрізняються за рівнем абстракції, підходами до формування запитів, архітектурними принципами та впливом на продуктивність виконання операцій.

Огляд ORM-фреймворків .NET-екосистеми дозволяє систематизувати наявні рішення, визначити їхні концептуальні особливості та області доцільного застосування. Аналіз таких інструментів є важливим з огляду на те, що вибір ORM безпосередньо впливає на архітектуру інформаційної системи, складність її реалізації та показники швидкодії.

Тому для побудови інформаційної системи порівняння ORM були обрані та проаналізовані наступні фреймворки:

- *Entity Framework Core (EF Core).* Entity Framework Core є сучасним ORM-фреймворком для платформи .NET, орієнтованим на кросплатформені застосунки та модульну архітектуру. З концептуальної точки зору EF Core реалізує підхід Data Mapper і забезпечує високий рівень абстракції доступу до даних. Формування запитів здійснюється за допомогою LINQ, що дозволяє інтегрувати логіку доступу до даних безпосередньо в об’єктно-орієнтований код.

EF Core підтримує різні підходи до проєктування моделей даних, зокрема ініціалізацію моделі з коду або на основі існуючої бази даних. Водночас складність внутрішніх механізмів відстеження стану об’єктів і генерації SQL може призводити до помітних накладних витрат, що робить питання продуктивності особливо актуальним у контексті його використання [3].

- *NHibernate.* NHibernate є ORM-фреймворком, що історично сформувався як адаптація ідей Hibernate для платформи .NET. Він реалізує класичний підхід Data Mapper і орієнтований на складні доменні моделі з багатим набором зв’язків між сутностями. NHibernate підтримує широкий спектр стратегій кешування, механізмів лінивого завантаження та конфігурації мапінгу.

З теоретичної точки зору NHibernate характеризується високою гнучкістю та потужністю, однак ці переваги досягаються за рахунок складності налаштування і значних накладних витрат. Тому він частіше застосовується у великих корпоративних системах, де складність предметної області переважає вимоги до максимальної швидкодії [4].

- *Linq2Db.* Linq2Db займає проміжне положення між повноцінними ORM та мікро-ORM. Його ключовою особливістю є орієнтація на LINQ як основний

інструмент формування запитів при мінімальному використанні механізмів автоматичного відстеження стану об’єктів. Це дозволяє досягти більш прогнозованої продуктивності та зменшити накладні витрати.

З архітектурної точки зору Linq2Db надає розробнику більший контроль над структурою SQL-запитів порівняно з класичними ORM, зберігаючи при цьому зручність типізованого доступу до даних. Такий підхід робить його придатним для систем, у яких важливий баланс між зручністю розробки та ефективністю виконання операцій [5].

- *Dapper.* Dapper належить до класу мікро-ORM і є прикладом мінімалістичного підходу до об’єктно-реляційного відображення. Він не намагається приховати SQL від розробника, а натомість фокусується на швидкому перетворенні результатів запитів у об’єкти мови програмування.

Теоретично Dapper можна розглядати як інструмент, що усуває лише найнижчий рівень рутинних операцій, пов’язаних із мапінгом даних, залишаючи повний контроль над запитами на стороні розробника. Завдяки цьому Dapper демонструє високу продуктивність і часто використовується у високонавантажених інформаційних системах або як доповнення до повноцінних ORM [6].

- *ServiceStack.OrmLite.* ServiceStack.OrmLite є легковаговим ORM-фреймворком, орієнтованим на простоту, прозорість та явне управління SQL-запитами. Він використовує код-орієнтований підхід до опису моделей і надає чітке відображення між класами та таблицями без складних механізмів відстеження стану об’єктів.

З концептуальної точки зору OrmLite займає проміжну позицію між мікро-ORM та класичними ORM-фреймворками. Його архітектура мінімізує накладні витрати, що позитивно впливає на продуктивність, але водночас вимагає від розробника більшої уваги до проєктування запитів і транзакцій [7].

Розглянуті ORM-фреймворки демонструють різні підходи до реалізації об’єктно-реляційного відображення – від повної автоматизації доступу до даних до мінімалістичних інструментів із явним використанням SQL. Така

різноманітність підтверджує, що вибір ORM не може бути універсальним і має ґрунтуватися на вимогах до продуктивності, складності предметної області та архітектури інформаційної системи. Саме ці відмінності створюють підґрунтя для науково обґрунтованого порівняння ORM-фреймворків за показниками ефективності виконання операцій.

1.4 Основні операції ORM та їх вплив на продуктивність

Основні операції в ORM відповідають стандарту CRUD (Create, Read, Update, Delete). Крім них є ще додаткові, які призначені для підтримки транзакцій та керування контекстом. Хоч усі вони дозволяють керувати даними у базі, та незважаючи на уніфікований інтерфейс, кожна з таких операцій має власні особливості реалізації, що безпосередньо впливають на продуктивність інформаційної системи. Теоретичний аналіз цих операцій є необхідною передумовою для коректного порівняння ORM-фреймворків за показниками швидкодії [1-2].

- *Операції створення даних (Create).* Операції створення нових записів у базі даних в ORM-контексті зазвичай реалізуються шляхом ініціалізації об’єкта предметної області та його подальшої синхронізації з реляційною моделлю. При цьому ORM виконує низку додаткових дій, зокрема визначення відповідності між властивостями об’єкта та стовпцями таблиці, формування SQL-оператора вставки та керування транзакцією.

З погляду продуктивності, найбільший вплив мають механізми відстеження стану об’єктів і пакетна обробка операцій. У разі масового додавання даних використання стандартних ORM-методів без оптимізації може призводити до значних накладних витрат через індивідуальну обробку кожного об’єкта. Це зумовлює необхідність застосування спеціалізованих механізмів bulk-insert або альтернативних стратегій доступу до даних [1-2].

- *Операції читання даних (Read).* Операції вибірки даних є одними з найкритичніших з точки зору впливу на загальну продуктивність системи. У

ORM-фреймворках вони реалізуються шляхом перетворення високорівневих запитів (наприклад, LINQ або спеціалізованих API) у SQL-вирази, які виконуються на стороні СКБД. Додатковим етапом є мапінг результатів запиту у відповідні об’єкти.

На продуктивність операцій читання істотно впливають такі фактори, як складність сформованого SQL-запиту, кількість звернень до бази даних та стратегія завантаження пов’язаних сутностей. Зокрема, використання ледачого завантаження може зменшувати обсяг переданих даних, але водночас створювати проблему множинних запитів. Навпаки, жадібне завантаження зменшує кількість звернень до бази, проте може призводити до надмірних вибірок [1-2].

- *Операції оновлення даних (Update).* Оновлення даних в ORM-системах пов’язане з механізмами контролю змін об’єктів. Перед виконанням SQL-оператора оновлення ORM визначає, які властивості об’єкта були змінені, і формує відповідний запит. Такий підхід підвищує зручність програмування, проте створює додаткове навантаження на обчислювальні ресурси.

З теоретичної точки зору, найбільші накладні витрати виникають у разі складних доменних моделей з великою кількістю зв’язаних об’єктів. У таких випадках оновлення одного об’єкта може спричиняти каскадні операції, що негативно впливає на час виконання транзакцій і масштабованість системи [1-2].

- *Операції видалення даних (Delete).* Операції видалення в ORM-фреймворках можуть мати різну семантику залежно від налаштувань мапінгу та правил каскадного видалення. Видалення одного об’єкта часто супроводжується перевіркою обмежень цілісності та обробкою залежних сутностей, що збільшує кількість внутрішніх операцій.

З позиції продуктивності такі операції є потенційно ресурсоємними, особливо у випадках, коли каскадні правила реалізуються на рівні ORM, а не безпосередньо в базі даних. Це вимагає додаткових звернень до СКБД та збільшує загальний час виконання операції [1-2].

- *Транзакції та керування контекстом.* Важливим аспектом роботи ORM є підтримка транзакційності. ORM-фреймворки зазвичай беруть на себе управління

межами транзакцій і забезпечують атомарність виконання групи операцій. Хоча це спрощує розробку, некоректне використання транзакцій може призводити до блокувань, зростання часу відгуку та зниження пропускної здатності системи.

Окремої уваги заслуговує керування життєвим циклом контексту доступу до даних. Надмірно довгоживучі контексти можуть накопичувати значну кількість об’єктів, що негативно впливає на споживання пам’яті та швидкодію.

Таким чином, основні операції ORM – створення, читання, оновлення та видалення даних – мають різний характер впливу на продуктивність інформаційних систем. Додаткові механізми, які забезпечують зручність розробки, одночасно створюють накладні витрати, що особливо помітно в умовах високого навантаження або великих обсягів даних. Це свідчить про необхідність системного підходу до аналізу продуктивності ORM-фреймворків і обґрунтованого вибору інструментів залежно від характеру виконуваних операцій.

1.5 Аналіз існуючих підходів до порівняння продуктивності ORM-фреймворків

Порівняння продуктивності ORM-фреймворків – це багатогранна задача, яка виходить далеко за межі простого зіставлення часу виконання конкретних функцій. ORM-інструменти впливають на різні аспекти роботи програмних систем: формування SQL-запитів, обробку результатів, кешування, керування транзакціями та інше. У зв’язку з цим для оцінки їх ефективності розроблено різноманітні підходи, які можна класифікувати за кількома основними критеріями.

1. Прямі бенчмарки на основі виконання CRUD-операцій.

Найпростіший підхід до порівняння ORM-фреймворків – це вимірювання часу виконання базових CRUD операцій. Такі бенчмарки характеризуються запуском стандартного набору запитів для різних ORM і порівнянням отриманих метрик (наприклад, середнього часу відповіді чи пропускної здатності). У

подібних тестах важливо контролювати умови експерименту: розмір і склад бази даних, кількість ітерацій, розмір транзакцій тощо [8].

Окрім ручних бенчмарків, існують спеціалізовані проєкти для порівняння ORM, які автоматизують виконання таких тестів. Наприклад, проект orm-performance-comparator на GitHub містить набір тестів, що вимірюють загальний час виконання REST-операцій для кількох ORM одночасно. Результати зберігаються у вигляді даних, що дозволяє стандартизовано порівнювати швидкодію різних інструментів [8].

2. Використання стандартних галузевих бенчмарків.

Більш формалізовані підходи до оцінювання продуктивності застосовують загальноприйняті галузеві бенчмарки, такі як TPC-C чи TPC-H, які моделюють навантаження при транзакційній обробці або складному аналітичному доступі до даних. У контексті ORM це означає не лише запуск стандартних SQL-запитів, а й оцінювання поведінки ORM-шару під реалістичним навантаженням, що включає одночасні транзакції, конкурентні запити та складні зв'язки між таблицями.

Прикладом такого підходу є використання TPC-C для тестування Python ORM: у одному з описаних випадків аналіз продуктивності показував, що різні ORM суттєво відрізняються за кількістю транзакцій за одиницю часу через відмінності в оптимізації генерації SQL та повторному використанні попередньо згенерованих запитів [9].

3. Багатокритеріальна оцінка продуктивності.

Крім оцінки швидкодії за часом виконання, існують підходи, які включають кілька вимірів якості та продуктивності одночасно. Такі методи враховують обсяг споживаної пам'яті, навантаження на процесор, використання кешу, а іноді навіть складність підтримки коду. Один із таких прикладів – методи багатокритеріальної оптимізації (наприклад, метод Electre), які дозволяють створювати зважені або ранжовані оцінки ORM-рішень на основі декількох показників. Це особливо корисно при виборі ORM для великих проєктів, де питання ресурсів та масштабованості не менш важливі, ніж миттєва швидкість виконання запитів [10].

4. Контекстно-залежні підходи.

Деякі дослідження підкреслюють, що продуктивність ORM значною мірою залежить від контексту використання, включаючи типи даних, які обробляються, структуру запитів та частоту транзакцій. У таких підходах аналіз продуктивності проводиться шляхом моделювання конкретних робочих навантажень – наприклад, складних зв'язків між сутностями або інтенсивного читання та запису при високому рівні конкурентного доступу. Тестування у таких умовах дозволяє розкрити слабкі сторони конкретних ORM у реальних сценаріях роботи, де абстрактні бенчмарки можуть бути недостатньо інформативними [11].

5. Порівняння продуктивності ORM з іншими підходами доступу до даних.

Ще один напрям аналізу – порівняння ORM із ручним SQL або іншими засобами доступу до даних, такими як безпосереднє використання драйверів або більш легкі маппери. Такі порівняння дозволяють оцінити накладні витрати, пов'язані саме з рівнем абстракції ORM. Як показують деякі дослідження в Golang, прямі SQL-запити часто значно швидші, тоді як ORM випереджають їх у зручності використання, але поступаються в продуктивності [11].

Системний аналіз існуючих підходів до порівняння продуктивності ORM-фреймворків свідчить про те, що немає єдиного «універсального» методу оцінки. Найбільш інформативними є комбіновані підходи, які поєднують прямі бенчмарки, галузеві тестові набори, багатокритеріальні оцінки та контекстно-залежні сценарії. Такий підхід дозволяє оцінити не лише час виконання операцій, але й ширший спектр характеристик, що визначають ефективність ORM у конкретних умовах. Такий аналіз може надати рекомендації та обґрунтування вибору технологій для розробки продуктивних і масштабованих інформаційних систем.

1.6 Недоліки наявних методів і засобів оцінювання продуктивності ORM

Питання порівняння продуктивності ORM-фреймворків є важливим етапом при виборі технологій для розроблення сучасних інформаційних систем. Проте,

перед вибором підходів порівняння продуктивності необхідно враховувати, що вони мають низку недоліків, пов’язаних із спрощенням тестових сценаріїв, обмеженим набором метрик, залежністю від середовища виконання та ігноруванням архітектурних особливостей самих ORM. Унаслідок цього отримані показники не завжди коректно будуть відображати реальну ефективність фреймворків у практичних умовах експлуатації. До наявних недоліків, як правило відносять наступні.

1. Обмежена репрезентативність стандартних бенчмарків.

Базові бенчмарки, що засновані на вимірюванні часу виконання окремих CRUD-операцій, найчастіше не враховують складності реальних робочих навантажень. У більшості випадків такі тести виконуються над спрощеними наборами даних і штучними сценаріями, що не репрезентують реальні вимоги багаторівневих інформаційних систем. Це створює ілюзію адекватної оцінки продуктивності ORM, проте не дозволяє виявити поведінку фреймворка у реальних умовах, наприклад при роботі з великою кількістю зв’язків між сутностями чи високим рівнем конкурентного доступу [8].

2. Неврахування комплексності запитів.

Ще одним суттєвим недоліком є те, що багато існуючих методів вимірювання продуктивності обмежуються порівнянням «простих» запитів і не аналізують поведінку ORM при генерації складних SQL-виразів із численними JOIN-операціями, підзапитами або умовами фільтрації. При цьому у реальних прикладних системах саме такі запити формують основну частину навантаження на СУБД, і їх вплив на продуктивність ORM значно відрізняється від поведінки при простих вибірках. Відсутність оцінки складних запитів призводить до неповного розуміння сильних і слабких сторін конкретного ORM-інструмента [9].

3. Ігнорування впливу архітектурних особливостей.

Стандартні тести часто не беруть до уваги архітектурні особливості фреймворків – зокрема механізми кешування, стратегії загрузки, управління транзакціями та відстеження стану об’єктів. Наприклад, ледаче завантаження (lazy loading) може істотно зменшити обсяг переданих даних, але при

неправильному використанні створює проблему *N+1*, що значно збільшує кількість SQL-запитів та знижує продуктивність. Базові тести, що вимірюють лише загальний час виконання, не дозволяють оцінити вплив цих механізмів на реальні робочі навантаження. Це робить результати порівнянь неповними та може призвести до неправильних висновків [12].

4. Обмеження в оцінці ресурсних характеристик.

Багато методів зосереджуються винятково на часу виконання запитів, ігноруючи при цьому інші ключові показники продуктивності, такі як використання пам’яті, рівень навантаження на процесор або частота звернень до дискового вводу-виводу. В сучасних багаторівневих системах ці параметри можуть істотно впливати на загальну ефективність, особливо в умовах високого навантаження або обмежених апаратних ресурсів. Ігнорування таких метрик зменшує практичну цінність оцінки ORM-фреймворків [10].

5. Неефективність при порівнянні в різних середовищах.

Порівняння результатів тестів, виконаних у різних операційних середовищах або на різних конфігураціях апаратного забезпечення, часто призводить до неспівставних результатів. Зміни в параметрах середовища виконання – кількість ядер процесора, доступна пам’ять, налаштування СКБД – можуть мати значний вплив на час виконання, що ускладнює об’єктивне порівняння ORM-фреймворків. Таким чином, відмінності у операційних середовищах можуть спотворити реальні показники продуктивності і ускладнити вибір оптимального інструмента [10].

6. Відсутність уніфікованих стандартів оцінки.

Однією з ключових проблем оцінювання ORM є відсутність загальноприйнятого набору стандартних сценаріїв та метрик, що дозволяли б виконувати порівняння в єдиному узгодженому форматі. Це призводить до того, що кожний дослідник або розробник застосовує власний набір тестів і параметрів, що унеможливлює безпосереднє зіставлення результатів. Неефективність стандартних підходів уніфікації тестових процедур є серйозною перешкодою для побудови репрезентативних порівнянь ORM-фреймворків [8].

7. Ігнорування особливостей предметної області.

Ще одним недоліком існуючих підходів є те, що вони майже не враховують специфіку предметної області застосування. Наприклад, деякі ORM можуть відмінно працювати в системах із низькою частотою записів, але значно гірше – у додатках із високою частотою оновлень і складними транзакціями. Оскільки загальні тести не моделюють конкретні сценарії бізнес-логіки, це призводить до того, що результати вимірювань не завжди корелюють із реальною продуктивністю у конкретному застосуванні [9].

Ці недоліки обґрунтовують необхідність розроблення більш комплексних підходів до оцінки ORM, які поєднували б репрезентативні сценарії навантаження, багатокритеріальні метрики та стандартизовані процедури вимірювання. Такий системний підхід створює основу для об’єктивнішого й практично значущого порівняння ORM-фреймворків у сучасних інформаційних системах.

1.7 Постановка задачі розроблення інформаційної системи порівняння ORM-фреймворків

Об’єктно-реляційне відображення як технологія значно спрощує розробку програмних систем, дозволяючи «перекладати» об’єктно-орієнтовані моделі у реляційні структури баз даних. Проте обрана стратегія доступу до даних і конкретний ORM-фреймворк можуть суттєво впливати на продуктивність застосунку, особливо у середовищах із високим навантаженням чи складною бізнес-логікою. Це породжує практичну потребу в інструментах, здатних об’єктивно порівнювати продуктивність різних ORM-рішень на основі кількісних показників, щоб надати розробникам та архітекторам інформаційних систем обґрунтовані рекомендації щодо вибору технології.

У таких умовах виникає потреба в розробленні інформаційної системи порівняння ORM-фреймворків, на основі показників продуктивності виконання

операцій. Система повинна надавати рекомендації по наступним характеристикам:

- Продуктивність;
- Функціональні можливості;
- Зручність використання та розширюваність;
- Надійність і безпека;
- Інтелектуальний аспект.

Дані характеристики забезпечать комплексний та гнучкий механізм порівняння та вибору необхідних ORM-фреймворків на основі їх репрезентативних метрик.

1.8 Висновок до першого розділу

У першому розділі магістерської роботи було проведено комплексний аналіз сучасних ORM-фреймворків та проблем, пов’язаних з оцінюванням їх продуктивності в контексті розроблення інформаційних систем. У ході дослідження розкрито теоретичні та практичні аспекти об’єктно-реляційного відображення, а також обґрунтовано його важливу роль у сучасних програмних рішеннях. Встановлено, що ORM-технології суттєво спрощують процес взаємодії між прикладною логікою та реляційними базами даних, однак водночас можуть створювати додаткові накладні витрати, що впливають на продуктивність.

Проведена класифікація та огляд популярних ORM-фреймворків різних екосистем (Java, .NET, Python та інших) показали значну різноманітність підходів до реалізації ORM, рівнів абстракції та архітектурних рішень. Виявлено, що відмінності між фреймворками зумовлюють різний вплив на швидкодію, споживання ресурсів і масштабованість інформаційних систем, що ускладнює їх пряме та універсальне порівняння.

У межах аналізу основних операцій ORM встановлено, що продуктивність значною мірою залежить від реалізації CRUD-операцій, механізмів керування транзакціями, роботи зі зв’язками між сутностями та використання кешування. Ці

фактори мають комплексний характер і можуть по-різному впливати на ефективність виконання операцій залежно від сценарію використання та умов навантаження.

Дослідження існуючих підходів до порівняння продуктивності ORM-фреймворків виявило, що більшість із них базуються на спрощених бенчмарках і обмеженому наборі метрик. Такі підходи не завжди відображають реальні умови експлуатації інформаційних систем і часто ігнорують архітектурні особливості ORM, вплив середовища виконання та специфіку предметної області. Визначено ключові недоліки наявних методів і засобів оцінювання, серед яких відсутність уніфікованих стандартів, низька репрезентативність тестових сценаріїв та обмежена практична цінність отриманих результатів.

На основі проведеного аналізу сформульовано постановку задачі розроблення інформаційної системи порівняння ORM-фреймворків, яка має забезпечити комплексний та гнучкий механізм вибору необхідних ORM-фреймворків на основі показників продуктивності виконання операцій.

2 МЕТОДИ ТА МОДЕЛІ ПОБУДОВИ ІНФОРМАЦІЙНОЇ СИСТЕМИ ПОРІВНЯННЯ ORM-ФРЕЙМВОРКІВ

2.1 Вибір показників продуктивності виконання операцій ORM-фреймворків

Вибір показників продуктивності виконання операцій ORM-фреймворків є одним із ключових етапів у процесі їх порівняльного аналізу. Оскільки ORM виступає проміжним рівнем між прикладною логікою та реляційною базою даних, саме набір обраних метрик визначає, наскільки коректно та об’єктивно буде оцінено його вплив на швидкодію інформаційної системи. Некоректно підібрані показники можуть спотворити результати дослідження та призвести до помилкових висновків щодо ефективності конкретного фреймворка.

Важливість правильного вибору показників продуктивності полягає в необхідності врахування не лише часу виконання окремих операцій, а й комплексного впливу ORM на використання ресурсів, масштабованість та поведінку системи в умовах реального навантаження. Саме тому обґрунтований набір метрик створює основу для достовірного аналізу, дозволяє порівнювати різні ORM-фреймворки в однакових умовах і забезпечує практичну цінність отриманих результатів для подальшого проєктування та оптимізації інформаційних систем.

В рамках досліджень проведених у роботі, для проектування інформаційної системи порівняння було обрано наступні характеристики:

- Продуктивність;
- Функціональні можливості;
- Зручність використання та розширюваність;
- Надійність і безпека;
- Інтелектуальний аспект.

Кожна з даної характеристики включає в себе ряд параметрів, по яким інформаційна система буде оцінювати ORM-фреймворк.

Продуктивність.

- *Час виконання CRUD-запитів.* CRUD-операції (Create, Read, Update, Delete) формують основу взаємодії прикладної логіки з базою даних у більшості інформаційних систем. Фактично цей параметр відображає сумарний вплив ORM-шару на швидкодію доступу до даних і дозволяє кількісно оцінити накладні витрати, що виникають унаслідок використання абстракції об'єктно-реляційного відображення.

Час виконання CRUD-запитів включає не лише безпосередній час виконання SQL-операцій у СКБД, але й додаткові витрати, пов'язані з роботою ORM-фреймворку. До них належать процеси формування SQL-запитів на основі об'єктної моделі, параметризація запитів, управління станом об'єктів, відстеження змін, а також перетворення результатів запиту з реляційного представлення у об'єктні структури прикладного коду. Саме сукупність цих етапів визначає реальну затримку між ініціюванням операції в коді та отриманням результату.

Оцінювання часу виконання CRUD-операцій дозволяє виявити відмінності між ORM-фреймворками у реалізації ключових механізмів доступу до даних. Наприклад, різні ORM можуть по-різному оптимізувати операції читання великих наборів даних, використовувати пакетну обробку під час вставки або оновлення записів, а також застосовувати різні стратегії кешування. Унаслідок цього навіть за однакової структури бази даних та однакових запитів фактичний час виконання операцій може суттєво відрізнятися, що безпосередньо впливає на загальну продуктивність системи [1-2].

Особливе значення цей параметр має для систем з інтенсивним обміном даними, таких як веб-застосунки, корпоративні інформаційні системи та сервіси з великою кількістю одночасних користувачів. У таких умовах навіть незначне збільшення часу виконання окремих CRUD-операцій може призвести до накопичення затримок, зниження пропускної здатності та погіршення

користувацького досвіду. Тому аналіз цього параметра є критично важливим для оцінювання здатності ORM-фреймворку ефективно працювати під навантаженням.

Крім того, час виконання CRUD-запитів є зручним для стандартизованого порівняння, оскільки CRUD-операції присутні практично в будь-якому прикладному сценарії. Це дозволяє використовувати даний параметр як універсальну основу для бенчмарків та експериментальних досліджень продуктивності. Водночас його аналіз у поєднанні з іншими показниками (кількість SQL-запитів, використання ресурсів, поведінка транзакцій) дає змогу глибше зрозуміти причини відмінностей у продуктивності між ORM-фреймворками [1-2].

- *Вплив Lazy Loading та Eager Loading на швидкість виконання операцій.*

Даний параметр безпосередньо визначає спосіб і момент завантаження пов’язаних даних з бази даних у прикладний код. Обрана стратегія завантаження суттєво впливає як на час виконання окремих запитів, так і на загальну ефективність роботи інформаційної системи.

Lazy Loading (ледаче завантаження) передбачає відкладене отримання пов’язаних сутностей – дані завантажуються лише тоді, коли до них фактично звертається прикладний код. З одного боку, це дозволяє зменшити початкове навантаження та скоротити час виконання простих операцій, коли пов’язані дані не потрібні. З іншого боку, надмірне використання Lazy Loading може призводити до виникнення проблеми N+1 запитів, коли один базовий запит породжує велику кількість додаткових звернень до бази даних. У результаті загальний час виконання операцій суттєво зростає, а продуктивність системи погіршується, особливо за умов великої кількості об’єктів або високого навантаження [13].

Eager Loading (жадібне завантаження), навпаки, передбачає отримання всіх необхідних пов’язаних даних заздалегідь, зазвичай у межах одного або обмеженої кількості SQL-запитів із використанням об’єднань (JOIN). Такий підхід зменшує кількість звернень до бази даних і дозволяє уникнути проблеми множинних запитів. Проте Eager Loading може негативно впливати на продуктивність у

випадках, коли завантажуються великі обсяги даних, що фактично не використовуються. Це призводить до збільшення часу виконання запиту, обсягу переданих даних та додаткового навантаження на пам’ять [14].

Важливість цього параметра при оцінюванні ORM-фреймворку полягає в тому, що різні ORM реалізують механізми Lazy та Eager Loading по-різному. Вони можуть відрізнятися за способом генерації SQL-запитів, ефективністю обробки зв’язків між сутностями, підтримкою комбінованих стратегій завантаження та можливостями тонкого налаштування. Унаслідок цього одна й та сама модель даних може демонструвати суттєво різні показники швидкодії залежно від обраного ORM та стратегії завантаження.

Аналіз впливу Lazy та Eager Loading на швидкість дозволяє оцінити не лише продуктивність ORM-фреймворку, а й його здатність ефективно працювати зі складними об’єктними графами. Для реальних інформаційних систем, у яких дані мають багаторівневі зв’язки, цей параметр є критичним, оскільки він визначає баланс між кількістю запитів, обсягом переданих даних і часом їх обробки.

- *Масштабність при великих об’ємах даних.* Масштабність ORM-фреймворку проявляється у тому, наскільки стабільно змінюється час виконання операцій зі збільшенням обсягу даних. Важливу роль тут відіграють механізми генерації SQL-запитів, оптимізація роботи з індексами, підтримка пакетних операцій та ефективність обробки результатів запитів. Якщо ORM не оптимізований для великих наборів даних, це може призводити до лінійного або навіть експоненціального зростання часу виконання операцій, що негативно впливає на загальну продуктивність системи [14].

Окремим аспектом масштабності є здатність ORM-фреймворку працювати з великими об’єктними графами без надмірного споживання пам’яті. Під час завантаження значних обсягів даних ORM виконує перетворення реляційних структур у об’єктні моделі, що супроводжується додатковими накладними витратами. Недостатньо ефективні механізми відстеження стану об’єктів або кешування можуть призводити до перевантаження оперативної пам’яті та зниження швидкодії, особливо у довготривалих сесіях роботи.

Важливість параметра масштабності також пов’язана з поведінкою ORM у багатокористувацьких середовищах. За великих обсягів даних та одночасного доступу багатьох користувачів зростає навантаження на систему керування базами даних і мережеву інфраструктуру. ORM-фреймворк має ефективно управляти підключеннями, транзакціями та конкурентним доступом до даних, не створюючи зайвих блокувань і не погіршуючи відгук системи. Відсутність таких механізмів суттєво обмежує масштабованість застосунку [15].

Оцінювання масштабності при великих об’ємах даних дозволяє виявити приховані обмеження ORM-фреймворків, які не проявляються під час тестування на малих вибірках. Саме цей параметр є показовим для довгострокових проєктів, де очікується зростання бази даних і підвищення вимог до продуктивності. Тому врахування масштабності є необхідною умовою об’єктивної оцінки ORM-фреймворків і обґрунтованого вибору технології для високонавантажених інформаційних систем.

Функціональні можливості.

- *Підтримка складних запитів (JOIN, агрегатні операції).* Даний параметр визначає здатність інструмента коректно та ефективно відображати складну реляційну логіку бази даних у прикладний рівень. У реальних інформаційних системах більшість бізнес-завдань виходить за межі простих CRUD-операцій і потребує виконання багатотабличних з’єднань, групування даних, обчислення статистичних показників та використання вкладених запитів.

Підтримка JOIN-операцій є критично важливою, оскільки реляційна модель даних ґрунтується на зв’язках між таблицями. ORM-фреймворк має надавати зручні та виразні засоби для опису таких зв’язків на рівні об’єктної моделі й автоматично транслювати їх у коректні SQL-запити. Обмежена або незручна підтримка JOIN змушує розробника використовувати сирий SQL, що знижує рівень абстракції, ускладнює супровід коду та нівелює переваги ORM як концепції. Тому здатність ORM прозоро працювати зі складними з’єднаннями є показником його функціональної зрілості [16].

Не менш важливим аспектом є підтримка агрегатних операцій, таких як підрахунок кількості записів, обчислення середніх значень, сум, мінімумів і максимумів, а також групування результатів. Такі операції широко використовуються у звітності, аналітичних модулях та системах підтримки прийняття рішень. ORM-фреймворк повинен дозволяти формувати агрегатні запити декларативним способом, не втрачаючи при цьому контролю над логікою обчислень і структурою результатів. Обмеження в цій сфері суттєво знижують придатність ORM для використання у складних прикладних сценаріях.

Важливість цього параметра для оцінювання функціональних можливостей ORM також полягає в тому, що різні фреймворки по-різному реалізують механізми побудови складних запитів. Вони можуть відрізнятися рівнем підтримки вкладених запитів, умовних виразів, підзапитів у SELECT або WHERE, а також можливістю оптимізації таких запитів. Чим ширший спектр підтримуваних конструкцій і чим ближче абстракції ORM до можливостей SQL, тим більш гнучким є фреймворк з точки зору розв’язання нетривіальних завдань.

Окрім цього, підтримка складних запитів тісно пов’язана з читабельністю та підтримуваністю коду. ORM, який дозволяє виразно описувати JOIN та агрегати на рівні мови програмування, сприяє кращому розумінню логіки застосунку та зменшує кількість помилок. Це особливо важливо для великих команд і довготривалих проєктів, де підтримка та розвиток системи є не менш значущими, ніж її початкова реалізація [16].

- *Міграції бази даних.* У сучасних програмних проєктах модель даних рідко залишається незмінною: з’являються нові сутності, змінюються зв’язки між ними, оптимізуються типи полів і додаються індекси. Наявність розвиненого механізму міграцій дозволяє керувати цими змінами системно та контрольовано [17].

Міграції бази даних забезпечують узгодженість між об’єктною моделлю застосунку та фізичною схемою реляційної бази даних. ORM-фреймворк із підтримкою міграцій дозволяє автоматизувати процес створення та модифікації таблиць, стовпців, обмежень і зв’язків на основі змін у коді. Це зменшує ризик

помилок, пов’язаних із ручним редагуванням SQL-скриптів, і підвищує надійність змін у структурі даних, що є важливою складовою функціональної повноти ORM.

Важливість цього параметра також проявляється у підтримці життєвого циклу розроблення. У командних проєктах та багатосередовищних конфігураціях (локальне середовище, тестування, продуктивна експлуатація) механізми міграцій дозволяють відтворювати однакову структуру бази даних у різних середовищах. ORM-фреймворки з розвиненою системою міграцій забезпечують зміни версій, можливість відкату до попереднього стану та послідовне застосування оновлень, що суттєво спрощує супровід і розгортання застосунків [17].

Крім того, підтримка міграцій впливає на гнучкість і масштабованість інформаційної системи. Здатність швидко адаптувати структуру бази даних до нових бізнес-вимог без значних простоїв є важливою вимогою до сучасних систем. ORM-фреймворк, який надає інструменти для безпечного виконання міграцій, дозволяє зменшити час оновлення системи та мінімізувати вплив змін на продуктивне середовище.

З точки зору функціональних можливостей ORM, механізми міграцій також демонструють рівень інтеграції фреймворку з інструментами розроблення та процесами автоматизації. Підтримка командного рядка, інтеграція з CI/CD-пайплайнами та можливість налаштування сценаріїв оновлення бази даних свідчать про зрілість ORM-рішення та його придатність для використання в реальних промислових проектах [17].

- *Кешування та асинхронні запити.* Кешування у межах ORM-фреймворку забезпечує повторне використання вже отриманих даних без необхідності кожного разу звертатися до бази даних. Це дозволяє суттєво зменшити кількість SQL-запитів, знизити навантаження на СУБД та скоротити час відповіді застосунку. Підтримка кешування на різних рівнях (кеш першого рівня в межах сесії, кеш другого рівня між сесіями, інтеграція з зовнішніми кеш-сховищами) розширює функціональні можливості ORM і надає розробнику інструменти для гнучкого керування життєвим циклом даних. Відсутність або обмежені

можливості кешування значно знижують ефективність ORM у сценаріях з частими повторними запитами.

Не менш важливою є підтримка асинхронних запитів, яка дозволяє виконувати операції доступу до даних без блокування основного потоку виконання. У сучасних веб- і сервіс-орієнтованих застосунках асинхронна модель взаємодії з базою даних є фактичним стандартом, оскільки вона підвищує пропускну здатність системи та забезпечує кращу масштабованість. ORM-фреймворк, що підтримує асинхронні операції на рівні API, дозволяє інтегрувати доступ до даних у загальну асинхронну архітектуру застосунку без використання додаткових низькорівневих механізмів [18].

З функціональної точки зору важливим є те, наскільки органічно кешування та асинхронні запити інтегровані в ORM-фреймворк. Зручні та узгоджені API, можливість конфігурації стратегій кешування, підтримка асинхронних операцій для всіх основних типів запитів і транзакцій істотно розширюють спектр завдань, які може вирішувати ORM. Це особливо актуально для систем із високими вимогами до швидкодії та стабільності роботи за великої кількості одночасних користувачів.

Крім того, наявність цих механізмів впливає на архітектурні рішення при проєктуванні застосунків. ORM із підтримкою кешування та асинхронного доступу до даних дозволяє будувати більш гнучкі та адаптивні системи, у яких продуктивність досягається не лише за рахунок оптимізації SQL-запитів, а й завдяки ефективному використанню ресурсів.

Зручність використання та розширюваність.

- *Крива навчання та документація.* Крива навчання відображає складність концепцій, які необхідно засвоїти розробнику для ефективної роботи з ORM-фреймворком. До таких концепцій належать об’єктно-реляційне відображення, управління життєвим циклом сутностей, робота з контекстами або сесіями, конфігурація зв’язків і стратегій завантаження даних. ORM із пологою кривою навчання дозволяє швидше інтегрувати нових розробників у проєкт, зменшити кількість помилок на початкових етапах і підвищити загальну продуктивність

команди. Це особливо важливо для великих або довготривалих проєктів, де склад команди може змінюватися з часом.

Документація є основним засобом передачі знань про можливості та особливості ORM-фреймворку. Повна, структурована та актуальна документація значно полегшує використання інструмента, дозволяє швидко знаходити відповіді на практичні питання та сприяє коректному застосуванню складних функцій. З точки зору розширюваності, якісна документація з описом внутрішніх механізмів, точок розширення та прикладів кастомізації є необхідною умовою для адаптації ORM під специфічні вимоги проекту [19].

Важливість цього параметра також проявляється у здатності ORM-фреймворку підтримувати масштабування та еволюцію системи. Чим краще задокументовані механізми налаштування, оптимізації та розширення, тим простіше розробникам реалізовувати нестандартні сценарії, інтегрувати сторонні бібліотеки або модифікувати поведінку ORM без порушення цілісності архітектури. Відсутність чітких рекомендацій ускладнює такі процеси й підвищує ризик створення технічного боргу.

Крім того, документація та навчальні матеріали формують спільноту користувачів навколо ORM-фреймворку. Активна спільнота, приклади використання, рекомендації з найкращих практик і типові шаблони рішень суттєво знижують поріг входу та сприяють швидшому вирішенню проблем. Це підвищує привабливість ORM-фреймворку для нових проєктів і полегшує його довготривале використання.

- *Можливість кастомізації та інтеграції з .NET.* Можливість кастомізації визначає, наскільки гнучко ORM-фреймворк дозволяє змінювати або розширювати свою стандартну поведінку. До таких аспектів належать налаштування мапінгу сутностей, управління життєвим циклом об’єктів, перехоплення запитів, додавання власної логіки обробки даних, а також використання спеціалізованих типів і конвертерів. ORM, який надає зрозумілі та стабільні механізми розширення, значно підвищує зручність використання,

оскільки розробник може адаптувати його до конкретних потреб без обходу стандартного API або використання неофіційних рішень [20].

Інтеграція з .NET-платформою є не менш важливою складовою цього параметра. Зручний ORM повинен органічно поєднуватися з ключовими компонентами .NET, зокрема з механізмами Dependency Injection, конфігурацією застосунку, логуванням, асинхронним програмуванням та інструментами тестування. Чим глибша й прозоріша така інтеграція, тим менше додаткових зусиль потребує включення ORM у загальну архітектуру системи. Це безпосередньо впливає на зручність використання, оскільки знижує кількість допоміжного коду та спрощує підтримку застосунку [20].

З точки зору розширюваності, цей параметр дозволяє оцінити, наскільки ORM придатний для довготривалих і еволюційних проєктів. У процесі розвитку системи часто виникає потреба у впровадженні нових підходів до зберігання даних, інтеграції з зовнішніми сервісами або оптимізації продуктивності. ORM-фреймворк із розвиненими можливостями кастомізації та тісною інтеграцією з .NET дозволяє реалізовувати такі зміни поступово, без радикальної перебудови архітектури або відмови від обраного інструмента.

Крім того, цей параметр має важливе значення для командної розробки. Чітко визначені точки розширення та стандартні механізми інтеграції сприяють уніфікації підходів до роботи з даними в межах проєкту. Це підвищує зрозумілість коду, полегшує його супровід і зменшує залежність від окремих розробників, що є важливим аспектом зручності використання у великих командах.

Надійність і безпека.

- *Підтримка транзакцій і rollback.* Підтримка транзакцій в ORM-фреймворку визначає, наскільки ефективно він здатний координувати виконання кількох SQL-операцій у межах однієї логічної операції прикладного рівня. ORM виступає посередником між кодом застосунку та СУБД, тому від якості реалізації транзакцій залежить коректність взаємодії з механізмами керування транзакціями бази даних. Наявність прозорої підтримки транзакцій дозволяє розробнику

реалізовувати складну бізнес-логіку без необхідності ручного керування кожним етапом виконання запитів [15].

Механізми rollback відіграють особливо важливу роль у забезпеченні надійності системи. У разі виникнення помилки, виняткової ситуації або порушення бізнес-правил rollback дозволяє повернути базу даних до попереднього узгодженого стану. ORM-фреймворк, який коректно реалізує rollback, зменшує ризик частково виконаних операцій і втрати даних. Це є критичним для систем, що обробляють фінансові операції, персональні дані або іншу чутливу інформацію.

З точки зору безпеки, підтримка транзакцій і rollback також має непрямий, але важливий вплив. Забезпечення цілісності даних є однією з базових вимог інформаційної безпеки, оскільки пошкоджені або неконсистентні дані можуть призвести до помилкових рішень або зловживань. ORM-фреймворки з розвиненими транзакційними механізмами дозволяють реалізувати контроль доступу та перевірку прав користувачів у межах транзакцій, зменшуючи ризик несанкціонованих змін у базі даних [15].

Важливість цього параметра також проявляється у здатності ORM працювати в багатокористувацькому середовищі. За одночасного доступу до даних кількох користувачів правильне управління транзакціями допомагає уникати конфліктів, втрати оновлень і некоректних результатів виконання запитів. ORM-фреймворк має коректно підтримувати різні рівні ізоляції транзакцій і інтегрувати їх із механізмами СУБД, забезпечуючи передбачувану поведінку системи.

- *Захист від SQL Injection.* ORM-фреймворки покликані абстрагувати розробника від ручного формування SQL-запитів, і саме ця абстракція є ключовим чинником підвищення безпеки. Надійний ORM автоматично використовує параметризовані запити або підготовлені вирази, у яких дані користувача передаються окремо від структури SQL-запиту. Такий підхід унеможливлює інтерпретацію введених значень як частини SQL-коду, що істотно

знижує ризик SQL Injection. Тому наявність і коректність реалізації цього механізму є критерієм безпеки ORM [21].

Важливість цього параметра полягає також у зменшенні людського фактора. Навіть досвідчені розробники можуть припускатися помилок під час ручної побудови SQL-запитів, особливо у складних або динамічних сценаріях. ORM-фреймворк, який за замовчуванням забезпечує захист від SQL Injection, дозволяє зменшити залежність безпеки системи від індивідуальних навичок розробника та уніфікувати підхід до обробки даних у межах усього застосунку.

З точки зору надійності, захист від SQL Injection сприяє збереженню цілісності та доступності даних. Атаки цього типу можуть не лише призводити до витоку інформації, а й порушувати нормальне функціонування системи, викликаючи помилки, некоректні транзакції або повне знищення даних. ORM-фреймворк, який забезпечує безпечне формування запитів, зменшує ймовірність таких інцидентів і підвищує загальну стійкість системи до зовнішніх загроз [21].

Крім того, цей параметр є важливим у контексті відповідності стандартам і рекомендаціям з інформаційної безпеки. Багато галузевих стандартів і практик прямо вказують на необхідність використання параметризованих запитів і уникнення динамічного SQL. ORM-фреймворки, що реалізують ці підходи на рівні архітектури, спрощують дотримання вимог безпеки та зменшують витрати на аудит і перевірку коду.

Інтелектуальний аспект.

- *Виявлення закономірностей у продуктивності.* Інтелектуальний аспект оцінювання продуктивності полягає в інтерпретації результатів експериментів та пошуку повторюваних моделей поведінки ORM-фреймворку. Наприклад, аналіз закономірностей може показати, як змінюється продуктивність при зростанні кількості сутностей, ускладненні зв’язків між таблицями або зміні стратегій завантаження даних. Такі закономірності дозволяють не лише порівнювати ORM між собою, а й прогнозувати їхню поведінку в майбутніх або масштабованих сценаріях використання [22].

Важливість цього параметра полягає також у можливості виявлення нелінійних ефектів, які неочевидні при поверхневому аналізі. ORM-фреймворк може демонструвати стабільну роботу за малих обсягів даних, але різке погіршення продуктивності після досягнення певного порогового значення. Виявлення подібних закономірностей є критично важливим для прийняття архітектурних рішень і запобігання проблемам продуктивності на пізніх етапах життєвого циклу системи.

З точки зору інтелектуального аналізу, виявлення закономірностей у продуктивності створює основу для застосування методів аналітики та елементів data-driven підходу. Узагальнення результатів тестування дозволяє формувати правила та рекомендації щодо оптимального використання ORM, вибору конфігурацій, стратегій кешування або типів запитів. Таким чином, ORM оцінюється не лише як інструмент виконання запитів, а як складна система з передбачуваною поведінкою [22].

Крім того, цей параметр підвищує практичну цінність результатів оцінювання. Розуміння закономірностей у продуктивності дозволяє розробникам і архітекторам обґрунтовано вибирати ORM-фреймворк залежно від специфіки проєкту, а не покладатися лише на узагальнені рейтинги або окремі бенчмарки. Це особливо важливо для складних інформаційних систем, де продуктивність визначається сукупністю взаємопов’язаних факторів.

- *Рекомендації для вибору ORM для різних сценаріїв.* Інтелектуальний аспект цього параметру полягає в здатності інформаційної системи порівняння враховувати різноманітність сценаріїв застосування ORM-фреймворків. Різні проєкти мають відмінні вимоги до продуктивності, масштабованості, складності доменної моделі та частоти доступу до даних. Наприклад, для високонавантажених сервісів із великою кількістю простих запитів доцільними можуть бути легковагові ORM або мікро-ORM, тоді як для корпоративних систем зі складною бізнес-логікою – повнофункціональні ORM із розвиненими механізмами роботи з транзакціями та зв’язками між сутностями. Формування

таких рекомендацій потребує не лише знання характеристик ORM, а й розуміння закономірностей їхньої поведінки в різних умовах.

Важливість цього параметра також полягає у зменшенні ризиків помилкового вибору технології. Неправильно підібраний ORM може призвести до прихованих проблем продуктивності, ускладнення супроводу або зростання витрат на оптимізацію в майбутньому. Рекомендації, сформовані на основі систематичного аналізу та експериментальних даних, дозволяють заздалегідь зіставити можливості ORM із вимогами конкретного сценарію та уникнути таких проблем.

З позиції інтелектуального аналізу, параметр рекомендацій передбачає синтез результатів за кількома групами критеріїв: продуктивність, функціональні можливості, зручність використання, надійність і безпека. Саме інтеграція цих характеристик у вигляді узагальнених висновків і сценарних порад свідчить про високий рівень інтелектуальності системи оцінювання. Таким чином, ORM-фреймворк оцінюється не ізольовано, а в контексті реальних потреб розробників та архітекторів [22].

Крім того, наявність рекомендацій для різних сценаріїв сприяє підвищенню відтворюваності та об’єктивності вибору ORM. Замість суб’єктивних уподобань або неформальних порад, розробники отримують обґрунтовані висновки, засновані на аналізі даних і встановлених закономірностях. Це особливо важливо в умовах швидкого розвитку технологій, коли кількість ORM-фреймворків і варіантів їх конфігурації постійно зростає.

Узагальнюючи розглянуті характеристики та їх параметри, можна зазначити, що комплексна оцінка ORM-фреймворків повинна базуватися на поєднанні кількох взаємодоповнювальних аспектів (рис. 2.1). У сукупності ці характеристики та параметри створюють цілісну основу для об’єктивного й обґрунтованого порівняння ORM-фреймворків.

Рис. 2.1 Показники продуктивності виконання операцій ORM-фреймворків.

2.2 Вибір програмних засобів і технологій реалізації системи

Для реалізації інформаційної системи порівняння ORM-фреймворків важливим є обґрунтований вибір програмних засобів і технологій, що забезпечують ефективність розробки, масштабованість і зручність користування системою. Основні програмні компоненти архітектури інформаційної системи представлені на рис. 2.2.

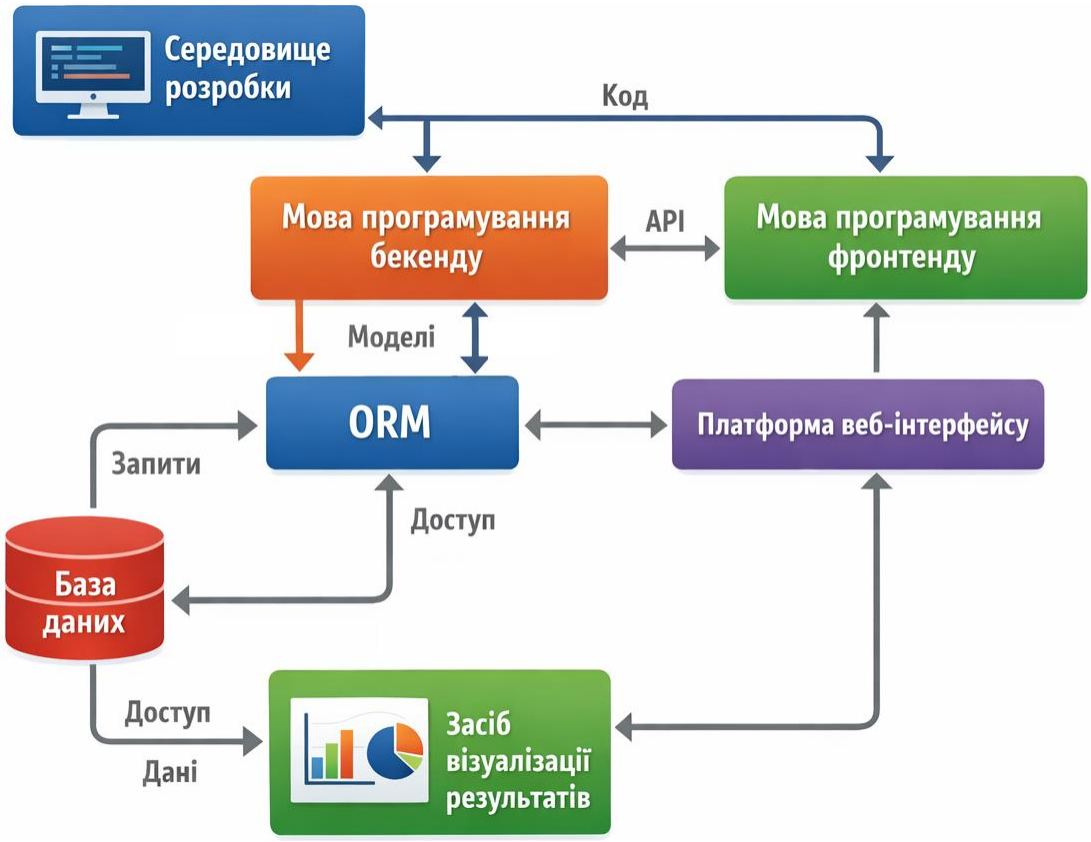


Рис. 2.2 Компоненти програмної архітектури інформаційної системи.

2.2.1 Середовище розробки

Сучасна розробка програмного забезпечення (ПЗ) є складним багатоетапним процесом, що поєднує проєктування архітектури, написання та тестування коду, налагодження, документування й подальший супровід програмних продуктів. У цих умовах ключову роль відіграє середовище розробки програмного забезпечення (IDE, Integrated Development Environment), яке виступає основним інструментом взаємодії розробника з кодом та інфраструктурою проєкту.

Середовище розробки безпосередньо впливає на продуктивність праці програміста, якість коду, швидкість виявлення та усунення помилок, а також на загальну ефективність життєвого циклу програмного забезпечення. Наявність у IDE таких функцій, як інтелектуальне автодоповнення, статичний аналіз коду, вбудовані засоби налагодження, тестування та інтеграції з системами контролю

версій, дозволяє суттєво зменшити кількість рутинних операцій і зосередитися на вирішенні прикладних та архітектурних задач (рис. 2.3).

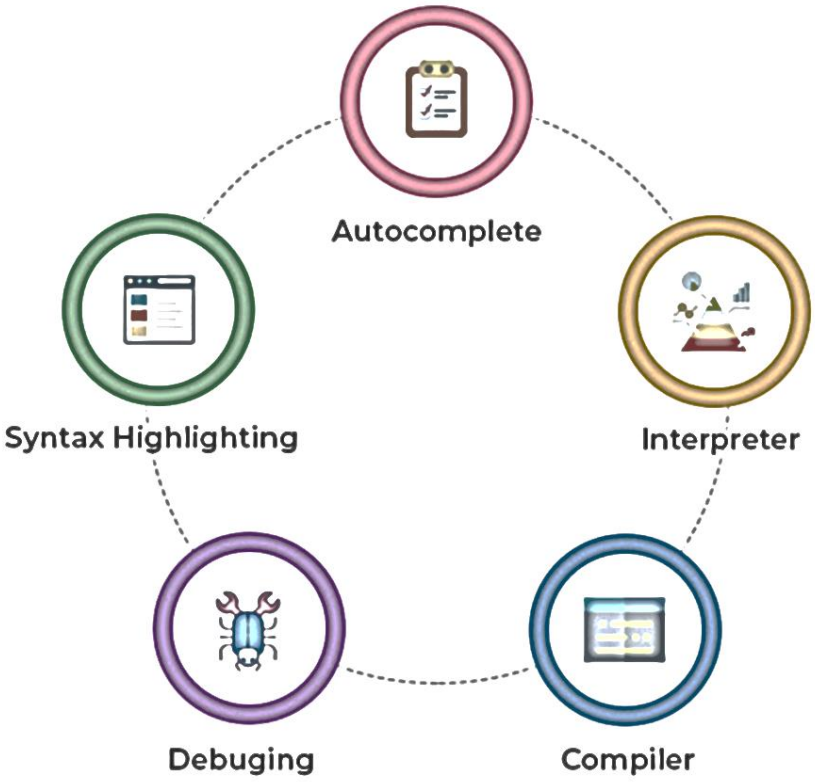


Рис. 2.3 Основні характеристики Integrated Development Environment

Особливої актуальності вибір середовища розробки набуває при створенні великих, масштабованих і довготривалих програмних систем, де помилки проєктування або неефективні інструменти можуть призвести до значних витрат часу й ресурсів. У таких проєктах IDE виконує не лише роль редактора коду, а й стає комплексною платформою для управління залежностями, командної розробки, автоматизації збірки та розгортання програмного забезпечення.

Крім того, різні середовища розробки орієнтовані на різні технологічні стеки, мови програмування та платформи, що зумовлює необхідність усвідомленого вибору інструменту залежно від специфіки завдання. Наприклад, корпоративні .NET-рішення, мобільні додатки, веб-сервіси чи наукові обчислення мають різні вимоги до функціональності IDE, її продуктивності та розширюваності.

На сьогоднішній день існує велика кількість різноманітних інтегрованих середовищ розробки, які відрізняються за функціональністю, підтримуваними мовами програмування, платформами, рівнем розширюваності та вимогами до апаратних ресурсів. Крім того, різні середовища розробки орієнтовані на різні технологічні стеки, мови програмування та цільові платформи (табл. 2.1).

Таке різноманіття зумовлене стрімким розвитком інформаційних технологій, появою нових парадигм програмування та зростанням потреб у спеціалізованих інструментах для веб-, мобільної, корпоративної та наукової розробки. У результаті розробники мають змогу обирати IDE, що найкраще відповідає конкретним вимогам проєкту та особистим уподобанням.

Таблиця 2.1

Порівняння популярних IDE

Середовище	**Платформи**	**Мови**	**Потужність**	**Розширюваність**	**Кросплатформність**	**Ідеальне використання**
Visual Studio	Windows, частково Mac	C#, C++, VB.NET, F#	Дуже висока	Висока	Частково	Корпоративні .NET проєкти, Azure, Windows
Rider / IntelliJ	Windows, Mac, Linux	C#, Java, Kotlin	Висока	Висока	Так	Мультиплатформові проєкти, професійна розробка
Eclipse	Windows, Mac, Linux	Java, C/C++, Python	Середня	Висока	Так	Java/C++ проєкти, універсальна IDE
VS Code	Windows, Mac, Linux	Будь-яка через плагіни	Низька-середня	Дуже висока	Так	Легка розробка, веб, скрипти
Xcode	Mac	Swift, Objective-C	Середня	Обмежена	Ні	Розробка iOS/macOS
NetBeans	Windows, Mac, Linux	Java, JavaScript, PHP, C/C++	Середня	Середня	Так	Java і веб-проєкти, GUI-дизайн
PyCharm	Windows, Mac, Linux	Python, Django, Flask	Висока	Висока	Так	Python, наукові проєкти, веб
Android Studio	Windows, Mac, Linux	Java, Kotlin, C++	Висока	Середня	Так	Android-додатки, мобільна розробка

З огляду на різноманіття сучасних інтегрованих середовищ розробки, вибір інструменту для створення програмного забезпечення повинен ґрунтуватися на комплексному аналізі функціональних можливостей, продуктивності, зручності використання та відповідності вимогам конкретного проєкту.

У даній роботі в якості середовища розробки обрано Visual Studio, що зумовлено низкою технічних, організаційних і практичних переваг цього інструмента.

Visual Studio є одним із найбільш функціонально насичених середовищ розробки, яке забезпечує повний цикл створення програмного забезпечення – від проєктування та написання коду до тестування, налагодження й розгортання готового продукту. Завдяки глибокій інтеграції з платформою .NET, середовище надає потужні засоби для розробки настільних, веб- та серверних застосунків, що є особливо важливим для корпоративних і масштабованих інформаційних систем [23] (рис. 2.4).

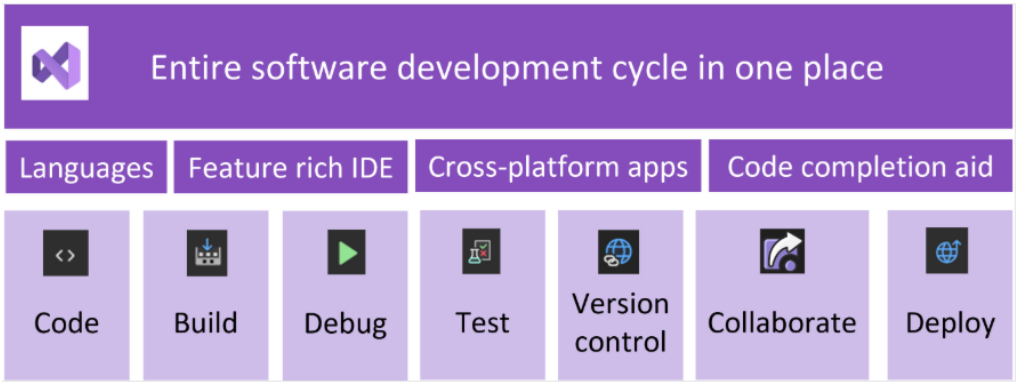


Рис. 2.4 Основні характеристики Visual Studio

Важливою перевагою Visual Studio є високий рівень інтелектуальної підтримки розробника. Вбудовані механізми автодоповнення коду (IntelliSense), статичного аналізу, автоматичного рефакторингу та перевірки типів дозволяють суттєво зменшити кількість синтаксичних і логічних помилок на ранніх етапах розробки. Це сприяє підвищенню якості програмного коду та скороченню часу, необхідного для його налагодження.

Окрему увагу заслуговують засоби налагодження та тестування, які реалізовані у Visual Studio. Потужний вбудований дебагер підтримує покрокове виконання програм, аналіз стеку викликів, перегляд значень змінних у режимі реального часу та роботу з багатопотоковими застосунками. Крім того, середовище містить інструменти для модульного тестування, що дозволяє автоматизувати перевірку коректності роботи програмних компонентів і забезпечити стабільність програмного забезпечення в процесі його розвитку.

Суттєвою перевагою Visual Studio є також інтеграція з сучасними інструментами командної розробки. Підтримка систем контролю версій (зокрема Git), засобів керування задачами та хмарних сервісів Microsoft Azure забезпечує ефективну взаємодію між учасниками проєкту та спрощує процеси спільної розробки й розгортання програмних рішень [23].

Крім функціональних можливостей, Visual Studio вирізняється стабільністю, масштабованістю та широкою підтримкою спільноти. Регулярні оновлення, детальна офіційна документація та велика кількість навчальних матеріалів роблять це середовище зручним як для початківців, так і для досвідчених розробників. Наявність безкоштовної версії Visual Studio Community також є важливим фактором, що забезпечує доступність середовища для освітніх і некомерційних проєктів.

Також вибір Visual Studio обумовлений популярністю цієї IDE. За даними щорічного опитування розробників Stack Overflow Developer Survey 2025 популярність використання Visual Studio серед середовищ розробки та редакторів кодів знаходиться на другому місці серед усіх Dev IDEs (рис. 2.5). А якщо говорити про повноцінні інтегровані середовища – то перше місце [24].

Visual Studio – це повноцінне інтегроване середовище розробки, яке з самого початку містить широкий набір вбудованих інструментів: компілятори, дизайнер інтерфейсу, засоби тестування, профілювання та налагодження. Воно орієнтоване насамперед на великі проєкти, корпоративну розробку та екосистему .NET.

Visual Studio Code – це легкий редактор коду з можливістю розширення, побудований за модульним принципом. Базова версія містить мінімальний набір функцій, а більшість IDE-можливостей додаються через розширення. VS Code створений як універсальний інструмент для швидкої та гнучкої роботи з кодом.

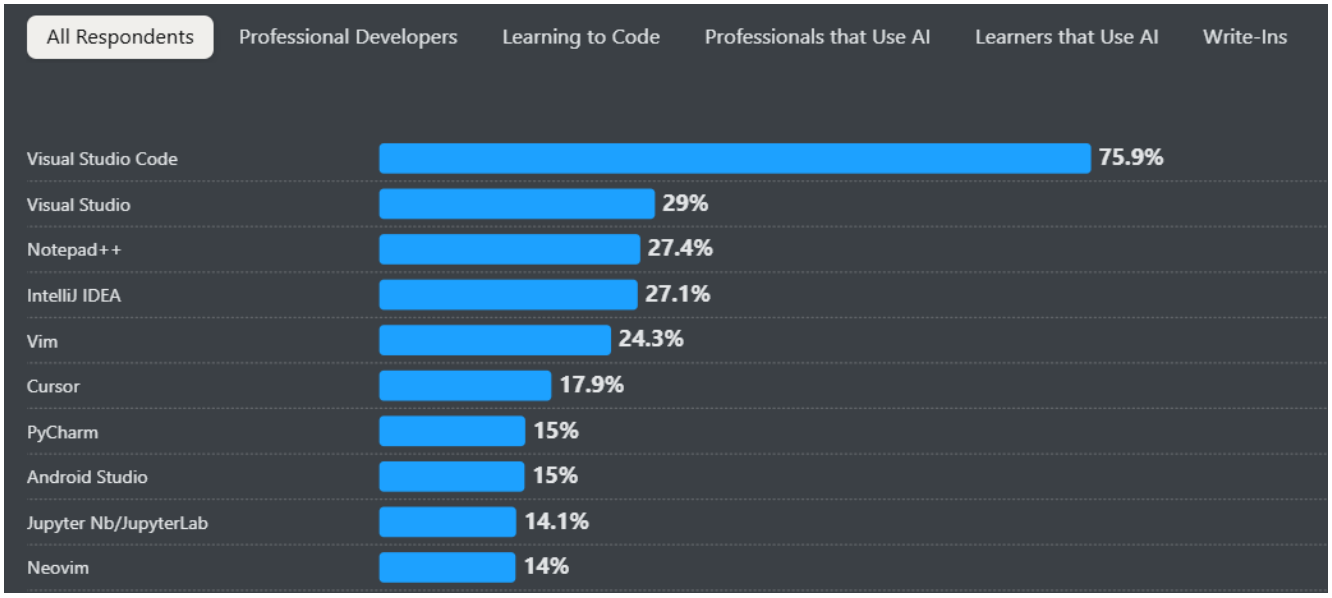


Рис. 2.5 Популярність використання Dev IDEs у 2025 році (перша десятка) за даними Stack Overflow Developer Survey

2.2.2 Мови реалізації проекту

Наступним важливим рішенням після вибору середовища розробки – є вибір мови програмування, оскільки саме вона визначає архітектурні підходи, можливості масштабування, рівень безпеки, швидкість розробки та зручність подальшого супроводу програмного забезпечення.

На сучасному етапі розвитку інформаційних технологій існує велика кількість мов програмування, кожна з яких орієнтована на вирішення певного класу задач. Вибір мови програмування залежить від багатьох факторів, зокрема типу програмного забезпечення, цільової платформи, вимог до продуктивності, безпеки та доступності бібліотек і фреймворків. Невдалий вибір мови може призвести до ускладнення реалізації проєкту, зростання витрат на розробку та зниження ефективності програмного продукту в цілому.

Особливо важливим питання вибору мови програмування є при створенні клієнт-серверних і веб-орієнтованих систем, де програмне забезпечення зазвичай поділяється на дві логічні частини – фронтенд і бекенд. Кожна з цих частин має власні функціональні завдання, вимоги та технологічні особливості, що зумовлює використання різних мов програмування.

Мови реалізації бекенду.

Бекенд-частина ПЗ є ключовим компонентом сучасних інформаційних систем, оскільки відповідає за реалізацію бізнес-логіки, обробку запитів клієнтів, збереження та обробку даних, забезпечення безпеки і взаємодію з зовнішніми сервісами.

Враховуючи, що обране середовище розробки Visual Studio орієнтоване насамперед на пдатформу .NET, бекенд-розробка в ній характеризується високим рівнем стандартизації, масштабованості та підтримкою сучасних архітектурних підходів.

Основною мовою програмування для бекенду в екосистемі .NET є C#, яка поєднує об’єктно-орієнтовану, компонентну та функціональну парадигми програмування. C# відзначається строгою типізацією, високою читабельністю коду та розвиненими механізмами управління пам’яттю, що робить її придатною для створення складних серверних систем із високими вимогами до надійності та продуктивності.

Також C# є однією з ключових технологій для створення серверної частини програмного забезпечення на платформі .NET. У бекенд C# відповідає за:

- обробку HTTP-запитів і відповідей;
- реалізацію бізнес-логіки;
- взаємодію з базами даних;
- автентифікацію та авторизацію користувачів;
- інтеграцію із зовнішніми сервісами;
- забезпечення масштабованості та безпеки.

Особливістю роботи C# є те, ця мова програмування працює через проміжний код і кероване середовище виконання, що забезпечує баланс між

продуктивністю, безпекою та кросплатформеністю у межах платформи .NET (рис. 2.6).

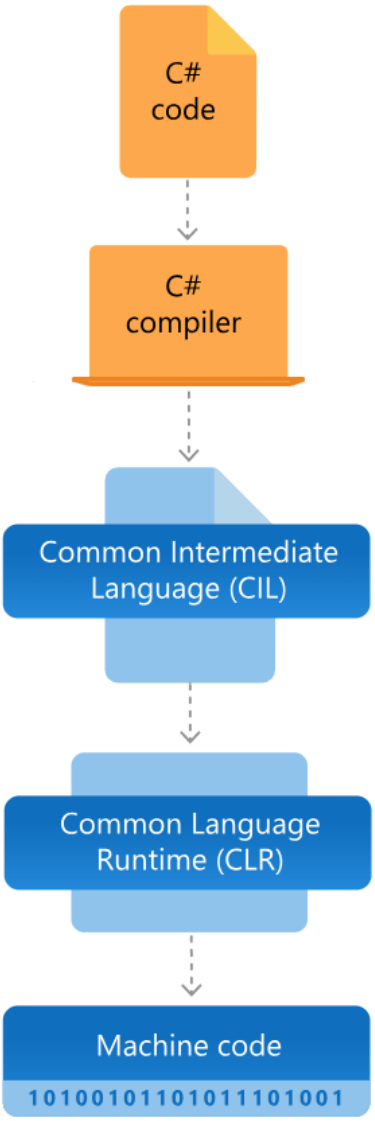


Рис. 2.6 Реалізація C# в середовищі .NET

C# є потужною, гнучкою та сучасною мовою програмування, яка поєднує продуктивність і безпеку з високим рівнем абстракції. Її структура, багаті мовні можливості та тісна інтеграція з платформою .NET роблять C# ефективним інструментом для створення масштабованого та надійного програмного забезпечення.

Мови реалізації фронтенду.

Фронтенд є клієнтською частиною програмного забезпечення, яка відповідає за взаємодію з користувачем, відображення інтерфейсу та обробку

користувацьких дій. Основними вимогами до фронтенд-мов є висока швидкодія в браузері, зручність створення інтерактивних інтерфейсів і сумісність із різними пристроями та операційними системами.

Фронтенд-розробка для пдатформи .NET має низку особливостей, пов'язаних із використанням як традиційних веб-технологій, так і спеціалізованих інструментів і мов програмування, інтегрованих у екосистему Microsoft.

Базовими технологіями фронтенду, незалежно від платформи, залишаються HTML, CSS, які забезпечують структуру та стилізацію користувацького інтерфейсу. Однак ключову роль у створенні інтерактивної логіки фронтенду відіграють мови програмування, що дозволяють реалізувати динамічну поведінку інтерфейсу, обробку подій та взаємодію з серверною частиною застосунку. В цьому аспекті лідирує мова JavaScript.

Разом ці технології утворюють фундамент клієнтської частини застосунків і забезпечують взаємодію між користувачем та програмною логікою (рис. 2.7). Їх широке застосування обумовлене універсальністю, кросплатформеністю та тісною інтеграцією з браузерними й серверними середовищами.

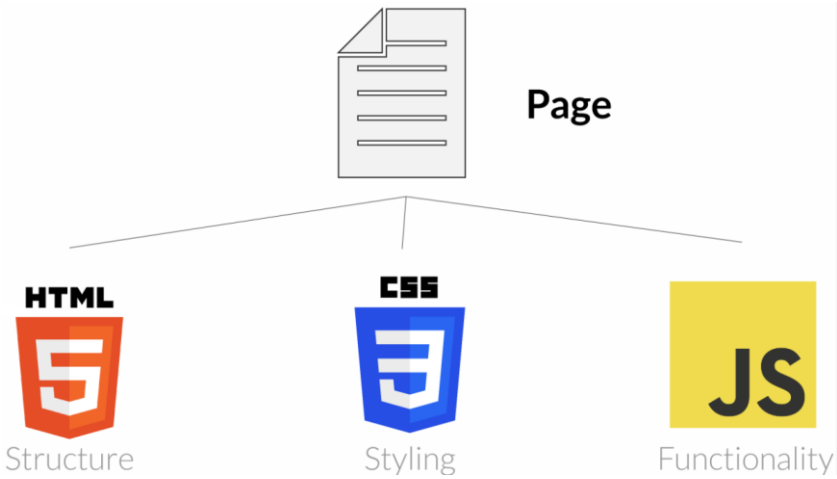


Рис. 2.7 Фронтенд структура веб-сторінки

HTML – мова гіпертекстової розмітки (HyperText Markup Language) відповідає за логічну структуру та семантичне представлення вмісту застосунку. Він визначає елементи інтерфейсу – заголовки, форми, кнопки, таблиці,

мультимедійні об’єкти – та встановлює їх ієрархію. Семантична розмітка HTML підвищує доступність застосунків, полегшує індексацію пошуковими системами та створює основу для подальшої стилізації і програмної обробки.

З технічної точки зору HTML виступає каркасом інтерфейсу, на який накладаються стилі та сценарії. Без чітко організованої HTML-структури неможливо забезпечити масштабованість і підтримуваність клієнтської частини застосунку.

CSS – каскадні таблиці стилів (Cascading Style Sheets) використовується для візуального оформлення та адаптації інтерфейсу. За допомогою CSS реалізуються кольори, шрифти, відступи, анімації та адаптивні макети, що дозволяють коректно відображати застосунок на різних пристроях і розмірах екранів.

Технічна важливість CSS полягає у відокремленні логіки від представлення, що відповідає принципам модульності та повторного використання коду. Сучасні можливості CSS, такі як Flexbox, Grid та медіа-запити, дозволяють створювати складні інтерфейси без використання додаткових програмних засобів, що позитивно впливає на продуктивність і швидкість завантаження застосунків.

JavaScript є основною мовою програмування клієнтської частини застосунків і відповідає за реалізацію інтерактивної поведінки. За допомогою JavaScript обробляються події користувача, виконується динамічна зміна вмісту сторінки, здійснюється обмін даними із сервером та реалізується клієнтська логіка.

З технічної точки зору JavaScript дозволяє створювати реактивні інтерфейси, оптимізувати роботу з даними та зменшувати навантаження на серверну частину. Крім того, JavaScript широко використовується не лише у браузері, але й на сервері, що сприяє уніфікації технологічного стеку та повторному використанню коду.

Популярність зазначених технологій обумовлена кількома ключовими факторами:

- кросплатформеність, оскільки вони підтримуються всіма сучасними браузерами;

- низький поріг входу та доступність навчальних ресурсів;
- висока гнучкість і масштабованість;
- розвинена екосистема фреймворків і бібліотек;
- підтримка сучасних стандартів безпеки та продуктивності.

У сукупності HTML, CSS і JavaScript формують універсальний інструментарій, який дозволяє створювати як прості веб-сторінки, так і складні клієнтські застосунки з багатою функціональністю.

У сучасних програмних системах клієнтська частина, реалізована за допомогою HTML, CSS та JavaScript, тісно взаємодіє з бекендом через API. Такий підхід дозволяє чітко розділити відповідальність між інтерфейсом користувача та серверною логікою, підвищуючи безпеку, продуктивність і масштабованість застосунків.

Таким чином, вибір C# для бекенд-розробки є доцільним завдяки високій продуктивності, надійності, строгій типізації та тісній інтеграції з платформою .NET, що забезпечує безпеку, масштабованість і зручність підтримки серверних застосунків. Використання HTML, CSS і JavaScript для фронтенду обґрунтоване їх універсальністю, кросплатформенністю та здатністю створювати структуровані, адаптивні й інтерактивні користувацькі інтерфейси. Поєднання цих мов дозволяє чітко розділити клієнтську і серверну логіку, що сприяє розробці сучасних, ефективних і масштабованих програмних систем.

За даними щорічного опитування розробників Stack Overflow Developer Survey 2025 популярність використання C#, HTML/CSS, JavaScript серед мов програмування, сценаріїв та розмітки знаходиться в першій десятці (рис. 2.8) [24].

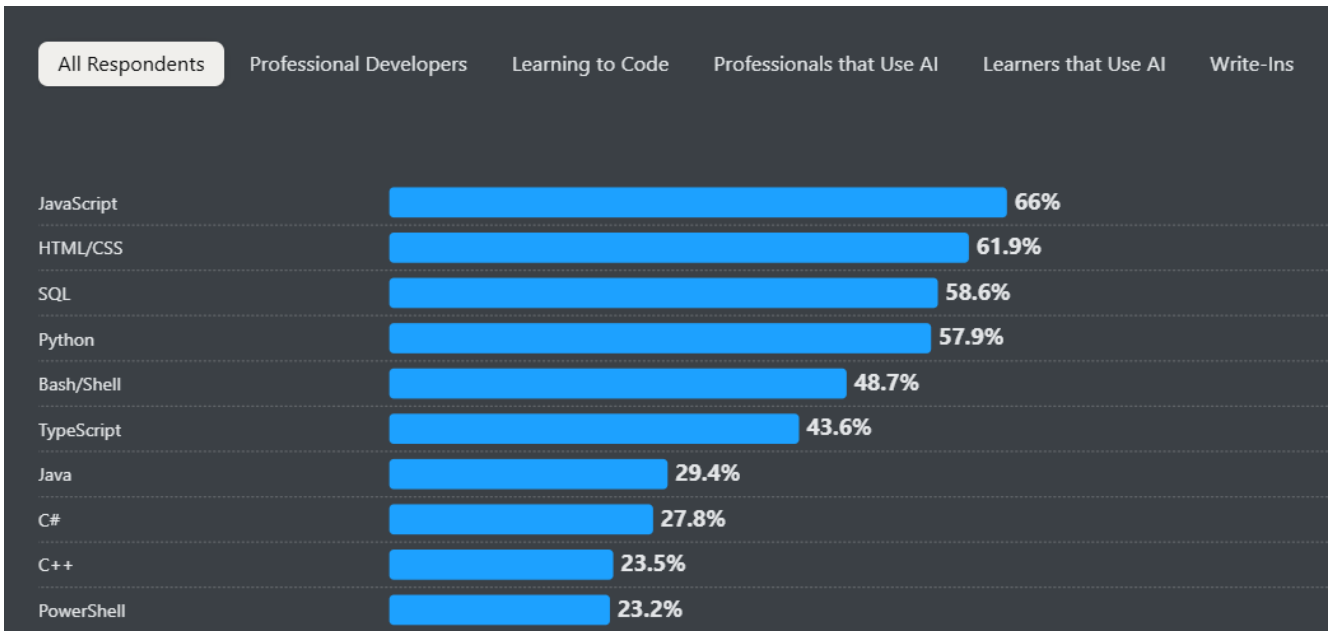


Рис. 2.8 Популярність використання C#, HTML/CSS, JavaScript у 2025 році серед мов програмування, сценаріїв та розмітки (перша десятка) за даними Stack Overflow Developer Survey

2.2.3. Платформа побудови веб-інтерфейсу

Інтерактивний веб-інтерфейс є ключовим елементом сучасних веб-застосунків і інформаційних систем, оскільки вони забезпечують ефективну взаємодію між користувачем і програмним забезпеченням у режимі реального часу. На відміну від статичних веб-сторінок, інтерактивні веб-інтерфейси дозволяють динамічно реагувати на дії користувача без необхідності повного перезавантаження сторінки, що суттєво підвищує зручність використання та продуктивність роботи.

Основне призначення інтерактивних веб-інтерфейсів полягає у забезпеченні швидкого та інтуїтивного доступу до функціональних можливостей застосунку. Вони використовуються для:

- введення та обробки користувацьких даних;
- навігації між функціональними модулями системи;
- відображення змін даних у реальному часі;
- взаємодії з серверною логікою через API або постійні з’єднання;

- зменшення затримок у взаємодії з користувачем.

Інтерактивний інтерфейс виступає основним засобом комунікації між користувачем і прикладною логікою системи.

Інтерактивні веб-інтерфейси реалізують низку важливих функцій, зокрема:

- обробка подій (кліки, введення даних, жести);
- динамічне оновлення вмісту без перезавантаження сторінки;
- клієнтська валідація даних для зменшення навантаження на сервер;
- відображення станів застосунку (завантаження, помилки, успішні операції);
- забезпечення зворотного зв’язку користувачеві у вигляді повідомлень, анімацій і підказок.

З технічної точки зору ці функції реалізуються через маніпуляцію об’єктної моделі документу (DOM, англ., Document Object Model), асинхронні запити до сервера та механізми реактивного оновлення інтерфейсу.

Сучасні інтерактивні веб-інтерфейси мають низку ключових характеристик:

- реактивність, тобто автоматичне оновлення інтерфейсу при зміні даних;
- асинхронність, що дозволяє виконувати запити до сервера без блокування інтерфейсу;
- адаптивність, забезпечення коректної роботи на різних пристроях;
- масштабованість, можливість розширення функціональності без повної переробки інтерфейсу;
- доступність, відповідність стандартам доступності для різних категорій користувачів.

Ці характеристики є критично важливими для створення надійних і довготривалих програмних рішень.

Інтерактивні веб-інтерфейси реалізуються з використанням клієнтських і серверних технологій. На клієнтському рівні застосовуються HTML для структури, CSS для візуального представлення та JavaScript для реалізації інтерактивної поведінки. На серверному рівні інтерфейси інтегруються з бізнес-логікою, базами даних і системами безпеки.

У сучасних підходах активно використовуються фреймворки, які автоматизують синхронізацію стану інтерфейсу та даних, що значно спрощує розробку та підвищує стабільність роботи застосунків.

Враховуючи попередній вибір платформи застосунку та мови бекенду, актуальним для побудови інтерактивного веб-інтерфейсу у роботі буде використання фреймворку Blazor Server.

Blazor Server є сучасною технологією платформи .NET, призначеною для розробки інтерактивних вебінтерфейсів із використанням мови програмування C# на серверній стороні. Його поява стала відповіддю на зростаючу потребу у створенні динамічних вебзастосунків із високим рівнем інтерактивності при зменшенні складності клієнтської частини.

Основне призначення Blazor Server полягає у спрощенні процесу розробки інтерактивних інтерфейсів шляхом перенесення більшості логіки з браузера на сервер. У цій моделі:

- інтерфейс користувача описується декларативно;
- обробка подій виконується на сервері;
- клієнт отримує лише результати змін інтерфейсу.

Такий підхід дозволяє створювати вебінтерфейси, які за рівнем інтерактивності наближаються до настільних застосунків, але працюють у браузері.

Досягається це наступним чином (рис. 2.9) [25]. На стороні сервера Blazor Server запускає компонент Razor, який, окрім обов'язкового шаблону Razor (теги HTML із вбудованим C#), також може включати файл коду (на C#). Хоча розробник програмного забезпечення використовує шаблон Razor для створення повного веб-сайту, вся ця сторінка надсилається до браузера лише один раз, під час першого запиту. Крім того, під час першого завантаження до браузера надсилається бібліотека JavaScript від Microsoft (blazor.server.js) [25].

Blazor зберігає початковий HTML-код, надісланий до браузера, на веб-сервері як віртуальне представлення вмісту браузера, тобто об'єктну модель документа (DOM). Ця віртуальна DOM потім використовується для виявлення

будь-яких змін DOM та передачі цих змін лише до веб-браузера. Бібліотека blazor.server.js відповідає на стороні клієнта за інтеграцію цих сповіщень про зміни у фактичну DOM веб-браузера [25].

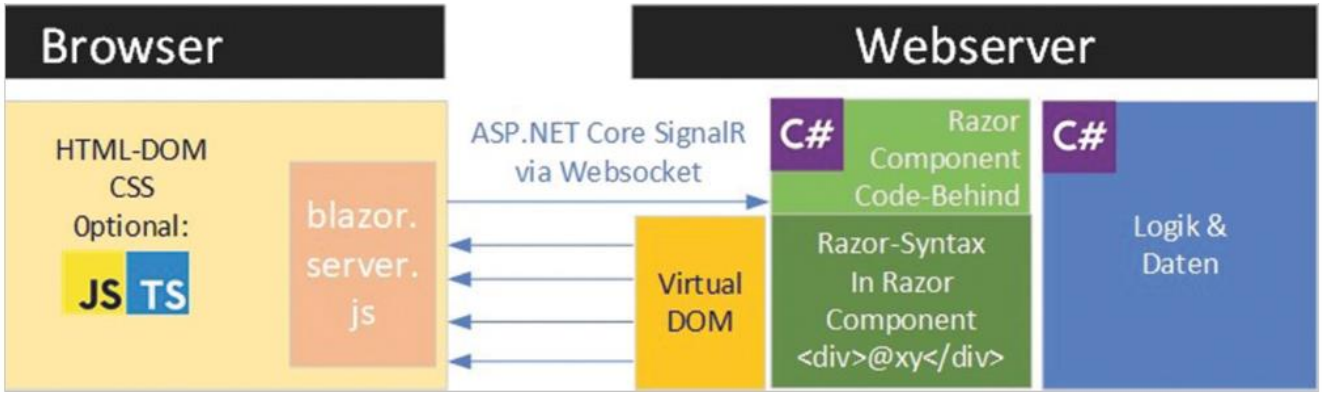


Рис. 2.9 Архітектура та функціональність Blazor Server.

Аналогічно, blazor.server.js відповідає за передачу взаємодії користувача з браузером (введення з клавіатури та кліки миші) на веб-сервер та генерацію подій у компоненті Razor. Після обробки події результуючі зміни DOM передаються клієнту. Передача між веб-браузером та веб-сервером відбувається у форматі, визначеному Microsoft, через середовище .NET [25].

Blazor Server використовує постійне двостороннє з’єднання між клієнтом і сервером для синхронізації стану інтерфейсу. З технічної точки зору це забезпечує:

- централізоване керування станом застосунку;
- швидке реагування на дії користувача;
- мінімальний обсяг клієнтського коду;
- автоматичне оновлення інтерфейсу без прямої маніпуляції DOM.

Завдяки цьому розробник працює з компонентами та станом, а не з низькорівневими браузерними механізмами.

Популярність Blazor Server обумовлена низкою практичних переваг:

- використання єдиної мови програмування C# для клієнтської та серверної частин;

- зменшення необхідності у JavaScript;
- спрощена архітектура застосунку;
- висока продуктивність при роботі з формами та подіями;
- зручність тестування та підтримки коду.

Ці переваги особливо важливі для команд, що вже використовують платформу .NET.

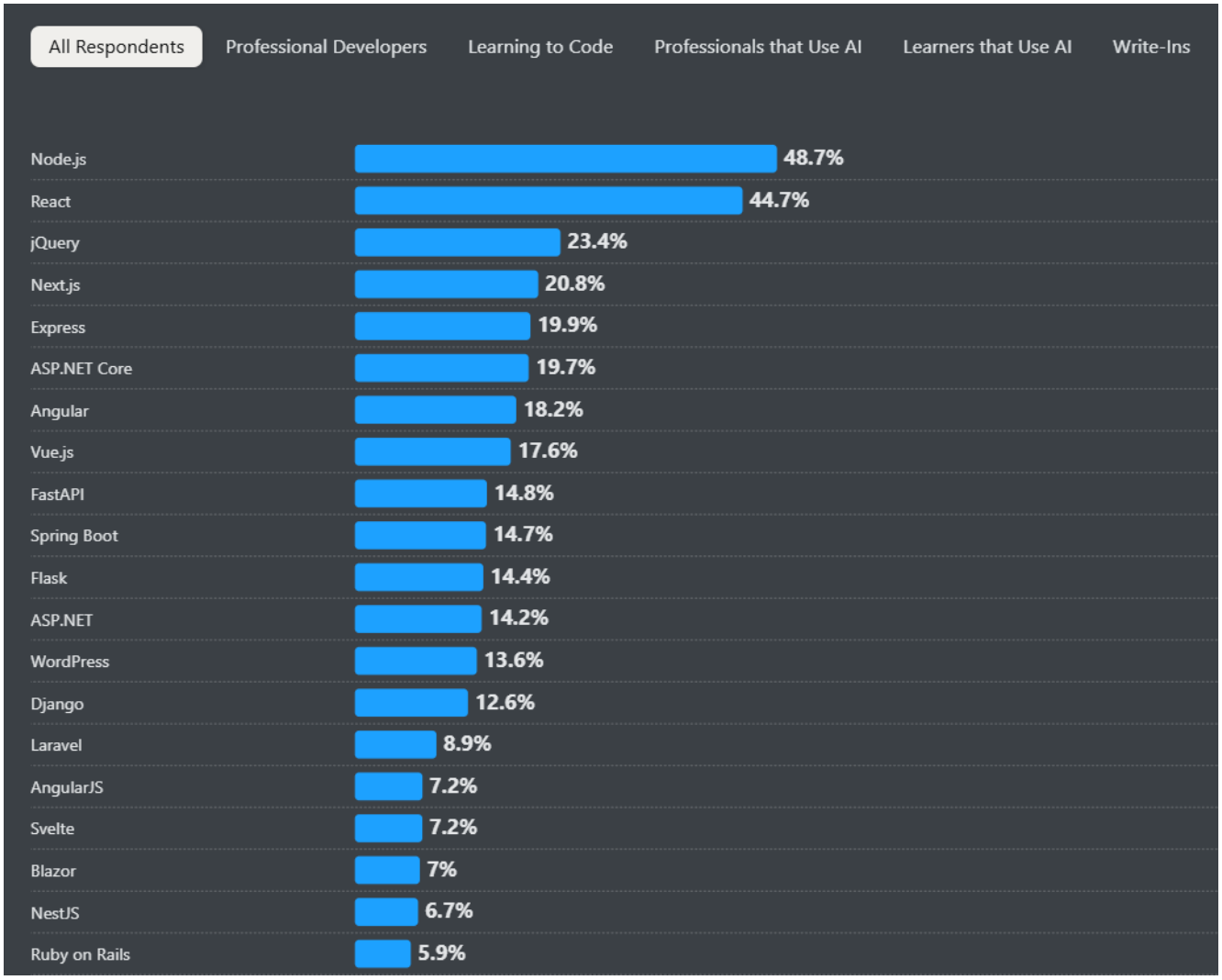


Рис. 2.10 Популярність використання Blazor у 2025 році серед веб-технологій та фреймворків (перша двадцятка) за даними Stack Overflow Developer Survey.

Хоча популярність використання Blazor Server у 2025 році за даними Stack Overflow Developer Survey знаходиться в кінці першої двадцятки (рис. 2.10), її динаміка з кожним роком зростає (рис. 2.11) [24].

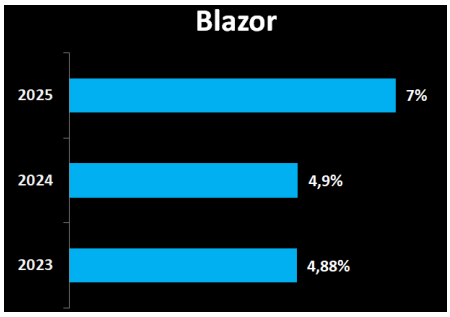


Рис. 11 Динаміка популярності використання Blazor за останні три роки за даними Stack Overflow Developer Survey.

Зростання динаміки пояснюється наступними факторами:

- активний розвиток екосистеми .NET;
- зростаючий попит на інтерактивні вебінтерфейси;
- прагнення зменшити складність клієнтського коду;
- підтримка сучасних стандартів веброзробки.

Blazor Server поступово займає свою нішу поряд із класичними фреймворками, пропонуючи альтернативний підхід до створення інтерактивних веб-інтерфейсів.

2.2.4. База даних

Однією з ключових складових сучасних інформаційних систем є бази даних, оскільки саме вони забезпечують надійне зберігання, впорядкування та обробку великих обсягів даних. У процесі функціонування програмного забезпечення інформаційні системи постійно оперують різноманітними даними, подіями, транзакціями та бізнес-процесами, що зумовлює необхідність використання ефективних механізмів керування цими даними. Застосування баз даних дозволяє

забезпечити цілісність, доступність і актуальність інформації незалежно від масштабу системи.

Основне призначення баз даних полягає у централізованому зберіганні та керуванні структурованою інформацією. Вони використовуються для:

- накопичення та довготривалого зберігання даних;
- забезпечення швидкого доступу до інформації;
- підтримки одночасної роботи багатьох користувачів;
- збереження логічних зв’язків між даними;
- інтеграції різних компонентів інформаційної системи.

Як правило, база даних виступає ядром інформаційної системи, навколо якого будується прикладна та бізнес-логіка.

Бази даних реалізують низку функцій, необхідних для стабільної роботи інформаційних систем:

- збереження даних у структурованому вигляді;
- обробка запитів для отримання, оновлення та видалення інформації;
- керування транзакціями, що гарантує коректність операцій;
- забезпечення цілісності даних за допомогою обмежень і правил;
- керування доступом і безпекою інформації;
- резервне копіювання та відновлення даних.

Реалізація цих функцій дозволяє знизити ризик втрати інформації та помилок обробки даних.

Сучасні бази даних мають низку технічних характеристик, які визначають їх ефективність:

- надійність, що забезпечує збереження даних у разі збоїв;
- продуктивність, яка впливає на швидкість виконання запитів;
- масштабованість, можливість обробки зростаючих обсягів даних;
- цілісність, підтримка логічної узгодженості даних;
- безпека, захист від несанкціонованого доступу;
- відмовостійкість, здатність працювати в умовах часткових збоїв.

Ці характеристики є критичними для інформаційних систем, що функціонують у реальному часі або обслуговують велику кількість користувачів.

У типовій архітектурі інформаційної системи база даних розташовується на окремому рівні, взаємодіючи з серверною частиною через чітко визначені інтерфейси. Такий підхід дозволяє:

- розділити бізнес-логіку та зберігання даних;
- спростити супровід і модернізацію системи;
- підвищити рівень безпеки;
- оптимізувати використання ресурсів.

Завдяки цьому бази даних забезпечують стабільність і довготривалу експлуатацію інформаційних систем.

В умовах цифровізації та зростання обсягів інформації бази даних набувають особливої важливості. Вони є основою для реалізації аналітики, звітності, автоматизації процесів і прийняття управлінських рішень. Без використання баз даних сучасні інформаційні системи не здатні ефективно обробляти та зберігати дані в необхідних обсягах і з потрібним рівнем надійності.

У сучасних інформаційних системах використовується широкий спектр систем керування базами даних, які відрізняються архітектурою, моделлю даних, рівнем продуктивності, масштабованістю та сферою застосування. Існують реляційні, об’єктно-реляційні та NoSQL-орієнтовані СКБД, кожна з яких оптимізована для розв’язання певного класу задач – від транзакційних корпоративних систем до розподілених сховищ великих даних. За таких умов вибір СКБД безпосередньо впливає на ефективність, надійність і подальший розвиток програмного забезпечення, що зумовлює необхідність їх порівняльного аналізу за ключовими технічними характеристиками з метою обґрунтованого вибору для конкретного застосунку (таблиця 2.2).

Таблиця 2.2

Порівняння популярних СКБД

СКБД	Тип	Модель даних	Підтримка ACID	Масштабованість	Продуктивність	Інтеграція з .NET	Типові сценарії використання
Microsoft SQL Server	Реляційна	Таблична	Повна	Вертикальна + кластеризація	Висока	Дуже висока	Корпоративні ІС, ERP, CRM, BI
Oracle Database	Реляційна	Таблична	Повна	Дуже висока	Дуже висока	Обмежена	Критичні корпоративні системи
PostgreSQL	Об’єктно-реляційна	Таблична + типи	Повна	Висока	Висока	Середня	Складні БД, аналітика
MySQL	Реляційна	Таблична	Часткова / повна (InnoDB)	Середня	Середня	Середня	Веб-застосунки
MariaDB	Реляційна	Таблична	Повна	Середня	Середня	Середня	Open-source проєкти
IBM Db2	Реляційна	Таблична	Повна	Висока	Висока	Обмежена	Банківські та корпоративні ІС
MongoDB	NoSQL	Документо-орієнтована	Часткова	Дуже висока	Висока	Середня	JSON-дані, гнучкі схеми
Cassandra	NoSQL	Колонкова	Обмежена	Дуже висока	Висока	Низька	Big Data, розподілені системи
Redis	NoSQL (in-memory)	Ключ-значення	Обмежена	Висока	Дуже висока	Середня	Кешування, сесії
SQLite	Реляційна (вбудована)	Таблична	Повна	Відсутня	Низька	Середня	Мобільні та локальні застосунки

Як видно з таблиці 2.2, Microsoft SQL Server є оптимальним вибором для .NET-орієнтованих застосунків. Тісна інтеграція SQL Server із середовищем .NET створює ефективну основу для побудови інформаційних систем різного рівня складності.

Microsoft SQL Server забезпечує широкий спектр функцій, необхідних для стабільної роботи інформаційних систем розроблених для .NET. До них можна віднести наступні:

- керування структурованими даними з підтримкою складних зв’язків;

- виконання транзакцій із дотриманням принципів ACID;
- оптимізацію запитів для підвищення продуктивності;
- керування доступом на рівні користувачів і ролей;
- резервне копіювання та відновлення даних;
- підтримку високої доступності та відмовостійкості.

Ці функції дозволяють зменшити ризик втрати інформації та забезпечити безперервну роботу системи.

Особливість роботи Microsoft SQL Server полягає в розподілі відповідальності між обробкою логіки запитів, керуванням пам'яттю та забезпеченням транзакційної цілісності даних (рис. 2.12) [26].

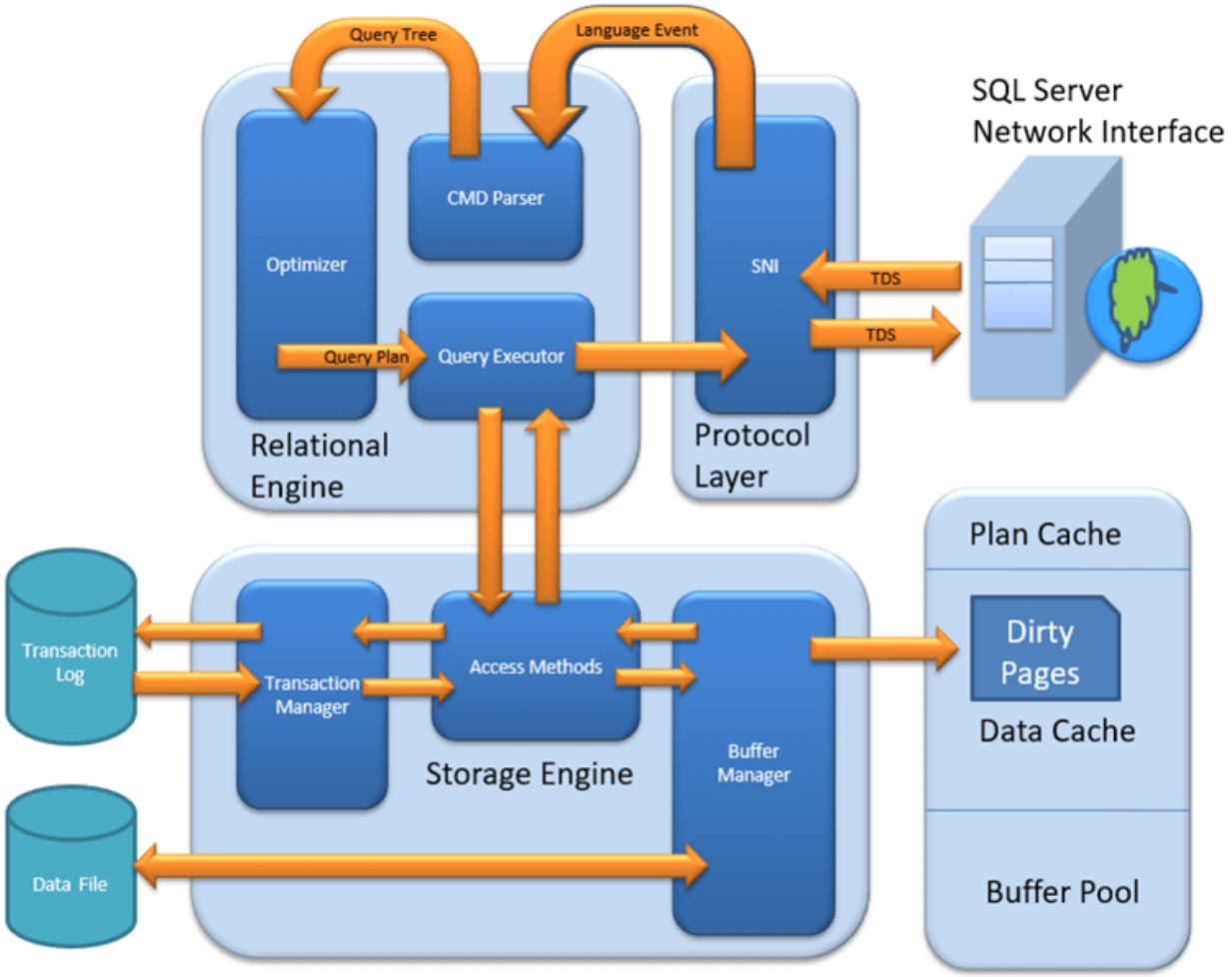


Рис. 2.12 Схема архітектури Microsoft SQL Server

Як можна побачити з рисунку 2.12 – SQL-запит, отриманий через мережевий інтерфейс TDS, передається до реляційного рушія, де він проходить

етап парсингу (CMD Parser), етап оптимізації (Optimizer) та етап формування плану виконання (Query Plan). Далі модуль (Query Executor) передає план у модуль зберігання (Storage Engine), який через (Access Methods, Buffer Manager) та (Transaction Manager) взаємодіє з кешем даних, буферним пулом (Buffer Pool), файлами даних і журналом транзакцій. Даний підхід забезпечує високу продуктивність, ефективне використання пам'яті та надійну транзакційну обробку, що є критично важливим для стабільної роботи корпоративних інформаційних систем [26].

Таким чином, використання Microsoft SQL Server в екосистемі .NET є доцільним завдяки їх тісній технологічній інтеграції, спільній підтримці з боку Microsoft та оптимізованій взаємодії з мовою C#. Таке поєднання забезпечує високу продуктивність, надійну транзакційну обробку даних, підвищений рівень безпеки та спрощений супровід, що робить MS SQL Server ефективним вибором для створення масштабованих і стабільних інформаційних систем на платформі .NET.

Вибір Microsoft SQL Server також підтверджується популярністю даного СКБД (рис. 2.13) [24].

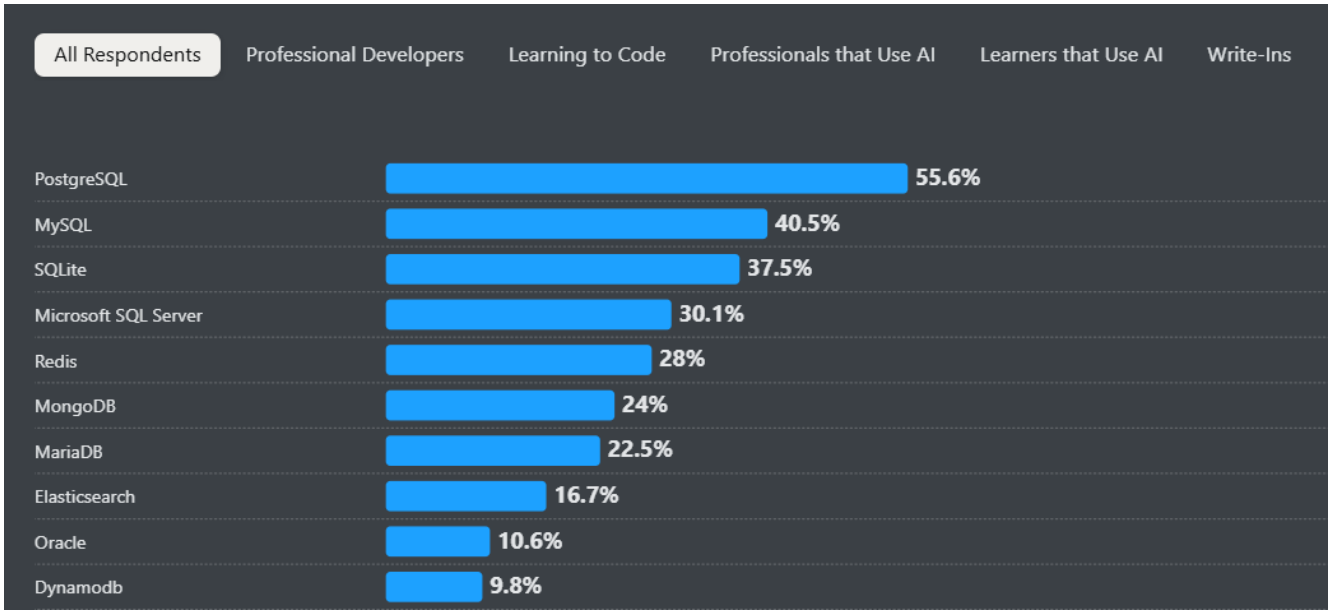


Рис. 2.13. Популярність використання СКБД у 2025 році (перша десятка) за даними Stack Overflow Developer Survey.

2.2.5. Object-Relational Mapping

Як було відмічено раніше, однією з ключових переваг використання Microsoft SQL Server є його тісна інтеграція з .NET-технологіями. В екосистемі .NET для доступу до даних MS SQL Server застосовуються наступні підходи:

- Entity Framework Core як ORM-засіб;
- ADO.NET для низькорівневого керування з'єднаннями;
- мова запитів LINQ, що забезпечує типобезпечну роботу з даними.

Такі підходи спрощують розробку, підвищують читабельність коду та зменшують кількість помилок при роботі з даними.

Entity Framework Core є найпоширенішим ORM-засобом та ключовим компонентом платформи .NET, що реалізує об'єктно-реляційне відображення між прикладною логікою, написаною мовою C#, та реляційною базою даних Microsoft SQL Server.

Його основне призначення полягає у зменшенні розриву між об'єктною моделлю застосунку та табличною структурою бази даних шляхом автоматичного перетворення об'єктів, їх властивостей і зв'язків у відповідні таблиці, стовпці та зовнішні ключі.

Entity Framework Core дозволяє розробникам працювати з даними на рівні предметної області, використовуючи класи, колекції та навігаційні властивості, без необхідності ручного написання SQL-запитів. Запити до бази формуються за допомогою запитів інтегрованих в мову LINQ (англ.. Language Integrated Query), які фреймворк транслює у оптимізований SQL-код з урахуванням специфіки Microsoft SQL Server (рис. 2.14). Такий підхід підвищує читабельність коду, спрощує його супровід і знижує ймовірність помилок, пов'язаних із синтаксисом або типізацією даних [27].

Важливою особливістю Entity Framework є механізм відстеження змін, який автоматично фіксує всі операції додавання, зміни або видалення об'єктів. На основі цієї інформації фреймворк формує необхідні SQL-операції для синхронізації стану об'єктної моделі з базою даних. Це значно спрощує

реалізацію транзакційної логіки та забезпечує узгодженість даних у багатокористувацьких середовищах.

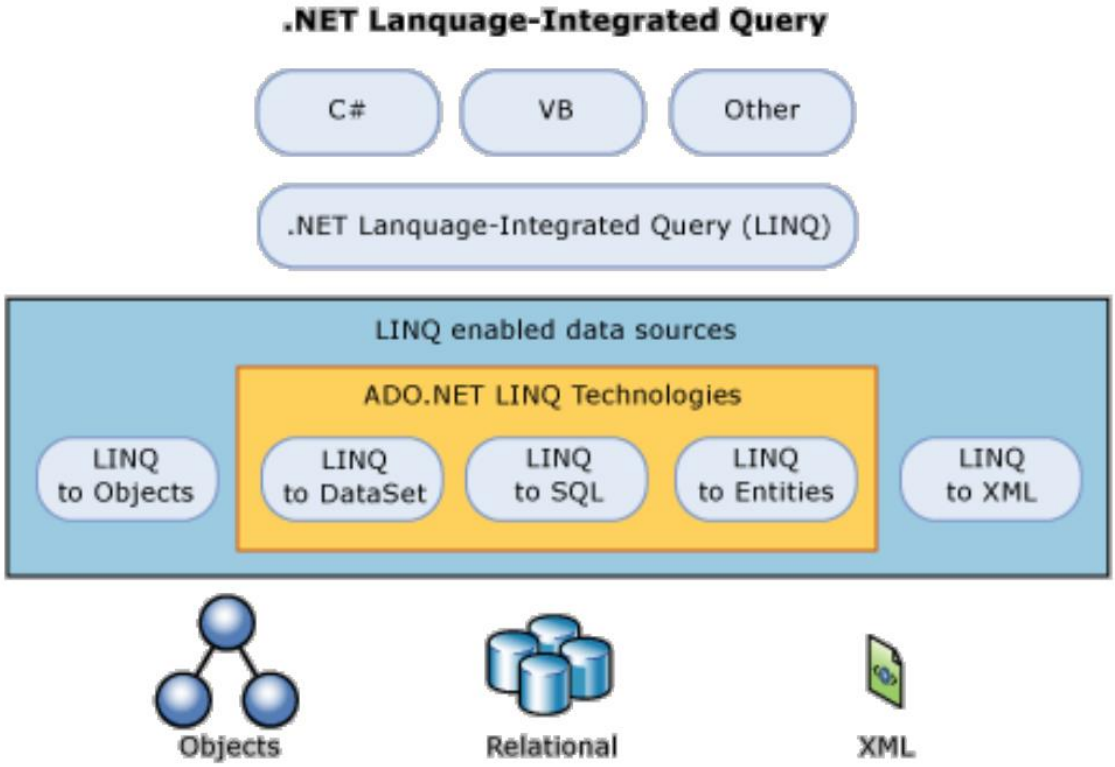


Рис. 2.14 Зв'язок LINQ з високорівневими мовами програмування та джерелами даних.

Entity Framework також надає інструменти для керування схемою бази даних за допомогою механізму міграцій. Міграції дозволяють версіонувати структуру бази, автоматично застосовувати зміни до MS SQL Server і підтримувати відповідність між моделлю застосунку та фізичним сховищем даних. Це особливо важливо для довготривалих проєктів, які постійно еволюціонують.

Завдяки тісній інтеграції з екосистемою .NET, Entity Framework підтримує асинхронні операції, механізми безпеки, транзакції та розширені можливості налаштування продуктивності. У разі потреби він дозволяє поєднувати об’єктно-орієнтований підхід із прямим виконанням SQL-запитів або викликом збережених процедур, що забезпечує баланс між зручністю використання ORM і контролем над виконанням запитів.

Entity Framework виступає ключовою сполучною ланкою між мовою програмування C#, платформою .NET та реляційною СКБД Microsoft SQL Server, забезпечуючи цілісну та узгоджену модель роботи з даними. Завдяки ORM-підходу Entity Framework дозволяє реалізовувати доступ до бази даних на об’єктно-орієнтованому рівні, автоматично транслюючи логіку C#-застосунку у оптимізовані SQL-запити (рис. 2.15) [28].

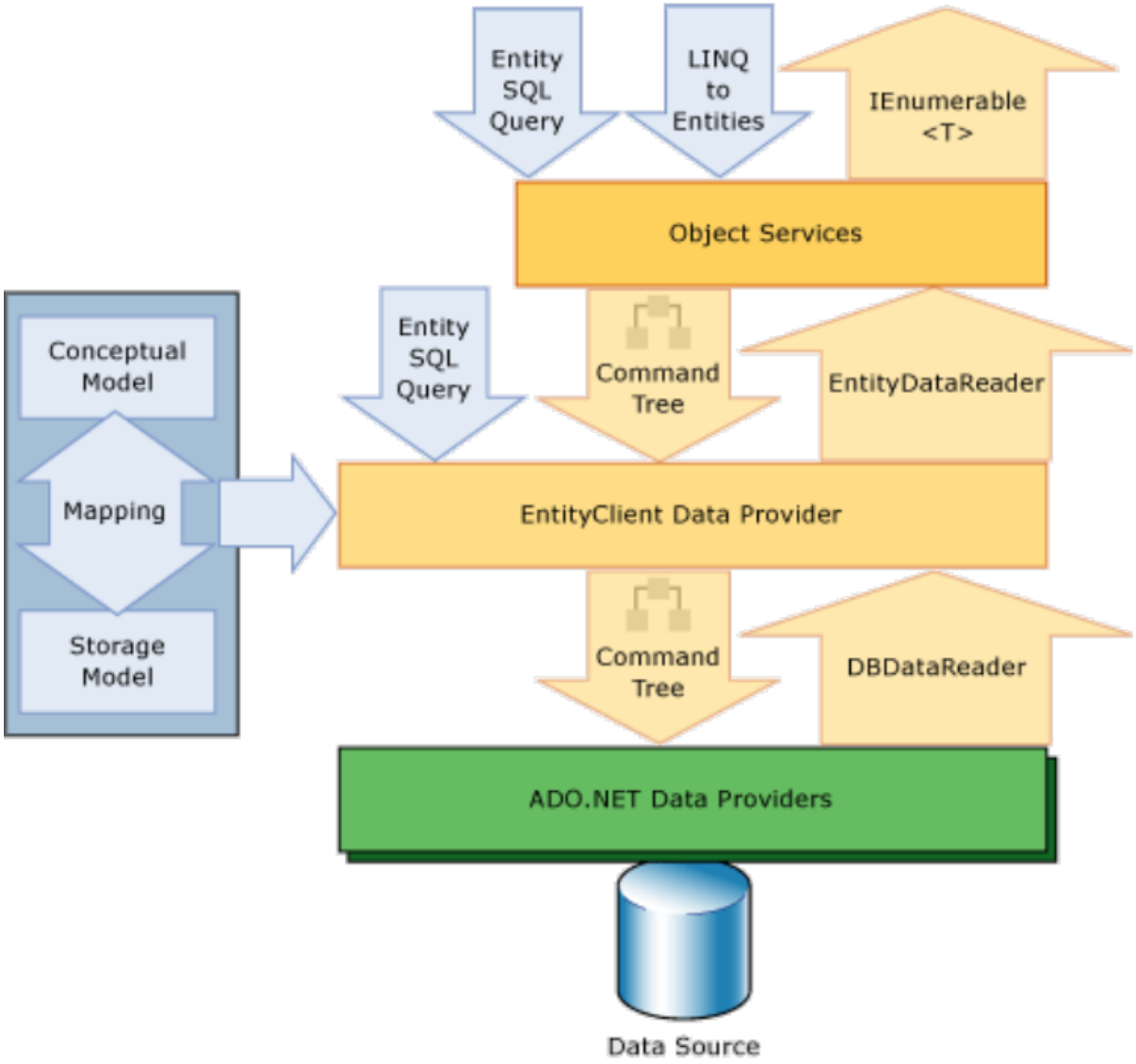


Рис. 2.15 Архітектура Entity Framework для доступу до даних.

Тісна інтеграція з .NET і повна підтримка можливостей MS SQL Server гарантують транзакційну надійність, типобезпечність і високу продуктивність, що робить Entity Framework ефективним інструментом для розробки сучасних, масштабованих та підтримуваних інформаційних систем.

2.2.6. Візуалізація результатів

В сучасних інформаційних системах візуалізація результатів відіграє ключову роль у процесі аналізу, інтерпретації та прийняття управлінських рішень. Зі зростанням обсягів і складності даних традиційні текстові або табличні форми подання інформації стають малоефективними. Саме тому використання спеціалізованих середовищ візуалізації є необхідним елементом побудови сучасних програмних рішень.

Основним призначенням середовищ візуалізації є перетворення структурованих і неструктурованих даних у наочну графічну форму, зрозумілу для користувача. Вони дозволяють швидко виявляти закономірності, тенденції, відхилення та взаємозв'язки між показниками, що складно або неможливо зробити при аналізі «сирих даних».

В інформаційних системах такі середовища виконують роль інтерфейсу між даними та користувачем, забезпечуючи ефективну взаємодію з результатами обчислень і аналітичних процесів.

Основні функції середовищ візуалізації

- *Графічне подання даних.* Відображення результатів у вигляді діаграм, графіків, гістограм, таблиць, карт і панелей індикаторів, що підвищує швидкість сприйняття інформації.
- *Інтерактивність.* Забезпечення можливості фільтрації, масштабування, деталізації (drill-down) та динамічного оновлення даних у режимі реального часу.
- *Аналіз і порівняння.* Порівняння показників за різними періодами, категоріями або сценаріями, що підтримує аналітичні та прогнозні функції системи.
- *Підтримка прийняття рішень.* Надання узагальненої та візуально зрозумілої інформації для керівників і аналітиків.
- *Контроль і моніторинг.* Відображення ключових показників ефективності та стану системи в режимі реального часу.

Ефективні середовища візуалізації результатів характеризуються такими технічними властивостями:

- *Масштабованість* – здатність працювати з великими обсягами даних без втрати продуктивності;
- *Гнучкість налаштування* – можливість адаптації візуальних елементів під потреби користувачів;
- *Інтерактивність* – підтримка подієво-орієнтованої взаємодії;
- *Інтегрованість* – сумісність з базами даних, сервісами та аналітичними модулями інформаційної системи;
- *Продуктивність* – мінімальні затримки при оновленні та відображенні даних;
- *Юзабіліті* – інтуїтивно зрозумілий інтерфейс і чітка структура подання інформації.

З архітектурної точки зору середовища візуалізації розташовуються на рівні представлення та взаємодіють із серверною логікою і базами даних через API або сервіси. Вони забезпечують відокремлення логіки обробки даних від способів їх відображення, що підвищує модульність і підтримуваність системи.

На сьогодні на ринку ПЗ існує багато рішень щодо візуалізації даних. Одним із таких є легка та гнучка JavaScript-бібліотека Chart.js.

Вона призначена для побудови інтерактивних графіків і діаграм на основі стандарту HTML5 Canvas. Її використання під час розробки програмного забезпечення дає змогу швидко інтегрувати візуальні елементи у вебінтерфейс застосунку без значних витрат ресурсів та складних налаштувань.

Важливість Chart.js полягає в можливості динамічного відображення результатів обчислень, статистичних показників і аналітичних даних у реальному часі. Бібліотека підтримує різні типи візуалізацій, зокрема лінійні, стовпчикові, кругові та комбіновані діаграми, що дозволяє адаптувати подання інформації до конкретних вимог предметної області (рис. 2.16). Крім того, інтерактивність графіків сприяє кращому сприйняттю даних і підвищує інформативність програмного продукту [29].

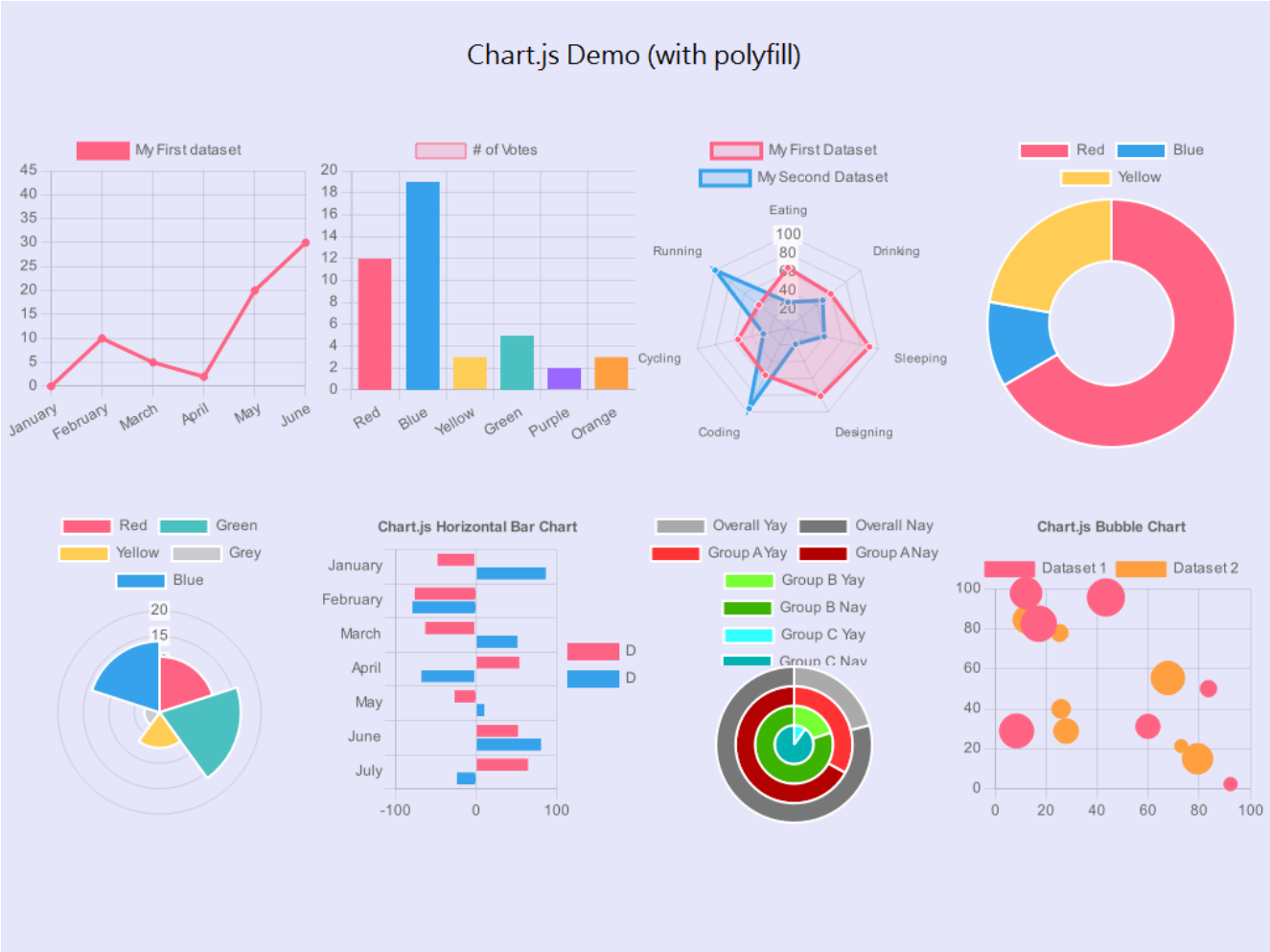


Рис. 2.16 Приклади візуалізації даних за допомогою Chart.js

Також Chart.js може легко інтегруватися з Blazor Server, що дозволяє поєднати серверну логіку платформи .NET, реалізовану мовою C#, із клієнтською візуалізацією на основі JavaScript. У такій архітектурі Blazor Server відповідає за обробку даних, бізнес-логіку та взаємодію з базою даних, тоді як Chart.js використовується для відображення результатів у браузері користувача. Обмін даними між C# та JavaScript здійснюється через механізм JavaScript Interop, що забезпечує синхронізацію стану інтерфейсу з серверними даними в режимі реального часу.

Застосування Chart.js у поєднанні з Blazor Server підвищує інформативність вебінтерфейсу та дозволяє створювати інтерактивні панелі візуалізації без перевантаження клієнтської частини. Такий підхід є особливо доцільним для

інформаційно-аналітичних і корпоративних систем, де важливими є централізоване керування логікою, безпека даних і зручність їх представлення.

2.3 Висновок до другого розділу

З огляду на архітектурні та функціональні вимоги до сучасних інформаційних систем, обраний стек технологій є технічно обґрунтованим і доцільним. Використання Visual Studio як середовища розробки забезпечує комплексну підтримку платформи .NET, зручні засоби налагодження, тестування та супроводу програмного коду. Мова програмування C# доцільно використовується як основна мова реалізації бекенду завдяки своїй типобезпечності, продуктивності та тісній інтеграції з .NET. Для побудови клієнтської частини застосунку застосовуються HTML, CSS та JavaScript як базові вебтехнології, що забезпечують коректне відображення інтерфейсу та взаємодію з користувачем.

Платформа Blazor Server використовується для створення інтерактивного вебінтерфейсу з централізованою серверною логікою, що підвищує безпеку та спрощує керування станом застосунку. Microsoft SQL Server обрано як надійний засіб зберігання даних, здатний забезпечити транзакційну цілісність, масштабованість і високу продуктивність. Entity Framework виконує роль ORM-шару між прикладним кодом і базою даних, спрощуючи доступ до даних та підвищуючи підтримуваність програмного забезпечення. Для наочного подання результатів обробки даних використовується Chart.js, який реалізує функції візуалізації та підвищує інформативність користувацького інтерфейсу (рис. 2.17).

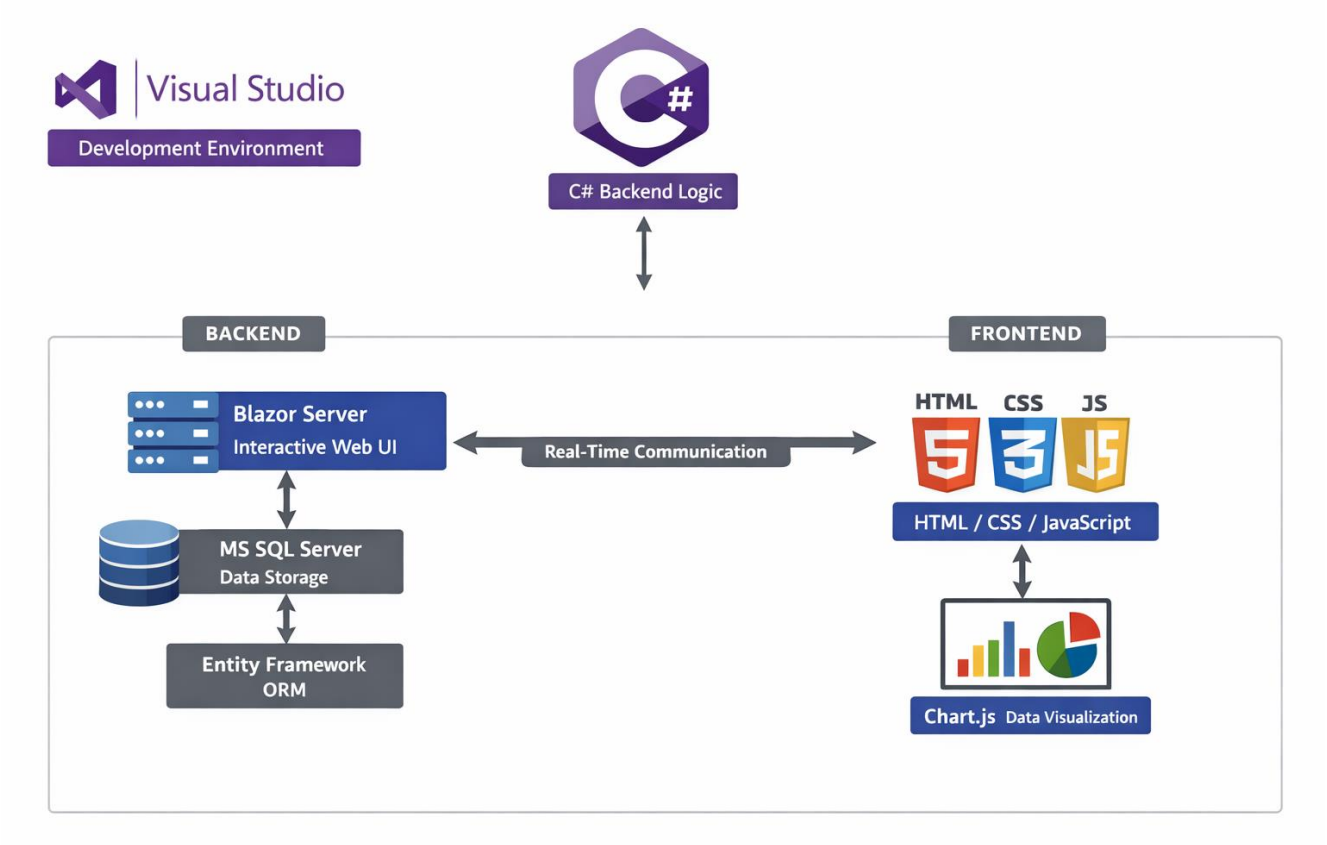


Рис. 2.17 Стек технологій реалізації інформаційної системи порівняння ORM-фреймворків

Таким чином, поєднання зазначених інструментів і технологій формує цілісне, масштабоване та ефективне рішення для розробки сучасних веборієнтованих інформаційних систем.

3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Загальна характеристика інформаційної системи вибору ORM для платформи .NET

У процесі проєктування та розробки сучасних інформаційних систем особливу роль відіграє вибір технологій доступу до даних. Для платформи .NET існує значна кількість ORM-фреймворків, кожен з яких має власні переваги, обмеження та сферу доцільного застосування. Неправильний вибір ORM може призвести до зниження продуктивності системи, ускладнення супроводу коду або збільшення часу розробки.

На практиці вибір ORM часто здійснюється інтуїтивно або на основі особистого досвіду розробника, що не завжди є оптимальним підходом, особливо для складних або довготривалих проєктів. Саме тому актуальним є використання інформаційних систем, які дозволяють формалізувати знання та досвід у певній предметній області й автоматизувати процес прийняття рішень.

У межах даної роботи розроблено інформаційну систему вибору ORM-фреймворку для платформи .NET. Основним призначенням системи є надання рекомендацій щодо найбільш доцільного ORM на основі заданих користувачем характеристик проєкту. Система аналізує низку параметрів, що відображають особливості майбутньої інформаційної системи, та на основі вбудованої моделі знань обчислює інтегральну оцінку для кожного ORM-фреймворку.

Розроблена система реалізована у вигляді односторінкового вебзастосунку з використанням технології Blazor Server та бібліотеки компонентів MudBlazor. Такий підхід забезпечує інтерактивність інтерфейсу, миттєве оновлення результатів при зміні вхідних параметрів і зручність використання без необхідності перезавантаження сторінки.

Експертна система орієнтована як на початківців, так і на досвідчених розробників. Вона не лише пропонує рекомендований ORM-фреймворк, але й пояснює причини такого вибору, зокрема визначає найбільш значущі параметри, які мали вирішальний вплив на результат. Додатково система надає візуалізацію результатів у вигляді графіків, що дозволяє порівняти альтернативні рішення між собою.

Таким чином, розроблена експертна система дозволяє підвищити обґрунтованість прийняття технічних рішень, зменшити суб'єктивний фактор під час вибору ORM-фреймворку та слугує прикладом практичного застосування експертних систем у сфері програмної інженерії.

3.2 Функціональні вимоги до інформаційної системи

Функціональні вимоги визначають перелік можливостей, які повинна забезпечувати інформаційна система вибору ORM-фреймворку для коректної та зручної роботи користувача. Формування чітких функціональних вимог є необхідним етапом проєктування інформаційної системи, оскільки вони визначають поведінку системи та очікувані результати її роботи.

Розроблена інформаційна система повинна забезпечувати виконання таких основних функцій:

1. *Збір вхідних даних від користувача.* Система повинна надавати користувачу можливість задавати характеристики майбутньої інформаційної системи шляхом вибору рівня критичності окремих параметрів. До таких параметрів належать:

- обсяг даних;
- частота звернень до бази даних;
- складність об'єктної моделі;
- потреба у складних SQL-запитах;
- критичність продуктивності;
- вимоги до підтримуваності коду.

2. *Валідація введених параметрів.* Значення кожного параметра повинні задаватися у межах визначеної шкали (від 1 до 5), де мінімальне значення відповідає низькій важливості параметра, а максимальне – критичній важливості. Система повинна запобігати введенню некоректних значень.

3. *Автоматичне обчислення рекомендацій.* Після зміни будь-якого параметра система повинна автоматично виконувати перерахунок результатів без необхідності додаткових дій з боку користувача. Це забезпечує інтерактивність та дозволяє миттєво оцінювати вплив окремих характеристик на фінальний вибір ORM-фреймворку.

4. *Оцінювання альтернативних ORM-фреймворків.* Для кожного ORM, що розглядається системою (Entity Framework Core, Dapper, NHibernate, Linq2Db, ServiceStack.OrmLite), повинна обчислюватися інтегральна оцінка, яка відображає ступінь відповідності цього фреймворку заданим вимогам користувача.

5. *Формування пояснення результатів.* Система повинна надавати текстове пояснення отриманого результату, в якому зазначається:

- рекомендований ORM-фреймворк;
- причина його вибору;
- параметр, що мав найбільший вплив на прийняття рішення.

6. *Візуалізація результатів.* Для підвищення наочності система повинна відображати результати оцінювання у графічному вигляді, зокрема у формі стовпчикової діаграми, яка дозволяє порівняти інтегральні оцінки всіх ORM-фреймворків.

7. *Зручний користувацький інтерфейс.* Інтерфейс системи повинен бути інтуїтивно зрозумілим, не перевантаженим зайвими елементами та адаптованим для роботи у веб-браузері без додаткового програмного забезпечення.

Таким чином, функціональні вимоги визначають основні сценарії взаємодії користувача з інформаційною системою та забезпечують досягнення її головної мети – підтримки прийняття обґрунтованого рішення щодо вибору ORM-фреймворку для платформи .NET.

3.3 Архітектура та логічна структура інформаційної системи

Архітектура інформаційної системи визначає принципи її побудови, взаємодію основних компонентів та розподіл відповідальності між ними. Правильно обрана архітектура забезпечує масштабованість, зручність супроводу та можливість подальшого розширення функціональності системи.

Розроблена інформаційна система вибору ORM-фреймворку реалізована на основі архітектурної моделі клієнт–сервер з використанням технології Blazor Server. У межах даної моделі користувацький інтерфейс виконується у веб-браузері, тоді як обчислювальна логіка та обробка даних розміщені на сервері.

Загальна архітектурна схема системи

На логічному рівні систему можна поділити на такі основні компоненти (рис. 3.1):

- *Користувацький інтерфейс (UI).* Реалізований за допомогою компонентів MudBlazor. Забезпечує введення параметрів експертної оцінки, відображення результатів та візуалізацію даних у вигляді графіків.
- *Компонент керування сценарієм оцінювання.* Відповідає за збереження поточного стану вхідних параметрів, ініціює перерахунок результатів при їх зміні та передає дані до експертної логіки.
- *Експертний сервіс.* Центральний компонент системи, який реалізує алгоритм оцінювання ORM-фреймворків. Саме тут відбувається обчислення інтегральних оцінок та формування рекомендацій.
- *Конфігураційний модуль.* Містить початкові експертні ваги параметрів для кожного ORM-фреймворку. Такий підхід дозволяє змінювати модель знань без модифікації програмного коду.
- *Модуль візуалізації результатів.* Формує структури даних, необхідні для побудови графіків, та передає їх до відповідних UI-компонентів.

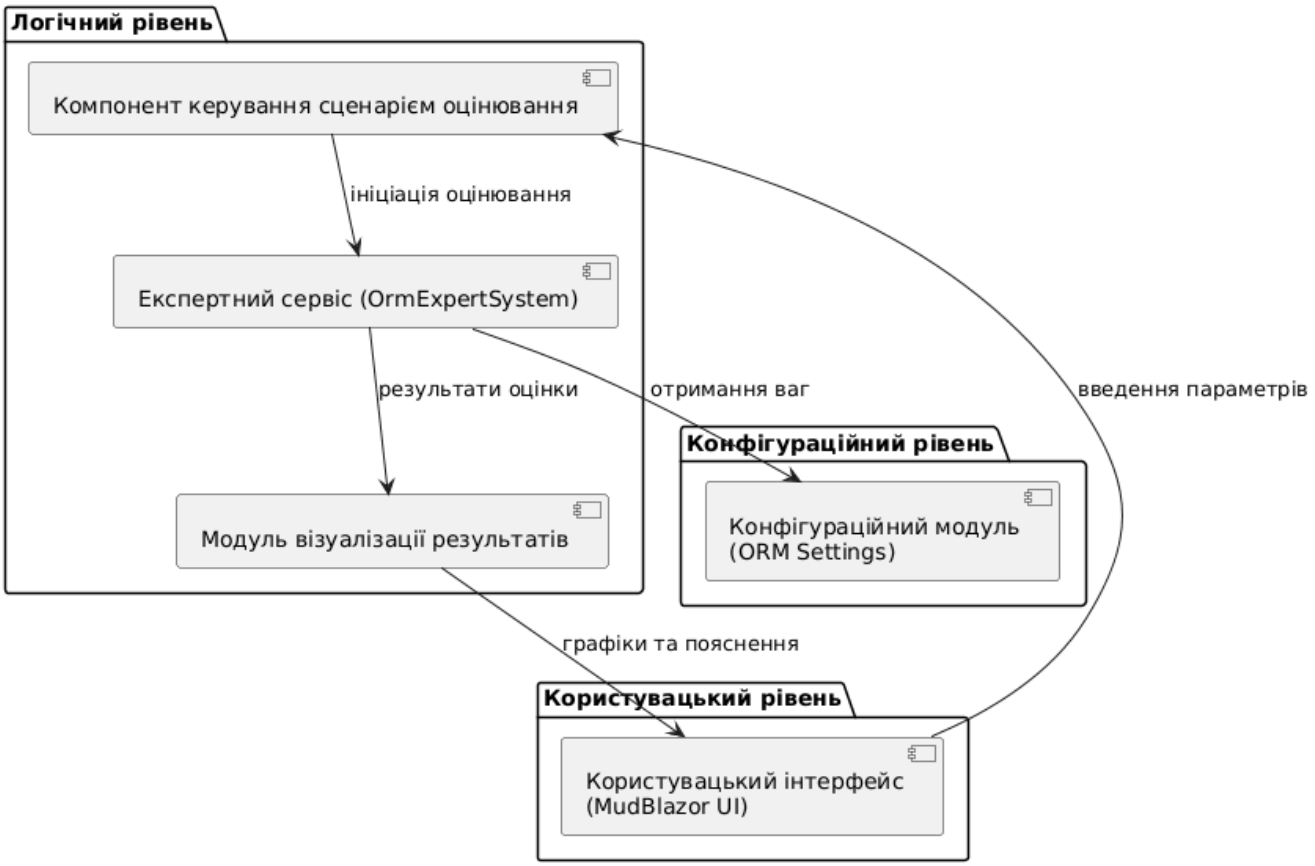


Рис. 3.1 Архітектурна схема системи

Логічна структура програмних компонентів

З точки зору програмної реалізації система має модульну структуру, у якій кожен компонент відповідає за окрему функцію. Основними програмними сутностями є:

- моделі даних, що описують вхідні параметри сценарію та результати оцінювання;
- сервіс експертної логіки, який реалізує алгоритм прийняття рішення;
- UI-компоненти для введення параметрів та відображення результатів;
- допоміжні класи для локалізації, відображення назв параметрів та пояснень.

Такий підхід відповідає принципам слабкого зв’язування та підвищує читабельність і підтримуваність коду.

3.4 Алгоритм роботи інформаційної системи

Алгоритм роботи інформаційної системи визначає послідовність обробки вхідних даних, метод оцінювання альтернатив та правила формування рекомендацій. У межах даної роботи використано підхід багатокритеріального оцінювання, який дозволяє врахувати вплив кількох параметрів одночасно.

Вхідні параметри експертної оцінки

Для формування рекомендації користувач задає значення шести основних параметрів, кожен з яких відображає певну характеристику майбутньої інформаційної системи:

- *Обсяг даних* – характеризує очікувану кількість записів та загальний розмір бази даних.
- *Частота запитів* – визначає інтенсивність звернень до бази даних у процесі експлуатації.
- *Складність моделей* – відображає кількість сутностей та зв’язків між ними.
- *Складні SQL-запити* – описує потребу у використанні JOIN, агрегацій та підзапитів.
- *Критичність продуктивності* – показує, наскільки важливим є мінімальний час виконання операцій.
- *Підтримуваність* – визначає важливість зручності супроводу та читабельності коду.

Значення кожного параметра задається за п’ятибальною шкалою, де 1 відповідає мінімальній важливості, а 5 – максимальній.

Інтегральна оцінка ORM-фреймворків

Для порівняння ORM-фреймворків у системі використовується інтегральна оцінка, яка є узагальненим числовим показником відповідності конкретного ORM заданим вимогам.

Інтегральна оцінка для кожного ORM обчислюється як зважена сума значень усіх параметрів:

$$S = \sum_{i=1}^{n} w_i \cdot p_i$$

де:

S – інтегральна оцінка ORM-фреймворку;

w_i – експертна вага i-го параметра для конкретного ORM;

p_i – значення параметра, задане користувачем;

n – кількість параметрів оцінювання.

Експертні ваги параметрів задаються заздалегідь та відображають сильні та слабкі сторони кожного ORM-фреймворку. Наприклад, для ORM, орієнтованих на максимальну продуктивність, більша вага надається параметрам продуктивності та складних SQL-запитів, тоді як для ORM з акцентом на зручність розробки – параметру підтримуваності.

Визначення найбільш значущого параметра

Окрім вибору найкращого ORM-фреймворку, система визначає параметр, який мав найбільший вплив на кінцевий результат. Для цього аналізується вклад кожного параметра в інтегральну оцінку:

$$C_i = w_i \cdot p_i$$

де:

C_i – вклад i-го параметра в загальну оцінку.

Параметр з максимальним значенням C_i вважається найбільш значущим і відображається користувачу у текстовому поясненні результату.

Послідовність роботи алгоритму

Алгоритм роботи інформаційної системи можна описати такою послідовністю кроків (рис. 3.2):

1. Користувач задає значення параметрів експертної оцінки.
2. Система автоматично фіксує зміни вхідних даних.
3. Для кожного ORM-фреймворку обчислюється інтегральна оцінка.
4. Визначається ORM з максимальною інтегральною оцінкою.
5. Аналізується внесок кожного параметра у фінальний результат.

6. Формується рекомендація та пояснення для користувача.

7. Результати візуалізуються у графічному вигляді.

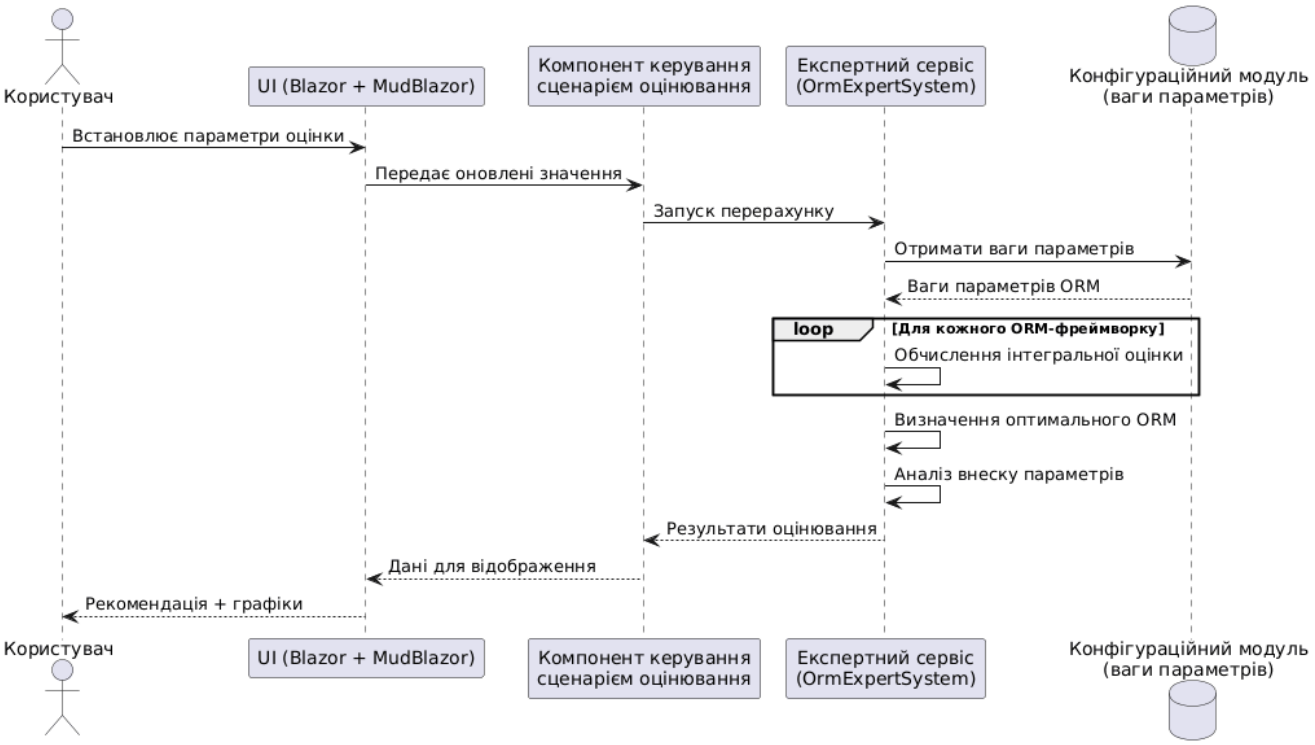


Рис. 3.2 Діаграма послідовностей роботи алгоритму

3.5 Реалізація користувацького інтерфейсу та взаємодія з системою

Користувацький інтерфейс інформаційної системи відіграє ключову роль у забезпеченні зручності її використання та правильного введення вихідних даних. Оскільки система орієнтована на підтримку прийняття рішень, інтерфейс повинен бути інтуїтивно зрозумілим, наочним та не перевантаженим технічними деталями.

Розроблений інтерфейс реалізовано у вигляді односторінкового вебзастосунку з використанням компонентної бібліотеки MudBlazor, що дозволило забезпечити сучасний вигляд, адаптивність та інтерактивність.

Загальна структура інтерфейсу

Інтерфейс експертної системи логічно поділяється на дві основні області (рис. 3.3):

- область введення параметрів;
- область відображення результатів експертної оцінки.

Такий підхід дозволяє користувачу одночасно змінювати параметри та спостерігати за змінами результатів у реальному часі, що значно підвищує наочність роботи системи.

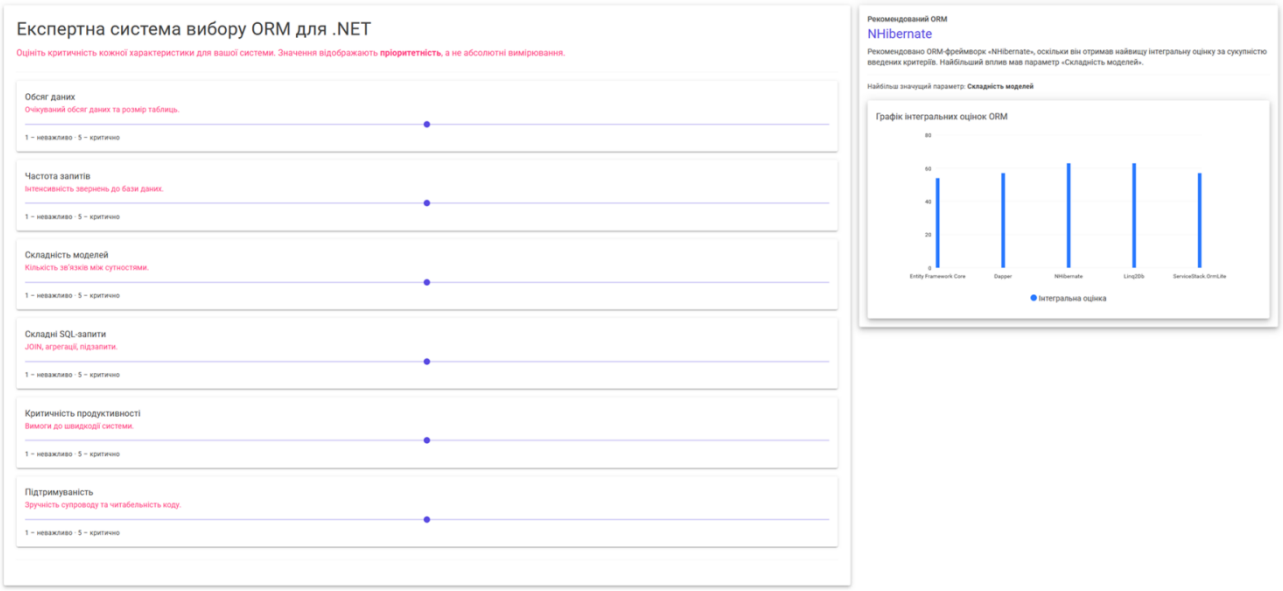


Рис. 3.3 Головна сторінка інформаційної системи

Введення параметрів експертної оцінки

Для введення значень параметрів використано елементи керування типу повзунок (slider). Кожен параметр супроводжується текстовою назвою та коротким описом, який пояснює його зміст і вплив на роботу інформаційної системи.

Користувачу пропонується оцінити критичність кожного параметра за п’ятибальною шкалою:

- «1» – параметр майже не має значення;
- «3» – середній рівень важливості;
- «5» – параметр є критично важливим.

Такий спосіб введення дозволяє уникнути використання абсолютних числових показників, які можуть бути складними для інтерпретації, та зосередитися на відносній важливості характеристик.

Для кожного параметра відображається підпис із поясненням шкали, що зменшує ймовірність помилок під час введення даних (рис. 3.4).

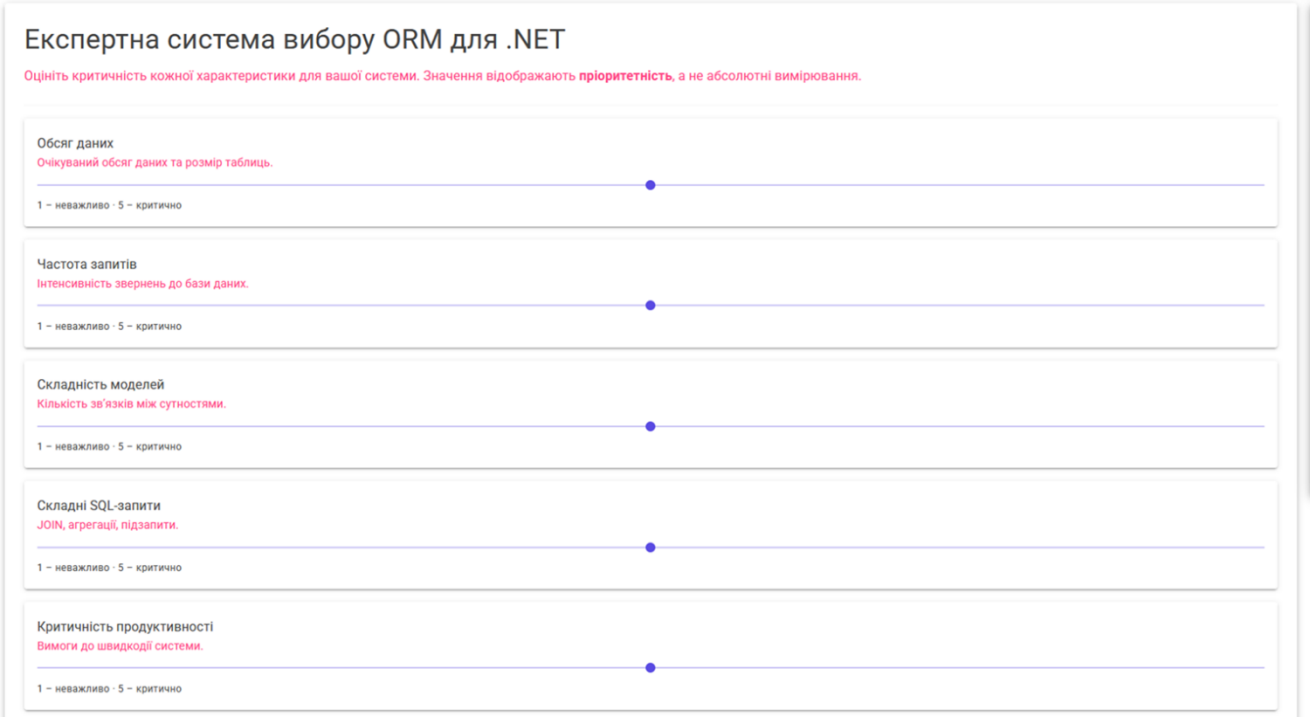


Рис. 3.4 Приклад введення параметрів експертної оцінки)

Динамічний перерахунок результатів

Однією з ключових особливостей розробленої системи є автоматичний перерахунок результатів при зміні будь-якого параметра. Завдяки використанню технології Blazor Server обчислення виконуються на сервері, а оновлення інтерфейсу відбувається миттєво без перезавантаження сторінки.

Це дозволяє користувачу експериментувати з різними значеннями параметрів і одразу бачити, як змінюється рекомендований ORM-фреймворк та його інтегральна оцінка. Такий підхід сприяє кращому розумінню впливу окремих характеристик на процес вибору технології доступу до даних.

Відображення результатів експертної оцінки

Результати роботи інформаційної системи відображаються в окремій області інтерфейсу, розташованій праворуч від блоку введення параметрів. Це дозволяє компактно розмістити всі елементи на одній сторінці та зменшити необхідність прокручування.

Користувачу відображаються такі результати (рис. 3.5):

- назва рекомендованого ORM-фреймворку;
- текстове пояснення вибору;
- найбільш значущий параметр, який мав вирішальний вплив на результат;
- стовпчикова діаграма інтегральних оцінок усіх ORM-фреймворків.

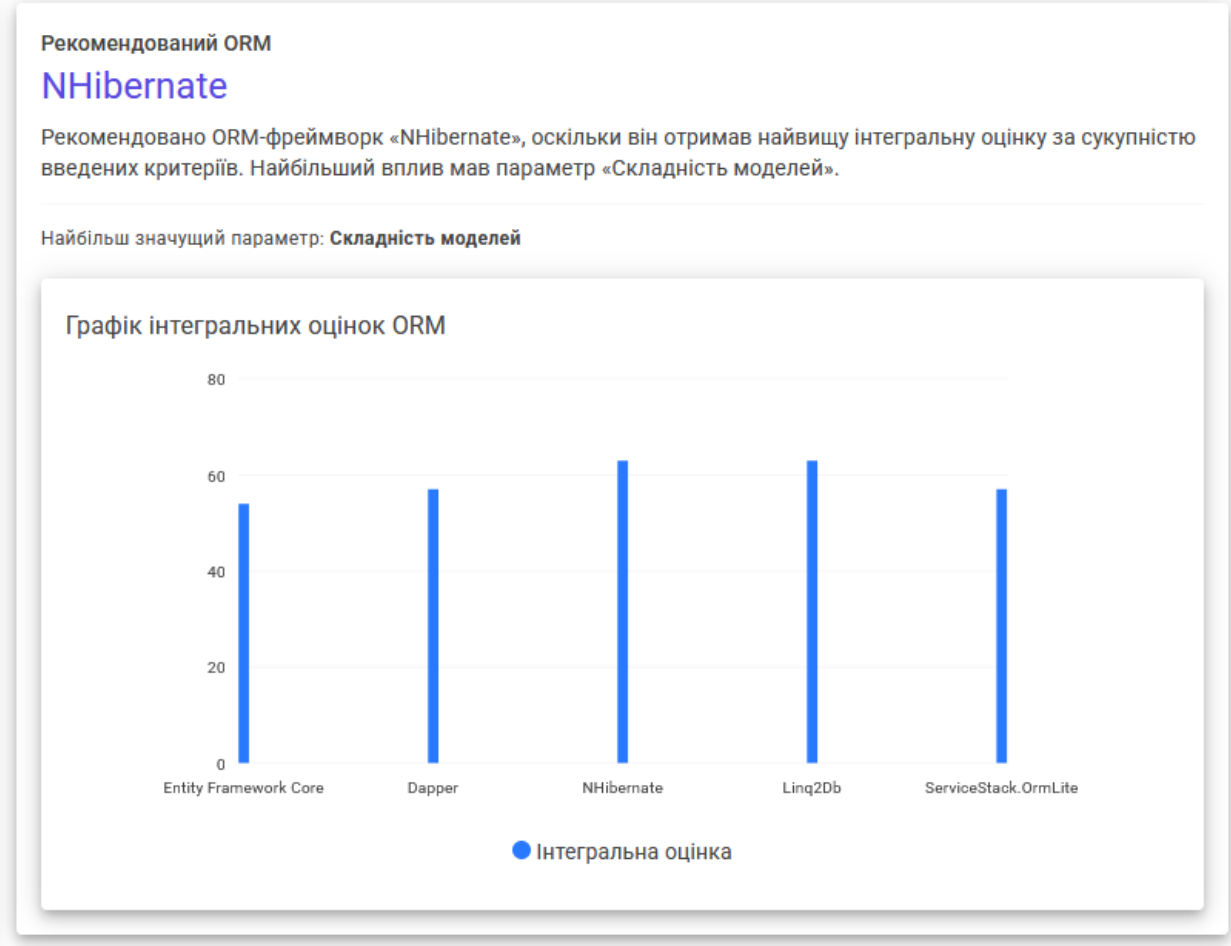


Рис. 3.5 Приклад відображення результатів експертної оцінки

Візуалізація результатів за допомогою діаграм

Для підвищення наочності результатів у системі використано стовпчикову діаграму. Кожен стовпчик відповідає окремому ORM-фреймворку, а його висота відображає інтегральну оцінку.

Використання графічного подання дозволяє:

- швидко порівняти альтернативні ORM між собою;
- оцінити, наскільки суттєво один фреймворк переважає інші;

- наочно продемонструвати вплив змін параметрів на результати.

Графік оновлюється синхронно зі зміною вхідних даних, що забезпечує інтерактивний характер системи.

3.6 Архітектура та логіка роботи системи

Розроблена інформаційна система реалізована у вигляді односторінкового вебзастосунку, що дозволяє користувачу взаємодіяти з системою без повного перезавантаження сторінки. Такий підхід позитивно впливає на швидкодію, зручність використання та загальний користувацький досвід.

В основі архітектури системи лежить клієнт-серверна модель. Клієнтська частина відповідає за візуалізацію даних, обробку введених користувачем параметрів та миттєве оновлення результатів оцінювання. Серверна логіка, у свою чергу, може використовуватись для збереження конфігурацій, розширених розрахунків або масштабування системи в майбутньому.

Основними компонентами архітектури є:

- *інтерфейс користувача* (UI), реалізований за допомогою компонентного підходу;
- *модель вхідних параметрів*, яка зберігає поточні значення характеристик системи;
- *модуль обчислення інтегральної оцінки*, що виконує математичну обробку даних;
- *модуль візуалізації результатів*, який відображає підсумкові значення у вигляді графіків та діаграм.

Взаємодія між цими компонентами організована таким чином, що будь-яка зміна параметра одразу ініціює повторний розрахунок та оновлення візуальних елементів без додаткових дій з боку користувача.

Модель вхідних параметрів та їх призначення

Для оцінювання характеристик програмної або інформаційної системи користувачу пропонується набір параметрів, кожен з яких описує окремий аспект

її функціонування. Значення параметрів задаються за допомогою інтерактивних повзунків (slider), що дозволяє інтуїтивно зрозуміло керувати процесом налаштування.

До основних параметрів належать:

- *Обсяг даних.* Характеризує очікуваний обсяг інформації, що зберігається в системі, а також розміри таблиць у базі даних. Високі значення цього параметра вказують на необхідність оптимізації зберігання та доступу до даних.
- *Частота запитів.* Визначає інтенсивність звернень до бази даних або сервісів. Цей параметр напряму впливає на вимоги до продуктивності та масштабованості системи.
- *Складність моделей даних.* Описує кількість зв'язків між сутностями, рівень нормалізації та загальну складність структури даних.
- *Складні SQL-запити.* Враховує використання JOIN-операцій, агрегацій, підзапитів та інших ресурсоємних механізмів обробки даних.
- *Критичність продуктивності.* Відображає вимоги до швидкодії системи, допустимі затримки та реакцію на навантаження.
- *Підтримуваність.* Характеризує зручність супроводу, читабельність коду та простоту внесення змін у майбутньому.

Кожен з перелічених параметрів має числове значення в заданому діапазоні, що дозволяє уніфікувати подальші обчислення.

Механізм розрахунку інтегральної оцінки

Інтегральна оцінка є узагальненим показником, який дозволяє кількісно оцінити загальний рівень складності та вимог до системи на основі набору вхідних параметрів. Її використання спрощує аналіз та дозволяє порівнювати різні конфігурації між собою.

Процес розрахунку інтегральної оцінки включає такі етапи:

1. зчитування поточних значень усіх параметрів;
2. нормалізація значень до єдиного масштабу;
3. застосування вагових коефіцієнтів (за потреби);
4. обчислення підсумкового значення за заданою формулою;

5. відображення результату у числовій та графічній формі.

Важливою особливістю системи є те, що розрахунок виконується автоматично в режимі реального часу. При зміні будь-якого параметра користувач миттєво бачить оновлений результат, що значно підвищує наочність та зручність аналізу.

3.7 Реалізація роботи з системою

Розроблена система орієнтована на кінцевого користувача, який не потребує спеціальних технічних знань у галузі програмування або адміністрування баз даних. Інтерфейс системи є інтуїтивно зрозумілим та не потребує попереднього навчання.

Після відкриття веб-застосунку користувач одразу отримує доступ до основного екрану, на якому розміщені всі елементи керування та відображення результатів.

Головний екран системи

Головний екран містить такі основні області:

- панель введення параметрів;
- блок з інтегральною оцінкою;
- область візуалізації результатів у вигляді діаграми.

У верхній частині сторінки розташована заголовна інформація, яка коротко описує призначення системи. Нижче розміщені елементи керування у вигляді повзунків для задавання значень параметрів.

Налаштування вхідних параметрів

Для зміни характеристик системи користувач використовує повзунки параметрів. Кожен повзунок має:

- назву параметра;
- короткий опис його призначення;
- числове значення, яке змінюється при переміщенні повзунка.

Зміна положення повзунка виконується шляхом перетягування миші або за допомогою клавіатури. Після зміни значення система автоматично запускає процес повторного обчислення інтегральної оцінки.

Користувач може змінювати параметри в будь-якій послідовності, при цьому всі зміни одразу відображаються в результатах.

Автоматичний перерахунок результатів

Однією з ключових особливостей системи є відсутність необхідності натискання кнопок підтвердження або запуску обчислень. Перерахунок виконується автоматично після кожної зміни параметра.

Такий підхід дозволяє:

- швидко аналізувати вплив окремих параметрів;
- порівнювати різні сценарії налаштувань;
- зменшити кількість дій користувача.

У результаті користувач отримує інтерактивний інструмент для дослідження залежностей між параметрами та підсумковою оцінкою.

Відображення інтегральної оцінки

Інтегральна оцінка відображається у вигляді числового показника, який оновлюється в реальному часі. Значення оцінки дозволяє користувачу швидко оцінити загальний рівень складності або вимог до системи.

Окрім числового значення, система може додатково відображати:

- кольорову індикацію рівня (низький, середній, високий);
- графічне представлення у вигляді стовпчикової або лінійної діаграми.

Візуалізація результатів аналізу

Для підвищення наочності отриманих результатів у системі використовується графічна візуалізація даних. Основним інструментом візуалізації є діаграма, яка відображає значення окремих параметрів або підсумкові показники.

Використання діаграм дозволяє:

- краще сприймати числові дані;
- швидко виявляти параметри з найбільшим впливом;

- порівнювати поточні та попередні значення.

Графік оновлюється синхронно зі зміною вхідних параметрів, що забезпечує цілісність та актуальність відображеної інформації.

3.8 Візуальне моделювання інформаційної системи

Для детального опису структури та логіки роботи інформаційної системи були побудовані UML-діаграми, які дозволяють візуально оцінити взаємодію компонентів та послідовність операцій.

Діаграма варіантів використання (Use Case)

Діаграма Use Case демонструє основні сценарії використання системи кінцевим користувачем (рис. 3.6). Основні дії включають:

- *Встановлення параметрів* – користувач задає значення характеристик системи.
- *Перегляд рекомендації* – система обчислює інтегральну оцінку та пропонує оптимальний ORM.
- *Візуалізація результатів* – система відображає графіки та пояснює найбільш значущий параметр.
- *Аналіз сценаріїв* – користувач може змінювати параметри та оцінювати вплив на результат.

Рис. 3.6 UML Use Case діаграма

Використання діаграми варіантів застосування дозволяє наочно продемонструвати функціональні можливості системи та взаємодію користувача з ключовими модулями.

Діаграма послідовностей (Sequence Diagram)

Діаграма послідовностей відображає поточний сценарій взаємодії користувача та системи під час вибору ORM-фреймворку (рис. 3.7):

1. Користувач задає параметри через UI.

2. Система отримує оновлені значення та передає їх у сервіс оцінки.

3. Сервіс обчислює інтегральні оцінки для всіх ORM-фреймворків.

4. Визначається ORM з максимальною інтегральною оцінкою та найбільш значущий параметр.

5. Результати повертаються до UI.

6. UI оновлює числові показники та графічні елементи у реальному часі.

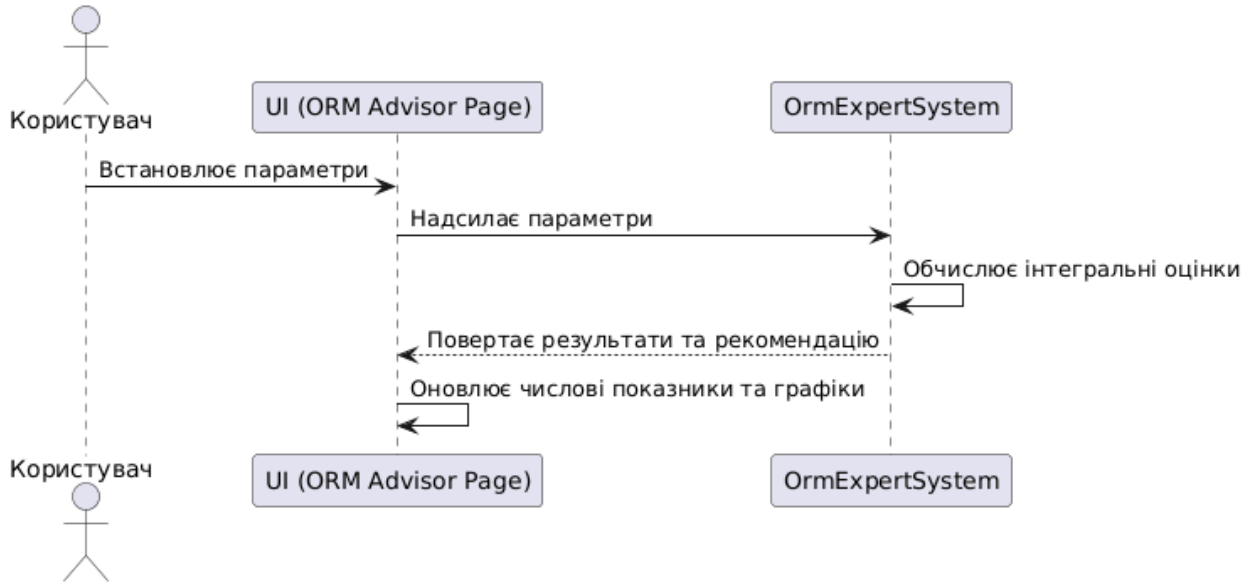


Рис. 3.7 UML Sequence діаграма)

Діаграма дозволяє оцінити послідовність викликів та взаємодію між компонентами системи, що є важливим для розуміння логіки роботи експертної системи.

Діаграма класів (Class Diagram)

Діаграма класів показує структуру основних об’єктів системи та їхні взаємозв’язки (рис. 3.8). Основні класи включають:

- *OrmScenarioInput* – зберігає вхідні параметри користувача.
- *OrmExpertSystem* – сервіс, який здійснює обчислення інтегральної оцінки та визначає оптимальний ORM.
- *OrmRecommendationResult* – містить результати оцінки, пояснення та інформацію про найбільш значущий параметр.
- *OrmEvaluationDetails* – деталізація оцінки для кожного ORM, включаючи внесок окремих параметрів.

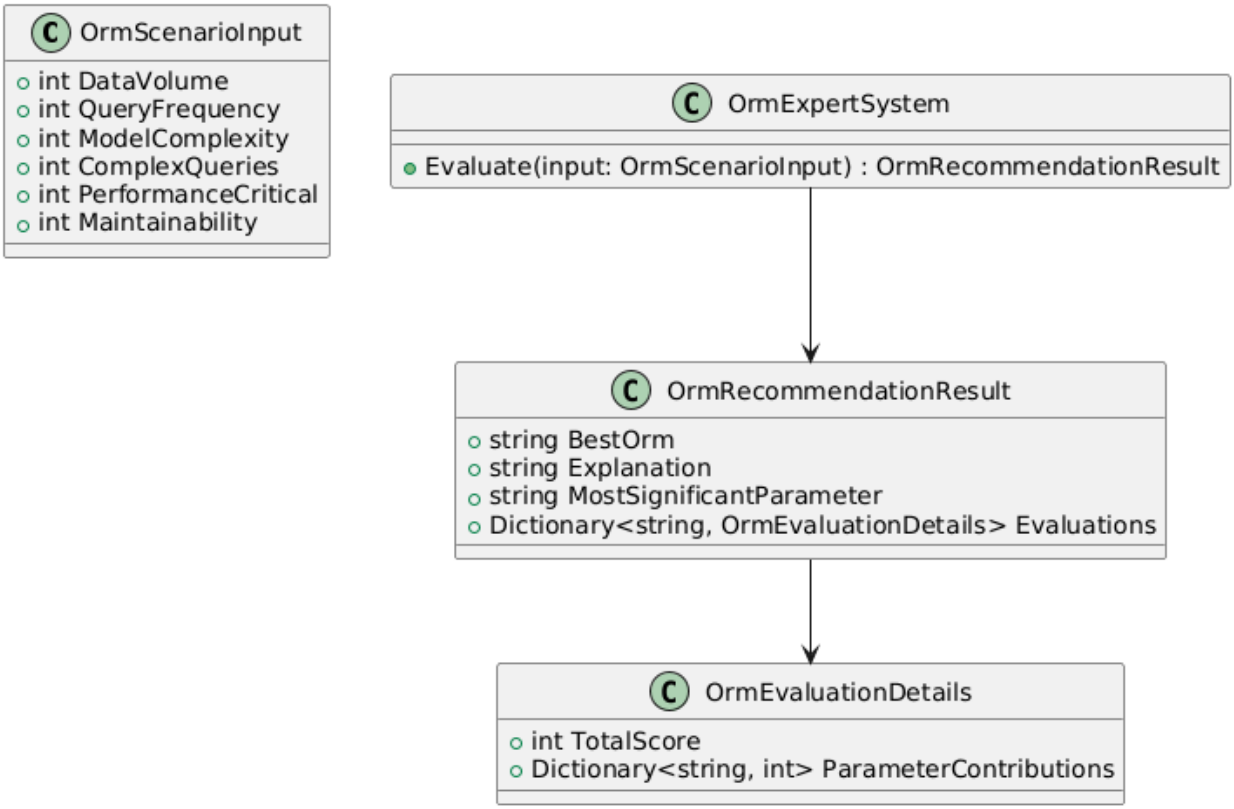


Рис. 3.8 Діаграма класів системи

Ця діаграма дозволяє систематизувати інформацію про внутрішні об’єкти системи, їхні атрибути та методи, що є важливим для розробки та подальшого супроводу.

3.9 Загальні рекомендації по роботі з інформаційною системою

1. *Для невеликих проектів або прототипів*

- Варто віддавати перевагу легким та швидким ORM, які забезпечують простоту розробки та підтримки.
- Рекомендується оцінювати параметри «Швидкість розробки» та «Підтримуваність» як пріоритетні.

2. *Для великих та високонавантажених систем*

- Основну увагу слід приділяти параметрам «Обсяг даних», «Частота запитів» та «Критичність продуктивності».

- Рекомендовано вибирати ORM із високою продуктивністю та можливістю оптимізації складних запитів.

3. *Для складних об'єктних моделей*

- Параметр «Складність моделей» має критичне значення.
- Слід обирати фреймворки, які дозволяють гнучко налаштовувати мапінг та підтримувати складні зв'язки між сутностями.

4. *Загальні поради*

- Перед остаточним вибором рекомендується протестувати кілька ORM у реальному проєкті з типовими даними.
- Використовувати систему як експертний інструмент для прийняття обґрунтованого рішення, а не як єдине джерело істини.

3.10 Висновок до третього розділу

Розроблена інформаційна система для вибору ORM-фреймворків у середовищі .NET дозволяє користувачу швидко та обґрунтовано визначати оптимальний інструмент для конкретного проєкту. Під час роботи системи були реалізовані ключові принципи:

1. Інтерактивність та динамічність

- Система миттєво реагує на зміну вхідних параметрів, автоматично перераховує інтегральні оцінки та оновлює візуалізацію результатів.
- Такий підхід дозволяє користувачу проводити «що-якщо» аналіз та порівнювати різні сценарії роботи системи.

2. Зручний інтерфейс користувача

- Використання сучасних компонентів UI (MudBlazor) забезпечує інтуїтивно зрозуміле управління параметрами.
- Відображення пояснень до кожного параметра та інтегральних оцінок допомагає користувачу розуміти логіку рекомендацій.

3. Науково обґрунтований підхід

- Розрахунок інтегральної оцінки ґрунтується на зваженому врахуванні всіх параметрів системи.
- Найбільш значущі параметри виділяються окремо, що дозволяє аргументувати вибір ORM для конкретного проєкту.

4. Візуалізація даних

- Вбудовані графіки та діаграми забезпечують наочне представлення результатів оцінки.
- Це дозволяє швидко оцінювати вплив окремих характеристик системи на фінальний вибір.

Розроблена система:

- забезпечує швидке та обґрунтоване визначення оптимального ORM-фреймворку;
- допомагає користувачу оцінювати важливість окремих параметрів;
- є наочним і інтерактивним інструментом для прийняття рішень у проєктуванні програмних систем на платформі .NET.

Таким чином, система може бути використана як у навчальних цілях, так і у реальних проєктах для оптимізації процесу вибору ORM та підвищення ефективності розробки.

ВИСНОВКИ

У ході виконання роботи було всебічно досліджено теоретичні та практичні аспекти використання ORM-технологій у сучасних інформаційних системах. Актуальність розробленої системи зумовлена зростаючою складністю програмних продуктів, обсягами даних і вимогами до продуктивності, масштабованості та підтримуваності програмного забезпечення.

У роботі було проаналізовано ключові показники продуктивності ORM-фреймворків, зокрема час виконання CRUD-операцій, вплив стратегій завантаження даних, а також здатність системи ефективно працювати з великими об’ємами інформації. Окрему увагу приділено функціональним можливостям ORM, таким як підтримка складних SQL-запитів, механізми міграції, кешування та асинхронна обробка запитів, що безпосередньо впливають на продуктивність і зручність розробки. Також у межах дослідження розглянуто зручність використання та розширюваність ORM-фреймворків.

Запропонована інформаційна система дозволяє не лише порівнювати ORM-рішення за формальними критеріями, а й підтримувати обґрунтоване прийняття технічних рішень на етапі проєктування програмного забезпечення.

У результаті виконання роботи сформовано цілісне уявлення про роль ORM-фреймворків у сучасних .NET-орієнтованих системах, визначено їх сильні та слабкі сторони з погляду продуктивності, функціональності та безпеки. Отримані результати можуть бути використані під час розробки інформаційних систем різного масштабу, а також слугувати основою для подальших досліджень у напрямі оптимізації доступу до даних та застосування інтелектуальних методів аналізу продуктивності ORM-технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Фаулер М. *Patterns of Enterprise Application Architecture* / M. Fowler. — Boston : Addison-Wesley, 2002. — 560 с.

2. Амблер С. *Mapping Objects to Relational Databases* / S. Ambler. — New York : John Wiley & Sons, 2002. — 325 с.

3. *Entity Framework Core* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/ru-ru/ef/core/. – Дата звернення: 19.12.2025.

4. *NHibernate. The object-relational mapper for .NET* [Електронний ресурс] – Режим доступу: https://nhibernate.info/. – Дата звернення: 19.12.2025.

5. *LINQ to DB* [Електронний ресурс] – Режим доступу: https://github.com/linq2db/linq2db. – Дата звернення: 19.12.2025.

6. *DapperLib* [Електронний ресурс] – Режим доступу: https://github.com/DapperLib. – Дата звернення: 19.12.2025.

7. *Servicestack. Documentation* [Електронний ресурс] – Режим доступу: https://docs.servicestack.net/ormlite/#getting-started. – Дата звернення: 19.12.2025.

8. *ORM performance comparator measure the speed* [Електронний ресурс] – Режим доступу: https://www.daobab.io/orm-performance-comparator.html?utm_source=chatgpt.com. – Дата звернення: 19.12.2025.

9. *Тестування продуктивності Python ORM методом, заснованим на бенчмарку TPC-C* [Електронний ресурс] – Режим доступу: https://habr.com/ru/articles/496116/. – Дата звернення: 19.12.2025.

10. Krapuhina N.V., Kurnikov P.A., Tarkhanov I.A. Multi-criteria method of estimating performance ORM components in information systems / [Proceedings of the Institute of Systems Analysis]. — 2015. — 65(22) : 105–109.

11. *Порівняння продуктивності звичайного SQL, ORM та GraphQL у Golang у контексті принципів «радикальної простоти»* [Електронний ресурс] – Режим доступу: https://habr.com/ru/companies/cloud4y/articles/707650/. – Дата звернення: 19.12.2025.

12. *HTML Standard (Living Standard). Section «Lazy loading attributes»* [Електронний ресурс] – Режим доступу: https://html.spec.whatwg.org/multipage/urls-and-fetching.html#lazy-loading-attributes – Дата звернення: 19.12.2025.

13. Bauer C., King G., Gregory G. *Java Persistence with Hibernate*. — Manning Publications, 2016.

14. Амблер С. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. — Wiley, 2003.

15. Kleppmann M. *Designing Data-Intensive Applications*. — O'Reilly Media, 2017.

16. Date C. J. *An Introduction to Database Systems*. — Addison-Wesley, 2003.

17. *Migrations Overview* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/ru-ru/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli. – Дата звернення: 19.12.2025.

18. *Asynchronous programming with async and await* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/. – Дата звернення: 19.12.2025.

19. Spinellis D. *Code Quality: The Open Source Perspective*. — Addison-Wesley, 2006.

20. *.NET dependency injection* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. – Дата звернення: 19.12.2025.

21. *Data Security. Stop SQL Injection Attacks Before They Stop You* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/archive/msdn-magazine/2004/september/data-security-stop-sql-injection-attacks-before-they-stop-you. – Дата звернення: 19.12.2025.

22. Han J., Kamber M., Pei J. *Data Mining: Concepts and Techniques*. — Morgan Kaufmann, 2012.

23. *Що таке Visual Studio?* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=visualstudio. – Дата звернення: 19.12.2025.

24. *Stack Overflow Survey 2025* [Електронний ресурс] – Режим доступу: https://survey.stackoverflow.co/2025/ – Дата звернення: 19.12.2025.

25. *Blazor Server: Der erste Blazor-Streich* [Електронний ресурс] – Режим доступу: https://entwickler.de/dotnet/der-erste-blazor-streich-001. – Дата звернення: 19.12.2025.

26. *Архітектура SQL Server* [Електронний ресурс] – Режим доступу: https://coderlessons.com/tutorials/bazy-dannykh/uchebnik-sql-server/3-arkhitektura-sql-server. – Дата звернення: 19.12.2025.

27. *LINQ та ADO.NET* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/framework/data/adonet/linq-and-ado-net. – Дата звернення: 19.12.2025.

28. *Огляд Entity Framework* [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/framework/data/adonet/ef/overview. – Дата звернення: 19.12.2025.

29. *Чудові приклади Chart.js, які можна використовувати на своєму веб-сайті* [Електронний ресурс] – Режим доступу: https://wordpress.mediadoma.com/uk/chudovi-prikladi-chart-js-jaki-mozhna-vikoristovuvati-na-svoiemu-veb-sajti/. – Дата звернення: 19.12.2025.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Інформаційна система порівняння ORM-фреймворків на основі показників продуктивності виконання операцій»

Виконав: студент групи ПДМ-62 Владислав СМІРНОВ

Керівник: канд. техн. наук, професор кафедри ІТ Максим КУКЛІНСЬКИЙ

Київ - 2026

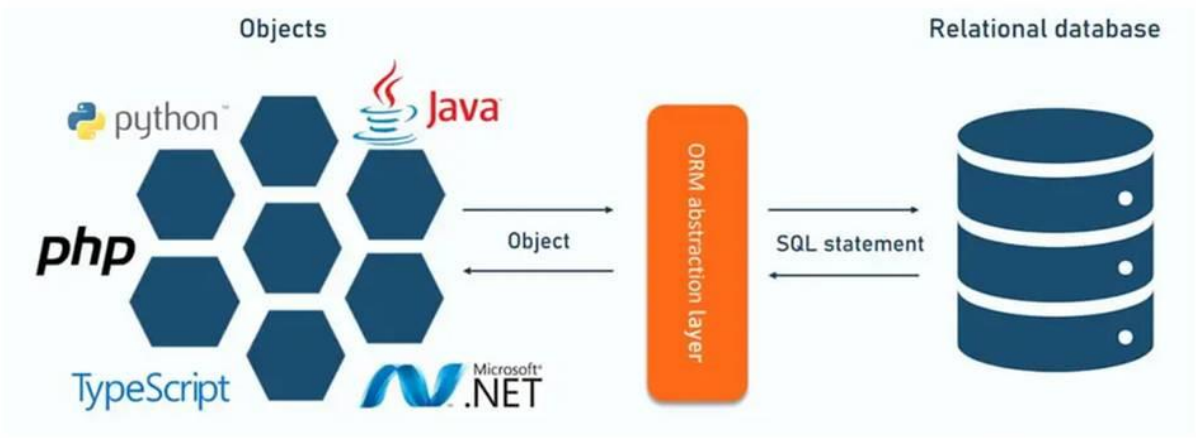
Мета, Об’єкт та Предмет дослідження

Мета роботи: розробка інформаційної системи, що дозволяє аналізувати та порівнювати ефективність використання ORM-фреймворків та підтримувати процес прийняття рішень при виборі ORM-технологій.

Об’єкт дослідження: процес порівняння показників ORM-фреймворків.

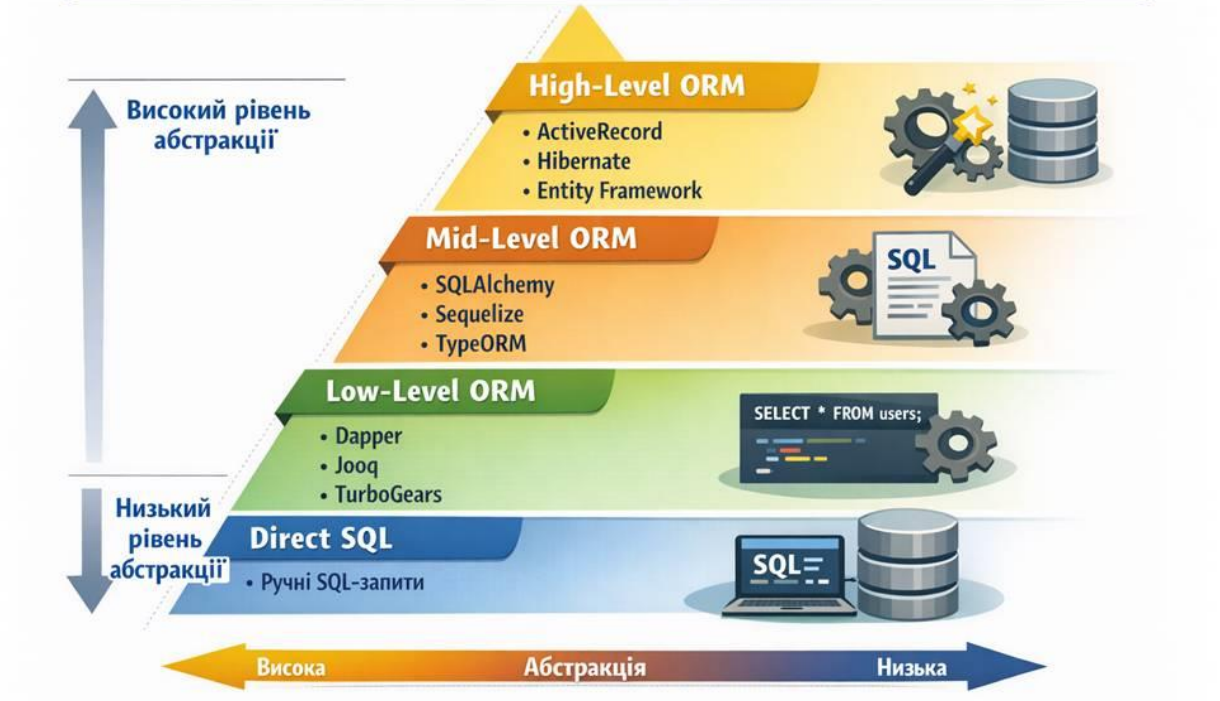
Предмет дослідження: показники оцінювання продуктивності виконання операцій ORM-фреймворками.

ORM як проміжний шар доступу до даних



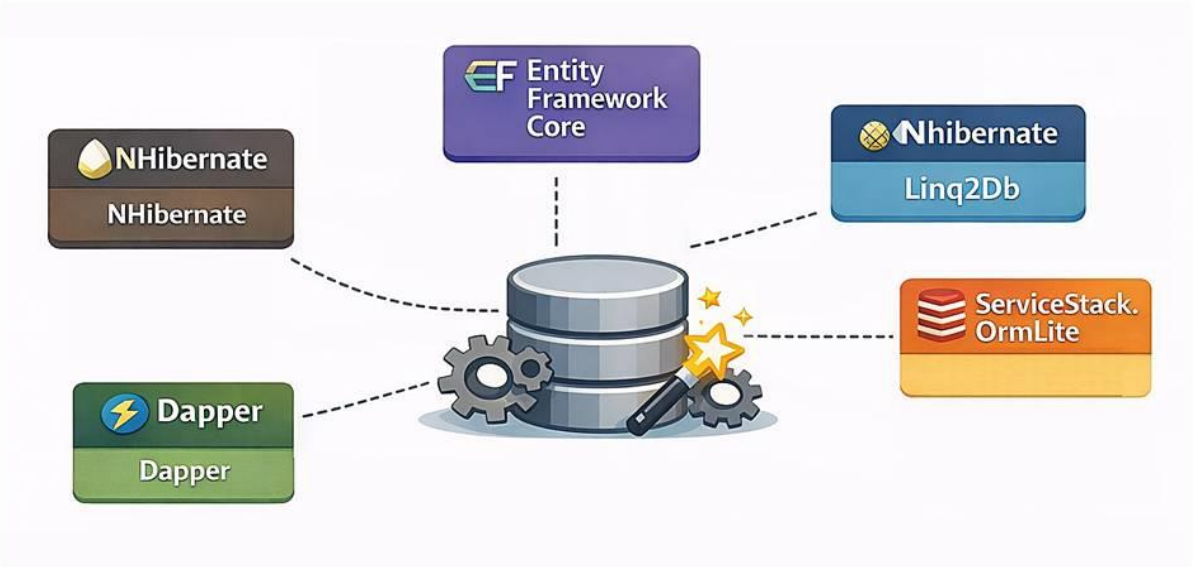
3

Класифікація ORM-фреймворків за рівнем абстракції



4

ORM-фреймворки .NET-екосистеми



5

Основні ORM-операції

6

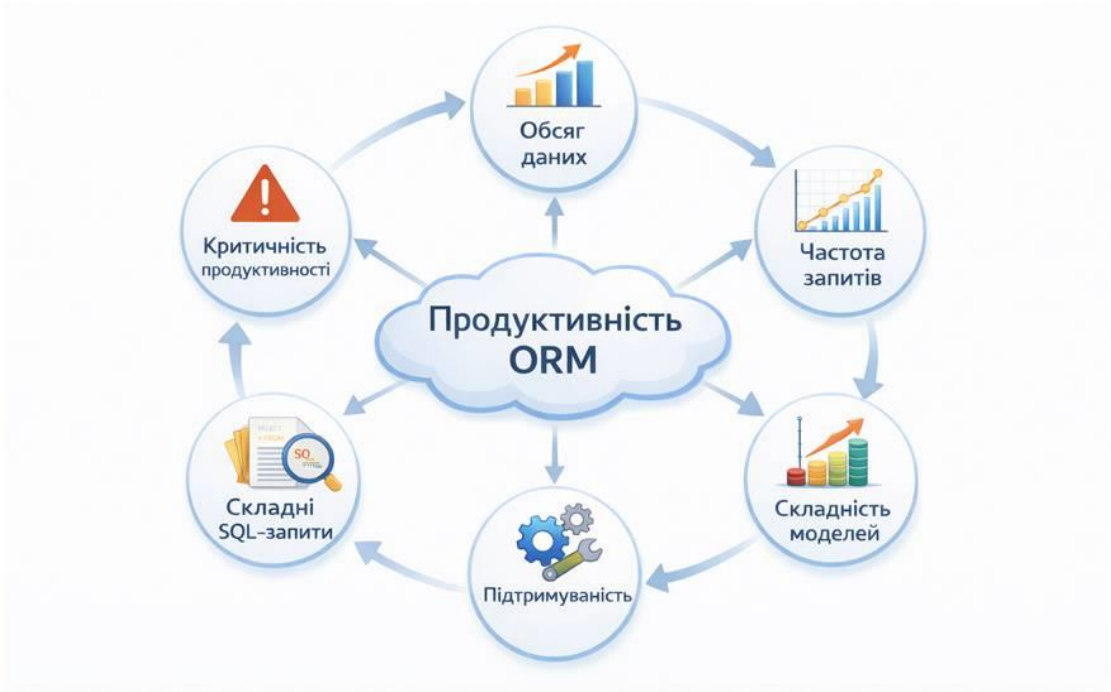
Стратегії завантаження даних: Lazy та Eager Loading

7

Постановка задачі

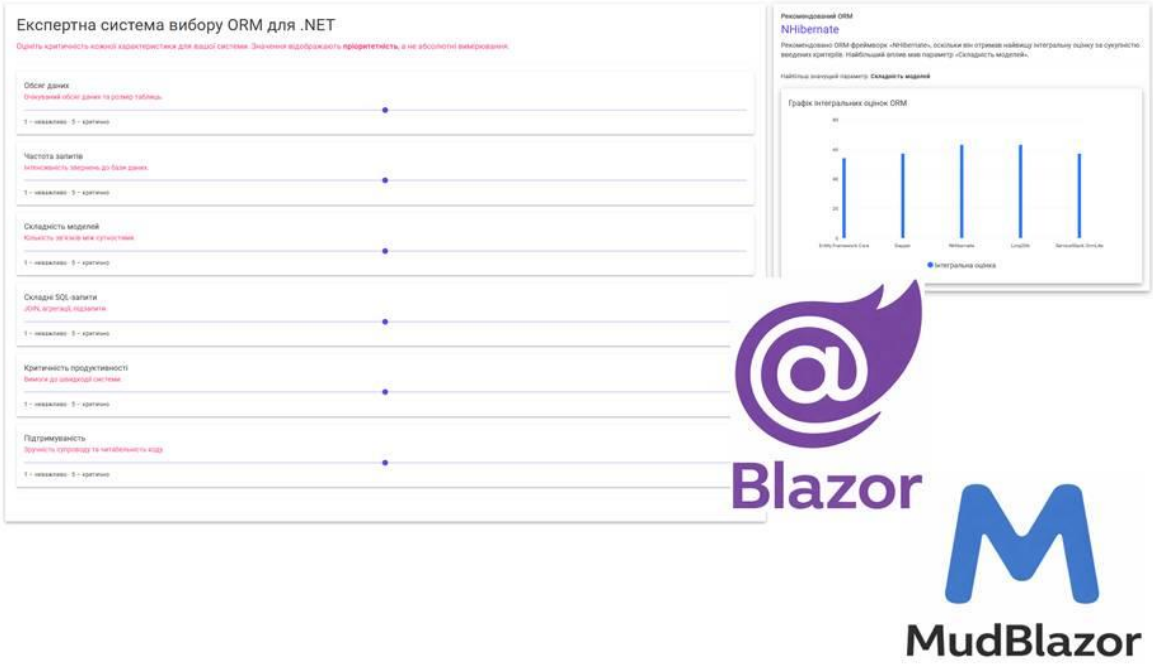
8

Критерії оцінювання ORM-фреймворків



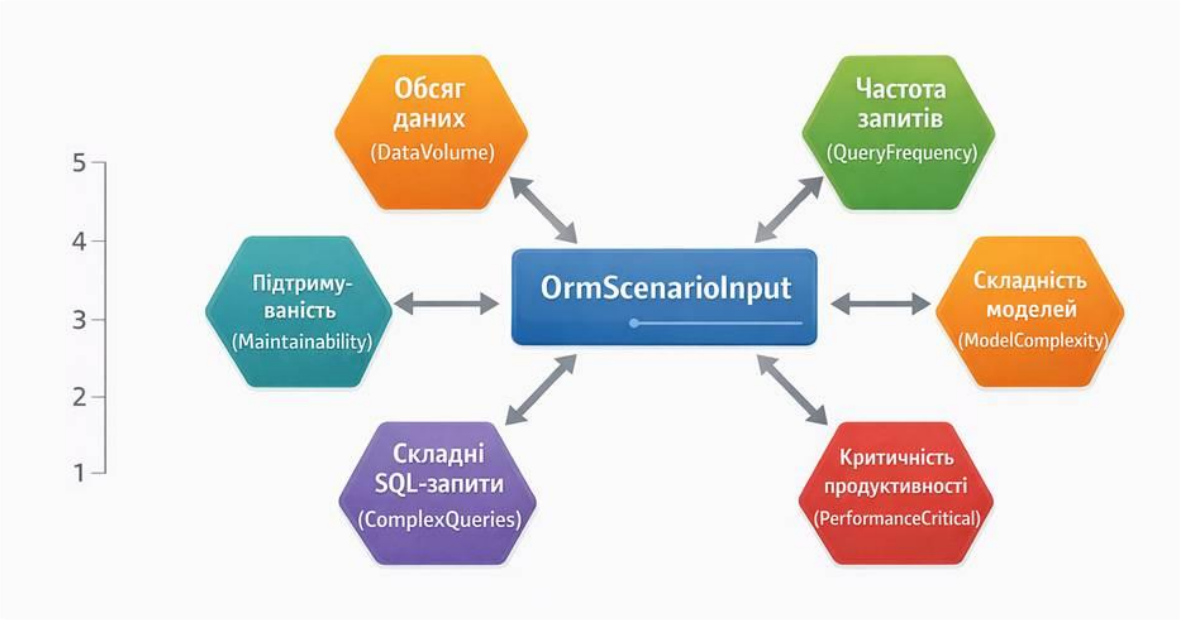
9

Інтерфейс користувача системи



10

Вхідні параметри сценарію



11

Алгоритм рекомендації

OrmExpertSystem.Evaluate

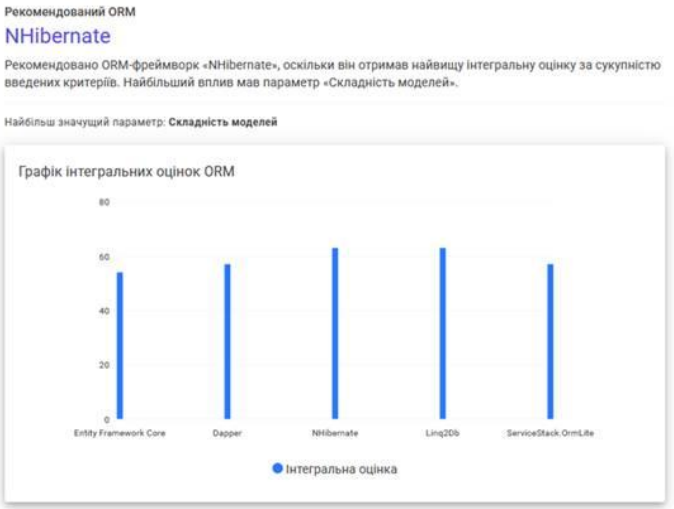
$$S = \sum_{i=1}^{n} w_i \cdot p_i$$

S — інтегральна оцінка ORM-фреймворку;

w_i — експертна вага i-го параметра для конкретного ORM;

p_i — значення параметра, задане користувачем;

n — кількість параметрів оцінювання.



12

ВИСНОВКИ

1. **Проаналізовано сучасні ORM-фреймворки платформи .NET та підходи до оцінювання їх продуктивності.**
   Встановлено, що більшість існуючих порівнянь ORM ґрунтуються на спрощених бенчмарках і не враховують специфіку реальних сценаріїв використання, таких як складність об'єктної моделі, інтенсивність запитів та вимоги до підтримуваності коду.
2. **Проаналізовано існуючі методи та інструменти порівняння ORM-фреймворків.**
   Визначено, що наявні підходи здебільшого орієнтовані на окремі показники (наприклад, швидкодію CRUD-операцій) і не забезпечують комплексної оцінки, необхідної для обґрунтованого вибору ORM у практичних .NET-застосунках.
3. **Визначено критерії оцінювання ORM-фреймворків та фактори, що впливають на їх ефективність.**
   До таких критеріїв віднесено продуктивність, складність моделей, характер SQL-запитів, інтенсивність звернень до бази даних та вимоги до підтримуваності.
   Розроблено метод інтегральної оцінки ORM, що дозволяє враховувати пріоритетність кожного критерію залежно від сценарію використання системи.
4. **Розроблено інформаційну систему вибору ORM-фреймворку для платформи .NET.**
   Система реалізує автоматизований розрахунок інтегральних оцінок на основі заданих користувачем параметрів та забезпечує формування рекомендацій із поясненням прийнятого рішення.
   Використання експертної системи дозволяє зменшити суб'єктивність вибору ORM, підвищити обґрунтованість рішень та спростити процес проєктування інформаційних систем.

13

ДЯКУЮ ЗА УВАГУ!

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

1. Program.cs

```
using Cursova.Configuration;
using Cursova.Services;
using Microsoft.AspNetCore.Components;
using
Microsoft.AspNetCore.Components.Web;
using MudBlazor.Services;

var builder =
WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddServerSideBlazor();

builder.Services.AddMudServices();

builder.Services.Configure<OrmExpertSettings
>(

builder.Configuration.GetSection("OrmExpertS
ettings"));

builder.Services.AddScoped<OrmExpertSystem
>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
   app.UseExceptionHandler("/Error");
   app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.MapBlazorHub();
app.MapFallbackToPage("/_Host");

app.Run();
```

2. OrmExpertSettings.cs

```
public class OrmExpertSettings
{
  public List<OrmFrameworkConfig> Frameworks {
  get; set; } = new();
}
```

3. OrmFrameworkConfig.cs

```
public class OrmFrameworkConfig
{
   public string Name { get; set; } = string.Empty;
   public Dictionary<string, int> Scores { get; set; } =
new();
}
```

4. OrmScenarioInput.cs

```
public class OrmScenarioInput
{
   public int DataVolume { get; set; } = 3;
   public int QueryFrequency { get; set; } = 3;
   public int ModelComplexity { get; set; } = 3;
   public int ComplexQueries { get; set; } = 3;
   public int PerformanceCritical { get; set; } = 3;
   public int Maintainability { get; set; } = 3;
}
```

5. OrmEvaluationDetails.cs

```
public class OrmEvaluationDetails
{
   public int TotalScore { get; set; }
   public Dictionary<string, int>
ParameterContributions { get; set; } = new();
}
```

6. OrmRecommendationResult.cs

```
public class OrmRecommendationResult
{
   public string BestOrm { get; set; } = string.Empty;
   public Dictionary<string, OrmEvaluationDetails>
Evaluations { get; set; } = new();
   public string Explanation { get; set; } =
string.Empty;
   public string MostSignificantParameter { get; set;
} = string.Empty;
}
```

7. OrmExpertSystem.cs

```
public class OrmExpertSystem
{
  private readonly OrmExpertSettings _settings;

  public
OrmExpertSystem(IOptions<OrmExpertSettings>
options)
  {
    _settings = options.Value;
  }

  public OrmRecommendationResult
Evaluate(OrmScenarioInput input)
  {
    var result = new
OrmRecommendationResult();

    foreach (var orm in _settings.Frameworks)
    {
      var details = new OrmEvaluationDetails();

      void Add(string param, int weight, int value)
      {
        var contribution = weight * value;
        details.ParameterContributions[param] =
contribution;
        details.TotalScore += contribution;
      }

      Add("DataVolume",
orm.Scores["DataVolume"], input.DataVolume);
      Add("QueryFrequency",
orm.Scores["QueryFrequency"],
input.QueryFrequency);
      Add("ModelComplexity",
orm.Scores["ModelComplexity"],
input.ModelComplexity);
      Add("ComplexQueries",
orm.Scores["ComplexQueries"],
input.ComplexQueries);
      Add("PerformanceCritical",
orm.Scores["PerformanceCritical"],
input.PerformanceCritical);
      Add("Maintainability",
orm.Scores["Maintainability"],
input.Maintainability);

      result.Evaluations[orm.Name] = details;
    }

    result.BestOrm = result.Evaluations
      .OrderByDescending(x =>
x.Value.TotalScore)
      .First().Key;

    var bestDetails =
result.Evaluations[result.BestOrm];
    var mostSignificant =
bestDetails.ParameterContributions
      .OrderByDescending(x => x.Value)
      .First();

    result.MostSignificantParameter =
mostSignificant.Key;

    result.Explanation =
      $"Рекомендовано ORM-фреймворк
«{result.BestOrm}», оскільки він отримав " +
      $"найвищу інтегральну оцінку за
сукупністю введених критеріїв. " +
      $"Найбільший вплив мав параметр
«{ParameterDisplayNames.Get(mostSignificant.Key
)}».";

    return result;
  }
}
```

8. MainLayout.razor

```
@inherits LayoutComponentBase

<MudThemeProvider />
<MudPopoverProvider />

<MudDialogProvider />
<MudSnackbarProvider />

<PageTitle>Cursova</PageTitle>

<div class="page">
  <main>
    <article class="content px-4">
      @Body
    </article>
  </main>
</div>
```

9. Index.razor

```
@page "/"
@using Cursova.Models
@using Cursova.Services
@inject OrmExpertSystem ExpertSystem

<PageTitle>ORM Advisor</PageTitle>
```

```
<MudContainer MaxWidth="MaxWidth.False" Class="mt-6">

  <MudGrid Spacing="4">

    <!-- ЛЕВАЯ КОЛОНКА -->
    <MudItem xs="12" md="8">

      <MudPaper Elevation="4" Class="pa-6">

        <MudText Typo="Typo.h4" GutterBottom>
          Експертна система вибору ORM для .NET
        </MudText>

        <MudText Typo="Typo.body1" Color="Color.Secondary" Class="mb-6">
          Оцініть критичність кожної характеристики для вашої системи.
          Значення відображають <b>пріоритетність</b>, а не абсолютні вимірювання.
        </MudText>

        <MudDivider Class="mb-4" />

        <MudStack Spacing="3">

          <ParameterSlider Title="Обсяг даних"
                           Description="Очікуваний обсяг даних та розмір таблиць."
                           Value="@Input.DataVolume"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.DataVolume)))" />

          <ParameterSlider Title="Частота запитів"
                           Description="Інтенсивність звернень до бази даних."

Value="@Input.QueryFrequency"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.QueryFrequency)))" />

          <ParameterSlider Title="Складність моделей"
                           Description="Кількість зв’язків між сутностями."

Value="@Input.ModelComplexity"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.ModelComplexity)))" />

          <ParameterSlider Title="Складні SQL-запити"
                           Description="JOIN, агрегації, підзапити."

Value="@Input.ComplexQueries"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.ComplexQueries)))" />

          <ParameterSlider Title="Критичність продуктивності"
                           Description="Вимоги до швидкодії системи."

Value="@Input.PerformanceCritical"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.PerformanceCritical)))" />

          <ParameterSlider Title="Підтримуваність"
                           Description="Зручність супроводу та читабельність коду."

Value="@Input.Maintainability"
                           ValueChanged="@(v => OnInputChanged(v, nameof(Input.Maintainability)))" />

        </MudStack>

        <MudDivider Class="my-6" />

      </MudPaper>

    </MudItem>

    <!-- ПРАВАЯ КОЛОНКА -->
    <MudItem xs="12" md="4">

      @if (Result is not null)
      {
        <MudPaper Elevation="6" Class="pa-4">

          <MudText Typo="Typo.subtitle2">
            Рекомендований ORM
          </MudText>
```

```
            <MudText Typo="Typo.h5"
Color="Color.Primary">
                @Result.BestOrm
            </MudText>

            <MudText Typo="Typo.body2"
Class="mt-2">
                @Result.Explanation
            </MudText>

            <MudDivider Class="my-3" />

            <MudText Typo="Typo.caption">
                Найбільш значущий параметр:

<b>@ParameterDisplayNames.Get(Result.MostSig
nificantParameter)</b>
            </MudText>

            <MudPaper Elevation="6" Class="pa-4
mt-4">
                <MudText Typo="Typo.subtitle1">
                    Графік інтегральних оцінок ORM
                </MudText>

                <MudChart
ChartType="ChartType.Bar"
                        ChartSeries="@Series"
                        XAxisLabels="@XAxisLabels"
                        Width="100%"
                        Height="350px" />
            </MudPaper>

        </MudPaper>
    }

</MudItem>

</MudGrid>

</MudContainer>


@code {
    OrmScenarioInput Input = new();
    OrmRecommendationResult? Result;

    private List<ChartSeries> Series = new ();
    private string[] XAxisLabels =
Array.Empty<string>();

    protected override void OnInitialized()
    {
        UpdateRecommendation();
    }

    void OnInputChanged(int newValue, string
propertyName)
    {
        // обновляем значение в модели
        switch (propertyName)
        {
            case nameof(Input.DataVolume):
Input.DataVolume = newValue; break;
            case nameof(Input.QueryFrequency):
Input.QueryFrequency = newValue; break;
            case nameof(Input.ModelComplexity):
Input.ModelComplexity = newValue; break;
            case nameof(Input.ComplexQueries):
Input.ComplexQueries = newValue; break;
            case nameof(Input.PerformanceCritical):
Input.PerformanceCritical = newValue; break;
            case nameof(Input.Maintainability):
Input.Maintainability = newValue; break;
        }

        // пересчитываем рекомендации и график
        UpdateRecommendation();
    }

    void UpdateRecommendation()
    {
        Result = ExpertSystem.Evaluate(Input);

        XAxisLabels =
Result.Evaluations.Keys.ToArray();

        Series = new List<ChartSeries>
        {
            new ChartSeries
            {
                Name = "Інтегральна оцінка",
                Data = Result.Evaluations.Values.Select(x
=> (double)x.TotalScore).ToArray()
            }
        };

        StateHasChanged();
    }

}
```

10. ParameterSlider.razor

```
@using MudBlazor

<MudPaper Elevation="2" Class="pa-4">

    <MudText Typo="Typo.subtitle1">
        @Title
```

```
    </MudText>

    <MudText Typo="Typo.body2"
Color="Color.Secondary" Class="mb-2">
        @Description
    </MudText>

    <MudSlider Min="1"
               Max="5"
               Step="1"
               ValueLabel="true"
               T="int"
               Value="Value"
ValueChanged="ValueChanged" />

    <MudText Typo="Typo.caption">
        1 – неважливо · 5 – критично
    </MudText>

</MudPaper>

@code {
    [Parameter] public string Title { get; set; } = "";
    [Parameter] public string Description { get; set; }
= "";
    [Parameter] public int Value { get; set; }
    [Parameter] public EventCallback<int>
ValueChanged { get; set; }
}
```