НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
Комп’ютерних наук
____________ _Голуб Б.Л.

“____” __________ 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА
на тему

Програмне забезпечення кластеризації даних про ДТП на дорогах міста Києва

Спеціальність 121 – «Інженерія програмного забезпечення»

Гарант освітньої програми
К.т.н., доцент _______________ Вайганг Г.О.

Керівник бакалаврської кваліфікаційної роботи
________к.т.н., доцент________ _______________ _Вайганг Г.О.
науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав __________________ ______Супрун Д.М.______
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп’ютерних наук
____________ _Голуб Б.Л.

“____” __________ 2025 р.

ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи студенту

Супруну Данилу Михайловичу

Спеціальність 121 «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи: Програмне забезпечення кластеризації даних про ДТП на дорогах міста Києва

затверджена наказом ректора НУБіП України № 2248 “С” від 16.12.2024

Термін подання завершеної роботи на кафедру 2025.06.03
(рік, місяць, число)

Вихідні дані до роботи: опис програмного забезпечення

Перелік питань що розглядаються:
1. Системний аналіз предметної області інформаційної системи
2. Проєктування інформаційного та програмного забезпечення
3. Розробка інформаційного та програмного забезпечення
4. Рекомендації щодо впровадження та експлуатації системи

Дата видачі завдання «_______»_________________2025 р.
Керівник бакалаврської кваліфікаційної роботи

к.т.н., доцент	_______________	Вайганг Г.О.
науковий ступінь та вчене звання)	(підпис)	(ПІБ)

Завдання прийняв до виконання ____________ Супрун Д.М.
(підпис) (ПІБ студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ....5
ВСТУП....6
1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ....8
1.1 Опис предметної області....8
1.2 Аналіз вимог до програмної системи....9
1.3 Моделювання предметної області....12
1.4 Огляд інформаційних джерел та існуючих рішень....22
1.5 Постановка завдання....27
2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ....29
2.1 Логічна модель даних у вигляді ER-діаграми....29
2.2 Діаграма класів та діаграма кооперації....32
2.3 Діаграма пакетів....36
2.4 Діаграма компонентів....38
3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ....40
3.1 Система управління інформаційною базою....40
3.2 Розробка інформаційної бази....42
3.3 Вибір інструментарію для створення прикладного програмного забезпечення....44
3.4 Алгоритмізація та програмування програмних модулів....46
4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ....51
4.1 Тестування системи....51
4.2 Вимоги до апаратного та програмного забезпечення....54

4.3 Склад інсталяційного пакету....................58

ВИСНОВКИ....................60

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.................... 61

ДОДАТОК А....................63

ДОДАТОК Б....................65

ДОДАТОК В.................... 66

ДОДАТОК Г.................... 67

ДОДАТОК Ґ....................68

ДОДАТОК Д.................... 78

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ДТП	Дорожньо-транспортна пригода
СУБД	Система управління базами даних
БД	База даних
API	Application Programming Interface (інтерфейс прикладного програмування)
K-Means	Алгоритм кластеризації на основі методу K-середніх
CSV	Comma-Separated Values (формат текстових файлів із роздільниками)
ПК	Персональний комп’ютер
UI	User Interface (користувацький інтерфейс)
EF Core	Entity Framework Core (ORM фреймворк для .NET)
ORM	Object-Relational Mapping (об’єктно-реляційне відображення)
.NET	Платформа розробки програмного забезпечення від Microsoft
GIS	Geographic Information System (геоінформаційна система)
JSON	JavaScript Object Notation (формат обміну даними)

ВСТУП

Проблема безпеки дорожнього руху є однією з ключових у сучасних мегаполісах. Зі зростанням кількості транспортних засобів, підвищенням інтенсивності руху та ускладненням дорожньої інфраструктури, зростає і кількість дорожньо-транспортних пригод (ДТП). Актуальним завданням для органів місцевого самоврядування, поліції, урбаністів та дослідників стає ефективний аналіз і візуалізація таких інцидентів для виявлення небезпечних ділянок дороги, так званих «чорних точок».

Одним із сучасних підходів до аналізу просторових даних є використання методів кластеризації, що дозволяють виявити приховані структури та просторово-часові скупчення подій. Застосування алгоритму K-Means у поєднанні з геоінформаційними сервісами, такими як Leaflet, дає змогу побудувати інтерактивну аналітичну систему, здатну в режимі реального часу візуалізувати зони підвищеної аварійності. Це, у свою чергу, створює передумови для підвищення ефективності транспортного планування, покращення інфраструктурних рішень і підвищення рівня загальної безпеки на дорогах.

Метою роботи є розробка прикладного програмного забезпечення для кластеризації даних про дорожньо-транспортні пригоди у місті Києві з використанням алгоритму K-Means, подальшим збереженням результатів у базі даних PostgreSQL і візуалізацією їх на інтерактивній мапі. Реалізована система дозволяє автоматизувати процес обробки, аналізу та візуалізації даних про ДТП, забезпечуючи зручний інтерфейс для взаємодії з користувачем, можливість динамічного оновлення інформації, а також базові засоби інформаційної безпеки.

Об’єктом дослідження є інформаційна система обробки, кластеризації та візуалізації даних про дорожньо-транспортні пригоди.

Предметом дослідження є алгоритмічні та програмні засоби кластерного аналізу ДТП з метою ідентифікації небезпечних ділянок у транспортній інфраструктурі міста.

У ході дослідження розроблено інформаційну модель системи, реалізовано кластеризацію даних за допомогою Accord.NET, організовано зберігання даних у реляційній СУБД PostgreSQL, а також створено графічний інтерфейс для інтерактивної роботи з мапою кластерів. Здійснено тестування системи на синтетичних і реальних даних, що підтвердило її ефективність і стабільність функціонування.

Результати дипломного проєкту можуть бути застосовані як основа для впровадження в системи моніторингу безпеки дорожнього руху, використані в дослідницьких цілях або інтегровані у веб-сервіси муніципального рівня.

Структура пояснювальної записки охоплює чотири основні розділи: аналіз предметної області, проєктування програмного та інформаційного забезпечення, реалізацію програмного продукту та оцінку ефективності його впровадження. Робота містить 62 сторінки основного тексту, 23 джерел літератури, 31 рисунок, 15 таблиць та 6 додатків.

1 СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Предметна область даної бакалаврської кваліфікаційної роботи охоплює комплекс задач, пов'язаних з аналізом, структуризацією та обробкою інформації про дорожньо-транспортні пригоди (ДТП) у межах урбанізованого середовища, зокрема на території міста Києва. Актуальність тематики обумовлена високим рівнем аварійності на дорогах мегаполісів, що зумовлює необхідність системного підходу до ідентифікації небезпечних зон шляхом інтеграції геоінформаційного аналізу з інструментами кластерного моделювання [1].

ДТП розглядається не як ізольована подія, а як елемент багатовимірної інформаційної моделі, яка включає просторові (координати), часові (дата, час доби, день тижня), інфраструктурні (тип дороги, обмеження швидкості, наявність світлофорів) та контекстуальні (погодні умови, тип транспортного засобу, рівень тяжкості наслідків) характеристики. Сукупність цих параметрів дозволяє формалізувати події у вигляді структурованих об'єктів у базі даних PostgreSQL, що забезпечує ефективне зберігання, фільтрацію та подальший аналіз [2].

Враховуючи особливості урбаністичної структури Києва, проблема виявлення зон підвищеного ризику виникнення ДТП потребує використання сучасних аналітичних методів. Серед них особливе місце посідає кластеризація – процес групування об'єктів за подібністю ознак. У цьому проєкті реалізовано алгоритм K-Means, що дозволяє виявити просторові скупчення аварій, які мають спільні характеристики. Результати кластеризації відображаються у вигляді інтерактивної мапи, що дозволяє наочно ідентифікувати критичні ділянки дорожньої мережі [3].

Особливу роль у цій системі відіграє інтеграція з геоінформаційними сервісами (зокрема, Leaflet) та використання ORM-технологій, таких як Entity Framework Core, які забезпечують ефективний обмін даними між користувачем

і серверною частиною системи. Це створює підґрунтя для масштабованого застосування програмного продукту у міських службах управління дорожнім рухом, транспортному плануванні, а також у наукових дослідженнях, пов'язаних з безпекою руху [4].

Для візуального відображення структури предметної області було побудовано узагальнену схему взаємодії між основними сутностями системи (рис. 1.1), яка ілюструє ключові об'єкти: ДТП, Користувач, Кластер, Результат кластеризації та їхні взаємозв'язки.

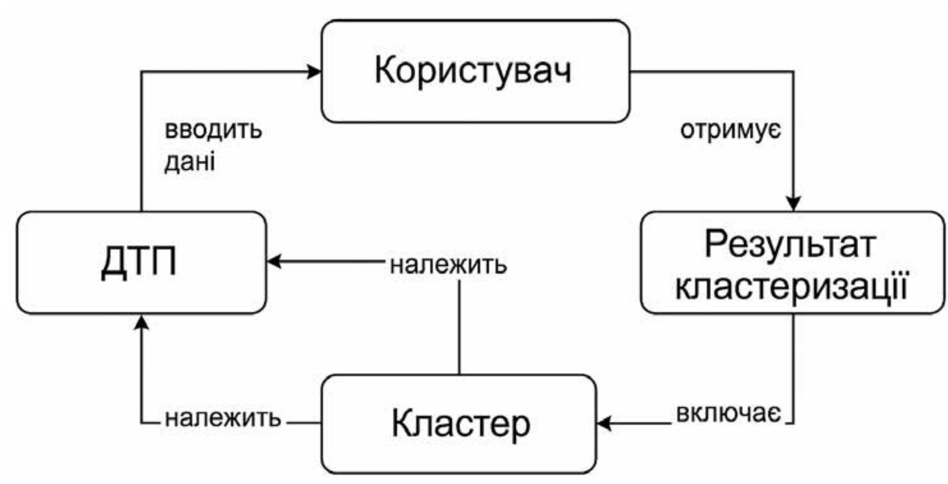


Рис. 1.1 Структура предметної області: взаємозв'язки між ключовими об'єктами системи

Системний підхід до формалізації ДТП у цифровому вигляді з урахуванням просторово-часових та контекстуальних параметрів дозволяє перейти від поверхневого статистичного аналізу до глибокого моделювання закономірностей виникнення аварій. Таким чином, предметна область дослідження є міждисциплінарною і охоплює компоненти геоінформатики, програмної інженерії, аналітики безпеки дорожнього руху та урбаністики.

1.2 Аналіз вимог до програмної системи

Ефективна реалізація програмного забезпечення для кластеризації даних про дорожньо-транспортні пригоди (ДТП) у місті Києві потребує чіткого

визначення вимог, які формують основу технічного завдання. Сформульовані вимоги охоплюють як функціональні, так і нефункціональні аспекти, що забезпечують коректність, стабільність, зручність та безпечність експлуатації системи.

У межах функціональних вимог система має реалізовувати повний цикл роботи з даними: завантаження інформації з CSV-файлів, обробку даних з подальшим проведенням кластерного аналізу за допомогою алгоритму K-Means, збереження результатів у реляційній базі даних PostgreSQL, а також виведення на інтерактивну мапу із можливістю фільтрації. Завдяки цьому забезпечується не лише обробка, але й аналітична інтерпретація вхідної інформації, що сприяє виявленню потенційно небезпечних зон.

З боку нефункціональних вимог ключовими є надійність системи, її масштабованість, ергономічність інтерфейсу користувача, продуктивність у роботі з великими обсягами даних, а також базові засоби забезпечення інформаційної безпеки. Усі ці вимоги є критично важливими для розробки систем, призначених для аналізу подій у реальному середовищі, де точність, швидкість і безпечність обробки даних мають вирішальне значення.

У табл. 1.1 подано формалізовані специфікації функціональних і нефункціональних вимог, які є основою для подальшого проєктування та реалізації системи.

Таблиця 1.1

Специфікація вимог

№	Назва вимоги	Опис функціоналу
1	2	3
Функціональні вимоги		
1	Імпорт даних	Завантаження CSV-файлу з інформацією про ДТП через інтерфейс користувача
2	Валідація структури	Перевірка відповідності структури файлу вимогам системи перед обробкою
3	Кластеризація	Виконання алгоритму K-Means для групування ДТП за просторово-часовими ознаками
4	Збереження результатів	Запис результатів кластеризації до бази даних PostgreSQL

Таблиця 1.1(продовження)

1	2	3
5	Візуалізація	Відображення ДТП та кластерів на інтерактивній мапі з можливістю фільтрації
6	Повторна кластеризація	Можливість перезапуску алгоритму з новими параметрами кластеризації
7	Перегляд детальної інформації	Надання розгорнутої інформації про конкретну подію ДТП при взаємодії з мапою
Нефункціональні вимоги		
1	Надійність	Система стабільно обробляє ≥1000 записів без збоїв і з прийнятним часом відповіді
2	Продуктивність	Середній час кластеризації набору з 1000 записів не перевищує 2 секунд
3	Масштабованість	Архітектура дозволяє подальше розширення функціональності та обробку великих даних
4	Зручність інтерфейсу	Інтуїтивно зрозумілий веб-інтерфейс для базових та просунутих користувачів
5	Конфіденційність даних	Хешування облікових даних, базове логування доступу
6	Захист від помилок	Система має логування помилок; валідація та обробка виключних ситуацій
7	Безпека взаємодії з БД	Використання ORM (Entity Framework Core) для безпечного доступу до бази даних

Загалом аналіз вимог сформував концептуальний каркас програмної системи, що забезпечує цілісність її архітектури та адаптивність до зміни обставин експлуатації. Сформульовані вимоги забезпечують не лише виконання безпосередніх функцій, а й формують підґрунтя для подальшого масштабування, оптимізації та впровадження програмного продукту в інституційне або муніципальне середовище.

Ретельно сформульовані функціональні та нефункціональні вимоги до програмної системи створюють основу для її надійної розробки, впровадження та масштабування. Вони дозволяють визначити межі системи, очікувану поведінку в різних сценаріях експлуатації, забезпечити безпечну обробку критичних даних і гарантувати зручність використання для кінцевих користувачів. Дотримання зазначених вимог у процесі реалізації є запорукою успішного досягнення поставленої мети – створення ефективного інструменту

для аналізу та візуалізації даних про дорожньо-транспортні пригоди в умовах великого міста.

Окрему роль у процесі проєктування відіграють користувацькі вимоги, які відображають очікування кінцевих користувачів щодо інтуїтивної взаємодії з системою, її доступності, безпечності та функціональної повноти. У межах реалізації системи інтерфейс має бути максимально простим у використанні, дозволяти завантаження файлів із даними про ДТП одним кліком, автоматично здійснювати перевірку структури та виконувати кластеризацію за заданими параметрами. Відображення результатів кластеризації реалізується у вигляді інтерактивної мапи з підтримкою масштабування, перегляду деталей і фільтрації за умовами події (погода, тип транспорту, день тижня). Окрім того, система повинна реалізовувати авторизацію користувачів із розмежуванням прав доступу (гості, аналітики, адміністратори), автентифікацію, зберігання облікових даних у хешованому вигляді, а також автоматичне збереження результатів до бази даних.

Функціональність системи має бути адаптована до потреб різних типів користувачів. Зокрема, передбачено реалізацію авторизації з ролями (гості, аналітики, адміністратори), кожна з яких має доступ лише до визначеного рівня функцій. Для забезпечення базового рівня інформаційної безпеки повинна бути реалізована автентифікація користувачів, хешування облікових даних та контроль доступу до критичних компонентів системи.

1.3 Моделювання предметної області

Моделювання предметної області є невіддільною складовою процесу розробки інформаційної системи, оскільки дозволяє формалізувати ключові об’єкти, їх атрибути, взаємозв’язки та поведінку у рамках заданого функціонального контексту. У межах даної роботи предметною областю є просторово-часовий аналіз дорожньо-транспортних пригод (ДТП) у межах міста Києва з використанням алгоритмів кластеризації. Для адекватного відображення логіки системи визначено основні сутності, побудовано UML-діаграми класів та

описано сценарії взаємодії користувача з системою. Це створює основу для подальшого етапу реалізації архітектури застосунку та бази даних.

З метою унаочнення функціональних можливостей системи та взаємодії основних учасників із програмним забезпеченням було побудовано UML-діаграму прецедентів. Вона відображає предметну область із точки зору зовнішніх акторів – користувача та адміністратора – та демонструє їхню взаємодію з ключовими функціями системи [5].

На діаграмі прецедентів зображено ключові сценарії взаємодії користувача та адміністратора з функціональністю системи кластеризації дорожньо-транспортних пригод. Вона дозволяє описати зовнішню поведінку системи на основі ролей учасників (акторів) і пов'язаних із ними прецедентів (use cases).

Основним актором є Користувач, який взаємодіє з системою за наступними сценаріями:

1. перегляд результатів на мапі – основна функція, що дозволяє бачити згруповані ДТП за кластерами;
2. перегляд детальної інформації про ДТП – деталізація кожного інциденту з мапи;
3. фільтрація результатів – обмеження видимих подій за часом, погодою, типом транспорту.
4. зміна параметрів кластеризації – дозволяє задати кількість кластерів або інші параметри алгоритму;
5. авторизація користувача – автентифікація для отримання доступу до персоналізованих функцій.

Процес перегляду результатів включає запуск кластеризації (K-Means), яка, у свою чергу, передбачає обов'язковий імпорт даних про ДТП (з CSV), що супроводжується валідацією структури файлу. Ці взаємозв'язки на діаграмі позначено відношенням «include», що вказує на необхідність виконання підпроцедур у межах вищого прецеденту.

Другим актором є Адміністратор, який має розширені повноваження, а саме:

1. управління обліковими записами – створення, редагування, деактивація користувачів;
2. перегляд журналів дій – моніторинг активності користувачів у системі.

Діаграма, яка представлена на рисунку 1.2, забезпечує структуроване уявлення про функціональні можливості системи з урахуванням ролей користувачів та логічної послідовності виконання дій.

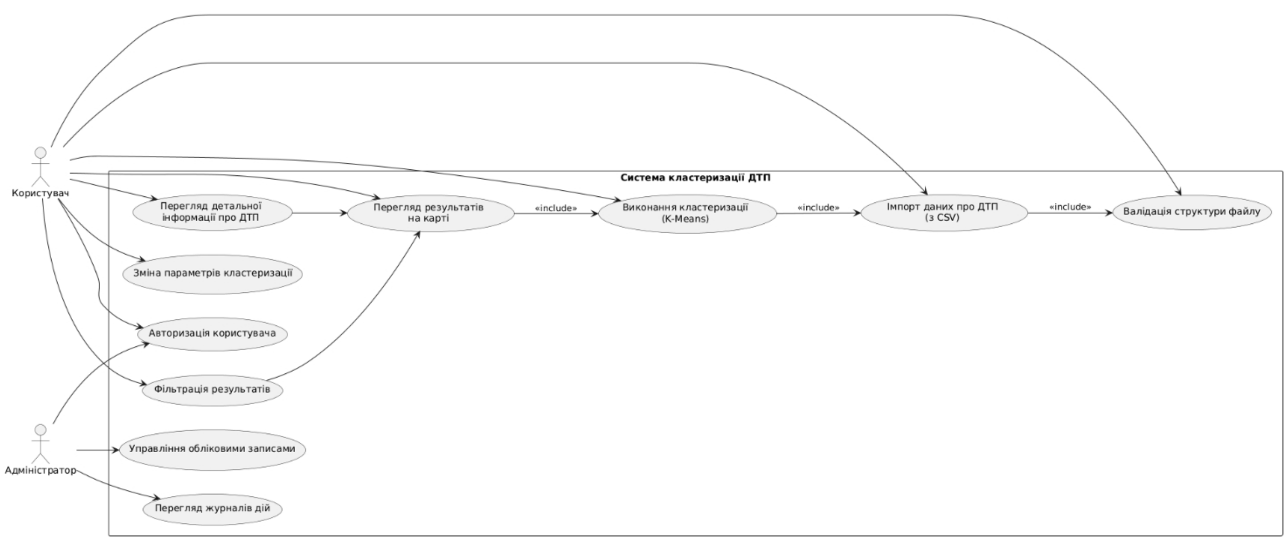


Рис. 1.2 UML-діаграма прецедентів предметної області системи кластеризації ДТП

Для формалізації внутрішньої структури предметної області системи кластеризації дорожньо-транспортних пригод та її реалізації у вигляді бази даних, побудовано UML-діаграму класів, що демонструє сутності, атрибути, зв’язки та їхню кратність. На відміну від діаграми прецедентів, яка описує взаємодію зовнішніх акторів із системою, діаграма класів відображає логічну модель даних, визначаючи внутрішні компоненти системи та їхню взаємодію. Вона є основою для створення схеми бази даних, а також для проектування взаємозв’язків між модулями програмного забезпечення.

На рисунку 1.3 зображено чотири основні сутності: Accident (ДТП), Cluster (Кластер), User (Користувач) та ClusteringResult (Результат кластеризації). Кожна з них має свій набір атрибутів, що описують її структурні характеристики.

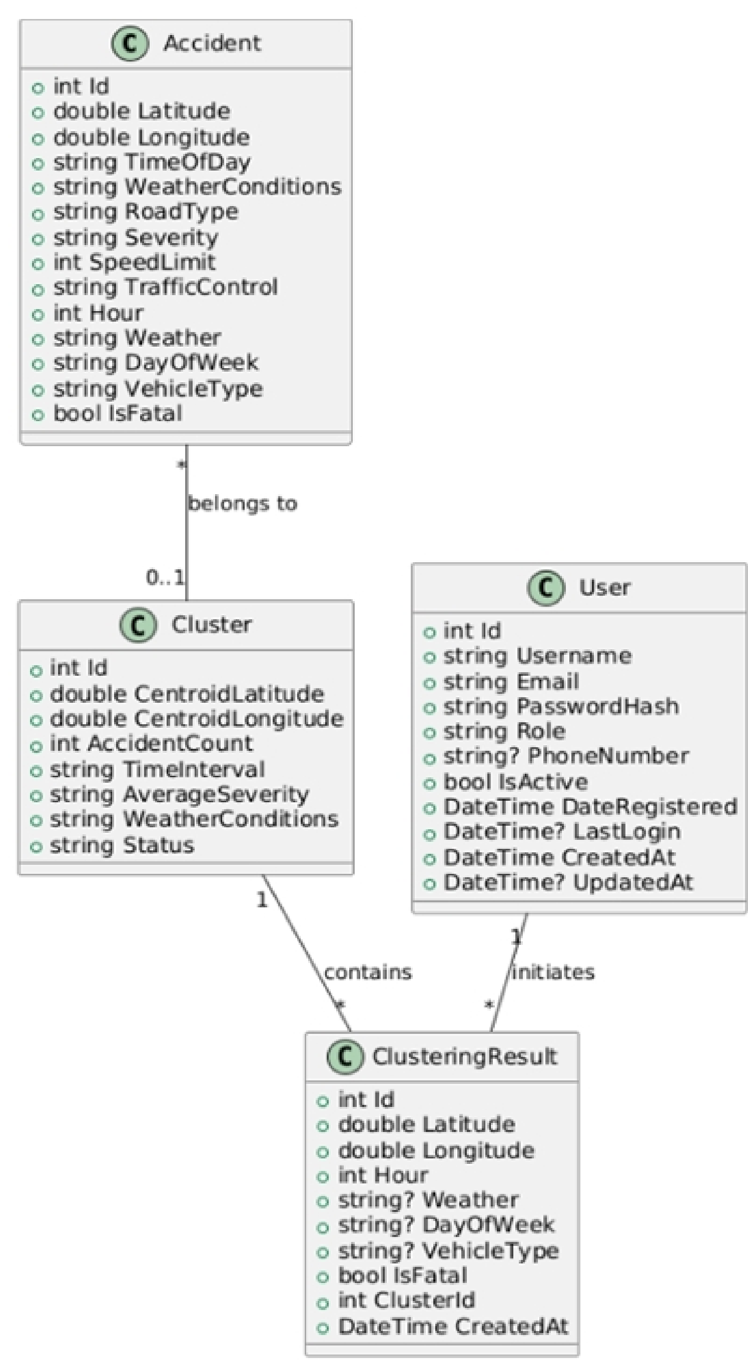


Рис. 1.3 UML-діаграма класів предметної області: сутності, атрибути та зв’язки

Сутність Accident зберігає інформацію про кожну зафіксовану дорожньо-транспортну пригоду. Атрибути включають координати, погодні умови, тип дороги, час події, тип транспортного засобу та рівень тяжкості. У табл. 1.2 наведено атрибутивну структуру цієї сутності. Значення координат, часу доби й погодних умов дають змогу не лише проводити аналітику, а й використовувати їх як ключові ознаки під час кластеризації.

Таблиця 1.2

Атрибути сутності "ДТП"

№	Атрибут	Тип	Опис
1	Id	int	Унікальний ідентифікатор ДТП
2	Latitude	double	Географічна широта місця ДТП
3	Longitude	double	Географічна довгота місця ДТП
4	TimeOfDay	string	Час доби (наприклад, «ранок», «вечір»)
5	Hour	int	Година доби у форматі 0–23
6	WeatherConditions	string	Погодні умови під час ДТП
7	Weather	string	Погодні умови у спрощеному форматі (наприклад, «дощ»)
8	DayOfWeek	string	День тижня, коли сталося ДТП
9	RoadType	string	Тип дороги (магістраль, міська вулиця тощо)
10	Severity	string	Рівень тяжкості ДТП (легка, тяжка, фатальна)
11	SpeedLimit	int	Обмеження швидкості на місці ДТП (км/год)
12	TrafficControl	string	Наявність дорожніх знаків, світлофорів тощо
13	VehicleType	string	Тип транспортного засобу, що брав участь
14	IsFatal	bool	Показник, чи було ДТП фатальним (з загиблими)

Сутність Cluster відображає логічне об’єднання ДТП, що мають схожі ознаки. Кожен кластер визначається координатами центроїда, кількістю подій, погодними умовами та часовими характеристиками (табл. 1.3). Кластери використовуються для узагальнення, виявлення тенденцій та візуалізації небезпечних зон на мапі.

Таблиця 1.3

Атрибути сутності "Кластер"

№	Атрибут	Тип	Опис
1	2	3	4
1	Id	int	Унікальний ідентифікатор кластера
2	CentroidLatitude	double	Географічна широта центру кластера
3	CentroidLongitude	double	Географічна довгота центру кластера
4	AccidentCount	int	Кількість ДТП у кластері
5	AverageSeverity	string	Середній рівень тяжкості ДТП у кластері
6	AverageWeather	string	Середні погодні умови, що характеризують кластер
7	TimeInterval	string	Часовий проміжок, охоплений кластером
8	Status	string	Стан кластера (наприклад, попередній, підтверджений)

Сутність User є відповідальною за автентифікацію, персоналізацію доступу й логування активностей. Записи користувачів зберігають як ідентифікаційні дані, так і інформацію про роль та активність (табл. 1.4).

Таблиця 1.4

Атрибути сутності "Користувач"

№	Поле	Тип даних	Опис	Примітки
1	Id	int	Унікальний ідентифікатор користувача	Первинний ключ
2	Username	string	Логін користувача	Унікальний
3	Email	string	Електронна пошта	Унікальний
4	PasswordHash	string	Хеш пароля для аутентифікації	Для безпеки паролі не зберігаються у відкритому вигляді
5	Role	string	Роль користувача (адмін, аналітик, гість)	Визначає рівень доступу
6	PhoneNumber	string?	Телефонний номер	Опціонально
7	IsActive	bool	Статус активності облікового запису	Вказує, чи користувач активний
8	DateRegistered	DateTime	Дата реєстрації користувача	Автоматично задається
9	LastLogin	DateTime?	Час останнього входу користувача	Опціонально
10	CreatedAt	DateTime	Дата створення профілю	Може співпадати з DateRegistered
11	UpdatedAt	DateTime?	Дата останнього оновлення даних профілю	Опціонально

Сутність ClusteringResult представляє підсумки роботи алгоритму кластеризації. Вона зберігає інформацію про те, які саме ДТП були згруповані, до яких кластерів вони належать, а також фіксує умови подій і метаінформацію запуску (табл. 1.5).

Таблиця 1.5

Атрибути сутності "Результат кластеризації"

№	Атрибут	Тип	Опис
1	2	3	4
1	Id	int	Унікальний ідентифікатор результату
2	Latitude	double	Географічна широта ДТП, що увійшла до кластера
3	Longitude	double	Географічна довгота ДТП
4	Hour	int	Година доби ДТП
5	Weather	string?	Погодні умови (nullable)
6	DayOfWeek	string?	День тижня (nullable)
7	VehicleType	string?	Тип транспортного засобу (nullable)

Таблиця 1.5 (продовження)

1	2	3	4
8	IsFatal	bool	Показник смертельності ДТП
9	Cluster	int	Ідентифікатор кластера, до якого належить ДТП
10	CreatedAt	DateTime	Час створення запису у системі
11	ClusterCount	int	Загальна кількість кластерів у результаті (з ClusteringResultHistoryDto)
12	ShortResults	string	Короткий опис або статистика результатів кластеризації

Зв’язки між сутностями відображено на UML-діаграмі класів:

1. один користувач може ініціювати багато результатів кластеризації (зв’язок 1 до N);
2. один кластер може містити багато записів результатів кластеризації (1 до N);
3. кожне ДТП може належати до одного або жодного кластера (0..1).

Для візуального представлення ключових сценаріїв функціонування системи кластеризації ДТП побудовано UML-діаграми послідовності, які демонструють часову залежність обміну повідомленнями між учасниками процесу. Ці діаграми дозволяють простежити, як саме відбувається ініціація кластеризації, обробка даних, збереження результатів і їхнє відображення користувачеві. Дотримання послідовності взаємодії між модулями системи забезпечує цілісність логіки роботи та правильну реалізацію взаємозв’язків між компонентами програмного забезпечення.

На першому етапі користувач відкриває інтерфейс системи, переходить до панелі кластеризації, де вводить основні параметри – кількість кластерів, кількість ітерацій тощо. Це зображено на рисунку 1.4. Після натискання кнопки запуску, веб-інтерфейс передає параметри до контролера кластеризації, який перевіряє їх на коректність і ініціює виконання алгоритму.

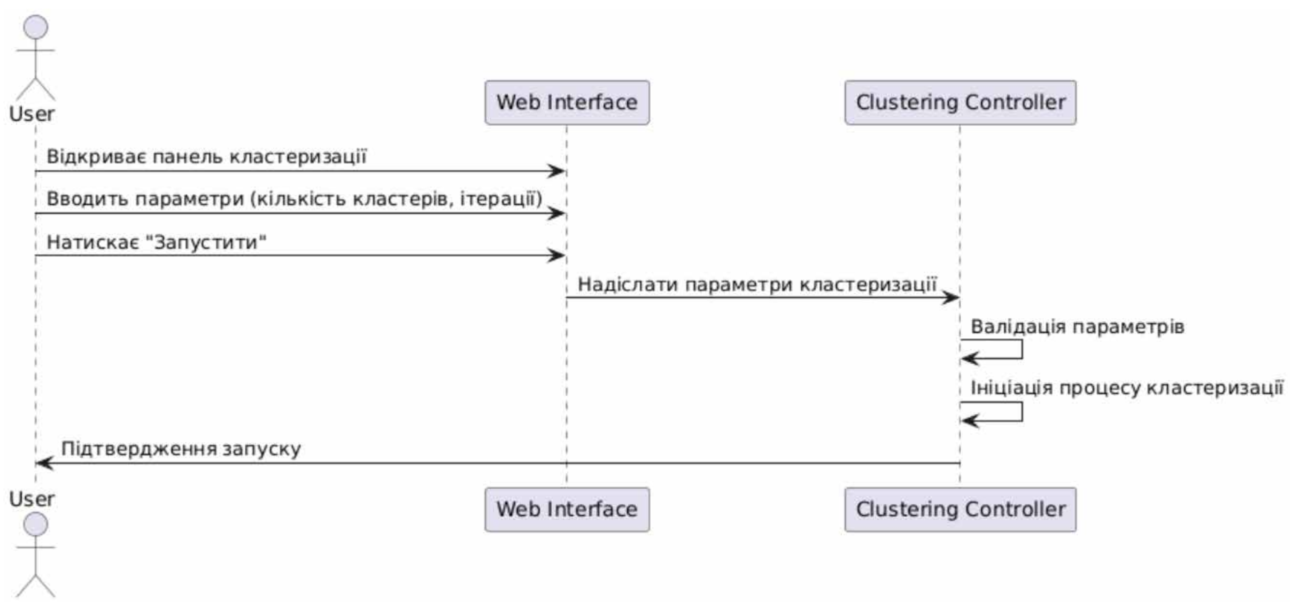


Рис. 1.4 Послідовність дій при запуску кластеризації користувачем

Після прийому параметрів система формує SQL-запит до бази даних PostgreSQL через ORM (Entity Framework Core), з урахуванням заданих фільтрів (час, тип транспорту, погода тощо). На рисунку 1.5 показано, що запит виконується з урахуванням вимог продуктивності, а отримані дані передаються до контролера для подальшої обробки.

Рис. 1.5 Послідовність отримання та фільтрації даних ДТП з бази

Контролер обробляє отримані дані, ініціюючи виконання кластеризації на основі алгоритму K-Means, реалізованого з використанням бібліотеки Accord.NET. Після завершення кластерного аналізу результати фіксуються у відповідній таблиці бази даних із зазначенням приналежності кожної події до

певного кластеру, координатою, часом доби, погодою тощо. Крім того, зберігаються метадані запуску, включаючи параметри, час та ініціатора процесу. Всі ці дії наведено на рисунку 1.6.

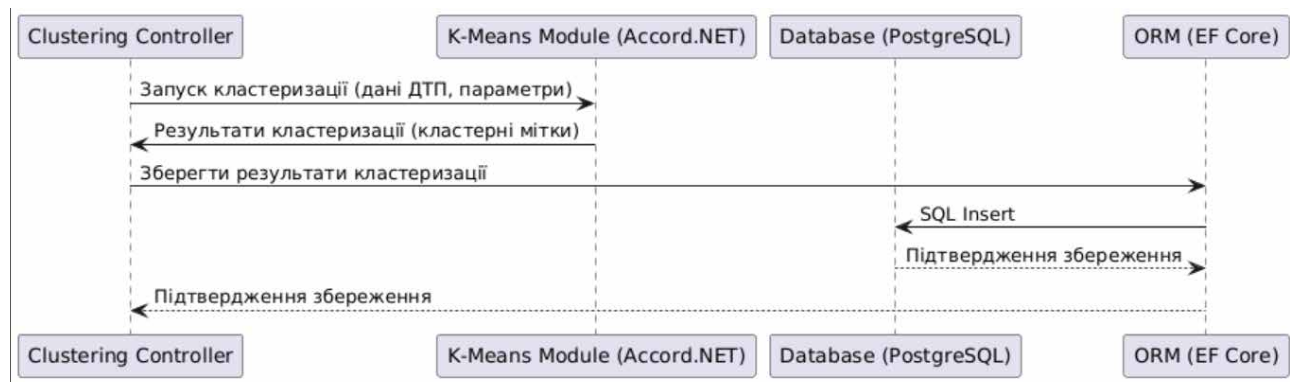


Рис. 1.6 – Послідовність виконання алгоритму K-Means і збереження результатів у базу

Після завершення обчислень користувач може ініціювати запит на перегляд результатів, що відображено на рисунку 1.7.

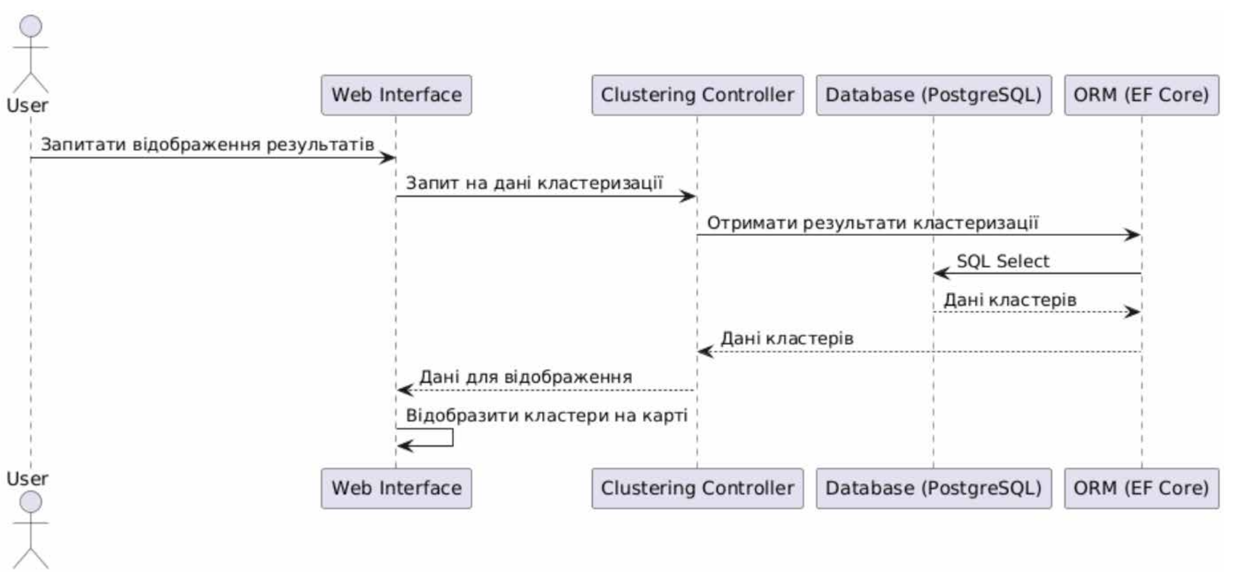


Рис. 1.7 Послідовність відображення результатів кластеризації на мапі

Інтерфейс надсилає запит до контролера, який звертається до бази даних і отримує результат кластеризації. Далі дані передаються у вигляді кластерів назад до інтерфейсу, де вони візуалізуються на інтерактивній мапі. Кожен кластер

відображається кольоровим маркером, що дозволяє користувачу інтуїтивно аналізувати просторову структуру подій. Передбачена можливість детального перегляду кожного кластеру, а також експорт результатів для подальшого аналізу.

На завершення моделювання предметної області доцільно представити загальну логіку виконання процесу кластеризації у вигляді діаграми активності. Діаграма активності, представлена на рисунку 1.8, відображає динамічну логіку процесу кластеризації дорожньо-транспортних пригод у системі. Вона ілюструє послідовність дій, що виконує користувач, а також реакцію системи на введені параметри й подальше опрацювання даних до моменту відображення результатів на інтерактивній мапі.

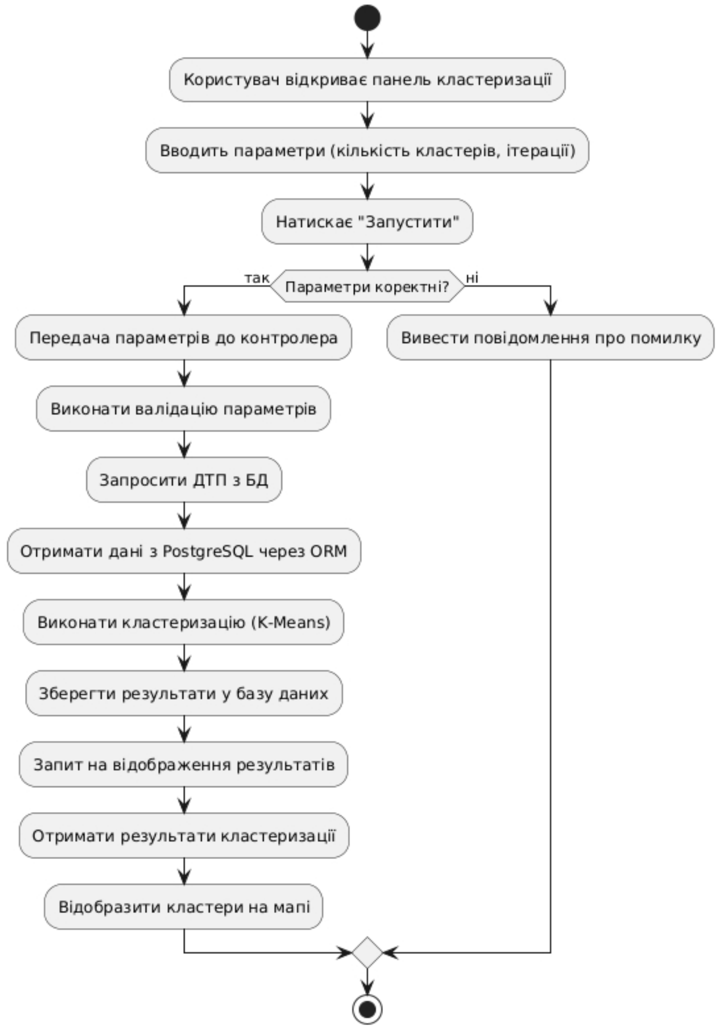


Рис. 1.8 Діаграма активності процесу кластеризації ДТП

Процес починається з ініціативи користувача, який відкриває відповідну панель кластеризації у веб-інтерфейсі. Після цього відбувається введення параметрів – таких як кількість кластерів або кількість ітерацій алгоритму – і натискання кнопки «Запустити». На цьому етапі система перевіряє правильність введених значень.

Якщо параметри є некоректними, відразу формується повідомлення про помилку, і виконання припиняється. У разі успішної валідації параметрів вони передаються до контролера, де здійснюється логічна перевірка і підготовка до кластеризації. Далі формуються запити до бази даних PostgreSQL через ORM-технологію (Entity Framework Core), що забезпечує отримання релевантної інформації про ДТП відповідно до заданих критеріїв.

Після надходження даних система запускає алгоритм кластеризації (K-Means), результати якого автоматично зберігаються до бази даних. Наступним кроком є запит на відображення цих результатів, що ініціюється веб-інтерфейсом користувача. Відповідні дані отримуються з бази та виводяться на інтерактивну мапу, де кожен кластер позначено візуально з урахуванням просторових координат, інтенсивності та додаткових аналітичних ознак.

Діаграма активності дає змогу узагальнити сценарій виконання ключової бізнес-логіки – від взаємодії користувача із системою до обробки даних і відображення результатів аналізу.

1.4 Огляд інформаційних джерел та існуючих рішень

У процесі розробки системи кластеризації ДТП доцільним є вивчення існуючих рішень, які вже реалізували подібний функціонал. Аналіз таких систем дозволяє виявити їхні сильні сторони, обмеження та визначити потенційні напрями вдосконалення розроблюваного застосунку. Нижче розглянуто приклади програмного забезпечення, що використовують методи обробки даних про дорожньо-транспортні пригоди з подальшою візуалізацією результатів.

Accident Explorer (MIOsoft) – це веб-орієнтований інструмент, розроблений для аналізу фатальних ДТП у США на основі бази FARS [6]. Його

функціональність охоплює кластеризацію аварій та виведення результатів на інтерактивну карту, реалізовану через LeafletJS та Mapbox. Особливістю є простота користування без потреби у знаннях із машинного навчання (рис. 1.9).

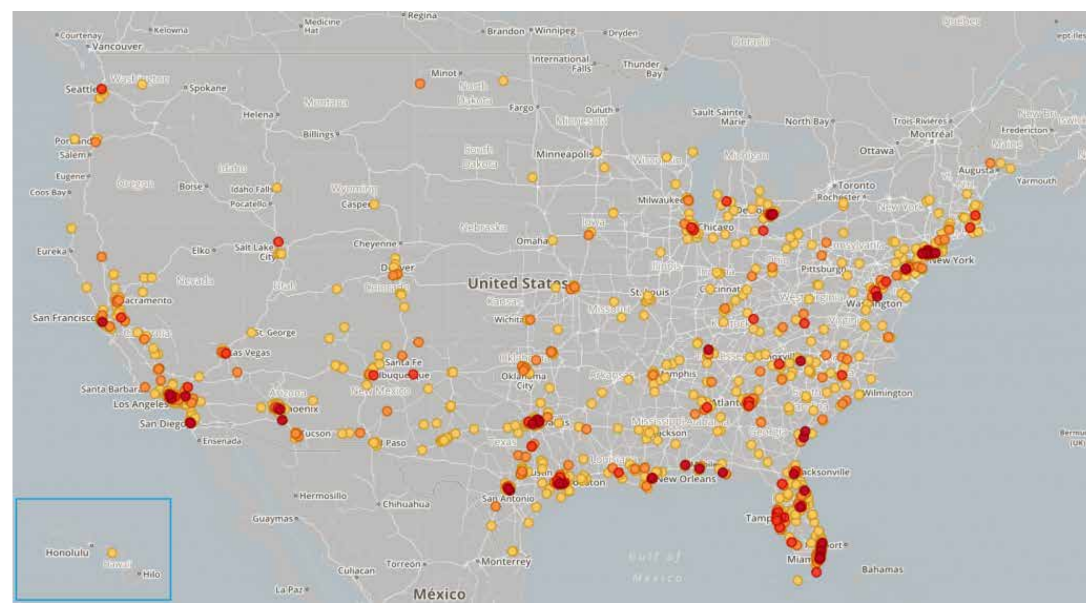


Рис. 1.9 Accident Explorer (MIOsoft)

Incident Cluster Explorer (ICE), створений CATT Lab, є більш аналітичним інструментом для кластеризації транспортних інцидентів [7]. Система передбачає побудову діаграм і мап для глибшого розуміння просторово-часових закономірностей подій (рис. 1.10).

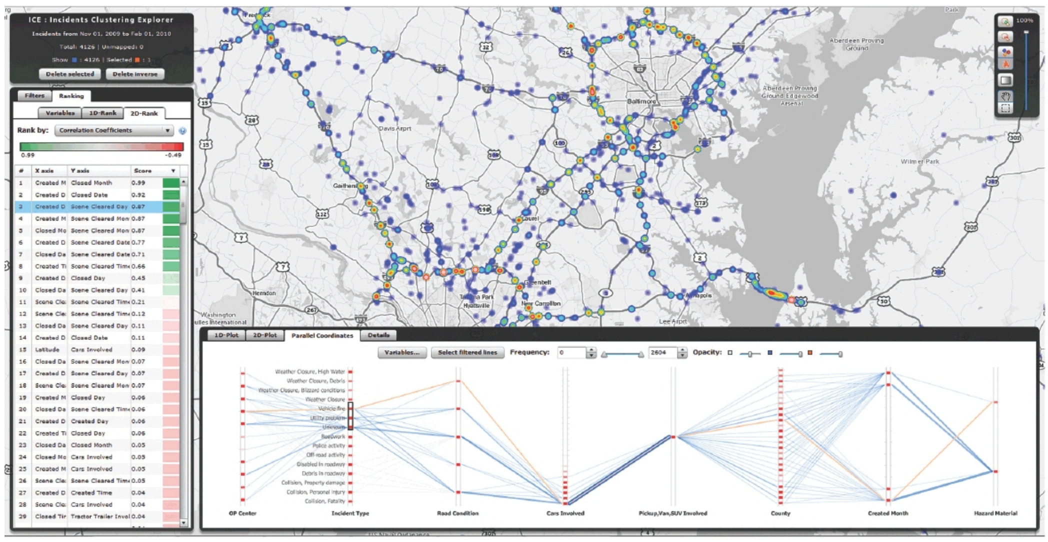


Рис. 1.10 Incident Cluster Explorer (ICE) – CATT Lab

iMAAP (TRL Software) підтримує детальний аналіз аварій, зокрема геопросторову візуалізацію, виявлення небезпечних зон та оцінку тенденцій [8]. Рішення орієнтоване на професійних користувачів у сфері транспорту та інфраструктури (рис. 1.11).

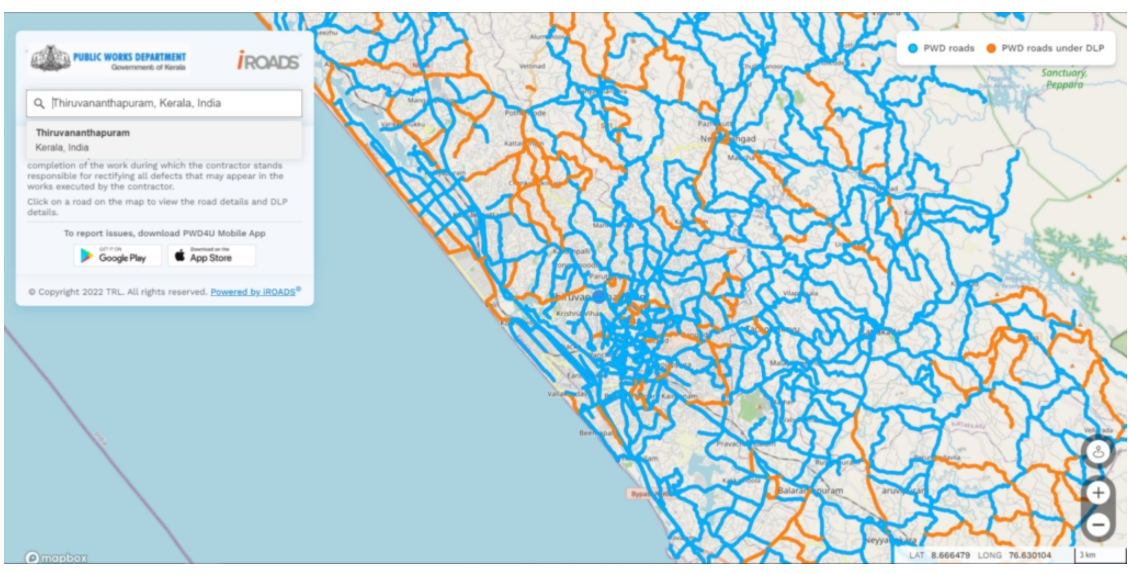


Рис. 1.11 – Accident Explorer (MIOsoft)

KeyACCIDENT – приклад програмного забезпечення, що використовується у розслідуваннях ДТП [9]. Система реалізує алгоритмічну ідентифікацію місць концентрації аварій для ухвалення рішень у сфері безпеки руху (рис. 1.12).

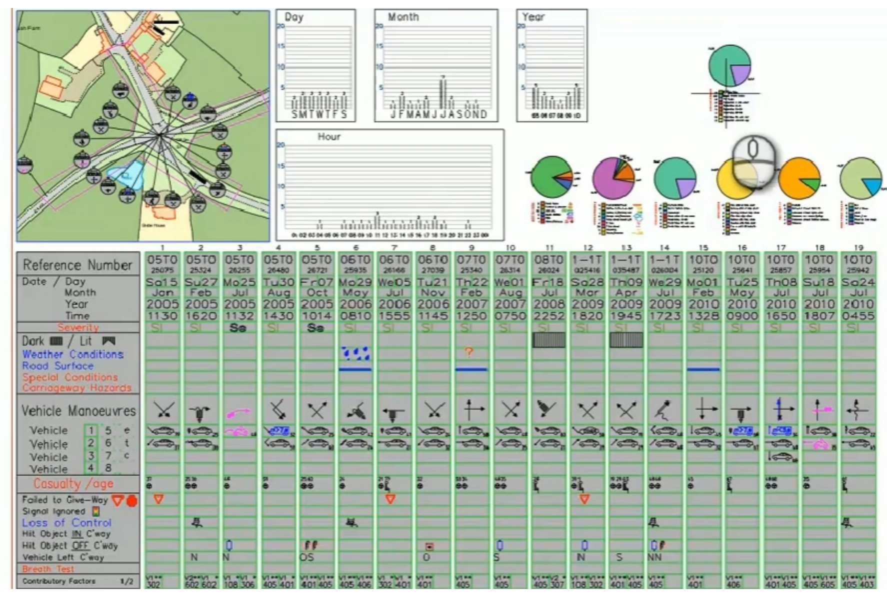


Рис. 1.12 KeyACCIDENT

CRASH (Citian) поєднує елементи аналітики, штучного інтелекту та картографії для формування рекомендацій щодо підвищення безпеки дорожнього руху у містах [10]. Завдяки динамічним інформаційним панелям, користувачі мають змогу оперативно реагувати на актуальні ризики (рис. 1.13).

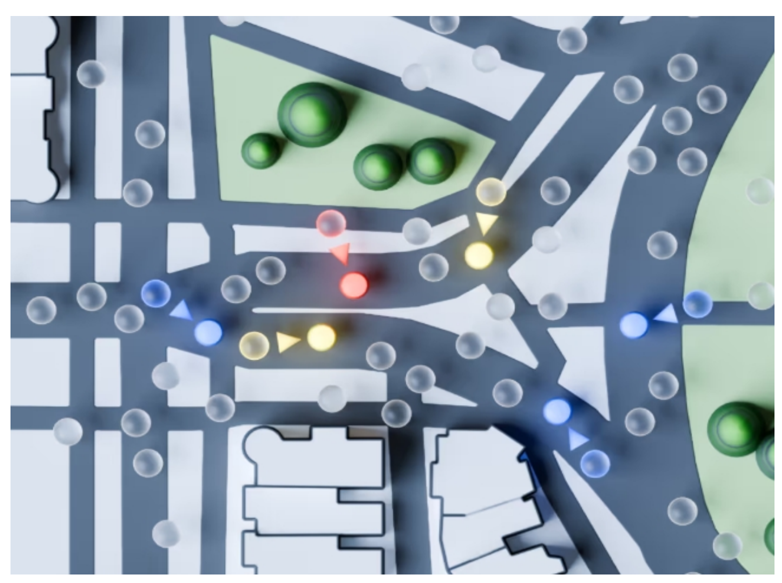


Рис. 1.13 – CRASH – Citian

TIMS (Transportation Injury Mapping System) – платформа візуалізації та аналізу ДТП, що зосереджена на даних з Каліфорнії [11]. Забезпечує доступ до статистики, порівняння періодів, мапування даних і розрахункових звітів для планувальників та дослідників (рис. 1.14).

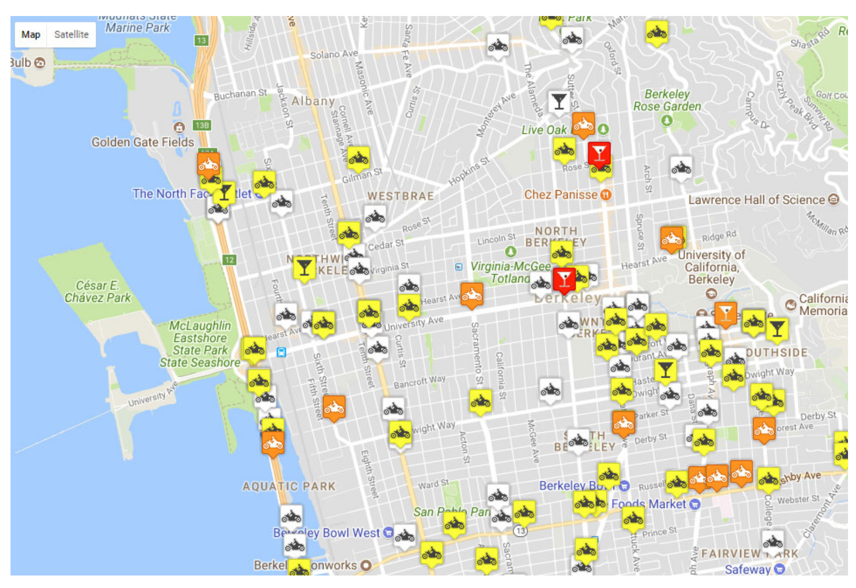


Рис. 1.14 Transportation Injury Mapping System (TIMS)

У табл. 1.6 представлено коротке порівняння функціональних характеристик вищенаведених систем.

Таблиця 1.6

Порівняння функціональності існуючих рішень

№	Назва	Кластеризація	Інтерактивна мапа	Налаштування алгоритму	Призначення
1	Accident Explorer	Так	Так	Обмежене	Демонстрація кластеризації
2	Incident Cluster Explorer	Так	Так	Обмежене	Аналіз транспортних інцидентів
3	iMAAP	Так	Так	Обмежене	Аналіз дорожніх аварій
4	KeyACCIDENT	Так	Так	Обмежене	Розслідування ДТП
5	CRASH – Citian	Так	Так	Обмежене	Планування міських покращень
6	TIMS	Обмежене	Так	Ні	Візуалізація та порівняння даних

Крім того, доцільно зіставити особливості інтерфейсів та рівень гнучкості алгоритмічних налаштувань у порівнянні з розроблюваним застосунком, що дозволяє обґрунтувати переваги пропонованої архітектури (табл. 1.7).

Таблиця 1.7

Порівняльний аналіз інтерфейсу та налаштувань кластеризації

№	Критерій	Розроблюваний застосунок	Існуючі рішення
1	2	3	4
1	Інтерфейс	Веб-інтерфейс на ASP.NET Core з інтерактивною мапою (Leaflet)	Більшість також використовують веб-інтерфейси з мапами, але менш гнучкі (часто фіксовані панелі або iframe-інтеграції)
2		Простий дизайн, орієнтований на користувача	Багато рішень орієнтовані на аналітиків/фахівців, тому менш інтуїтивні
3		Інтерактивне обрання кластерів, фільтрація за параметрами	В деяких інструментах (напр. iMAAP) можливості фільтрації обмежені
4	Налаштування алгоритму	Реалізована можливість обрати кількість кластерів (K у K-Means)	У більшості аналогів алгоритм прихований або автоматичний без налаштувань користувачем
5		Алгоритм кластеризації інтегрований напряму у систему	Часто використовують кластеризацію «за сценою», без пояснень або гнучкості

Таблиця 1.7 (продовження)

1	2	3	4
6		У перспективі можлива заміна/розширення алгоритмів (напр. DBSCAN)	Обмежене розширення в готових SaaS-рішеннях
7	Прозорість логіки	Повна контрольованість алгоритму з боку розробника та користувача	У більшості аналогів алгоритми закриті, без можливості налаштування або перегляду внутрішньої логіки

Розглянуті платформи демонструють високу функціональність у галузі аналізу ДТП. Проте більшість із них мають обмеження у частині налаштування алгоритмів та адаптації до специфіки нових міст чи форматів даних. Запропонована у цій роботі система покликана поєднати простоту використання з відкритістю конфігурації та розширюваністю алгоритмічного ядра.

1.5 Постановка завдання

У результаті аналізу предметної області, користувацьких і системних вимог, а також існуючих рішень у сфері кластеризації дорожньо-транспортних пригод сформульовано основну задачу дослідження: розробити програмне забезпечення для автоматизованого аналізу ДТП із застосуванням алгоритмів кластеризації та інтерактивною візуалізацією результатів на карті міста.

Проєкт передбачає створення веб-застосунку, який надає користувачам можливість завантажувати дані про дорожньо-транспортні пригоди, налаштовувати параметри кластеризації, аналізувати результати у вигляді мапи кластерів і здійснювати фільтрацію інформації за обраними ознаками. Система має забезпечити простий у використанні інтерфейс, підтримку кількох ролей доступу (гостьовий режим, аналітик, адміністратор), базовий рівень захисту даних, а також можливість масштабування і адаптації до різних джерел інформації.

Відповідно до поставленої мети, вирішуються такі основні завдання:

– сформулювати вимоги до функціональності та архітектури системи;

- розробити інформаційну модель предметної області з використанням UML-діаграм;
- побудувати логічну модель бази даних та реалізувати її в СУБД PostgreSQL;
- розробити веб-інтерфейс із можливістю завантаження, перегляду, фільтрації та візуалізації даних про ДТП;
- інтегрувати алгоритм кластеризації (K-Means) з можливістю налаштування параметрів;
- реалізувати механізми авторизації, ролей користувачів та обмеженого доступу до функцій системи;
- протестувати систему на реальному наборі даних та проаналізувати ефективність кластеризації.

У результаті реалізації вищезазначених завдань очікується створення сучасного програмного інструменту, який сприятиме автоматизації аналізу ДТП, надасть зручний інтерфейс для користувачів різних рівнів підготовки та дозволить приймати обґрунтовані управлінські рішення у сфері безпеки дорожнього руху.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних у вигляді ER-діаграми

Для опису логічної структури зберігання даних у розроблюваній системі кластеризації ДТП було побудовано модель сутностей і зв’язків (ER-модель), яка слугує основою для проєктування реляційної бази даних. Побудова цієї моделі дозволяє формалізувати відношення між ключовими об’єктами, які беруть участь у функціонуванні системи, і визначити основні атрибути кожної сутності.

Як показано на рисунку 2.1, логічна модель охоплює п’ять основних сутностей: користувач (User), результат кластеризації (ClusteringResult), кластер (Cluster), дорожньо-транспортна пригода (Accident) та журнал подій (Log).

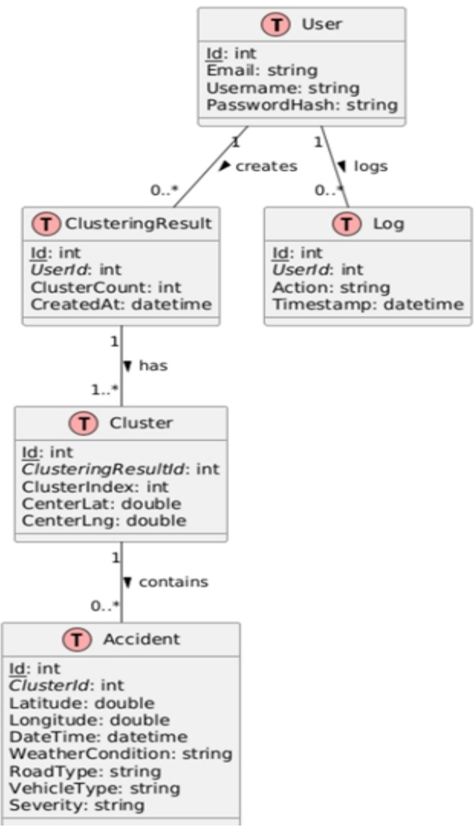


Рис. 2.1 Логічна модель даних системи кластеризації ДТП у вигляді ER-діаграми

Між цими сутностями встановлено чіткі зв’язки, що описують, як дані взаємодіють у процесі виконання кластеризації, збереження результатів та контролю дій користувача.

Сутність User відіграє роль джерела взаємодії з системою. Вона містить облікові дані користувача, включно з адресою електронної пошти, іменем користувача та хешем пароля для автентифікації. Один користувач може створити кілька запусків кластеризації, а також здійснювати дії, які фіксуються у журналі.

Основні атрибути:

- Id – унікальний ідентифікатор користувача;
- Email – електронна пошта;
- Username – ім’я користувача;
- PasswordHash – хеш пароля для забезпечення безпеки.

ClusteringResult зберігає інформацію про кожен запуск алгоритму кластеризації. Для кожного результату зазначено кількість кластерів, що було створено, і дату його формування. Така структура дозволяє відстежувати історію виконання алгоритмів і формувати статистичні звіти.

Основні атрибути:

- Id – унікальний ідентифікатор результату кластеризації;
- UserId – зовнішній ключ, що посилається на користувача, який ініціював кластеризацію;
- ClusterCount – кількість кластерів, визначених при запуску;
- CreatedAt – дата та час створення результату кластеризації.

Наступна сутність – Cluster – описує просторові групи ДТП, сформовані в результаті кластеризації. Кожен кластер пов’язаний із певним результатом та містить географічні координати центру. Такий підхід дозволяє візуалізувати центри скупчення аварій на мапі міста.

Основні атрибути:

- Id – унікальний ідентифікатор кластера;

– ClusteringResultId – зовнішній ключ, що пов’язує кластер з відповідним результатом кластеризації;
– ClusterIndex – порядковий номер або індекс кластера;
– CenterLat / CenterLng – координати центру кластера.

У свою чергу, сутність Accident описує конкретні події ДТП, які було включено до кластерів. Для кожної події фіксуються координати місця, дата, погодні умови, тип транспортного засобу та тяжкість наслідків. Ці параметри є ключовими для аналізу ризиків і прийняття рішень щодо підвищення безпеки дорожнього руху.

Основні атрибути:

– Id – унікальний ідентифікатор ДТП;
– ClusterId – зовнішній ключ, що вказує на кластер, до якого належить ця ДТП;
– Latitude / Longitude – географічні координати;
– DateTime – дата та час події;
– WeatherCondition – погодні умови;
– RoadType – тип дороги;
– VehicleType – тип транспортного засобу;
– Severity – рівень тяжкості ДТП.

Сутність Log виконує функцію контролю активності користувача. Вона реєструє усі суттєві дії, пов’язані з запуском кластеризації, оновленням даних чи іншими подіями. Запис журналу включає час дії, її тип і посилання на відповідного користувача.

Основні атрибути:

– Id – унікальний ідентифікатор запису;
– UserId – зовнішній ключ на користувача;
– Action – опис дії (наприклад, запуск кластеризації);
– Timestamp – дата та час події.

Взаємозв’язки між сутностями логічної моделі забезпечують цілісність і структурну узгодженість даних:

- один користувач може ініціювати декілька результатів кластеризації;
- кожен результат кластеризації може містити один або кілька кластерів;
- кластер, у свою чергу, може включати багато записів ДТП;
- система зберігає історію дій кожного користувача у журналі подій.

Ця логічна структура реалізує принцип нормалізації бази даних, сприяє підвищенню узгодженості та зменшенню надлишковості збережених даних. Вона також забезпечує адаптивність і масштабованість програмного рішення у разі розширення функціональності або збільшення обсягів інформації [2, 12].

2.2 Діаграма класів та діаграма кооперації

Для моделювання архітектурної структури програмної системи кластеризації даних про дорожньо-транспортні пригоди (ДТП) застосовано UML-діаграму класів [14]. Цей тип діаграми надає можливість формалізовано відобразити взаємозв’язки між ключовими класами, їхні методи та обов’язки в межах бізнес-логіки застосунку, а також визначити функціональну розподіленість компонентів системи.

На рисунку 2.2 подано логічну структуру програмного ядра системи, яка охоплює контролери, сервіси, утилітарні компоненти та допоміжні класи [5, 13]. Основну функціональну роль відіграють сервіси, які реалізують бізнес-логіку, забезпечують обробку вхідних даних, збереження результатів, шифрування інформації та логування подій.

Клас ClusteringService є центральним елементом системи кластеризації. Він реалізує метод RunClustering(), у межах якого здійснюється повний цикл обробки даних: зчитування ДТП із CSV-файлів (через CsvReaderService), визначення оптимальної кількості кластерів (за допомогою ElbowMethod), логування подій (LoggerService) та ідентифікація географічного району (DistrictLocator). Така модульність дозволяє розширювати функціональність кожного блоку без порушення цілісності системи.

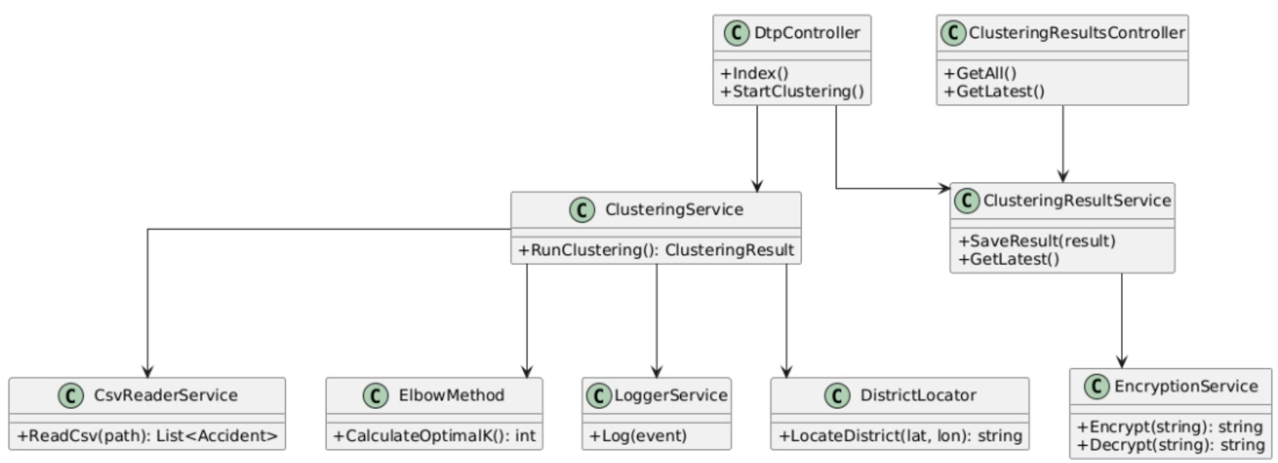


Рис. 2.2 Діаграма класів програмної системи кластеризації ДТП

Результати кластеризації обробляються окремим класом ClusteringResultService, що відповідає за їх збереження та надання доступу до останніх обчислень. Цей клас взаємодіє з декількома сервісами:

- CsvReaderService, що відповідає за зчитування даних з CSV-файлів;
- ElbowMethod, який реалізує метод «ліктя» для визначення оптимального значення K у K-Means;
- DistrictLocator, що використовується для встановлення району за координатами;
- LoggerService, який фіксує всі дії в системі для подальшого аналізу;
- ClusteringResultService, який забезпечує збереження отриманих результатів кластеризації та взаємодію з базою даних;
- EncryptionService, який виконує шифрування та розшифрування чутливих даних.

Контролери DtpController і ClusteringResultsController реалізують зовнішню взаємодію користувача із системою, зокрема – ініціюють кластеризацію та надають доступ до результатів. Вони викликають відповідні методи сервісів, забезпечуючи обробку запитів із фронтенду.

Зв’язки між класами реалізовані відповідно до принципів слабкого зв’язування та інверсії залежностей. Контролери не реалізовують логіку напряму, а делегують її відповідним сервісам, що відповідає сучасним підходам до побудови багаторівневих архітектур.

Для моделювання динамічної взаємодії об'єктів на різних етапах обробки даних було створено декілька діаграм кооперації (послідовностей), які деталізують логіку виклику методів.

Перший сценарій (рис. 2.3) описує процес ініціації кластеризації. Користувач надсилає запит через контролер DtpController, який передає його до ClusteringService. Далі відбувається послідовний виклик служб читання даних, розрахунку оптимального K, визначення районів та збереження результатів.

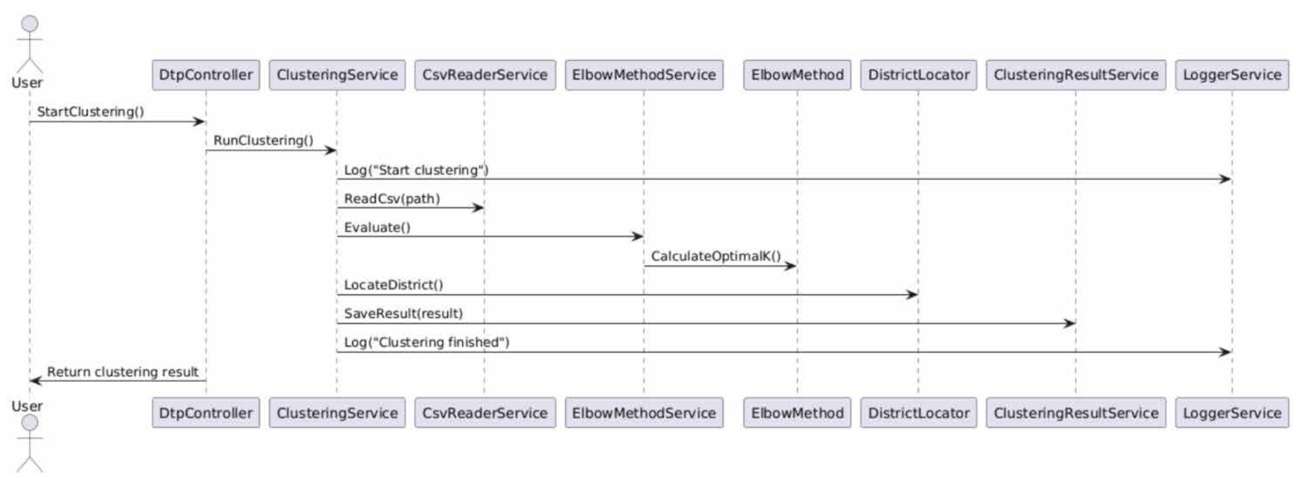


Рис. 2.3 Діаграма кооперації для сценарію запуску кластеризації ДТП

Другий сценарій (рис. 2.4) ілюструє процес отримання та розшифрування збережених результатів кластеризації. Запит надсилається через ClusteringResultsController, а далі відповідний результат обробляється за допомогою ClusteringResultService і EncryptionService перед поверненням користувачу.

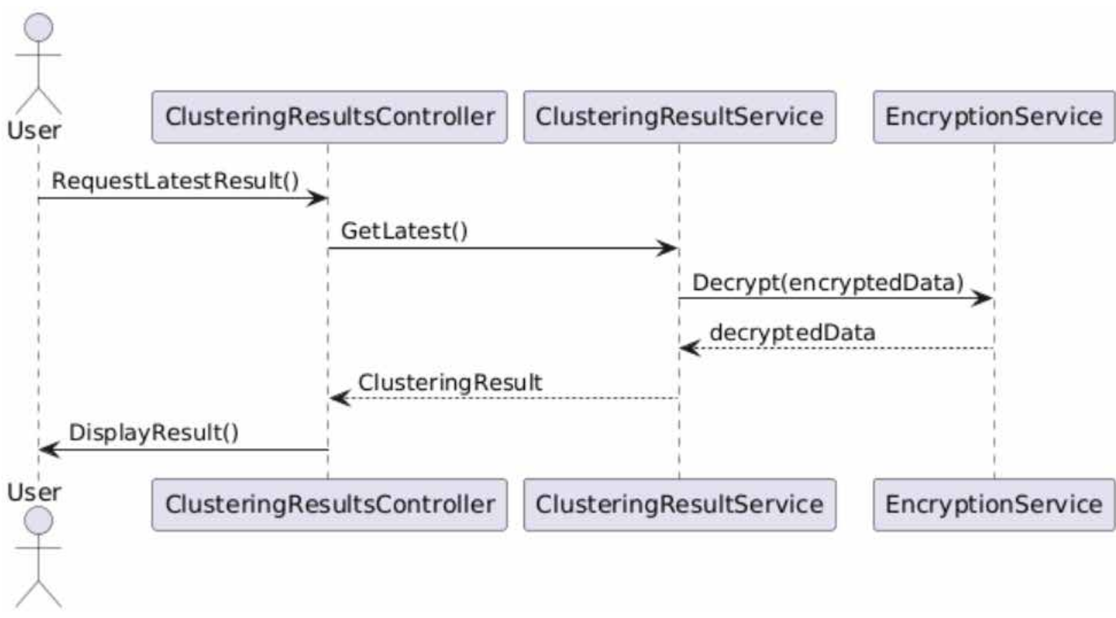


Рис. 2.4 Діаграма кооперації для сценарію перегляду результатів кластеризації

Третій сценарій (рис. 2.5) описує механізм логування дій. Будь-який сервіс, зокрема ClusteringService, може викликати метод LoggerService.Log() для фіксації події. Це забезпечує повну трасування та прозорість функціонування системи, що є важливим для безпеки та відлагодження.

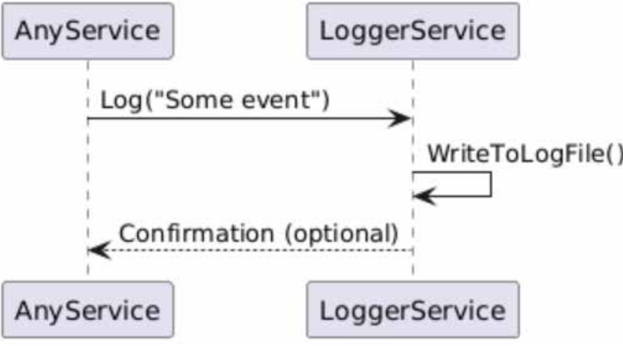


Рис. 2.5 Діаграма кооперації для сценарію логування подій системи

Запропонована структурна та поведінкова модель дозволяє забезпечити розширюваність, повторне використання компонентів та відповідність принципам SOLID, що є запорукою стабільного розвитку програмного продукту. Візуалізація кооперації сприяє глибшому розумінню взаємодії об’єктів на різних етапах роботи системи та підтверджує її архітектурну цілісність.

2.3 Діаграма пакетів

Для формалізації архітектурної побудови програмного забезпечення кластеризації дорожньо-транспортних пригод (ДТП) у місті Києві використано діаграму пакетів, яка відображає структуру системи у вигляді сукупності логічно ізольованих модулів (пакетів) та залежностей між ними. Така візуалізація дозволяє ефективно описати розподіл відповідальності між складовими частинами проєкту, а також підкреслити модульність і масштабованість архітектури [14].

Діаграма пакетів (рис. 2.6) охоплює ключові компоненти, згруповані відповідно до їх функціонального призначення: інтерфейс користувача, кластеризаційна логіка, сервісні функції, безпекові засоби та доступ до даних.

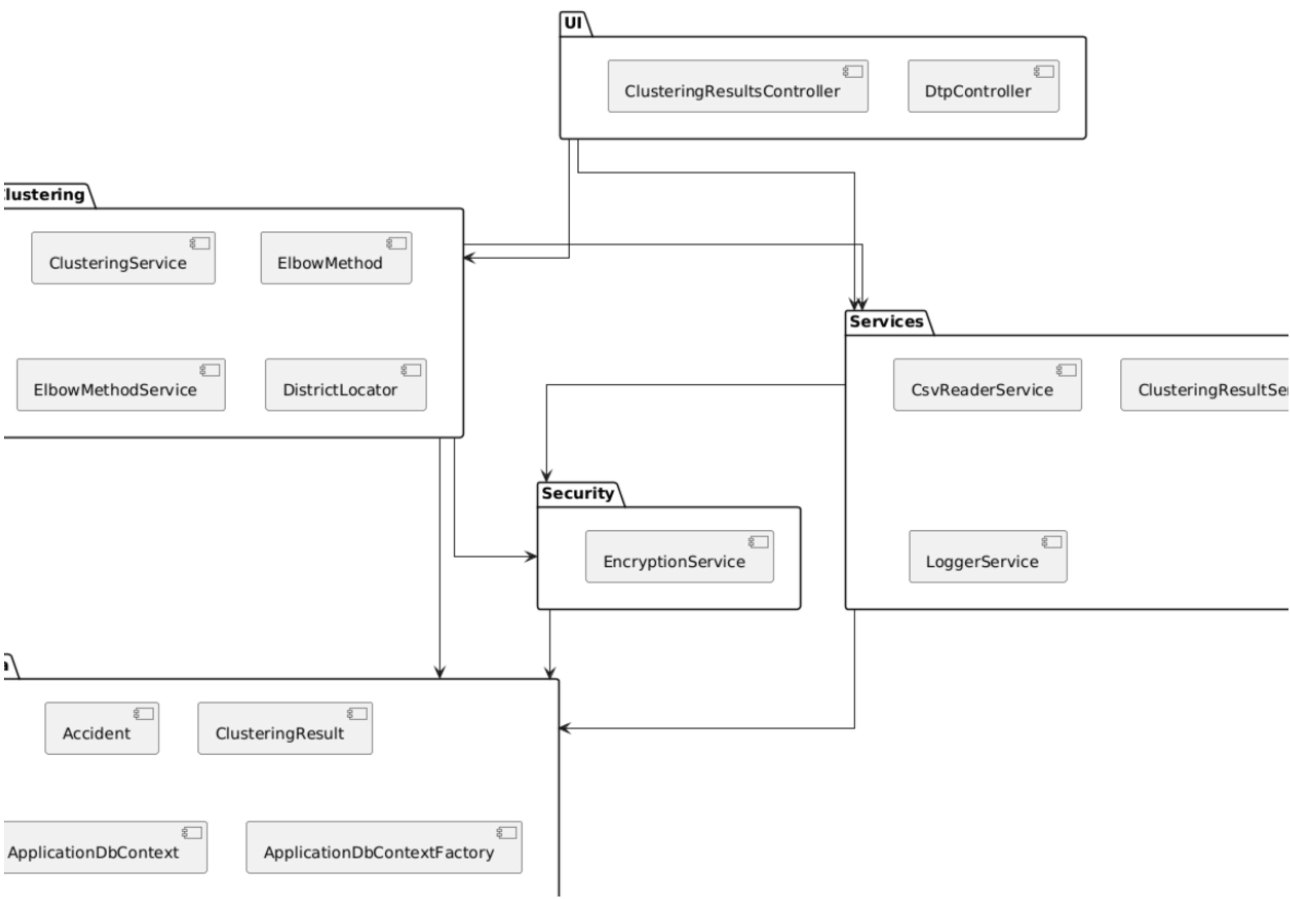


Рис. 2.6 UML-діаграма пакетів програмної системи кластеризації ДТП

Структуровану інформацію щодо вмісту кожного пакету подано у таблиці 2.1.

Таблиця 2.1

Структура програмної системи кластеризації ДТП за пакетами

№	Назва пакета	Опис функціоналу	Основні класи / сервіси	Залежності
1	UI	Реалізує контролери користувацького інтерфейсу для керування кластеризацією та перегляду результатів.	DtpController, ClusteringResultsController	Clustering, Services
2	Clustering	Втілює бізнес-логіку: кластеризація, оптимізація параметрів, геолокація.	ClusteringService, ElbowMethod, DistrictLocator	Services, Data, Security
3	Services	Містить допоміжні сервіси для роботи з файлами, логуванням і базою даних.	CsvReaderService, ClusteringResultService, LoggerService	Data, Security
4	Security	Забезпечує інформаційну безпеку шляхом шифрування та дешифрування.	EncryptionService	Data (непряма залежність)
5	Data	Визначає моделі предметної області та конфігурацію доступу до БД.	Accident, ClusteringResult, ApplicationDbContext	–

Архітектурне рішення, представлене на діаграмі пакетів, дозволяє досягти високого ступеня логічної ізоляції між компонентами, що спрощує модифікацію та супровід системи. Зокрема, сервіси зчитування, логування, кластеризації та обробки результатів винесені у окремі пакети, що підтримує принципи інверсії залежностей і сприяє реалізації тестованих і незалежних модулів.

Поділ системи за пакетами відповідає підходам побудови багаторівневих застосунків, зокрема – трирівневій архітектурі (presentation, business logic, data access), і може бути масштабований у разі розширення функціональності або змін у вимогах.

2.4 Діаграма компонентів

З метою представлення архітектурної організації програмного забезпечення було побудовано діаграму компонентів, яка відображає структуру високорівневих модулів системи та взаємозв’язки між ними. Така діаграма дозволяє наочно продемонструвати розподіл відповідальності, залежності між частинами системи та загальну логіку взаємодії між компонентами.

На рисунку 2.7 зображено UML-діаграму компонентів системи кластеризації дорожньо-транспортних пригод (ДТП), що реалізована за принципами багаторівневої архітектури. Компоненти згруповано відповідно до їх функціонального призначення в логічні блоки: інтерфейс користувача, бізнес-логіка, сервіси, доступ до даних і безпека [5, 14].

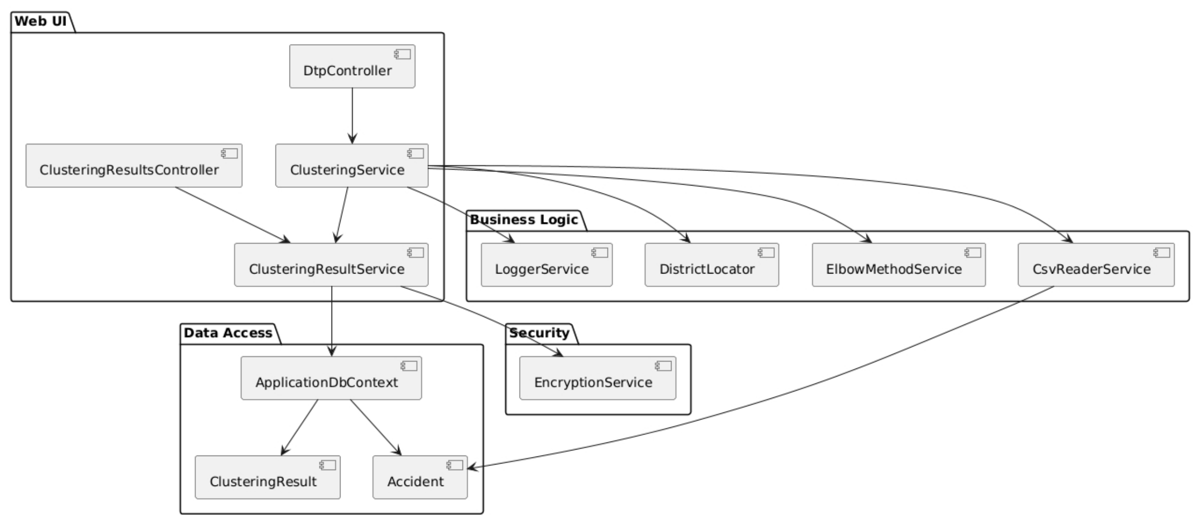


Рис. 2.7 UML-діаграма компонентів програмної системи кластеризації ДТП

До компоненту Web UI належать контролери, що реалізують інтерфейс взаємодії користувача із системою через HTTP-запити:

1. DtpController – відповідає за запуск алгоритму кластеризації;
2. ClusteringResultsController – надає доступ до історії результатів кластеризації та їх виведення, взаємодіють із відповідними сервісами бізнес-логіки (ClusteringService) і сервісами збереження (ClusteringResultService), як видно зі стрілок залежності на діаграмі.

Компонент Business Logic є центральним логічним блоком, який містить класи, що реалізують обробку даних і виконання алгоритмів:

1. ClusteringService виконує основні дії з обробки та групування даних;
2. ElbowMethodService та ElbowMethod реалізують логіку обчислення оптимальної кількості кластерів;
3. DistrictLocator визначає адміністративний район на основі координат;
4. LoggerService – веде журнал дій системи (наприклад, запуск кластеризації).

До службових сервісів належать компоненти Services, які забезпечують додаткову функціональність:

1. CsvReaderService – імпортує дані ДТП із CSV-файлів;
2. ClusteringResultService – відповідає за збереження та отримання результатів кластеризації;
3. LoggerService – логування усіх ключових дій системи, які спираються на компоненти доступу до даних та безпеки.

Компонент Security представлений сервісом EncryptionService, який забезпечує шифрування та дешифрування результатів кластеризації перед збереженням або передачею. Його використання дозволяє дотримуватися вимог інформаційної безпеки та конфіденційності даних.

Компонент Data Access доступу до даних включає:

1. ApplicationDbContext – контекст Entity Framework Core для взаємодії з базою даних;
2. ClusteringResult та Accident – моделі, що представляють результат кластеризації та події ДТП, є базовим для всіх сервісів, що взаємодіють з БД, і забезпечує централізовану роботу з даними.

Діаграма компонентів демонструє логічно поділену архітектуру системи з чітким розмежуванням обов’язків між модулями. Ц сприяє масштабованості, гнучкості та спрощує супровід програмного забезпечення. Окрема увага приділена забезпеченню слабкого зчеплення між компонентами, що дозволяє змінювати або замінювати окремі модулі без порушення цілісності системи.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління інформаційною базою

У межах розробленої системи кластеризації даних про дорожньо-транспортні пригоди (ДТП) було реалізовано інформаційну базу, яка ґрунтується на реляційній моделі даних. Як основну систему управління базами даних обрано PostgreSQL, що забезпечує оптимальний баланс між продуктивністю, масштабованістю та функціональною гнучкістю.

Вибір саме цієї СУБД зумовлений кількома ключовими факторами. Насамперед, PostgreSQL підтримує обробку географічних даних, що критично важливо для аналізу просторових параметрів ДТП. Інтеграція з розширенням PostGIS дозволяє використовувати геолокаційні координати для кластеризації та візуалізації на карті. Система також демонструє стабільну роботу з великими обсягами записів, що особливо важливо під час імпорту історичних даних із CSV-файлів.

Важливою перевагою є сумісність PostgreSQL із технологіями ASP.NET Core, зокрема підтримка через ORM-бібліотеки, такі як Entity Framework Core. Це дозволяє легко реалізовувати CRUD-операції, працювати з міграціями та зв’язками між таблицями на рівні об’єктно-орієнтованої моделі. Висока надійність і відповідність ACID-властивостям гарантують цілісність даних навіть у разі збоїв або нештатних ситуацій.

База даних проєкту включає логічно організовану структуру таблиць, кожна з яких представляє сутності предметної області. У таблиці Users зберігаються облікові записи користувачів, які ініціюють процеси кластеризації або взаємодіють із системою. З нею пов’язані таблиці ClusteringResults — що фіксують метадані кластеризацій — та Logs, де реєструються події, спричинені користувачем.

Кожен результат кластеризації (таблиця ClusteringResults) може бути пов’язаний із численними подіями ДТП, які зберігаються в таблиці Accidents. Такий зв’язок реалізується за допомогою зовнішніх ключів і відповідає відношенню типу «один до багатьох». Крім того, таблиця Accidents містить розширений набір полів для опису просторових, часових та описових характеристик інцидентів. Це дозволяє формувати аналітичні вибірки, групувати події за кластером або регіоном і будувати візуалізацію.

На рисунку 3.1 подано ER-діаграму, що відображає структуру основних таблиць бази даних та їхні зв’язки. Зокрема, чітко видно взаємозв’язки між користувачами, результатами кластеризації, подіями ДТП і логами — з урахуванням зовнішніх ключів і кардинальностей.

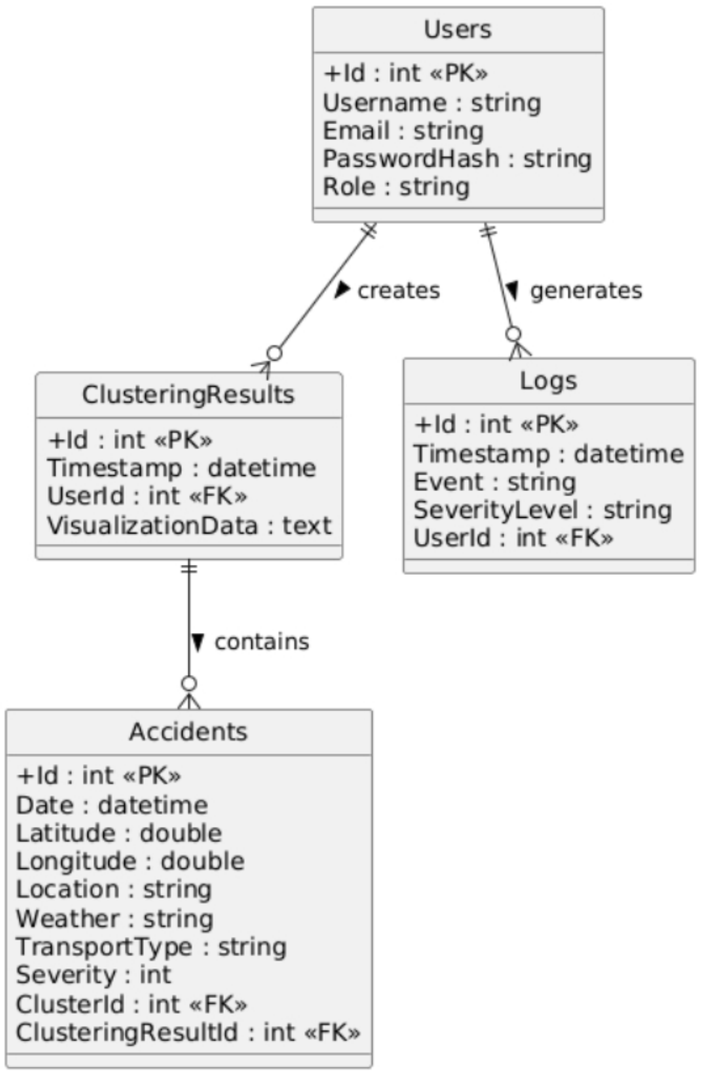


Рис. 3.1 ER-діаграма взаємозв’язків між таблицями бази даних PostgreSQL

Запропонована модель забезпечує повну структурованість даних і гнучкість їх використання як для поточних, так і для аналітичних задач. Завдяки такій архітектурі можлива подальша розширюваність системи, зокрема впровадження нових ролей користувачів, форм звітності або алгоритмів обробки.

3.2 Розробка інформаційної бази

Створення бази даних у межах проєкту реалізовано з використанням підходу Code First, що передбачає генерацію всієї структури бази безпосередньо з моделей, реалізованих мовою програмування C#. У цьому процесі було застосовано фреймворк Entity Framework Core, який дозволяє описати модель даних через класи, а далі автоматично згенерувати схему бази даних за допомогою механізму міграцій. Такий підхід сприяє кращому контролю версій і дозволяє уникнути помилок, властивих ручному написанню SQL-інструкцій.

Після створення моделей класів сутностей було ініційовано початкову міграцію за допомогою команди Add-Migration InitialCreate. В результаті було згенеровано відповідний файл міграції, що містить SQL-команди створення таблиць, їхніх зв’язків, індексів та обмежень. Реальне створення або оновлення бази даних здійснюється через метод context.Database.Migrate(), який викликається під час запуску програми.

Для забезпечення підтримки сценаріїв командного запуску або розгортання бази без запуску основного застосунку, було додано спеціалізований клас ApplicationDbContextFactory. Він реалізує інтерфейс IDesignTimeDbContextFactory<TContext> і дозволяє створювати об’єкт контексту бази даних без активації основної логіки програми.

Налаштування схем таблиць в основному здійснюється через атрибути або Fluent API, що задають типи стовпців, зовнішні ключі та зв’язки між таблицями. Це дозволяє дотримуватись правил нормалізації, зберігаючи логічну цілісність структури.

На рисунку 3.2 подано фрагмент початкової міграції, яка демонструє автоматичну генерацію структури бази даних на основі оголошених C#-моделей.

```
using Microsoft.EntityFrameworkCore.Migrations;
namespace DtpClusteringApp.Migrations
{

    public partial class InitialCreate : Migration
    {

        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Accidents",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Latitude = table.Column<double>(type: "float", nullable: false),
                    Longitude = table.Column<double>(type: "float", nullable: false),
                    TimeOfDay = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    WeatherConditions = table.Column<string>(type: "nvarchar(max)", nullable
                    RoadType = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    VehicleType = table.Column<string>(type: "nvarchar(max)", nullable: true
                    Severity = table.Column<string>(type: "nvarchar(max)", nullable: true),
                    SpeedLimit = table.Column<int>(type: "int", nullable: false),
                    TrafficControl = table.Column<string>(type: "nvarchar(max)", nullable: t
                    Date = table.Column<DateTime>(type: "datetime2", nullable: false)
```

Рис. 3.2 Фрагмент початкової міграції InitialCreate у проєкті на основі EF Core

Окрім створення структури, важливим етапом є імпорт початкових даних про ДТП. Ця процедура реалізована за допомогою спеціального сервісу CsvReaderService, який відповідає за обробку файлів формату CSV. Оскільки дані передаються в зашифрованому вигляді, попередньо реалізовано дешифрування за допомогою EncryptionService. Далі сервіс виконує розбір CSV-файлу за допомогою бібліотеки CsvHelper, перетворюючи рядки таблиці на об'єкти типу Accident.

Структура CSV-файлу містить такі дані, як координати місця події, погодні умови, час доби, тип транспорту, обмеження швидкості та інші параметри. У процесі обробки частина полів нормалізується або обчислюється програмно: виділяється година, класифікується погодна умова, визначається день тижня, а також встановлюється, чи була подія фатальною. Нижче наведено приклад обчислення:

Приклад коду обробки:

```
record.Hour = ParseHour(record.TimeOfDay);
record.Weather = ParseWeather(record.WeatherConditions).ToString();
record.DayOfWeek = ParseDayOfWeek(record.TimeOfDay);
record.IsFatal = record.Severity.ToLower().Contains("fatal");
```

Після обробки об'єкти передаються до бази даних методом AddRange() через відповідний сервіс або контролер. У поточній реалізації не передбачено перевірки на дублювання записів, тому можливе повторне внесення однакових записів при багаторазовому імпорті. Це обмеження планується усунути в майбутньому шляхом додавання перевірок унікальності за ключовими полями.

Для забезпечення стабільної продуктивності системи за великої кількості записів реалізовано **оптимізацію запитів**. Зокрема, на ключові поля — такі як Latitude, Longitude, Date, WeatherConditions — було накладено індекси, що суттєво прискорює фільтрацію й агрегацію даних. Це особливо актуально при виконанні запитів за часовими або географічними параметрами.

Наразі не впроваджено використання AsNoTracking() або пагінації, проте в планах — їх інтеграція задля підвищення ефективності обробки масивів даних. Це дозволить знизити навантаження на контекст і уникнути непотрібного збереження станів при читанні великих таблиць.

Таким чином, обрана архітектура бази даних у поєднанні з інструментами EF Core забезпечує керованість, гнучкість та підготовленість до подальшого масштабування й адаптації системи.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Проєктування та реалізація прикладного програмного забезпечення для кластеризації даних про дорожньо-транспортні пригоди потребує ретельного вибору інструментарію, який забезпечує відповідність вимогам продуктивності, масштабованості, розширюваності та інтеграції з сучасними аналітичними сервісами. У даному проєкті було сформовано технологічний стек, що охоплює

компоненти серверної логіки, клієнтського відображення, роботи з даними та реалізації кластеризаційних алгоритмів. Рішення про вибір того чи іншого інструмента приймалося на основі поєднання таких критеріїв, як функціональність, відповідність архітектурній моделі, наявність відкритого коду, сумісність із .NET середовищем та підтримка просторово-часової обробки інформації.

Узагальнена характеристика використаних інструментів наведена в таблиці 3.1.

Таблиця 3.1

Вибрані інструменти для реалізації програмного забезпечення

№	Інструмент / Технологія	Призначення та обґрунтування використання
1	C# / .NET 8	Основна мова та платформа реалізації серверної логіки; забезпечує об’єктно-орієнтовану структуру, асинхронність, інтеграцію з EF Core та підтримку Web API
2	PostgreSQL + PostGIS	Реляційна СУБД з підтримкою геоданих; використовується для зберігання ДТП, результатів кластеризації та облікових записів; забезпечує високу продуктивність, транзакційність та розширюваність
3	Entity Framework Core	ORM-технологія для реалізації Code First міграцій; дозволяє автоматизовано створювати та підтримувати структуру БД відповідно до моделей програми
4	Accord.NET	Бібліотека для машинного навчання; реалізує алгоритм кластеризації K-Means; інтегрується з C# без зовнішніх залежностей
5	Leaflet.js	JavaScript-бібліотека для відображення кластерів на інтерактивній карті з використанням маркерів, підказок і групування точок
6	CsvHelper	Бібліотека для імпорту CSV-файлів у форматі моделей C#; використовується для попередньої обробки вхідних даних
7	Visual Studio 2022	Основне середовище розробки з підтримкою .NET 8, Razor Pages, запуску Web API та інтеграції з системою керування базами даних
8	pgAdmin	Інструмент адміністрування PostgreSQL; використовується для перегляду структури БД, моніторингу запитів та ручного редагування таблиць
9	Google Maps API (опц.)	Альтернативна бібліотека для візуалізації кластерів на карті з додатковими можливостями кастомізації картографічного шару

Обраний інструментарій дозволяє забезпечити повний цикл обробки даних: від імпорту та кластеризації до інтерактивного відображення результатів. Така архітектура є гнучкою та масштабованою, що створює можливості для подальшого розширення, зокрема додавання нових алгоритмів кластеризації, покращення візуалізації або інтеграції з іншими геоінформаційними системами.

Підсумовуючи, вибір засобів розробки є технічно виваженим і забезпечує ефективну реалізацію програмного забезпечення, орієнтованого на обробку та аналіз просторово-часових даних у контексті безпеки дорожнього руху.

3.4 Алгоритмізація та програмування програмних модулів

Розробка програмного забезпечення для кластеризації дорожньо-транспортних пригод (ДТП) передбачала створення чіткої структури обробки даних, впровадження ефективного алгоритму кластеризації, а також побудову інфраструктури програмної реалізації, орієнтованої на модульність і розширюваність.

Основу вхідних даних становить CSV-файл generated_accidents.csv, що містить синтетичні записи про ДТП з типово притаманними ознаками: координати, час події, погодні умови, тип дороги, тип транспортного засобу, тяжкість пригоди тощо. Для геопросторової прив'язки також було використано файл Kyiv_districts.geojson, що визначає межі районів Києва. Приклад структури одного запису у форматі JSON представлено нижче:

```json
{
  "latitude": 50.4501,
  "longitude": 30.5234,
  "timestamp": "2023-11-20T08:15:00",
  "weather": "rain",
  "vehicle_type": "car",
  "severity": "medium"
}
```

Дані імпортуються до бази PostgreSQL за допомогою ORM Entity Framework Core. Таблиця 3.2 узагальнює атрибути вхідного датасету:

Таблиця 3.2

Основні атрибути записів про ДТП

№	Назва поля	Опис	Приклад значення
1	Latitude	Географічна широта	50.4361392
2	Longitude	Географічна довгота	30.4737296
3	Date	Дата події	2023-11-20
4	TimeOfDay	Час доби	08:15
5	DayOfWeek	День тижня	Monday
6	WeatherConditions	Погодні умови	Rain
7	VehicleType	Тип транспортного засобу	Car
8	Severity	Тяжкість аварії	Medium
9	SpeedLimit	Обмеження швидкості	50
10	TrafficControl	Наявність засобів регулювання руху	TrafficLight

Для генерації даних було використано спеціальну програму, яка формує записи про ДТП на основі географічної інформації про дорожню мережу Києва, отриманої з відкритих джерел у форматі GeoJSON (зокрема, з OpenStreetMap). Програма обирає координати випадкових точок із реальної дорожньої мережі, що дозволяє моделювати місця виникнення ДТП максимально наближено до реальних умов (рис. 3.3).

Date	Time	Latitude	Longitude	Weather	Vehicle	Severity
2023-11-20	08:15	50.4501	30.5234	Rain	Car	Medium
2023-11-21	18:05	50.4510	30.5241	Clear	Bus	Low

Рис. 3.3 Фрагмент вхідного датасету у форматі CSV, який використовується для кластеризації

Кожен запис у згенерованому наборі даних включає такі характеристики: координати широти та довготи, дату та час події (з випадковим розподілом у межах періоду з 2020 року по сьогодення), погодні умови, тип дороги, тип транспортного засобу, рівень тяжкості аварії, обмеження швидкості, тип регулювання руху (наприклад, світлофор чи дорожній знак), а також день тижня.

Такий комплекс атрибутів дозволяє враховувати не лише просторові, а й часові й контекстуальні фактори, які суттєво впливають на виникнення ДТП.

Згенеровані дані зберігаються у CSV-файлі, який потім імпортується у базу даних PostgreSQL з використанням ORM-системи Entity Framework Core. Завдяки цьому забезпечується ефективне зберігання, обробка та подальший аналіз даних.

Розробка програмного забезпечення передбачає не лише створення інтерфейсної частини або бази даних, а й побудову внутрішньої логіки, що реалізує основні функціональні можливості системи. Важливу роль у цьому процесі відіграє алгоритмізація – формалізація способів розв’язання поставлених завдань – та їх подальше програмування в межах модульної архітектури застосунку. Кожен модуль виконує окрему, чітко визначену функцію, що відповідає принципам структурованого, розширюваного та безпечного програмування.

Кластеризацію реалізовано із застосуванням бібліотеки Accord.NET. Основні кроки роботи алгоритму:

1. ініціалізація кількості кластерів K (вказується вручну або обчислюється автоматично);
2. вибір випадкових центроїдів;
3. призначення кожного запису до найближчого центроїда;
4. перерахунок центроїдів як середнього значення координат;
5. повторення до досягнення стабільності кластерів.

Налаштовуються такі параметри: кількість кластерів, максимальна кількість ітерацій, критерії зупинки. Вхідні дані нормалізуються і кодуються у вигляді числових векторів, включаючи координати, тип ТЗ, погодні умови тощо.

Результати кластеризації зберігаються у таблиці ClusteringResults, що дозволяє проводити аналітику за часовими проміжками, районами міста тощо.

Схема роботи алгоритму K-Means наведено на рисунку 3.4.

Рис. 3.4 Схема роботи алгоритму K-Means

Вся логіка алгоритму інкапсульована в окремий клас ClusteringService. Основні функції сервісу кластеризації (ClusteringService):

- обробка CSV-файлів;
- нормалізація даних;
- запуск K-Means;
- збереження результатів;
- формування GeoJSON для інтерактивної мапи.

Цей забезпечує модульність та тестованість системи. Сервіс реалізовано у вигляді залежності, яку можна легко вбудовувати до Web API або інтерфейсної частини.

Веб-сервер реалізовано з використанням REST API. API-Контролери дозволяють:

- запускати кластеризацію за заданими параметрами;
- завантажувати дані;

- переглядати кластери у вигляді списку чи мапи;
- отримувати GeoJSON для фронтенда.

Контролери відповідають за зв’язок між клієнтом і бізнес-логікою.

В системі реалізовано механізми безпеки та аудиту:

- LoggerService – журналювання всіх подій;
- EncryptionService – шифрування та дешифрування даних для забезпечення конфіденційності при обробці CSV-файлів.

Ці компоненти формують основу для майбутньої автентифікації та ролей користувачів.

Для відображення результатів кластеризації використовується інтерактивна мапа з бібліотекою Leaflet.js. Вивід результатів здійснюється у вигляді маркерів різного кольору, що позначають кластери на карті Києва (рис. 3.5).

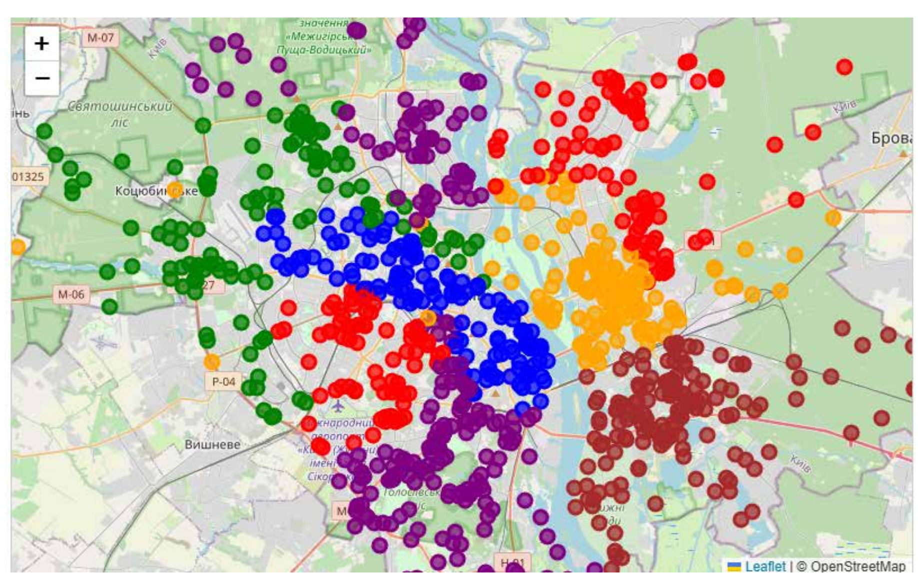


Рис. 3.5 Візуалізація кластерів ДТП на мапі Leaflet.js

Розроблений підхід до алгоритмізації та програмної реалізації забезпечує комплексне оброблення даних про ДТП, ефективну кластеризацію за просторово-часовими ознаками та наочну візуалізацію. Система структурована, масштабована та придатна до інтеграції з реальними джерелами даних у майбутньому.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування є невіддільною складовою процесу розробки програмного забезпечення, яка дозволяє забезпечити його коректну функціональність, стабільність і відповідність заявленим вимогам. У процесі розробки інформаційної системи для кластеризації дорожньо-транспортних пригод (ДТП) було здійснено комплексне тестування на різних рівнях реалізації. Метою було виявлення потенційних помилок, логічних хиб, а також оцінювання продуктивності та точності алгоритмів.

На етапі перевірки функціональності окремих компонентів системи застосовано модульне тестування із використанням бібліотеки xUnit. Зокрема, протестовано перетворення рядків у дату та час, визначення погодних умов, логіку шифрування та дешифрування даних, а також окремі обчислення, пов’язані з роботою алгоритму кластеризації K-Means. Модулі перевірялися як на коректних, так і на граничних вхідних даних.

Для верифікації взаємодії між компонентами застосовано інтеграційне тестування. Особливу увагу приділено правильності обміну даними між імпортом CSV, сервісом кластеризації, базою даних та клієнтським інтерфейсом. Це дозволило переконатися в коректності передачі, збереження та візуалізації даних.

У рамках функціонального тестування змодельовано ключові сценарії використання системи: імпорт файлів, запуск кластеризації з різними параметрами, перегляд результатів на мапі, повторне використання історичних даних тощо. Усі сценарії супроводжувалися моніторингом логування, перевіркою коректності виводу та стійкості до повторного використання.

Особливу увагу приділено перевірці роботи алгоритму K-Means на тестових наборах даних (табл. 4.1). Наприклад, у випадку з розподілом двох груп

точок у різних районах (Оболонь та Позняки) алгоритм коректно сформував два кластери. У іншому випадку – із трьома групами точок, що мали однакову геолокацію, але різну погоду – спостерігалося коректне групування за додатковими параметрами. При навантажувальному тестуванні з 100 тисячами точок система продемонструвала стабільну роботу без втрати даних та збоїв.

Таблиця 4.1

Приклади тестування алгоритму K-Means у системі кластеризації ДТП

№	Назва прикладу	Умови тесту	Очікуваний результат	Результат виконання
1	Два центри в протилежних районах	Два набори точок – Оболонь і Позняки; K=2	Алгоритм формує два чітко розділені просторові кластери	Чітке розмежування, без помилок
2	Три змішані кластери з варіативною погодою	Однакові координати, різна погода і час доби; K=3	Кластери формуються з урахуванням не тільки координат, а й погодних характеристик	Класифікація відповідає очікуванням
3	Навантажувальний тест	Генерація 100 000 точок; перевірка швидкості і стабільності	Алгоритм завершує кластеризацію без помилок, обробка виконується стабільно	Успішно оброблено без втрат даних

Результати тестування засвідчили відповідність системи вимогам за кількома ключовими характеристиками:

- стабільність – усі перевірені сценарії виконувалися без помилок навіть за умов повторного використання функцій. Витоків пам’яті або збоїв не зафіксовано.
- точність кластеризації – результати узгоджуються з очікуваними географічними зонами. Центроїди кластерів мали логічне положення, що підтверджувалося аналізом координат.
- продуктивність – обробка масиву у 10 000 записів тривала не більше 2–3 секунд, що відповідає прийнятному рівню для локальних обчислень.

– безпека – жодна інформація не зберігалася у відкритому вигляді. Шифрування й дешифрування реалізовані коректно. Уся обробка супроводжувалась системним логуванням.

Для покращення аналізу даних у системі реалізовано графічну візуалізацію результатів у вигляді діаграм, які відображають:

– кількість ДТП по районах – гістограма з розподілом подій за адміністративними одиницями (рис. 4.1), яка дозволяє виявити найбільш проблемні території з точки зору безпеки руху;

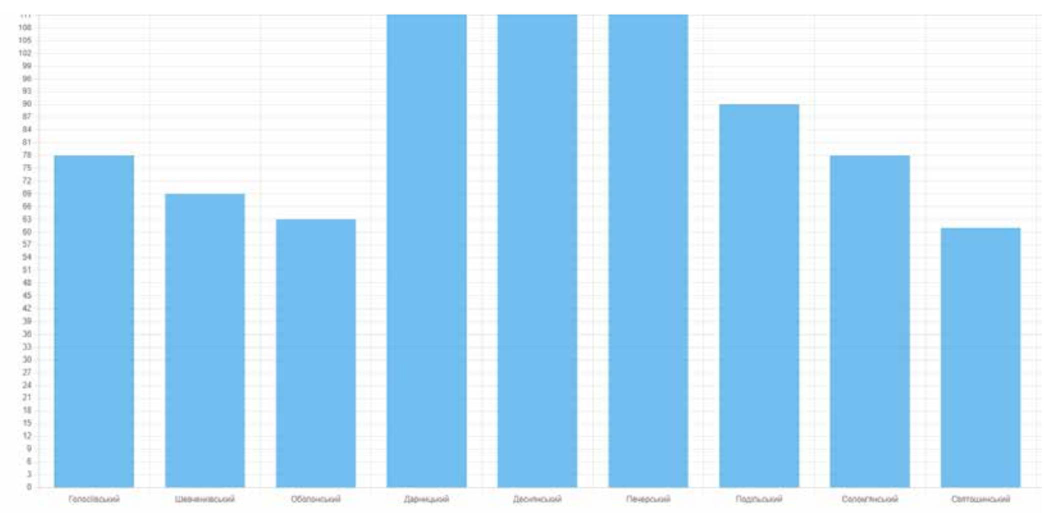


Рис. 4.1 Графік кількості ДТП по районах

– розподіл ДТП за місяцями – графік, що демонструє кількість подій у кожному місяці року (рис. 4.2), що дозволяє виявити сезонні закономірності у зміні рівня аварійності, що важливо для планування ресурсів і превентивних заходів;

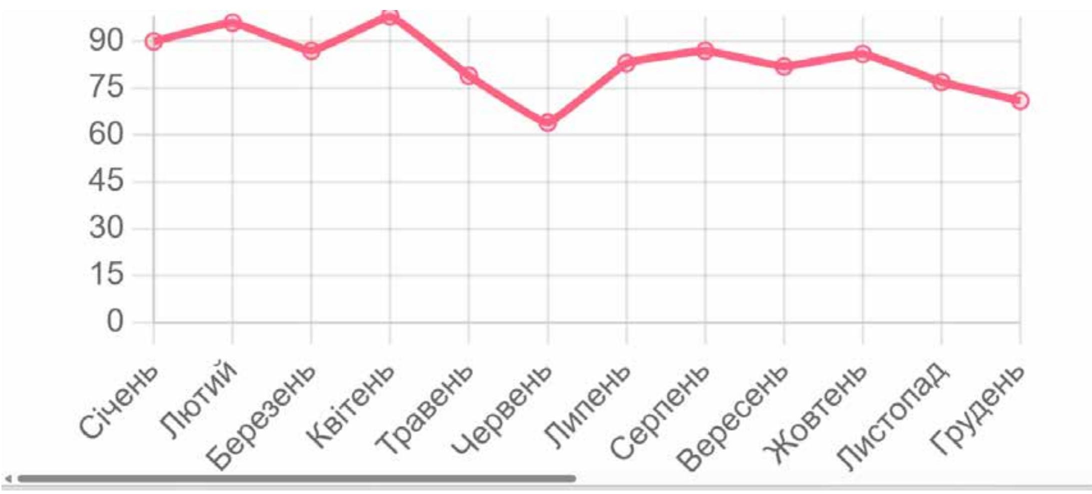


Рис. 4.2 Графік ДТП за місяцями

– розподіл ДТП за годинами доби – лінійний або стовпчиковий графік, який відображає інтенсивність ДТП упродовж доби (рис. 4.3), дає змогу виявити пікові години аварійності, що є важливим для оптимізації роботи дорожньої служби та організації безпеки руху.

Рис. 4.3 Графік ДТП за годинами доби

Кожен графік має інтерактивну природу, підтримує фільтрацію за параметрами (кластер, день тижня, погодні умови) й тісно інтегрується з мапою Leaflet.js, яка надає геопросторову інтерпретацію кластеризації.

Інтерактивність графіків дозволяє користувачу фільтрувати дані за певними ознаками (погода, кластер, день тижня), що покращує якість аналізу та

прийняття рішень. Візуалізація графіків разом з інтерактивною мапою кластерів на базі Leaflet.js створюють цілісну аналітичну панель для дослідження аварійності у місті.

4.2 Вимоги до апаратного та програмного забезпечення

Для забезпечення ефективного функціонування інформаційної системи кластеризації дорожньо-транспортних пригод необхідно враховувати апаратні та програмні характеристики середовища її використання. Системні вимоги охоплюють характеристики серверного обладнання, клієнтських пристроїв, інструментів виконання та бази даних. Правильно визначені вимоги гарантують сумісність, надійність і масштабованість застосунку.

Системні вимоги до роботи програмного забезпечення. У табл. 4.2 наведено порівняльні характеристики мінімальних і рекомендованих системних вимог.

Таблиця 4.2

Системні вимоги до розгортання програмного забезпечення

Компонент	Мінімальні вимоги	Рекомендовані вимоги
Операційна система	Windows 10 / Ubuntu 20.04	Windows 11 / Ubuntu 22.04
Процесор	2 ядра, 2.0 GHz	4 ядра, 3.0 GHz або Intel Xeon / AMD EPYC
Оперативна пам’ять	4 ГБ	8–16 ГБ
Накопичувач	1 ГБ вільного місця	SSD на 100 ГБ з підтримкою резервного копіювання
Платформа виконання	.NET 8 SDK + Runtime	.NET 8 SDK + Runtime
СУБД	PostgreSQL 13+	PostgreSQL 15+
Інтерактивна мапа	LeafletJS + OpenStreetMap	LeafletJS + Mapbox / Google Maps API
Браузер	Chrome 90+, Firefox 88+, Edge 91+	Chrome останньої версії

Вимоги до клієнтських пристроїв. Для локального використання системи на невеликих наборах даних достатньо стандартних офісних ПК або ноутбуків. В табл. 4.3 наведено рекомендовану мінімальну конфігурацію.

Таблиця 4.3

Мінімальні вимоги до клієнтського обладнання

Компонент	Мінімальна конфігурація
Процесор	Intel Core i3 або AMD Ryzen 3
Оперативна пам’ять	4 ГБ
Накопичувач	20 ГБ вільного місця на SSD або HDD
Графічний адаптер	Інтегрований, з підтримкою WebGL
Підключення до мережі	Доступ до Інтернету для API-запитів

Сумісність програмного середовища. Програмне забезпечення побудовано на кросплатформенній технології .NET 8, що дозволяє запуск як на Windows-, так і на Unix-системах. Основні вимоги до середовища наведено в табл. 4.4.

Таблиця 4.4

Програмні вимоги до середовища виконання

Компонент	Вимоги
Операційна система	Windows 10/11, Ubuntu 20.04+, macOS 11+
.NET Runtime	.NET 8 SDK та Runtime
Система управління базами	PostgreSQL 14 або новіша
Бібліотека доступу до БД	Npgsql.EntityFrameworkCore.PostgreSQL
Браузер	Chrome, Firefox, Edge

Рекомендації для серверної інфраструктури. Для стабільної роботи системи в умовах високого навантаження або при одночасному доступі кількох користувачів рекомендується організувати серверне розгортання відповідно до параметрів, які надано у табл. 4.5.

Таблиця 4.5

Рекомендації для серверного розгортання

Параметр	Рекомендоване значення
Серверна ОС	Ubuntu Server 22.04 або Windows Server 2019/2022
Процесор	≥ 4 ядра (Intel Xeon, AMD EPYC)

Оперативна пам’ять	8–16 ГБ
Сховище	SSD ≥ 100 ГБ, підтримка резервного копіювання
БД	Окрема інсталяція PostgreSQL з можливістю бекапу
Захист	HTTPS, API токени, регулярне шифрування даних
Моніторинг	Serilog, Health Checks, системи логування
Інфраструктура	VPS або хмара (Azure, AWS, DigitalOcean)

На рис. 4.5 представлено UML-діаграму розгортання системи кластеризації ДТП, яка демонструє компоненти програмного забезпечення, їх розміщення на фізичних вузлах та взаємозв’язки між ними.

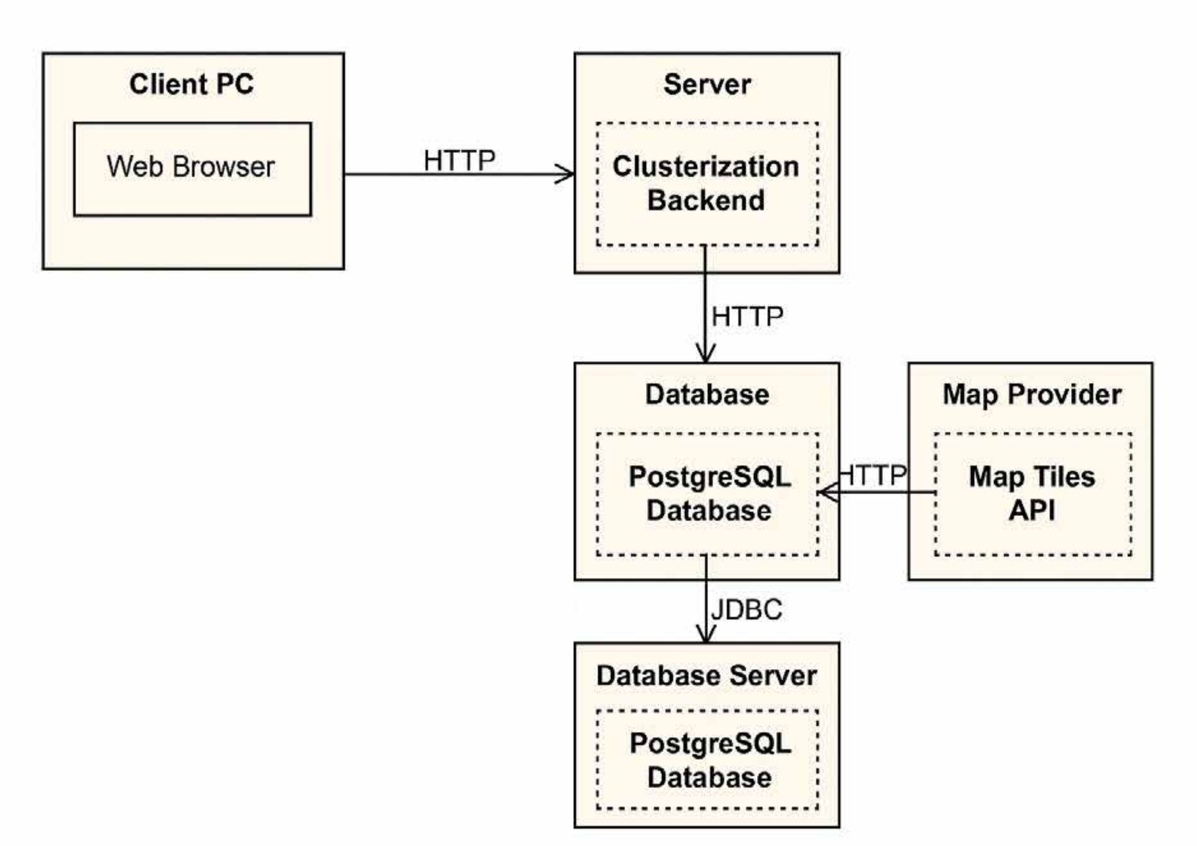


Рис. 4.5 Діаграма розгортання системи кластеризації ДТП у середовищі клієнт-сервер

Система реалізована за архітектурою клієнт-сервер. Основними вузлами є:

1. клієнтський пристрій (Client Browser) — взаємодіє з користувачем через веб-інтерфейс. Доступ до функціональності здійснюється через HTTP-запити до Web API;
2. вебсервер (ASP.NET Core App) — містить основну логіку застосунку, включно з контролерами (DtpController, ClusteringResultsController), сервісами кластеризації (ClusteringService, ClusteringResultService), імпорту даних, шифрування, логування та API-інтерфейсами;
3. база даних PostgreSQL — зберігає дані про ДТП, результати кластеризації, облікові записи користувачів та журнали подій. Інтеграція з .NET-застосунком забезпечується за допомогою ORM-бібліотеки Entity Framework Core (Npgsql).

Компоненти системи працюють у тісній взаємодії: дані, отримані з CSV-файлів, зберігаються в БД; результати кластеризації також записуються до БД і надсилаються у вебінтерфейс для візуалізації на мапі з використанням Leaflet.js.

Таким чином, розгортання програмного забезпечення базується на гнучкій, масштабованій архітектурі з чітким розмежуванням ролей клієнта, сервера й бази даних. Завдяки використанню сучасного стеку технологій і підтримці кросплатформенності, систему можливо ефективно розгортати як у локальних, так і в хмарних середовищах. Дотримання апаратно-програмних вимог забезпечує стабільність, безпеку та продуктивність застосунку на всіх рівнях його використання.

4.3 Склад інсталяційного пакету

Інсталяційний пакет розробленої системи не реалізований у вигляді класичного інсталятора (.msi або .exe), однак підготовлена збірка проєкту дозволяє виконати ручне розгортання застосунку на будь-якому сумісному комп’ютері.

Після виконання команди dotnet publish у відповідному режимі (наприклад, Release), формується набір файлів, необхідних для запуску:

– DtpClusteringApp.exe – основний виконуваний файл;

- *.dll – бібліотеки, необхідні для роботи застосунку;
- appsettings.json – конфігураційний файл із параметрами підключення до бази даних;
- wwwroot – директорія із статичними файлами (якщо використовується веб-інтерфейс);
- README.txt або інструкція (за наявності);
- додаткові ресурси, зокрема шаблони CSV-файлів, якщо вони передбачені у складі пакету.

Для запуску системи достатньо розпакувати опубліковану збірку на обрану машину. Перед першим запуском потрібно:

- переконатися, що на комп'ютері встановлено .NET Runtime відповідної версії (наприклад, .NET 8);
- встановити систему управління базами даних PostgreSQL (рекомендована версія – 15 і вище);
- створити базу даних згідно інструкції (вручну або через міграції за допомогою dotnet ef database update);
- за необхідності – відредагувати файл appsettings.json, вказавши коректний рядок підключення до бази.

Для успішного запуску застосунку користувач має:

- мати базові технічні навички (робота з конфігураційними файлами, терміналом);
- володіти обліковими даними доступу до PostgreSQL-сервера;
- мати достатні права для створення бази даних (або доступ до вже налаштованої);
- мати доступ до системних ресурсів, якщо застосунок передбачає локальну або мережеву взаємодію.

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було розроблено прикладне програмне забезпечення для кластеризації дорожньо-транспортних пригод у місті Києві з використанням сучасних інструментів платформи .NET 8, бібліотеки Accord.NET та СУБД PostgreSQL. Основною метою дослідження було створення ефективної інформаційної системи, здатної здійснювати просторово-часовий аналіз аварійних подій з подальшою візуалізацією результатів.

У роботі реалізовано підхід *Code First* для побудови бази даних, що дало змогу гнучко керувати її структурою та забезпечити стабільну інтеграцію з логікою застосунку. Імпорт даних із зашифрованих CSV-файлів реалізовано із використанням механізмів дешифрування та парсингу, що дозволяє обробляти великі обсяги записів з високим рівнем безпеки.

Алгоритм кластеризації K-Means було адаптовано для обробки просторових і погодних характеристик, що дало змогу ідентифікувати зони підвищеної аварійності та зробити висновки щодо впливу зовнішніх чинників на ДТП. Результати кластеризації інтерактивно відображаються на мапі за допомогою бібліотеки Leaflet.js, що забезпечує зручний візуальний аналіз.

У результаті тестування було підтверджено стабільність і точність роботи системи, зокрема правильність розподілу кластерів, відповідність очікуваним сценаріям і ефективність при обробці великих обсягів даних. Сформовані графіки розподілу ДТП за районами, місяцями та годинами доби дозволяють виявляти сезонні та часові тенденції.

Таким чином, поставлені завдання були успішно реалізовані. Розроблене програмне забезпечення може бути використане як інструмент підтримки прийняття рішень для служб безпеки дорожнього руху, аналітиків та органів міського планування. Подальший розвиток системи може передбачати впровадження методів прогнозування, класифікації та виявлення аномалій на основі історичних даних про ДТП.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.
2. PostgreSQL Documentation. – [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/ (дата звернення: 22.04.2025).
3. K-means Clustering Algorithm – GeeksForGeeks. – [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/k-means-clustering-introduction/ (дата звернення: 15.04.2025).
4. Entity Framework Core Documentation – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 25.04.2025).
5. UML – Unified Modeling Language – [Електронний ресурс]. – Режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/ (дата звернення: 04.05.2025).
6. MIOsoft Accident Explorer – [Електронний ресурс]. – Режим доступу: https://www.miosoft.com/solutions/accident-explorer.html (дата звернення: 02.05.2025).
7. CATT Lab – Incident Cluster Explorer – [Електронний ресурс]. – Режим доступу: https://www.cattlab.umd.edu/ (дата звернення: 02.05.2025).
8. TRL iMAAP – [Електронний ресурс]. – Режим доступу: https://trlsoftware.com/products/imaap/ (дата звернення: 02.05.2025).
9. KeyACCIDENT – [Електронний ресурс]. – Режим доступу: https://www.keysoftsolutions.co.uk/products/keyaccident (дата звернення: 02.05.2025).
10. Citian CRASH – [Електронний ресурс]. – Режим доступу: https://citian.co/crash/ (дата звернення: 01.05.2025).
11. TIMS – Transportation Injury Mapping System – [Електронний ресурс]. – Режим доступу: https://tims.berkeley.edu/ (дата звернення: 01.05.2025).

12. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Boston: Prentice Hall, 2017. – 432 p.
13. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Boston: Addison-Wesley, 1995. – 395 p.
14. UML: що це таке і як використовувати – [Електронний ресурс]. – Режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ – Дата звернення: 28.05.2025.
15. Accord.NET Framework. – [Електронний ресурс]. – Режим доступу: http://accord-framework.net (дата звернення: 05.05.2025).
16. Clustering Algorithms in Data Science – [Електронний ресурс]. – Режим доступу: https://scikit-learn.org/stable/modules/clustering.html (дата звернення: 25.04.2025).
17. Data.gov.ua – Портал відкритих даних України. – [Електронний ресурс]. – Режим доступу: https://data.gov.ua (дата звернення: 15.04.2025).
18. Leaflet.js Documentation. – [Електронний ресурс]. – Режим доступу: https://leafletjs.com (дата звернення: 05.05.2025).
19. Entity Framework Core Documentation – [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 25.04.2025).
20. GeoJSON Format Specification. – [Електронний ресурс]. – Режим доступу: https://geojson.org/ (дата звернення: 25.04.2025).
21. Войцеховський С. О., Міщенко О. І. Аналіз і візуалізація просторових даних. – К.: Ліра-К, 2021. – 224 с.
22. Голуб Б. Л., Вайганг Г. О. Методичні вказівки з розробки бакалаврської кваліфікаційної роботи для студентів спеціальності 121. – Київ: НУБіП України, 2025. – 58 с.
23. Мельник Ю. О., Іванченко С. П. Аналіз та обробка просторово-часових даних. – К.: НТУУ КПІ, 2022. – 164 с.

ДОДАТОК А

Основні класи програми

Клас Accident

Клас представляє сутність ДТП (дорожньо-транспортна пригода) з усіма основними характеристиками.

```
public class Accident
{
    public int Id { get; set; }                              // Унікальний ідентифікатор ДТП
    public double Latitude { get; set; }                     // Географічна широта місця ДТП
    public double Longitude { get; set; }                    // Географічна довгота місця ДТП
    public DateTime Date { get; set; }                        // Дата та час події
    public string WeatherConditions { get; set; }            // Погодні умови під час ДТП
    public string RoadType { get; set; }                      // Тип дороги (шосе, міська
                    вулиця тощо)
    public string VehicleType { get; set; }                   // Тип транспортного засобу
    public int Severity { get; set; }                          // Рівень тяжкості ДТП
    public int SpeedLimit { get; set; }                        // Обмеження швидкості на
                    ділянці дороги
    public string TrafficControl { get; set; }                // Засоби регулювання руху
                    (світлофор, знаки)
    public int Hour { get; set; }                                // Година доби, коли сталася
                    ДТП
    public bool IsFatal { get; set; }                            // Чи була ДТП зі
                    смертельними наслідками
    public string DayOfWeek { get; set; }                        // День тижня
}
```

Клас ClusteringResult

Клас зберігає інформацію про результати кластеризації ДТП.

```
public class ClusteringResult
{
    public int Id { get; set; }                // Унікальний ідентифікатор результату
                    кластеризації
    public int ClusterId { get; set; }         // Ідентифікатор кластера
    public int AccidentId { get; set; }        // Посилання на ДТП, що належить цьому
                    кластеру
    public double DistanceToCentroid { get; set; } // Відстань до центроїда кластера
}
```

Клас CsvReaderService

Сервіс для імпорту даних ДТП з CSV-файлу, використовуючи бібліотеку CsvHelper.

```
public class CsvReaderService
{
    private readonly ILogger _logger;

    public CsvReaderService(ILogger logger)
    {
        _logger = logger;
    }

    public List<Accident> ReadAccidentsFromCsv(string filePath)
    {
        var accidents = new List<Accident>();

        using (var reader = new StreamReader(filePath))
        using (var csv = new CsvReader(reader, CultureInfo.InvariantCulture))
```

```
        {
            var records = csv.GetRecords<Accident>().ToList();
            // Додаткова обробка полів
            foreach (var accident in records)
            {
                accident.Hour = accident.Date.Hour;
                accident.IsFatal = accident.Severity >= 4; // Припустимо, 4 і більше —
                    смертельні
                accident.DayOfWeek = accident.Date.DayOfWeek.ToString();
            }
            accidents.AddRange(records);
        }

        _logger.LogInformation($"Імпортовано {accidents.Count} записів ДТП.");
        return accidents;
    }
}
```

ДОДАТОК Б.

Структура бази даних

```
-- Таблиця дорожньо-транспортних пригод
CREATE TABLE Accidents (
    Id SERIAL PRIMARY KEY,
    Latitude DOUBLE PRECISION NOT NULL,
    Longitude DOUBLE PRECISION NOT NULL,
    TimeOfDay TEXT NOT NULL,
    WeatherConditions TEXT NOT NULL,
    RoadType TEXT NOT NULL,
    Severity TEXT NOT NULL,
    SpeedLimit INTEGER NOT NULL,
    TrafficControl TEXT NOT NULL,
    Hour INTEGER NOT NULL,
    Weather TEXT NOT NULL,
    DayOfWeek TEXT NOT NULL,
    VehicleType TEXT NOT NULL,
    IsFatal BOOLEAN NOT NULL
);

-- Таблиця результатів кластеризації
CREATE TABLE ClusteringResults (
    Id SERIAL PRIMARY KEY,
    Latitude DOUBLE PRECISION NOT NULL,
    Longitude DOUBLE PRECISION NOT NULL,
    Hour INTEGER NOT NULL,
    Weather TEXT,
    DayOfWeek TEXT,
    VehicleType TEXT,
    IsFatal BOOLEAN NOT NULL,
    Cluster INTEGER NOT NULL,
    CreatedAt TIMESTAMP WITHOUT TIME ZONE DEFAULT NOW()
);

-- Таблиця історії кластеризацій
CREATE TABLE ClusteringResultHistory (
    Id SERIAL PRIMARY KEY,
    ClusterCount INTEGER NOT NULL,
    CreatedAt TIMESTAMP WITHOUT TIME ZONE NOT NULL,
    ShortResults TEXT NOT NULL
);

-- Таблиця користувачів
CREATE TABLE Users (
    Id SERIAL PRIMARY KEY,
    Username TEXT NOT NULL UNIQUE,
    PasswordHash TEXT NOT NULL,
    Role TEXT NOT NULL
);

-- Таблиця логів подій
CREATE TABLE Logs (
    Id SERIAL PRIMARY KEY,
    Timestamp TIMESTAMP WITHOUT TIME ZONE DEFAULT NOW(),
    Level TEXT NOT NULL,
    Message TEXT NOT NULL,
    UserId INTEGER,
    FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE SET NULL
);
```

ДОДАТОК В

Зразок CSV-файлу з даними ДТП

Latitude,Longitude,TimeOfDay,WeatherConditions,RoadType,VehicleType,Severity,SpeedLimit,TrafficControl,Date,DayOfWeek
50.4361392,30.4737296,08:52,Rain,Residential,Car,Fatal,50,None,10/01/2020 00:00:00,Sunday
50.4212821,30.5553687,06:31,Snow,Residential,Motorcycle,Fatal,80,None,12/24/2020 00:00:00,Wednesday
50.4600681,30.3582913,23:28,Snow,Residential,Truck,Minor,100,TrafficLight,06/30/2024 00:00:00,Friday
50.2775339,30.5882332,15:15,Clear,Highway,Bicycle,Fatal,80,None,10/07/2020 00:00:00,Friday
50.4601942,30.5386080,10:32,Rain,Country,Bus,Fatal,100,None,01/17/2024 00:00:00,Saturday

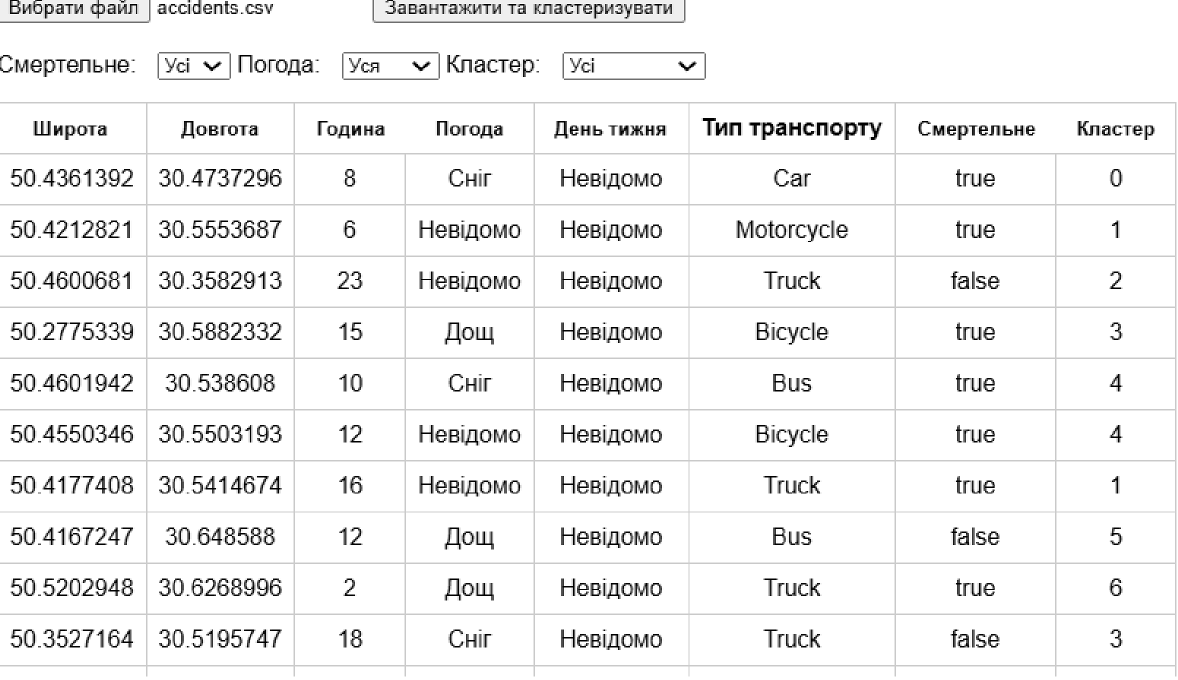
	A	B	C	D	E	F	G	H	I	J	K	L
1	Latitude,Longitude,TimeOfDay,WeatherConditions,RoadType,VehicleType,Severity,SpeedLimit,TrafficControl,Date,DayOfWeek											
2	50.4361392,30.4737296,08:52,Rain,Residential,Car,Fatal,50,None,10/01/2020 00:00:00,Sunday											
3	50.4212821,30.5553687,06:31,Snow,Residential,Motorcycle,Fatal,80,None,12/24/2020 00:00:00,Wednesday											
4	50.4600681,30.3582913,23:28,Snow,Residential,Truck,Minor,100,TrafficLight,06/30/2024 00:00:00,Friday											
5	50.2775339,30.5882332,15:15,Clear,Highway,Bicycle,Fatal,80,None,10/07/2020 00:00:00,Friday											
6	50.4601942,30.538608,10:32,Rain,Country,Bus,Fatal,100,None,01/17/2024 00:00:00,Saturday											
7	50.4550346,30.5503193,12:52,Windy,Urban,Bicycle,Fatal,50,TrafficLight,07/08/2024 00:00:00,Monday											

Рис. В.1 – Фрагмент CSV-файлу з даними про ДТП, відкритого у програмі Microsoft Excel

ДОДАТОК Г.

Результати кластеризації

Рис. Г.1 – Мапа з точками кластеризації



Кластеризація ДТП

Вибрати файл accidents.csv   Завантажити та кластеризувати

Смертельне: Усі Погода: Уся Кластер: Усі

Широта	Довгота	Година	Погода	День тижня	Тип транспорту	Смертельне	Кластер
50.4361392	30.4737296	8	Сніг	Невідомо	Car	true	0
50.4212821	30.5553687	6	Невідомо	Невідомо	Motorcycle	true	1
50.4600681	30.3582913	23	Невідомо	Невідомо	Truck	false	2
50.2775339	30.5882332	15	Дощ	Невідомо	Bicycle	true	3
50.4601942	30.538608	10	Сніг	Невідомо	Bus	true	4
50.4550346	30.5503193	12	Невідомо	Невідомо	Bicycle	true	4
50.4177408	30.5414674	16	Невідомо	Невідомо	Truck	true	1
50.4167247	30.648588	12	Дощ	Невідомо	Bus	false	5
50.5202948	30.6268996	2	Дощ	Невідомо	Truck	true	6
50.3527164	30.5195747	18	Сніг	Невідомо	Truck	false	3

Рис. Г.2 – Таблиця по кластеризованих даних

ДОДАТОК Ґ.

HTML-сторінка для взаємодії з результатами кластеризації ДТП

```
        <!DOCTYPE html>
<html lang="uk">
<head>

    <meta charset="UTF-8">
    <title>Кластеризація ДТП</title>
    <link rel="stylesheet" href="https://unpkg.com/leaflet/dist/leaflet.css" />
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

    <style>
        body {
            font-family: Arial,sans-serif;
            padding: 20px;
        }

        h1, h2 {
            margin-bottom: 20px;
        }

        table {
            width: 100%;
            border-collapse: collapse;
            margin-top: 15px;
        }

        th, td {
            border: 1px solid #ccc;
            padding: 8px;
            text-align: center;
        }

            th button {
                background: none;
                border: none;
                cursor: pointer;
                font-weight: bold;
            }

        .filters {
            margin-top: 20px;
        }

            .filters label {
                margin-right: 10px;
            }

        #map {
            height: 500px;
            width: 800px;
            margin: 30px auto;
            border: 1px solid #ccc;
        }

        #resultsTableWrapper {
            max-height: 400px;
            overflow-y: auto;
            display: block;
```

```
            position: relative;
        }

        #resultsTable {
            max-height: 400px;
            overflow-y: auto;
            display: block;
        }

            #resultsTable th {
                position: sticky;
                top: 0;
                background-color: #fff;
                z-index: 10;
                box-shadow: 0 1px 5px rgba(0,0,0, 0.1);
            }
    </style>
</head>
<body>
    <h1>Кластеризація ДТП</h1>

    <form id="uploadForm">
        <input type="file" id="csvFile" name="file" accept=".csv" required>
        <button type="submit">Завантажити та кластеризувати</button>
    </form>

    <div class="filters">
        <label for="isFatalFilter">Смертельне:</label>
        <select id="isFatalFilter">
            <option value="">Усі</option>
            <option value="true">Так</option>
            <option value="false">Ні</option>
        </select>

        <label for="weatherFilter">Погода:</label>
        <select id="weatherFilter">
            <option value="">Уся</option>
            <option value="0">Ясно</option>
            <option value="1">Дощ</option>
            <option value="2">Туман</option>
            <option value="3">Сніг</option>
        </select>

        <label for="clusterFilter">Кластер:</label>
        <select id="clusterFilter">
            <option value="">Усі</option>
        </select>
    </div>

    <table id="resultsTable">
        <thead>
            <tr>
                <th><button onclick="sortTable(0)">Широта</button></th>
                <th><button onclick="sortTable(1)">Довгота</button></th>
                <th><button onclick="sortTable(2)">Година</button></th>
                <th><button onclick="sortTable(3)">Погода</button></th>
                <th><button onclick="sortTable(4)">День тижня</button></th>
                <th>Тип транспорту</th>
                <th><button onclick="sortTable(6)">Смертельне</button></th>
                <th><button onclick="sortTable(7)">Кластер</button></th>
            </tr>
        </thead>
        <tbody></tbody>
    </table>
```

```
<div id="map"></div>

<h2>Кількість ДТП по районах</h2>
<canvas id="districtChart" width="600" height="400"></canvas>

<h2>Кількість ДТП по місяцях</h2>
<canvas id="monthlyChart" width="600" height="400"></canvas>

<h2>Кількість ДТП по годинах доби</h2>
<canvas id="hourlyChart"></canvas>


<h2>Історія кластеризацій</h2>
<button onclick="loadHistory()">Переглянути історію кластеризацій</button>


<table id="historyTable">
    <thead>
        <tr>
            <th>Дата</th>
            <th>Кількість кластерів</th>
            <th>Результати</th>
        </tr>
    </thead>
    <tbody></tbody>
</table>

<script src="https://unpkg.com/leaflet/dist/leaflet.js"></script>
<script>
    let clusters = [];
    let map;
    let markersLayer;

    document.getElementById("uploadForm").addEventListener("submit", async
             function (e) {
        e.preventDefault();
        const fileInput = document.getElementById("csvFile");
        const formData = new FormData();
        formData.append("file", fileInput.files[0]);

        try {
            const response = await fetch("/dtp/upload-and-cluster", {
                method: "POST",
                body: formData
            });

            let data;
            try {
                data = await response.json();
            } catch (err) {
                const text = await response.text();
                console.error("Сервер повернув не-JSON:", text);
                alert("Помилка з сервера: " + text);
                return;
            }

            clusters = data.clusters || [];
            populateClusterFilter(clusters);
            renderTable(clusters);
            renderMap(clusters);
            updateDistrictChart(clusters);  // <-- додайте це
            updateMonthlyChart(clusters);
            updateHourlyChart(clusters);  // Додай це!
```

```
    } catch (err) {
        console.error("Помилка:", err);
        alert("Не вдалося обробити файл.");
    }
});

function renderTable(data) {
    const tableBody = document.querySelector("#resultsTable tbody");
    tableBody.innerHTML = "";
    data.forEach(accident => {
        const row = document.createElement("tr");
        row.innerHTML = `
                                        <td>   ${accident.latitude}
            </td>
                                        <td>${accident.longitude} </td>
                                        <td>${accident.hour} </td>
                                        <td> ${accident.weatherConditions
            || 'Невідомо'}</td>

                                        <td>${accident.dayOfWeek ||
            'Невідомо'}</td>

                                        <td>${accident.vehicleType}</td>
                                        <td>    ${accident.isFatal} </td>

                                        <td
            class="cluster">${accident.cluster}</td>

                                      `;
        tableBody.appendChild(row);
    });
}

function mapWeather(code) {
    return {
        0: 'Ясно',

        1: 'Дощ',
        2: 'Туман',
        3: 'Сніг'
    }[code] || 'Невідомо';
}

function mapDayOfWeek(code) {
    return {

        0: 'Неділя',
        1: 'Понеділок',

        2: 'Вівторок',
        3: 'Середа',
        4: 'Четвер',

        5: 'П’ятниця',
        6: 'Субота'
    }[code] || 'Невідомо';
}

function populateClusterFilter(data) {
    const uniqueClusters = [...new Set(data.map(item => item.cluster))];
    const select = document.getElementById('clusterFilter');
    select.innerHTML = '<option value="">Усі</option>';
    uniqueClusters.forEach(c => {
        select.innerHTML += `<option value="${c}">Кластер ${c}</option>`
            ;
```

```
    });
}

document.getElementById('isFatalFilter').addEventListener('change',
            applyFilters);
document.getElementById('weatherFilter').addEventListener('change',
            applyFilters);
document.getElementById('clusterFilter').addEventListener('change',
            applyFilters);

function applyFilters() {
    let filtered = [...clusters];
    const isFatal = document.getElementById('isFatalFilter').value;
    const weather = document.getElementById('weatherFilter').value;
    const cluster = document.getElementById('clusterFilter').value;

    if (isFatal) filtered = filtered.filter(a => a.isFatal.toString() ===
            isFatal);
    if (weather) filtered = filtered.filter(a => a.weather.toString() ===
            weather);
    if (cluster) filtered = filtered.filter(a => a.cluster.toString() ===
            cluster);

    renderTable(filtered);
    renderMap(filtered);
    updateDistrictChart(filtered);  // <-- Оновити графік
    updateMonthlyChart(filtered);
    updateHourlyChart(filtered);  // Додай це!


}


function renderMap(data) {
    if (!map) {
        map = L.map('map').setView([50.45, 30.52], 11);
        L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; OpenStreetMap'
        }).addTo(map);
        setTimeout(() => map.invalidateSize(), 200);
    }

    if (markersLayer) {
        markersLayer.clearLayers();
    } else {
        markersLayer = L.layerGroup().addTo(map);
    }

    data.forEach(item => {
        if (!item.latitude || !item.longitude) return;
        const marker = L.circleMarker([item.latitude, item.longitude], {
            radius: 6,
            color: getClusterColor(item.cluster),
            fillOpacity: 0.7
        });
        marker.bindPopup(`
                                                <b>Кластер:</b>
            ${item.cluster}<br>
                                                <b>Смертельне:</b>
            ${item.isFatal}<br>
                                                <b>Погода:</b>
            ${mapWeather(item.weather)}<br>
                                                <b>Час:</b> ${item.hour}:00
                                            `);
        markersLayer.addLayer(marker);
```

```
    });
}

function getClusterColor(cluster) {
    const colors = ['red', 'blue', 'green', 'purple', 'orange', 'brown'];
    return colors[cluster % colors.length];
}

function countClustersFromTable() {
    const rows = document.querySelectorAll('#resultsTable tbody tr');
    const clusterCounts = {};

    rows.forEach(row => {
        const clusterCell = row.querySelector('td.cluster'); // додай клас
            `cluster` до відповідної td
        if (!clusterCell) return;

        const cluster = clusterCell.textContent.trim();
        if (cluster in clusterCounts) {
            clusterCounts[cluster]++;
        } else {
            clusterCounts[cluster] = 1;
        }
    });

    return clusterCounts;
}


function drawClusterChart() {
    const clusterCounts = countClustersFromTable();
    const labels = Object.keys(clusterCounts);
    const data = Object.values(clusterCounts);

    const ctx = document.getElementById('clusterChart').getContext('2d');

    new Chart(ctx, {
        type: 'bar',
        data: {
            labels: labels,
            datasets: [{
                label: 'Кількість ДТП по кластерах',
                data: data,
                backgroundColor: '#007bff'
            }]
        },
        options: {
            responsive: true,
            plugins: {
                legend: { display: false },
                title: {
                    display: true,
                    text: 'Кількість ДТП по кластерах'
                }
            }
        }
    });
}



async function loadHistory() {
    try {
        const response = await fetch("/dtp/history");
        const data = await response.json();
```

```
        const tableBody = document.querySelector("#historyTable tbody");
        tableBody.innerHTML = "";
        data.forEach(result => {
            const row = document.createElement("tr");
            row.innerHTML = `
                                                    <td>${new
             Date(result.createdAt).toLocaleString()}</td>

             <td>${result.clusterCount}</td>
                                                    <td><button
             onclick="viewResult(${result.id})">Переглянути</button></td>
                                                `;
            tableBody.appendChild(row);
        });
    } catch (err) {
        console.error("Помилка при завантаженні історії:", err);
    }
}

async function viewResult(id) {
    try {
        const response = await fetch(`/api/clusteringresults/${id}`);
        const result = await response.json();

        const clusters = JSON.parse(result.results);
        populateClusterFilter(clusters);
        renderTable(clusters);

        renderMap(clusters);
    } catch (err) {
        console.error("Помилка при перегляді результату:", err);
        alert("Сталася помилка при перегляді результату.");
    }
}


let districtChartInstance = null;
let monthlyChartInstance = null;
let hourlyChartInstance = null;


function updateMonthlyChart(data) {
    const monthCounts = new Array(12).fill(0); // Ініціалізуємо 12 місяців

    data.forEach(item => {
        const date = new Date(item.timestamp || item.date);
        if (!isNaN(date)) {
            const month = date.getMonth(); // 0 = січень
            monthCounts[month]++;
        }
    });

    const monthLabels = [
        'Січень', 'Лютий', 'Березень', 'Квітень', 'Травень', 'Червень',
        'Липень', 'Серпень', 'Вересень', 'Жовтень', 'Листопад', 'Грудень'
    ];

    const ctx = document.getElementById('monthlyChart').getContext('2d');

    if (monthlyChartInstance) {
        monthlyChartInstance.data.datasets[0].data = monthCounts;
        monthlyChartInstance.update();
    } else {
        monthlyChartInstance = new Chart(ctx, {
            type: 'line',
```

```
            data: {
                labels: monthLabels,
                datasets: [{
                    label: 'Кількість ДТП',
                    data: monthCounts,
                    fill: false,
                    borderColor: 'rgba(255, 99, 132, 1)',
                    backgroundColor: 'rgba(255, 99, 132, 0.2)',
                    tension: 0.1
                }]
            },
            options: {
                responsive: true,
                plugins: {
                    title: {
                        display: true,
                        text: 'Кількість ДТП по місяцях'
                    }
                },
                scales: {
                    y: {
                        beginAtZero: true,
                        ticks: {
                            stepSize: 1
                        }
                    }
                }
            }
        });
    }
}


function updateDistrictChart(data) {
    // рахуємо ДТП по кластерах з масиву об'єктів (а не з таблиці)
    const clusterCounts = {};

    data.forEach(item => {
        const c = item.cluster;
        clusterCounts[c] = (clusterCounts[c] || 0) + 1;
    });

    const clusterToDistrict = {
        0: "Голосіївський",
        1: "Шевченківський",
        2: "Оболонський",
        3: "Дарницький",
        4: "Деснянський",
        5: "Печерський",
        6: "Подільський",
        7: "Солом'янський",
        8: "Святошинський"
    };

    const labels = Object.keys(clusterCounts).map(id => clusterToDistrict[id]
            || `Кластер ${id}`);
    const counts = Object.values(clusterCounts);

    const ctx = document.getElementById('districtChart').getContext('2d');

    if (districtChartInstance) {
        districtChartInstance.data.labels = labels;
        districtChartInstance.data.datasets[0].data = counts;
        districtChartInstance.update();
    } else {
```

```
            districtChartInstance = new Chart(ctx, {
                type: 'bar',
                data: {
                    labels: labels,
                    datasets: [{
                        label: 'Кількість ДТП',
                        data: counts,
                        backgroundColor: 'rgba(54, 162, 235, 0.7)',
                        borderColor: 'rgba(54, 162, 235, 1)',
                        borderWidth: 1
                    }]
                },
                options: {
                    responsive: true,
                    scales: {
                        y: {
                            beginAtZero: true,
                            ticks: {
                                stepSize: 1
                            }
                        }
                    }
                }
            });
        }
    }


    function updateHourlyChart(data) {
        const hourCounts = new Array(24).fill(0);

        data.forEach(item => {
            // Замість item.TimeOfDay — беремо item.hour
            if (typeof item.hour === "number" && item.hour >= 0 && item.hour < 24)
                {
                hourCounts[item.hour]++;
            }
        });

        const hourLabels = [];
        for (let i = 0; i < 24; i++) {
            hourLabels.push(i.toString().padStart(2, '0') + ':00');
        }

        const ctx = document.getElementById('hourlyChart').getContext('2d');

        if (window.hourlyChartInstance) {
            hourlyChartInstance.data.datasets[0].data = hourCounts;
            hourlyChartInstance.update();
        } else {
            window.hourlyChartInstance = new Chart(ctx, {
                type: 'bar',
                data: {
                    labels: hourLabels,
                    datasets: [{
                        label: 'Кількість ДТП',
                        data: hourCounts,
                        backgroundColor: 'rgba(54, 162, 235, 0.6)',
                        borderColor: 'rgba(54, 162, 235, 1)',
                        borderWidth: 1
                    }]
                },
                options: {
                    responsive: true,
                    plugins: {
```

```
                        title: {
                            display: true,
                            text: 'Кількість ДТП по годинах доби'
                        }
                    },
                    scales: {
                        y: {
                            beginAtZero: true,
                            ticks: { stepSize: 1 }
                        }
                    }
                }
            });
        }
    }


    </script>

    <script src="js/site.js"></script>

</body>
</html>
```

ДОДАТОК Д.

Додатковий код для backand

ClusteringResultsController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using DtpClusteringApp.Data;
using DtpClusteringApp.Models;

namespace DtpClusteringApp.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ClusteringResultsController : ControllerBase
    {
        private readonly ApplicationDbContext _context;

        public ClusteringResultsController(ApplicationDbContext context)
        {
            _context = context;
        }

        // GET: api/ClusteringResults
        [HttpGet]
        public async Task<IActionResult> GetAllResults()
        {
            var results = await _context.ClusteringResults
                .OrderByDescending(r => r.CreatedAt)
                .ToListAsync();

            return Ok(results);
        }

        // GET: api/ClusteringResults/5
        [HttpGet("{id}")]
        public async Task<IActionResult> GetResult(int id)
        {
            var result = await _context.ClusteringResults.FindAsync(id);
            if (result == null)
            {
                return NotFound();
            }

            return Ok(result);
        }

        [HttpGet("history/{id}")]
        public async Task<IActionResult> GetHistoryById(int id)
        {
            var result = await _context.ClusteringResults.FindAsync(id);
            if (result == null)
            {
                return NotFound();
            }

            return Ok(result);
        }

    }
}
```

DtpController.cs

```
using DtpClusteringApp.Services;
using DtpClusteringApp.Models;
using Microsoft.AspNetCore.Mvc;

namespace DtpClusteringApp.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class DtpController : ControllerBase
    {
        private readonly CsvReaderService _csvReaderService;
        private readonly ClusteringService _clusteringService;
        private readonly ClusteringResultService _clusteringResultService;

        public DtpController(CsvReaderService csvReaderService, ClusteringService
clusteringService, ClusteringResultService clusteringResultService)
        {
            _csvReaderService = csvReaderService;
            _clusteringService = clusteringService;
            _clusteringResultService = clusteringResultService;
        }

        [HttpPost("upload-and-cluster")]
        public async Task<IActionResult> UploadAndCluster(IFormFile file)
        {
            if (file == null || file.Length == 0)
                return BadRequest("Файл не надано.");

            try
            {
                using var stream = new MemoryStream();
                await file.CopyToAsync(stream);
                stream.Position = 0;

                var accidents = _csvReaderService.LoadFromStream(stream);

                if (accidents.Count == 0)
                    return BadRequest("Файл не містить жодних даних для
кластеризації.");

                Console.WriteLine($"Завантажено {accidents.Count} записів.");

                var labels = _clusteringService.Clusterize(accidents);
                var clusterCount = labels.Distinct().Count();

                await _clusteringResultService.SaveClusteringResult(accidents, labels,
clusterCount);
                Console.WriteLine("Кластери успішно обчислено. Результати
збережено.");

                var results = accidents.Select((accident, index) => new
                {
                    accident.Latitude,
                    accident.Longitude,
                    accident.Hour,
                    accident.WeatherConditions,
                    accident.DayOfWeek,
                    accident.VehicleType,
                    accident.IsFatal,
                    accident.Date,
                    accident.WeatherCode,
                    Cluster = labels[index]
                }).ToList();
```

```
                return Ok(new { Clusters = results });
            }
            catch (Exception ex)
            {
                Console.WriteLine("Помилка кластеризації: " + ex.Message);
                return StatusCode(500, "Помилка кластеризації");
            }
        }

        [HttpGet("history")]
        public async Task<IActionResult> GetClusteringHistory()
        {
            try
            {
                var history = await _clusteringResultService.GetClusteringHistory();
                return Ok(history);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Помилка при отриманні історії: {ex.Message}");
                return StatusCode(500, "Помилка при отриманні історії кластеризацій");
            }
        }

        [HttpGet("results")]
        public async Task<IActionResult> GetResults()
        {
            var results = await _clusteringResultService.GetClusteringResultDtos();
            return Ok(results);
        }
    }
}
```

ApplicationDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using DtpClusteringApp.Models;

namespace DtpClusteringApp.Data
{
    public class ApplicationDbContext : DbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        { }

        public DbSet<ClusteringResult> ClusteringResults { get; set; } // таблиця для
збереження результатів кластеризації
        public DbSet<Accident> Accidents { get; set; } // нова таблиця

    }
}
```

ApplicationDbContextFactory.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.Configuration;
using System.IO;

namespace DtpClusteringApp.Data
{
    public class ApplicationDbContextFactory :
IDesignTimeDbContextFactory<ApplicationDbContext>
    {
        public ApplicationDbContext CreateDbContext(string[] args)
        {
            var configuration = new ConfigurationBuilder()
```

```
                .SetBasePath(Directory.GetCurrentDirectory()) // шукає
appsettings.json у поточній папці
                .AddJsonFile("appsettings.json")
                .Build();

            var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();

optionsBuilder.UseNpgsql(configuration.GetConnectionString("DefaultConnection"));

            return new ApplicationDbContext(optionsBuilder.Options);
        }
    }
}
```

Accident.cs

```
namespace DtpClusteringApp.Models
{
    public class Accident
    {
        public int Id { get; set; }
        public double Latitude { get; set; }
        public double Longitude { get; set; }

        public string TimeOfDay { get; set; } = string.Empty;
        public string WeatherConditions { get; set; } = string.Empty;
        public string RoadType { get; set; } = string.Empty;
        public string Severity { get; set; } = string.Empty;
        public int SpeedLimit { get; set; }
        public string TrafficControl { get; set; } = string.Empty;

        // Числові поля для кластеризації
        public int Hour { get; set; }

        // Замість числових Weather та DayOfWeek — використаємо string
        public string Weather { get; set; } = string.Empty;
        public string DayOfWeek { get; set; } = string.Empty;

        public string VehicleType { get; set; } = string.Empty;
        public bool IsFatal { get; set; }
    }
}
```

ClusteringResult.cs

```
using System.ComponentModel.DataAnnotations;

namespace DtpClusteringApp.Models
{
    public class ClusteringResult
    {
        [Key]
        public int Id { get; set; }

        public double Latitude { get; set; }
        public double Longitude { get; set; }
        public int Hour { get; set; }

        // Зроблено nullable типом
        public string? Weather { get; set; }  // Nullable string
        public string? DayOfWeek { get; set; }  // Nullable string
        public string? VehicleType { get; set; }  // Nullable string

        public bool IsFatal { get; set; }

        public int Cluster { get; set; }
        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    }
}
```

ClusteringResultHistoryDto.cs

```
namespace DtpClusteringApp.Models
{
    public class ClusteringResultHistoryDto
    {
        public int Id { get; set; }
        public int ClusterCount { get; set; }
        public DateTime CreatedAt { get; set; }
        public string ShortResults { get; set; } = string.Empty;
    }
}
```

EncryptionService.cs

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace DtpClusteringApp.Services.Security
{
    public class EncryptionService
    {
        private readonly byte[] _key;
        private readonly byte[] _iv;

        public EncryptionService()
        {
            // 256-бітовий ключ і IV — у продакшн-версії їх потрібно зберігати в
конфігурації
            _key = Encoding.UTF8.GetBytes("ThisIsASecretKeyThisIsASecretKey"); // 32
байти
            _iv = Encoding.UTF8.GetBytes("ThisIsAnInitVect"); // 16 байт
        }

        // Метод для шифрування тексту CSV
        public void EncryptCsv(string inputFile, string outputFile)
        {
            // Читання всього вмісту файлу
            string content = File.ReadAllText(inputFile);
            byte[] encryptedContent = Encrypt(Encoding.UTF8.GetBytes(content));

            // Запис зашифрованого вмісту в новий файл
            File.WriteAllBytes(outputFile, encryptedContent);
        }

        // Метод для дешифрування тексту CSV
        public void DecryptCsv(string inputFile, string outputFile)
        {
            // Читання зашифрованого вмісту
            byte[] encryptedContent = File.ReadAllBytes(inputFile);
            byte[] decryptedContent = Decrypt(encryptedContent);

            // Запис дешифрованого вмісту в новий файл
            File.WriteAllText(outputFile, Encoding.UTF8.GetString(decryptedContent));
        }

        // Шифрування
        public byte[] Encrypt(byte[] data)
        {
            using var aes = Aes.Create();
            aes.Key = _key;
            aes.IV = _iv;
            using var encryptor = aes.CreateEncryptor();
            return encryptor.TransformFinalBlock(data, 0, data.Length);
        }
```

```
        // Дешифрування
        public byte[] Decrypt(byte[] encrypted)
        {
            using var aes = Aes.Create();
            aes.Key = _key;
            aes.IV = _iv;
            using var decryptor = aes.CreateDecryptor();
            return decryptor.TransformFinalBlock(encrypted, 0, encrypted.Length);
        }
    }
}
```

LoggerService.cs

```
namespace DtpClusteringApp.Services.Security
{
    public class LoggerService
    {
        private readonly string logFilePath = "log.txt";

        public void Log(string message)
        {
            var line = $"[{DateTime.Now}] {message}";
            File.AppendAllText(logFilePath, line + Environment.NewLine);
        }
    }
}
```

ClusteringResultService.cs

```
using DtpClusteringApp.Data;
using DtpClusteringApp.Models;
using Microsoft.EntityFrameworkCore;

namespace DtpClusteringApp.Services
{
    public class ClusteringResultService
    {
        private readonly ApplicationDbContext _context;

        public ClusteringResultService(ApplicationDbContext context)
        {
            _context = context;
        }

        public async Task SaveClusteringResult(List<Accident> accidents, int[] labels,
int clusterCount)
        {
            var results = accidents.Select((accident, index) => new ClusteringResult
            {
                Latitude = accident.Latitude,
                Longitude = accident.Longitude,
                Hour = accident.Hour,
                Weather = accident.Weather,
                DayOfWeek = accident.DayOfWeek,
                VehicleType = accident.VehicleType,
                IsFatal = accident.IsFatal,
                Cluster = labels[index],
                CreatedAt = DateTime.UtcNow
            }).ToList();

            await _context.ClusteringResults.AddRangeAsync(results);
            await _context.SaveChangesAsync();
        }

        public async Task<List<ClusteringResult>> GetClusteringHistory()
        {
            return await _context.ClusteringResults
                .OrderByDescending(r => r.CreatedAt)
```

```
                .ToListAsync();
        }
    }
}
```

ClusteringService.cs

```
using DtpClusteringApp.Models;

namespace DtpClusteringApp.Services
{
    public class ClusteringService
    {
        private readonly DistrictLocator _districtLocator;
        private readonly Dictionary<string, int> _districtEncoding = new();

        public ClusteringService(DistrictLocator districtLocator)
        {
            _districtLocator = districtLocator;
        }

        public int[] Clusterize(List<Accident> accidents)
        {
            var labels = new int[accidents.Count];

            for (int i = 0; i < accidents.Count; i++)
            {
                var accident = accidents[i];
                string? district = _districtLocator.GetDistrictName(accident.Latitude,
accident.Longitude) ?? "Unknown";

                if (!_districtEncoding.ContainsKey(district))
                    _districtEncoding[district] = _districtEncoding.Count;

                labels[i] = _districtEncoding[district];
            }

            return labels;
        }

        // Більше не потрібні ParseWeather, ParseVehicleType тощо
    }
}
```

CsvReaderService.cs

```
using DtpClusteringApp.Models;
using CsvHelper;
using CsvHelper.Configuration;
using System.Globalization;
using System.IO;
using DtpClusteringApp.Services.Security;

namespace DtpClusteringApp.Services
{
    public class CsvReaderService
    {
        private readonly EncryptionService _encryptionService;
        private readonly LoggerService _logger;

        public CsvReaderService(EncryptionService encryptionService, LoggerService
logger)
        {
            _encryptionService = encryptionService;
            _logger = logger;
        }

        public List<Accident> LoadFromCsv(string filePath)
        {
```

```
            try
            {
                using var fs = new FileStream(filePath, FileMode.Open,
FileAccess.Read);
                byte[] encryptedData = new byte[fs.Length];
                fs.Read(encryptedData, 0, encryptedData.Length);

                byte[] decryptedData = _encryptionService.Decrypt(encryptedData);
                using var ms = new MemoryStream(decryptedData);

                _logger.Log($"Файл {filePath} дешифровано та оброблено.");
                return LoadFromStream(ms);
            }
            catch (Exception ex)
            {
                _logger.Log($"Помилка при завантаженні CSV: {ex.Message}");
                return new List<Accident>();
            }
        }

        public List<Accident> LoadFromStream(Stream stream)
        {
            var accidents = new List<Accident>();
            var config = new CsvConfiguration(CultureInfo.InvariantCulture)
            {
                HeaderValidated = null,
                MissingFieldFound = null
            };

            using var reader = new StreamReader(stream);
            using var csv = new CsvReader(reader, config);

            var records = csv.GetRecords<Accident>().ToList();

            foreach (var record in records)
            {
                record.Hour = ParseHour(record.TimeOfDay);
                record.Weather = ParseWeather(record.WeatherConditions).ToString();
                record.DayOfWeek = ParseDayOfWeek(record.TimeOfDay);
                record.IsFatal = record.Severity.ToLower().Contains("fatal");
                accidents.Add(record);
            }

            return accidents;
        }

        private string ParseDayOfWeek(string timeOfDay)
        {
            try
            {
                var date = DateTime.Parse(timeOfDay);
                return date.ToString("dddd", CultureInfo.InvariantCulture);
            }
            catch
            {
                return "Unknown";
            }
        }

        private int ParseHour(string time)
        {
            if (DateTime.TryParseExact(time, "HH:mm", CultureInfo.InvariantCulture,
DateTimeStyles.None, out var dt))
                return dt.Hour;
            return 0;
```

```
        }

        private int ParseWeather(string weather)
        {
            if (string.IsNullOrWhiteSpace(weather))
                return 0;

            var value = weather.ToLowerInvariant();
            if (value.Contains("clear")) return 1;
            if (value.Contains("cloud")) return 2;
            if (value.Contains("rain")) return 3;
            if (value.Contains("snow")) return 4;
            if (value.Contains("fog")) return 5;
            if (value.Contains("wind")) return 6;
            if (value.Contains("storm")) return 7;
            return 0;
        }
    }
}
```

ElbowMethod.cs

```
using Accord.MachineLearning;
using System;
using System.Linq;

namespace DtpClusteringApp.Services
{
    public class ElbowMethod
    {
        public int GetOptimalNumberOfClusters(double[][] data)
        {
            // Максимальна кількість кластерів
            int maxClusters = 10;
            double[] withinClusterSums = new double[maxClusters];

            // Обчислюємо суму квадратів відстаней для кожного k
            for (int k = 1; k <= maxClusters; k++)
            {
                var kmeans = new KMeans(k);
                var clusters = kmeans.Learn(data);

                // Обчислюємо середньоквадратичну помилку для даного k
                double error = 0;
                for (int i = 0; i < data.Length; i++)
                {
                    // Відстань між точкою і її центроїдом
                    double distance = EuclideanDistance(data[i],
clusters.Centroids[clusters.Decide(data)[i]]);
                    error += distance * distance;
                }

                withinClusterSums[k - 1] = error;
            }

            // Знайдемо точку "ліктя" (мінімальне зниження помилки)
            int optimalK = 1;
            double maxDiff = 0;
            for (int i = 1; i < withinClusterSums.Length - 1; i++)
            {
                double diff = withinClusterSums[i - 1] - withinClusterSums[i + 1];
                if (diff > maxDiff)
                {
                    maxDiff = diff;
                    optimalK = i + 1;
                }
            }
```

```
            return optimalK;
        }

        // Функція для обчислення евклідової відстані між двома точками
        private double EuclideanDistance(double[] point1, double[] point2)
        {
            double sum = 0;
            for (int i = 0; i < point1.Length; i++)
            {
                sum += Math.Pow(point1[i] - point2[i], 2);
            }
            return Math.Sqrt(sum);
        }
    }
}
```

ElbowMethodService.cs

```
using Accord.Statistics.Kernels;
using Accord.MachineLearning;
using System;
using System.Collections.Generic;

namespace DtpClusteringApp.Services
{
    public class ElbowMethodService
    {
        public List<double> FindOptimalK(double[][] data, int maxK = 10)
        {
            List<double> inertiaValues = new List<double>();

            for (int k = 1; k <= maxK; k++)
            {
                // Створюємо модель кластеризації
                var kmeans = new KMeans(k);
                var clusters = kmeans.Learn(data);

                // Погрішність (inertia) для поточного значення k
                inertiaValues.Add(kmeans.Error);
            }

            return inertiaValues;
        }
    }
}
```

DistrictLocator.cs

```
using NetTopologySuite.Features;
using NetTopologySuite.Geometries;
using NetTopologySuite.IO;
using Newtonsoft.Json;

public class DistrictLocator
{
    private readonly List<(string Name, Geometry Geometry)> _districts = new();

    public DistrictLocator(string geoJsonPath)
    {
        using var reader = new StreamReader(geoJsonPath);
        var geoJson = reader.ReadToEnd();

        var serializer = GeoJsonSerializer.Create();
        using var stringReader = new StringReader(geoJson);
        using var jsonReader = new JsonTextReader(stringReader);

        var featureCollection = serializer.Deserialize<FeatureCollection>(jsonReader)
```

```
            ?? throw new InvalidOperationException("GeoJSON не може бути прочитаний
або порожній.");

        foreach (var feature in featureCollection)
        {
            var name = feature.Attributes["NAME"]?.ToString() ?? "Unknown";
            _districts.Add((name, feature.Geometry));
        }
    }

    public string? GetDistrictName(double latitude, double longitude)
    {
        var point = new Point(longitude, latitude); // GeoJSON uses (lon, lat)

        foreach (var (name, geometry) in _districts)
        {
            if (geometry.Contains(point))
                return name;
        }

        return null;
    }
}
```

MigrationHelper.cs

```
using DtpClusteringApp.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.Extensions.DependencyInjection;

namespace DtpClusteringApp
{
    public class MigrationHelper
    {
        private readonly IServiceProvider _serviceProvider;

        public MigrationHelper(IServiceProvider serviceProvider)
        {
            _serviceProvider = serviceProvider;
        }

        public void ApplyMigration()
        {
            using (var scope = _serviceProvider.CreateScope())
            {
                var dbContext =
scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
                var migrator = dbContext.GetService<IMigrator>();
                migrator.Migrate();
            }
        }
    }
}
```

Program.cs

```
using DtpClusteringApp;
using DtpClusteringApp.Data;
using DtpClusteringApp.Services;
using DtpClusteringApp.Services.Security; // додано для доступу до нових сервісів
using Microsoft.EntityFrameworkCore;
using NetTopologySuite.IO;
using NetTopologySuite.Features;
using NetTopologySuite.Geometries;

var builder = WebApplication.CreateBuilder(args);
```

```
// Реєстрація нових сервісів для шифрування та логування
builder.Services.AddSingleton<EncryptionService>();
builder.Services.AddSingleton<LoggerService>();

// Реєстрація інших сервісів
builder.Services.AddScoped<ClusteringResultService>();
builder.Services.AddScoped<CsvReaderService>();
builder.Services.AddScoped<ClusteringService>();

// Додавання DistrictLocator
builder.Services.AddSingleton(new DistrictLocator("Data/Kyiv_districts.geojson"));

// Реєстрація контролерів та бази даних
builder.Services.AddControllers();
builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddTransient<MigrationHelper>();
builder.Services.AddRazorPages();

var app = builder.Build();

// Автоматичне застосування міграцій
using (var scope = app.Services.CreateScope())
{
    var migrationHelper = scope.ServiceProvider.GetRequiredService<MigrationHelper>();
    migrationHelper.ApplyMigration();
}

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

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();

app.MapRazorPages();
app.MapControllers();

app.Run();
```