ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «ІНТЕЛЕКТУАЛЬНА СИСТЕМА ГЕНЕРАЦІЇ SQL-ЗАПИТІВ НА ОСНОВІ ОБРОБКИ ПРИРОДНОЇ МОВИ»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерні системи та мережі»

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

_____ Владислав Бутенко
(підпис)

Виконав: здобувач вищої освіти групи КСДМ-62
Владислав БУТЕНКО

Керівник: Віталій ВОЛОХІН
к.т.н., доцент

Рецензент:
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Магістр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____ Наталія ЛАЩЕВСЬКА
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бутенку Владиславу Олександровичу

1. Тема кваліфікаційної роботи: «Інтелектуальна система генерації SQL-запитів на основі обробки природної мови»

керівник кваліфікаційної роботи Віталій ВОЛОХІН, к.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «___» _____ 2025 р. № _____.

2. Строк подання кваліфікаційної роботи «17» листопада 2025 р.

3. Вихідні дані до кваліфікаційної роботи: запити українською та англійською мовами довжиною до 200 слів, а також структура реляційних баз даних на основі SQLite 3 та PostgreSQL.

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

1. Аналіз теоретичні основи NLP, існуючих рішень NL2SQL та формулювання функціональних й нефункціональних вимог до системи.
2. Проєктування багаторівневої модульної архітектури.
3. Програмна реалізація прототипу системи на мові Python та проведення тестування точності генерації SQL.

5. Перелік ілюстративного матеріалу: *презентація*
    1. Актуальність роботи
    2. Мета та ключові виклики проєкту.
    3. Наукова новизна та архітектура.
    4. Діаграма компонентів.
    5. Висновки до магістерської роботи.

6. Дата видачі завдання «01» вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури та документації	*20.10 - 24.10.2025*	
2	Вивчення матеріалів для аналізу розвитку технологій NL2SQL	*20.10 – 24.10.2025*	
3	Дослідження методів перетворення природної мови в SQL запити	*20.10 – 24.10.2025*	
4	Порівняння технологій, які будуть використовуватись	*20.10 – 24.10.2025*	
5	Вибір технологій на основі аналізу	*20.10 – 24.10.2025*	
6	Проєктування багаторівневої модульної архітектури для перетворення природної мови в SQL запити	*02.11 - 21.11.2025*	
7	Оформлення роботи: вступ, висновки, реферат	*01.12 - 20.12.2025*	
8	Розробка демонстраційних матеріалів	*01.12 - 20.12.2025*	
9	Попередній захист роботи	*01.12 - 20.12.2025*	

Здобувач вищої освіти _____ Владислав БУТЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Віталій ВОЛОХІН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 60 стор., 10 рис., 40 джерел.

Мета роботи – розробка архітектури, програмна реалізація та створення прототипу інтелектуальної системи генерації SQL-запитів з природної мови (NL2SQL) з підтримкою української та англійської мов, можливістю локального розгортання та механізмами забезпечення точності й безпеки.

Об'єкт дослідження – процеси обробки природної мови та генерації структурованих SQL-запитів.

Предмет дослідження – архітектура інтелектуальної системи NL2SQL, що включає модулі NLP, семантичного аналізу, генерації та валідації SQL.

У роботі використано трансформерні архітектури, методи семантичного парсингу, аналіз вимог, моделювання архітектурних компонентів за допомогою UML-діаграм, побудову прототипу, а також емпіричне тестування за метриками exact match та execution accuracy.

Проведено аналіз теоретичних основ обробки природної мови (NLP) та сучасних архітектурних рішень для систем NL2SQL, зокрема моделей Seq2SQL, RAT-SQL та PICARD.

Розроблено інтегровану багаторівневу модульну архітектуру інтелектуальної системи, яка поєднує компактні трансформерні моделі, механізми schema linking та багаторівневу валідацію SQL. Реалізовано прототип системи з використанням мови Python, фреймворків PyTorch та FastAPI, бібліотек Transformers та spaCy, а також локальної СУБД SQLite 3.

КЛЮЧОВІ СЛОВА: NL2SQL, ОБРОБКА ПРИРОДНОЇ МОВИ (NLP), ТРАНСФОРМЕРИ, ГЕНЕРАЦІЯ SQL-ЗАПИТІВ, СЕМАНТИЧНИЙ ПАРСИНГ, УКРАЇНСЬКА МОВА, БЕЗПЕКА ДАНИХ, ЛОКАЛЬНЕ РОЗГОРТАННЯ.

ABSTRACT

Text part of the master's qualification work: 60 pages, 10 figures, 40 sources.

The aim of the work is the development of the architecture, software implementation, and prototype of an intelligent system for generating SQL queries from natural language (NL2SQL) with support for Ukrainian and English languages, local deployment capability, and mechanisms for ensuring accuracy and security.

The object of the study is the processes of natural language processing and structured SQL query generation.

The subject of the study is the architecture of an intelligent NL2SQL system, including modules for NLP, semantic analysis, SQL generation, and validation.

The work employs methods of machine learning, transformer architectures, semantic parsing methods, requirements analysis, architectural component modeling using UML diagrams, prototyping, as well as empirical testing and quality assessment using exact match and execution accuracy metrics.

An analysis of the theoretical foundations of natural language processing (NLP) and modern architectural solutions for NL2SQL systems, such as Seq2SQL, RAT-SQL, and PICARD models, has been carried out.

An integrated multi-level modular architecture of an intelligent system has been developed, combining compact transformer models, schema linking mechanisms, and multi-level SQL validation. A system prototype has been implemented using the Python language, PyTorch and FastAPI frameworks, Transformers and spaCy libraries, as well as the local SQLite 3 DBMS.

KEYWORDS: NL2SQL, NATURAL LANGUAGE PROCESSING (NLP), TRANSFORMERS, SQL QUERY GENERATION, SEMANTIC PARSING, UKRAINIAN LANGUAGE, DATA SECURITY, LOCAL DEPLOYMENT.

ЗМІСТ

ВСТУП 9
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ ТА ІСНУЮЧИХ РІШЕНЬ 11
1.1 Теоретичні основи обробки природної мови та генерації SQL-запитів 11
1.2. Аналіз існуючих систем і підходів 17
1.3. Постановка завдання та вимоги до системи.....21
РОЗДІЛ 2. АРХІТЕКТУРА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ.....26
2.1 Загальна архітектура системи 26
2.2 Вибір і адаптація моделей NLP.....32
2.3 Механізми генерації та валідації SQL-запитів 39
РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ.....46
3.1 Використані технології та інструменти 46
3.2 Реалізація ключових модулів 52
3.3 Тестування та оцінка ефективності 59
ВИСНОВКИ.....68
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....70
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ.....73
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ.....77

ВСТУП

Сучасні інформаційні системи накопичують дедалі більші обсяги даних, що вимагає ефективних засобів доступу та аналізу. Водночас, значна частина користувачів не володіє мовами запитів до баз даних, що створює бар'єри у використанні потенціалу аналітичних інструментів. Проблема взаємодії нетехнічних користувачів з реляційними базами даних особливо актуальна в умовах цифрової трансформації, коли інформаційні ресурси мають бути доступними максимально широкому колу фахівців. Одним з перспективних рішень даної проблеми є системи перетворення природномовних запитів в формальні SQL-запити, що дозволяють інтуїтивно отримувати дані без знання синтаксису та структури бази даних.

Розвиток обробки природної мови (NLP), зокрема технологій машинного та глибинного навчання, створив підґрунтя для появи систем NL2SQL, що забезпечують автоматичне генерування SQL-запитів на основі текстового опису наміру користувача. Еволюція від статистичних моделей до трансформерів та спеціалізованих архітектур для семантичного парсингу відкрила можливості для високої точності та універсальності подібних систем. Однак, низка проблем залишається невирішеною: недостатня адаптованість моделей до української мови, складність роботи з багатотабличними схемами, ризики синтаксичних помилок та SQL-ін'єкцій, а також висока ресурсомісткість сучасних мовних моделей.

Об'єкт дослідження – процеси обробки природної мови та генерації структурованих SQL-запитів.

Предмет дослідження – архітектура інтелектуальної системи NL2SQL, що включає модулі NLP, семантичного аналізу, генерації та валідації SQL.

Мета роботи – розробка архітектури, програмної реалізації та прототипу інтелектуальної системи генерації SQL-запитів з природної мови з підтримкою української та англійської мов, можливістю локального розгортання та механізмами забезпечення точності й безпеки.

Для досягнення мети сформульовано такі **завдання**:

– проаналізувати теоретичні основи NLP і сучасні підходи до NL2SQL;

– дослідити існуючі архітектури та їх обмеження;

– визначити вимоги до системи, включно з функціональними, нефункціональними та експлуатаційними

– обґрунтувати вибір моделей NLP та розробити схему їх адаптації;

– спроєктувати архітектуру інтелектуальної системи;

– реалізувати прототип з використанням Python та SQLite3;

– здійснити тестування та оцінку точності роботи системи;

– сформувати рекомендації щодо впровадження та подальшої модернізації.

Методи дослідження включають машинне навчання, трансформерні архітектури, методи семантичного парсингу, аналіз вимог, моделювання архітектурних компонентів, створення UML-діаграм, побудову прототипу, емпіричне тестування та оцінку якості за метриками exact match та execution accuracy.

Наукова новизна полягає в розробці інтегрованої архітектури інтелектуальної системи NL2SQL, що поєднує компактні трансформерні моделі, механізми schema linking, багаторівневу валідацію SQL та можливість повністю локального розгортання. Запропонований підхід формує гібридну модель генерації SQL, орієнтовану на україномовні запити та оптимізовану для роботи в умовах обмежених обчислювальних ресурсів.

Практичне значення полягає в створенні працюючого прототипу системи, який може використовуватися для бізнес-аналітики, навчальних платформ, внутрішніх корпоративних інструментів і дослідницьких задач. Результати роботи можуть бути застосовані для автоматизації взаємодії з базами даних, зниження навантаження на IT-персонал та підвищення доступності даних для нетехнічних користувачів.

Структура роботи складається зі вступу, трьох основних розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Теоретичні основи обробки природної мови та генерації SQL-запитів

Обробка природної мови, або Natural Language Processing (NLP), є однією з фундаментальних галузей штучного інтелекту, яка зосереджується на забезпеченні ефективної взаємодії між комп'ютерними системами та людською мовою в її природній формі. Дана дисципліна сформувалась на перетині лінгвістики, комп'ютерних наук, математики та когнітивних наук, дозволяючи машинам не лише розпізнавати текст чи мову, але й розуміти її семантику, контекст, наміри, а також генерувати відповіді, що імітують людське спілкування. NLP набуває особливого значення для створення інтелектуальних інтерфейсів, що спрощують доступ до великих обсягів даних. Зокрема, генерація SQL-запитів на основі природної мови (NL2SQL) є складним, але перспективним напрямом, що передбачає автоматичне перетворення неструктурованих текстових описів в структуровані команди мови структурованих запитів (SQL), яка є де-факто стандартом для взаємодії з реляційними базами даних, такими як PostgreSQL, MySQL чи Oracle [1].

Процес розпізнавання вимагає глибокого розуміння семантики мови, аби уникнути помилок, зумовлених неоднозначністю слів, синтаксичними конструкціями, контекстом чи іншими чинниками. Як зазначається в дослідженнях, присвячених інтелектуальним технологіям перетворення природної мови в SQL, такі системи дозволяють користувачам без глибоких технічних знань ефективно взаємодіяти з базами даних, спрощуючи процеси аналізу, звітності та прийняття рішень в бізнесі, науці, державному управлінні, тощо. Це підкреслює роль NLP в демократизації доступу до інформації, роблячи її доступною для широкого кола фахівців.

Історія розвитку NLP починається з середини XX століття, коли перші спроби автоматизувати переклад текстів між мовами призвели до створення

правило-орієнтованих систем. В 1950-х роках, під впливом робіт Алана Тьюрінга, Ноама Хомського та інших вчених, почали розроблятись моделі, засновані на формальних граматиках, де лінгвістичні правила вручну кодувалися для розбору тексту. Наприклад, токенізація, як базовий крок, передбачає розбиття тексту на окремі слова, фрази чи субтокени, що є необхідним для подальшого аналізу. За нею слідує морфологічний аналіз, який визначає частини мови – іменник, дієслово, прикметник тощо, та синтаксичний розбір, що будує дерево залежностей між елементами речення за допомогою контекстно-вільних граматик. Представлені методи хоча й ефективні для простих завдань, таких як базовий переклад чи розпізнавання шаблонів, стикалися з серйозними обмеженнями в обробці варіативності природної мови, включаючи синоніми, омоніми, ідіоматичні вирази, сарказм чи регіональні діалекти. З появою статистичних моделей в 1990-х роках, заснованих на ймовірнісних обчисленнях та великих об’ємах текстів, таких як Brown Corpus чи Penn Treebank, NLP перейшла до використання машинного навчання. Це дозволило автоматизувати низку процесів, серед яких, розпізнавання іменованих сутностей (Named Entity Recognition, NER), де система ідентифікує особи, місця, організації чи дати в тексті, або аналіз настрою, що визначає емоційний тон повідомлення, позитивний, негативний чи нейтральний [2].

Перехід до методів глибокого навчання, особливо рекурентних нейронних мереж (RNN), став справжньою революцією в NLP. RNN дозволяють моделювати послідовності даних, враховуючи контекст попередніх слів під час обробки поточного. Однак стандартні RNN страждають від проблеми зникання чи вибуху градієнтів, що ускладнює навчання на довгих послідовностях тексту, наприклад, в складних реченнях з різними мовленнєвими конструкціями. Вищезгадану проблему вдалось вирішити за рахуноку мереж з довгою короткостроковою пам’яттю (LSTM), запропонованих Сеппом Хохрайтером та Юргеном Шмідхубером в 1997 році, які вводять механізми воріт, вхідні, вихідні та забування для селективного збереження інформації. В контексті генерації SQL-запитів LSTM виявляється особливо корисним, оскільки дозволяє системі захоплювати довготривалі залежності в природній мові, наприклад, коли запит

містить умовні конструкції, підзапити чи посилання на попередні елементи ("покажи студентів, які вивчають інформатику і мають середній бал вище 80") [3].

За даними досліджень щодо інтелектуальної технології перетворення природної мови в SQL, використання LSTM забезпечує ефективне розуміння мови та передбачення відповідних SQL-структур, базуючись на глибокому навчанні. Це робить LSTM ключовим інструментом для задач NL2SQL, де точність залежить від здатності моделі обробляти послідовні дані без втрати контексту, особливо в умовах неоднозначності, наприклад, коли слово «банка» може означати фінансову установу чи контейнер.

Сучасний етап розвитку NLP домінується архітектурою трансформерів, запропонованою в 2017 році Ашішем Васвані та колегами в статті «Attention Is All You Need». Трансформери базуються на механізмі уваги (attention mechanism), який дозволяє моделі зважувати важливість різних слів в реченні незалежно від їх позиції, обробляючи весь вхідний текст паралельно. Це радикально прискорює навчання та покращує масштабованість порівняно з RNN. Моделі на кшталт BERT (Bidirectional Encoder Representations from Transformers), розробленої Google, навчаються на масивних кластерах текстів, таких як Wikipedia та BookCorpus, захоплюючи контекст з обох боків слова (двосторонній підхід), тоді як GPT (Generative Pre-trained Transformer) від OpenAI фокусується на генеративних завданнях, створюючи coherent текст на основі вхідних підказок [4].

В задачах NL2SQL трансформери дозволяють проводити fine-tuning на спеціалізованих датасетах, таких як Spider, WikiSQL чи ATIS, де природні запити зіставляються з SQL-еквівалентами. Наприклад, трансформери ефективно справляються з семантичним парсингом, перетворюючи запит «Які студенти вивчають інформатику і проживають у Києві?» на SQL-команду SELECT з відповідними JOIN, WHERE та можливими агрегатними функціями. Дослідження щодо застосування трансформерів в діалогових системах показують, як ці моделі інтегруються з базами даних для динамічної генерації запитів, забезпечуючи високу точність навіть складних діалогах. Фреймворки на кшталт Hugging Face

Transformers полегшують впровадження, дозволяючи швидко адаптувати моделі до конкретних доменів.

Генерація SQL-запитів на основі NLP стикається з низкою теоретичних викликів, які потребують комплексного підходу. По-перше, проблема доменної адаптації, яка полягає в тому, що модель, навчена на загальних текстах, може погано справлятися з технічними термінами баз даних, серед яких можна виділити назви таблиць, стовпців чи специфічні функції (наприклад, DATE_TRUNC в PostgreSQL). По-друге, неоднозначність мови вимагає розв’язання кореференції, де займенники чи еліпсиси заміщуються на конкретні сутності, а також обробки полісемії, коли одне слово має кілька значень залежно від контексту. По-третє, SQL як формальна мова має строгу синтаксичну структуру з чіткими правилами для SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY тощо, тому генерація повинна забезпечувати валідність запитів, уникаючи синтаксичних помилок, що могли б призвести до некоректних результатів, таймаутів чи навіть вразливостей, як SQL-ін’єкції. В бізнес-контексті, як зазначається в описах NLP від провідних компаній, ці виклики долаються за допомогою комбінації методів, включаючи токенізацію, лематизацію, векторизацію та семантичне зіставлення зі схемою бази даних [5]. Ключові етапи обробки представлені в таблиці 1.1.

Таблиця 1.1. Основні етапи обробки природної мови в контексті генерації SQL-запитів

Етап обробки	Опис	Застосування в NL2SQL	Приклад
Токенізація та нормалізація	Розбиття тексту на токени з видаленням шумів (стоп-слова, пунктуація, приведення до нижнього регістру).	Підготовка запиту для подальшого аналізу, виділення ключових слів як "select", "where", "середній бал".	"Покажи студентів з балом > 80" → ["покажи", "студентів", "з", "балом", ">", "80"]

Продовження таблиці 1.1

Морфологічний та синтаксичний аналіз	Визначення частин мови, побудова дерева залежностей.	Розпізнавання суб’єкта, об’єкта, умов для формування CLAUSE в SQL.	Визначення "студентів" як об’єкт, "бал > 80" як умову WHERE.
Семантичний аналіз та ембедінги	Перетворення слів у вектори (Word2Vec, GloVe, BERT embeddings), захоплення семантичної подібності.	Зіставлення природної мови зі схемою бази даних (таблиці, стовпці).	"середній бал" → зіставлення з колонкою "avg_score".
Генерація послідовності (seq2seq)	Використання енкодер-декодер архітектур для створення вихідної послідовності.	Перетворення вхідного тексту в SQL-код з урахуванням контексту.	Вихід: SELECT * FROM students WHERE avg_score > 80;
Валідація, виконання та корекція	Перевірка запиту на синтаксичну коректність, виконання в БД, обробка помилок.	Забезпечення безпеки, точності, інтеграція з інструментами як SQLAlchemy чи LangChain.	Виявлення помилки в JOIN, автоматична корекція.

Таблиця демонструє логічний ланцюжок, де кожен етап будується на попередньому, забезпечуючи перехід від неструктурованої мови до формалізованої. В роботі з базами даних, такими як PostgreSQL, що підтримують складні запити з вікнами, CTE чи JSON, NLP-моделі інтегруються з метаданими схеми (INFORMATION_SCHEMA) для підвищення точності. Теоретичні основи

також включають гібридні підходи, де правило-орієнтовані методи (наприклад, регулярні вирази для простих шаблонів) комбінуються з глибоким навчанням для кращої інтерпретовності, особливо в задачах, де потрібна висока точність, як у фінансовій аналітиці, охороні здоров'я чи державному управлінні [6].

Одним з ключових аспектів сучасного NL2SQL є використання фреймворків на кшталт LangChain, які дозволяють будувати складні ланцюжки обробки. Наприклад, процес починається з підключення до бази даних через спеціальні інструменти (SQLDatabaseToolkit), після чого велика мовна модель (LLM), така як GPT-4 чи Llama, генерує SQL-запит на основі користувацького вводу. Далі відбувається виконання запиту, валідація результату та перефразування в природну мову.

Це забезпечує не лише точність, але й контекстну пам'ять для обробки послідовних запитів, як в чат-ботах для аналітики. Переваги такого підходу включають доступність даних для нетехнічних користувачів, інтуїтивність інтерфейсу, гнучкість інтеграції з різними СУБД та підтримку few-shot learning, де модель навчається на кількох прикладах для адаптації до конкретного домену. Однак виклики, такі як точність генерації SQL при складних запитах з агрегатами, підзапитами чи часовими рядами, або витрати на використання LLM, вимагають постійного вдосконалення, наприклад, через RAG (Retrieval-Augmented Generation), де модель доповнюється релевантними прикладами з бази знань.

Застосування NLP для NL2SQL поширюється на різні галузі. В бізнес-аналітиці NLP автоматизує створення звітів про продажі, поведінку клієнтів чи логістику, дозволяючи менеджерам формулювати запити типу «які товари найчастіше купують в Києві влітку?» без знання SQL. В охороні здоров'я NLP обробляє медичні нотатки для створення запитів до баз пацієнтів, полегшуючи діагностику та моніторинг. Фінансова галузь використовує NLP для виявлення шахрайства через аналіз транзакцій, а нафтогазова, для інтерпретації геологічних звітів та прогнозування. В державному секторі NL2SQL спрощує доступ до відкритих даних, наприклад, в реєстрах чи статистичних базах. Інтеграція з

інструментами на кшталт Secoda чи DataHub додає управління метаданими, забезпечуючи безпеку, відповідність GDPR та масштабування [7].

Теоретичні основи NLP для генерації SQL-запитів ґрунтуються на еволюції від правило-орієнтованих та статистичних методів до рекурентних мереж, трансформерів та гібридних підходів, з акцентом на розуміння контексту, семантичне зіставлення та забезпечення синтаксичної коректності. Дані принципи не лише формують основу для розробки інтелектуальних систем, але й відкривають перспективи для подальших досліджень, де виклики неоднозначності мови, доменної адаптації та безпеки вирішуються за допомогою фреймворків на кшталт LangChain, RAG та інтеграції з метаданими баз даних.

1.2. Аналіз існуючих систем і підходів

Аналіз існуючих систем генерації SQL-запитів на основі обробки природної мови є важливим етапом для розуміння сучасного стану технологій, виявлення їх сильних та слабких сторін, а також визначення напрямів удосконалення в межах запропонованої архітектури. Задача NL2SQL (Natural Language to SQL) привертає увагу дослідників та розробників з початку 2010-х років, коли зросла потреба в інтуїтивних інтерфейсах для роботи з базами даних, особливо для користувачів без глибоких знань SQL. Системи, що реалізують такий функціонал, можна умовно поділити на кілька категорій [8]:

– академічні прототипи, побудовані на спеціалізованих датасетах;

– комерційні продукти великих технологічних компаній;

– інструменти з відкритим кодом, доступні для спільноти розробників.

Кожна з цих категорій має свої особливості, які впливають на точність, масштабованість та практичну застосовуваність.

Одним з датасетів, що стимулювали розвиток NL2SQL, є WikiSQL, представлений в 2017 році компанією Salesforce. Цей набір даних містить понад 80 тисяч пар «природний запит – SQL», згенерованих на основі таблиць Wikipedia. WikiSQL фокусується на простих запитах з однією таблицею, без

складних JOIN чи вкладених підзапитів, що робить його придатним для базового тестування моделей. На основі цього датасету було розроблено кілька систем, зокрема Seq2SQL – модель на базі рекурентних нейронних мереж з механізмом уваги, яка генерує SQL послідовно, крок за кроком. Seq2SQL досягає точності близько 60% за метрикою exact match, що означає повну відповідність згенерованого запиту еталонному. Перевагою підходу є відносна простота архітектури, яка включає енкодер для обробки природної мови та декодер для формування SQL. Однак обмеження WikiSQL в складності запитів робить такі системи малопридатними для реальних сценаріїв, де бази даних містять десятки таблиць з різноманітними зв'язками. Дослідження, присвячені аналізу Seq2SQL, показують, що модель погано справляється з агрегатними функціями та умовами, коли природний запит містить неоднозначні посилання на стовпці [9].

Значно складнішим є датасет Spider, розроблений в 2018 році командою Єльського університету за підтримки Yahoo. Spider включає понад 10 тисяч запитів до 200 баз даних з кількома таблицями, охоплюючи складні конструкції, такі як JOIN, GROUP BY, ORDER BY, підзапити та навіть віконні функції. Цей датасет став еталоном для оцінки крос-доменних систем NL2SQL, тобто тих, що працюють з базами даних з різних предметних областей – від освіти до фінансів. На базі Spider було створено RAT-SQL – одну з провідних моделей, яка інтегрує схему бази даних в графову структуру та використовує відношення (relation-aware transformers) для кращого зіставлення слів природної мови з елементами схеми. RAT-SQL досягає точності близько 70% за exact match на тестовому наборі Spider, що є значним прогресом. Архітектура моделі включає етап попередньої обробки, де схема бази даних кодується в граф, та трансформер, який враховує відношення між таблицями та стовпцями. Це дозволяє системі розуміти, наприклад, що запит «покажи студентів з вищим балом за середній» вимагає підзапиту з агрегатною функцією AVG. Недоліком RAT-SQL є висока обчислювальна складність, особливо при великій кількості таблиць, що робить її менш придатною для реального часу на слабких пристроях [10].

Комерційні рішення, такі як Google BigQuery NL або Microsoft Power BI Q&A, пропонують готові до використання інтерфейси для NL2SQL. BigQuery NL,

інтегрований в хмарну платформу Google Cloud, використовує великі мовні моделі (LLM), зокрема PaLM або Gemini, для генерації запитів. Користувач вводить текст українською чи англійською, система автоматично формує SQL, виконує його та візуалізує результати. Перевагою є висока точність в доменах, для яких модель була додатково навчена, а також підтримка складних аналітичних запитів. Однак комерційні системи мають обмеження. Вони працюють лише в межах власної екосистеми, вимагають підписки та не дозволяють повного доступу до коду для модифікації. Крім того, обробка конфіденційних даних в хмарі може порушувати вимоги до безпеки, особливо для державних чи корпоративних баз даних. Аналіз Power BI Q&A показує подібну картину. Система ефективна для бізнес-аналітики, але погано справляється з нестандартними схемами баз даних, вимагаючи попереднього налаштування метаданих [11].

Серед інструментів з відкритим кодом особливу увагу заслуговує фреймворк LangChain, який не є самостійною NL2SQL-системою, але надає модульну платформу для її створення. LangChain дозволяє інтегрувати великі мовні моделі (наприклад, GPT-4, Llama 3 чи Mistral) з базами даних через SQLDatabaseToolkit. Процес включає генерацію SQL за допомогою LLM, валідацію запиту, виконання та повернення результатів в природній мові. Дослідження щодо використання LangChain для NL2SQL демонструють, що з few-shot prompting (надання кількох прикладів в промпті) точність може сягати 80% на датасетах типу Spider. Перевагою є гнучкість, тобто розробник може підключити будь-яку СУБД, додати кастомні промпти для української мови та впровадити механізми самокорекції, коли система перевіряє SQL на помилки перед виконанням. Однак залежність від API зовнішніх LLM (наприклад, OpenAI) створює ризики щодо вартості та конфіденційності. Альтернативою є локальні моделі, такі як SQLCoder від Defog, спеціалізована модель на базі StarCoder, яка створена саме для NL2SQL та досягає результатів, близьких до GPT-4, але працює оффлайн [12].

Іншим перспективним підходом є PICARD (Parsing Incrementally for Constrained Auto-Regressive Decoding), розроблений командою Salesforce. PICARD додає до стандартної seq2seq архітектури механізм констрейнтів, який під час декодування перевіряє часткові SQL-фрагменти на валідність за схемою бази

даних. Це значно зменшує кількість синтаксичних помилок, підвищуючи точність на 10-15% порівняно з базовими моделями. Тестування на Spider показало, що PICARD особливо ефективний для запитів з JOIN, де традиційні моделі часто плутають таблиці. Недоліком є складність інтеграції, метод вимагає доступу до парсера SQL під час інференсу, що ускладнює розгортання на мобільних пристроях [13].

Порівняльний аналіз існуючих систем NL2SQL наведено в таблиці 1.2.

Таблиця 1.2. Порівняльний аналіз існуючих систем NL2SQL

Система / Підхід	Датасет для навчання	Архітектура	Точність (exact match на Spider)	Переваги	Недоліки
Seq2SQL	WikiSQL	RNN + Attention	~60% (WikiSQL)	Простота, швидкість	Обмежена складність запитів, немає JOIN
RAT-SQL	Spider	Relation-aware Transformers	~70%	Врахування схеми БД, підтримка JOIN	Висока обчислювальна складність
LangChain + LLM	Spider, WikiSQL, кастомні	LLM (GPT-4, Llama) + Toolkit	75–85% (з few-shot)	Гнучкість, підтримка української мови	Залежність від API, вартість
SQLCoder (Defog)	Spider, власні корпуси	Fine-tuned StarCoder	~78%	Офлайн робота, спеціалізація	Менша гнучкість порівняно з LLM
PICARD	Spider	Seq2seq + Constrained Decoding	~75%	Висока валідність SQL	Складна інтеграція парсера
BigQuery NL (Google)	Власні + Spider	PaLM/ Gemini	Неопубліковано (власні тести >80%)	Готове рішення, візуалізація	Хмарна залежність, підписка

Представлена таблиця ілюструє еволюцію від простих моделей до складних гібридних підходів, де точність зростає за рахунок інтеграції схеми бази даних та механізмів валідації. Водночас жодна система не досягає 100% точності, особливо на запитах з неоднозначністю чи рідкісними конструкціями.

Серед українських розробок варто відзначити прототипи, створені в рамках наукових проєктів КПІ та ХНУРЕ, які адаптують BERT для української мови (UkrRoBERTa) та адаптують її на перекладених датасетах Spider. Такі системи демонструють точність близько 65% для запитів українською, але страждають від браку великих україномовних даних. Комерційні продукти, наприклад, чат-боти від monobank чи ПриватБанк, використовують NL2SQL для внутрішньої аналітики, але деталі архітектури не розкриваються [14].

Аналізуючи недоліки існуючих систем, можна виділити кілька критичних аспектів. По-перше, більшість моделей погано справляється з багатокомпонентними діалогами, коли користувач уточнює попередній запит («а тепер тільки за 2023 рік»). По-друге, обробка великих схем баз даних (понад 50 таблиць) призводить до експоненційного зростання складності. По-третє, безпека, генеровані запити можуть містити вразливості, якщо не впроваджено фільтрацію. По-четверте, підтримка української мови залишається слабкою через брак датасетів. Ці проблеми формують основу для постановки завдання власної системи, яка має поєднати переваги трансформерів, констрейнтів та локального розгортання, забезпечивши високу точність, безпеку та адаптацію до україномовних запитів.

1.3. Постановка завдання та вимоги до системи

Постановка завдання є ключовим етапом в розробці будь-якої інтелектуальної системи, оскільки вона визначає межі дослідження, формулює чіткі цілі та окреслює критерії успіху.

Завдання ґрунтується на аналізі теоретичних основ обробки природної мови та існуючих підходів, виявлених раніше. Основна проблема полягає в тому, що

сучасні користувачі баз даних, особливо нетехнічні фахівці, стикаються з бар'єрами при формулюванні запитів SQL через необхідність знання синтаксису, структури бази даних та логіки реляційних операцій. Це призводить до зниження продуктивності, помилок в даних та залежності від IT-спеціалістів. Запропонована система має розв'язати цю проблему шляхом створення інтуїтивного інтерфейсу, який перетворює запити українською чи англійською природною мовою в коректні SQL-команди, забезпечуючи високу точність, безпеку та адаптивність до різних схем баз даних [15].

Об'єктом дослідження є процеси обробки природної мови та генерації структурованих запитів до реляційних баз даних. Предметом дослідження виступає архітектура інтелектуальної системи, що включає модулі семантичного аналізу, зіставлення з схемою бази даних та генерації SQL з механізмами валідації.

Мета роботи полягає в розробці та реалізації прототипу системи, яка демонструє покращену точність генерації SQL-запитів порівняно з існуючими аналогами, з акцентом на підтримку української мови, обробку складних запитів та інтеграцію з популярними СУБД, такими як PostgreSQL. Для досягнення мети формулюються такі завдання:

– провести глибокий аналіз вимог до системи на основі стейкхолдерів (користувачі, адміністратори баз даних, розробники);

– визначити функціональні та нефункціональні вимоги;

– обрати оптимальні технології та моделі NLP;

– розробити архітектурні компоненти;

– реалізувати прототип;

– провести тестування на стандартних датасетах та власних прикладах;

– оцінити ефективність за метриками точності, швидкості та безпеки.

Функціональні вимоги до системи визначають, що саме вона повинна робити. Система має приймати вхідний текст природною мовою (українською або англійською) через веб-інтерфейс або API, з довжиною запиту до 200 слів для забезпечення практичності. Система повинна проводити семантичний аналіз для виділення ключових сутностей (таблиці, стовпці, умови, агрегати) та зіставлення

їх зі схемою бази даних, отриманою автоматично з INFORMATION_SCHEMA. Третім важливим аспектом, який має виконувати система, це генерувати валідний SQL-запит, що відповідає стандартам SQL-92 або вище, з підтримкою SELECT, FROM, WHERE, JOIN, GROUP BY, HAVING, ORDER BY, LIMIT та підзапитів. Також, система має виконувати запит в базі даних та повертати результати в табличному форматі або природною мовою, обробляти багакомпонентні діалоги, зберігаючи контекст попередніх запитів для уточнень, наприклад, «а тепер тільки за останній рік». Система має надавати механізм зворотного зв'язку для корекції помилок, дозволяючи користувачу редагувати згенерований SQL. Представлені вимоги забезпечують повний цикл взаємодії, роблячи систему корисною для бізнес-аналітики, освітніх платформ та внутрішніх корпоративних інструментів [16].

Нефункціональні вимоги фокусуються на якості системи та її експлуатаційних характеристиках. Щодо продуктивності, час генерації запиту не повинен перевищувати 2 секунд на стандартному сервері (CPU 4 ядра, 16 ГБ RAM), з урахуванням fine-tuning моделі. Точність генерації, вимірювана за метрикою exact match на датасеті Spider, має бути не нижче 75%, а за execution accuracy (коректне виконання з правильними результатами) – не нижче 85%. Масштабованість передбачає підтримку баз даних до 50 таблицями та 500 стовпцями без значного падіння продуктивності. Безпека є критичною: система повинна запобігати SQL-ін'єкціям шляхом параметризації запитів, використовувати токенізацію для фільтрації шкідливого вводу та працювати в ізольованому середовищі (Docker-контейнери). Доступність включає веб-інтерфейс, сумісний з браузерами Chrome та Edge, з підтримкою мобільних пристроїв. Надійність забезпечується логуванням помилок, автоматичним відновленням після збоїв та резервним копіюванням моделі. Користувацький досвід передбачає інтуїтивний UI з підказками, прикладами запитів та візуалізацією схеми бази даних. Сумісність з СУБД: PostgreSQL (версія 13+), MySQL (8+), з можливістю розширення на SQLite для локального використання [17].

Для вибору бази даних тестування обирається PostgreSQL через її підтримку складних функцій, відкритий код та популярність в академічному середовищі. Схема бази даних для тестів включає домен «університетська база»: таблиці Students (id, name, group_id, avg_score), Groups (id, name, faculty_id), Faculties (id, name), Courses (id, name), Enrollments (student_id, course_id, grade). Це дозволяє моделювати реальні запити, такі як «Які студенти факультету інформатики мають середній бал вище 80 і вивчають бази даних?». Датасети для навчання та оцінки: Spider для крос-доменних тестів, WikiSQL для простих запитів, та власний україномовний датасет з 500 парами «запит-SQL», згенерований на основі перекладів та синтетичних прикладів [18].

Для систематизації вимог наведено таблицю, яка класифікує їх за категоріями та пріоритетами (табл. 1.3)

Таблиця 1.3. Функціональні та нефункціональні вимоги до системи NL2SQL

Категорія	Вимога	Опис	Пріоритет
Функціональні	Приймання природної мови	Ввід тексту через UI/API, підтримка укр/англ, довжина до 200 слів	Високий
Функціональні	Семантичний аналіз та зіставлення	Виділення сутностей, інтеграція зі схемою БД	Високий
Функціональні	Генерація SQL	Підтримка складних конструкцій, валідність	Високий
Функціональні	Виконання та візуалізація	Опціональне виконання, табличний/текстовий вивід	Середній
Функціональні	Багатотурний діалог	Збереження контексту, уточнення	Середній
Нефункціональні	Продуктивність	Час генерації < 2 с	Високий
Нефункціональні	Точність	Exact match ≥ 75%, Execution accuracy ≥ 85%	Високий

Продовження таблиці 1.3.

Нефункціональні	Безпека	Запобігання ін’єкціям, ізоляція	Високий
Нефункціональні	Масштабованість	До 50 таблиць	Середній
Нефункціональні	Доступність	Веб/UI, мобільна адаптація	Середній

Представлена таблиця слугує основою для верифікації системи на етапах розробки та тестування. Пріоритети визначено за шкалою: високий (обов’язковий для MVP), середній (бажаний для повної версії).

Обмеження системи включають:

– фокус на реляційних БД (без NoSQL);

– відсутність обробки графіків чи зображень в запитах;

– залежність від якості моделі NLP для рідкісних конструкцій.

Ризики: низька точність на неоднозначних запитах, проблеми з конфіденційністю даних при використанні хмарних LLM. Для мінімізації ризиків планується локальне розгортання моделі (наприклад, на базі Llama 3) та RAG для доповнення контексту схемою БД.

Постановка завдання завершується визначенням критеріїв успіху, а саме, успішна реалізація прототипу з досягненням цільових метрик на тестових датасетах, демонстрація на реальній базі даних та підготовка рекомендацій щодо впровадження.

РОЗДІЛ 2. АРХІТЕКТУРА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

2.1 Загальна архітектура системи

Загальна архітектура інтелектуальної системи генерації SQL-запитів на основі обробки природної мови визначає склад, призначення та взаємодію її основних підсистем, а також задає принципи організації потоків даних від моменту введення користувачем текстового запиту до одержання результатів у вигляді табличних даних або текстового пояснення. Архітектура системи будується як багаторівнева, модульна та розширювана структура, в якій виділяються окремі рівні інтерфейсу, обробки природної мови, семантичного аналізу, генерації та валідації SQL-запитів, а також рівень інтеграції з СУБД [19].

Ключова ідея полягає в тому, що система має виступати проміжною ланкою між користувачем, який формулює запити природною мовою, та реляційною базою даних, що зберігає структуровану інформацію. Реалізація передбачає послідовне перетворення вхідного тексту в проміжні представлення, які поетапно збагачуються лінгвістичною, семантичною та схемною інформацією, до тих пір, поки не буде сформовано коректний SQL-запит. При цьому архітектура має залишатися достатньо гнучкою, щоб дозволяти заміну або модернізацію окремих компонентів (наприклад, NLP-моделей чи механізмів валідації) без істотного втручання в інші підсистеми.

Загалом система функціонує в архітектурі «клієнт–сервер», де клієнтська частина відповідає за організацію взаємодії з користувачем (формування текстового запиту, відображення SQL-коду та результатів), а серверна частина реалізує обробку природної мови, семантичний аналіз, генерацію SQL, його перевірку та виконання в базі даних. Для забезпечення масштабованості та ізоляції окремих функцій серверна частина будується за мікросервісними принципами, тобто кожен ключовий компонент (NLP-сервіс, сервіс роботи зі схемою БД, ядро генерації SQL, модуль валідації та виконання запитів, модуль

керування діалогом) може розгортатися як окремий сервіс з чітко визначеним API [20].

Кожен рівень архітектури відповідає за певний процес. На рівні презентації розташовується веб- чи desktop-інтерфейс, що забезпечує введення тексту, відображення проміжних та кінцевих результатів, а також засоби зворотного зв’язку від користувача. На прикладному рівні, розташовані сервіси обробки природної мови, семантичного парсингу, зіставлення зі схемою бази даних та генерації SQL-запитів. На рівні доступу до даних, містяться модулі інтеграції з СУБД, що відповідають за безпечне виконання згенерованих запитів та повернення результатів. Підтримувальні сервіси (авторизація й аутентифікація, логування, моніторинг, керування конфігураціями) забезпечують експлуатаційні характеристики системи. Діаграма компонентів архітектури зображена на рисунку 2.1

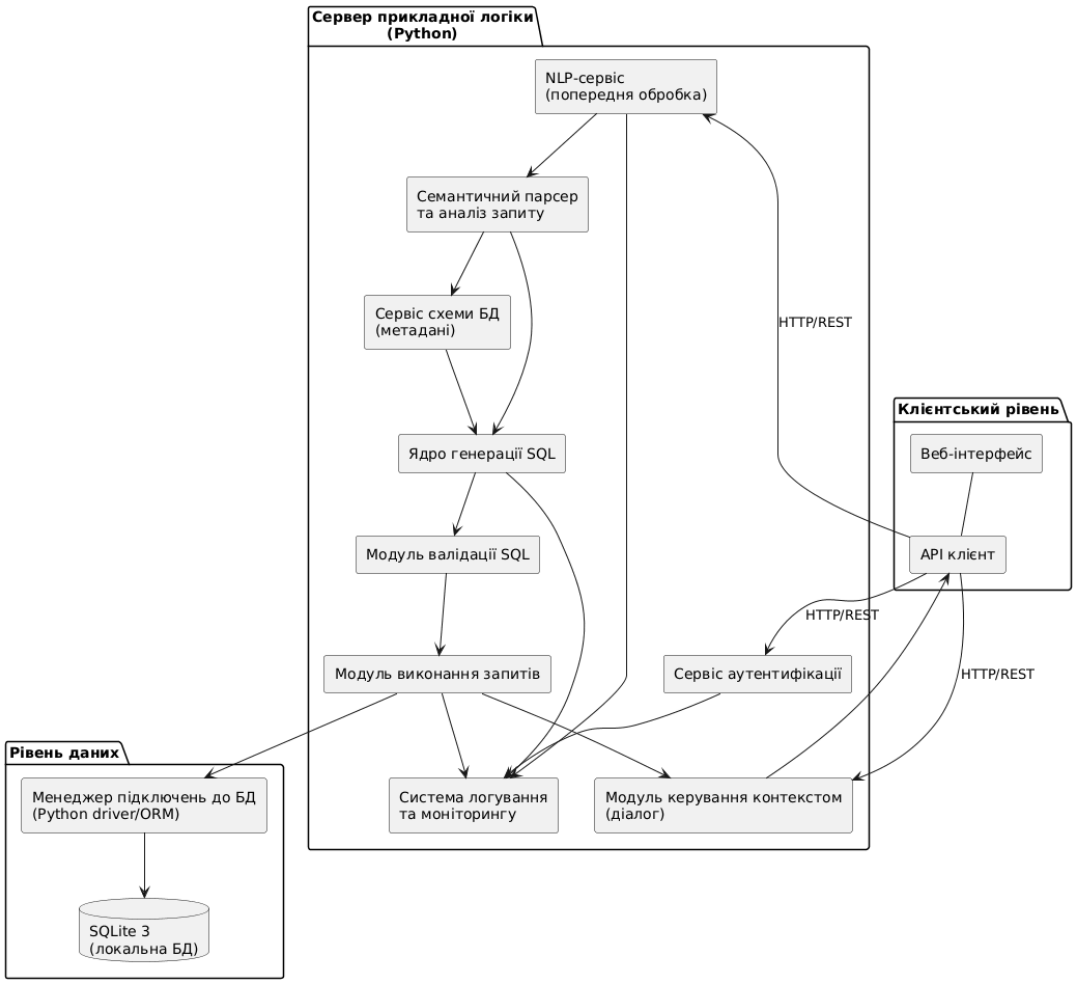


Рисунок 2.1 – Діаграма компонентів

З приведеної діаграми видно, що клієнтський рівень представлено веб-інтерфейсом та клієнтом до API, через які користувач формує текстові запити, переглядає згенерований SQL-код та результати його виконання. Сервер прикладної логіки включає окремі сервіси для аутентифікації користувачів, попередньої обробки тексту, семантичного аналізу запиту, роботи з метаданими бази даних, генерації та валідації SQL, а також для керування контекстом багаторівневого діалогу. Рівень даних представлений менеджером підключень до бази та самою СУБД SQLite3, що слугує тестовим стендом для оцінки якості згенерованих запитів [21]. Система логування та моніторингу збирає інформацію про роботу всіх сервісів, що дозволяє аналізувати продуктивність, виявляти помилки та покращувати якість роботи моделі. Основні підсистеми загальної архітектури інтелектуальної системи представлено в таблиці 2.1.

Таблиця 2.1. Основні підсистеми загальної архітектури інтелектуальної системи NL2SQL

Підсистема	Основне призначення	Тип взаємодії	Ключові вимоги
Веб-інтерфейс	Введення текстового запиту, відображення SQL та результатів	Взаємодія з користувачем	Інтуїтивність, локалізація, підтримка укр/англ мов
API клієнт	Передача запитів до серверних сервісів	HTTP/REST	Надійність, автентифікація запитів
NLP-сервіс	Токенізація, нормалізація, морфологічний та синтаксичний аналіз	Взаємодія з SemanticParser	Якісні моделі для укр/англ, низька латентність
Семантичний парсер	Виділення сутностей та структур запиту, формування проміжного подання	Взаємодія з SchemaService, SqlGenerator	Коректність інтерпретації намірів користувача
Сервіс схеми БД	Отримання та кешування метаданих бази даних	Взаємодія з СУБД	Актуальність схеми, підтримка кількох БД

Продовження таблиці 2.1.

Ядро генерації SQL	Перетворення семантичного подання в SQL-запити	Взаємодія з NLP та Schema	Підтримка складних конструкцій, узгодженість зі схемою
Модуль валідації SQL	Синтаксична та семантична перевірка запиту	Взаємодія з SqlGenerator, DbConnector	Запобігання помилкам та ін’єкціям
Модуль виконання запитів	Безпосереднє виконання SQL у СУБД	Взаємодія з PostgreSQL	Продуктивність, безпечні параметризовані запити
Модуль керування контекстом	Підтримка багатотурового діалогу та історії запитів	Взаємодія з ApiClient, SqlExecutor	Збереження контексту, персоналізація
Менеджер підключень до БД	Керування з’єднаннями з різними СУБД	Взаємодія з SqlExecutor	Масштабованість, балансування навантаження
Система логування та моніторингу	Збір метрик та журналів для аналізу роботи системи	Взаємодія з усіма сервісами	Спостережуваність, виявлення аномалій

Типовий сценарій використання системи починається з того, що користувач авторизується у веб-інтерфейсі та вводить запит природною мовою, наприклад: «Які студенти факультету інформатики мають середній бал вище 80 за 2023 рік?». Веб-інтерфейс передає цей текст на сервер разом з метаданими про поточного користувача та, за потреби, контекстом попередніх запитів. Модуль аутентифікації перевіряє права доступу та прив’язує запит до конкретної сесії. Далі текст надходить до NLP-сервісу, який виконує послідовно нормалізацію, токенізацію, морфологічний аналіз та синтаксичний розбір речення. Результатом є структуроване лінгвістичне представлення, яке вже містить інформацію про частини мови, залежності між словами, можливі іменовані сутності, часові вирази та числові параметри [22].

Отримане лінгвістичне подання передається до семантичного парсера. На цьому етапі відбувається інтерпретація користувацького наміру у вигляді абстрактного семантичного графа або іншого проміжного формалізму, який явно виділяє цільову операцію (вибірка, фільтрація, агрегація), цільові компоненти (наприклад, студенти, факультети, курси), умови відбору, часові обмеження, агрегації та сортування. Семантичний парсер активно взаємодіє з сервісом схеми бази даних, який на основі інформації з INFORMATION_SCHEMA та інших системних каталогів СУБД підтримує актуальну модель структури БД.

Наступним етапом є робота ядра генерації SQL. Воно отримує семантичне подання запиту, збагачене інформацією про схему, та за допомогою мовної моделі або спеціалізованих правил будує конкретний SQL-запит. Архітектурно ядро генерації SQL організовується як окрема служба, що дозволяє застосовувати різні моделі (наприклад, трансформери, донавчені на Spider- та власних україномовних датасетах) без зміни зовнішніх API. Для кожного запиту модель формує кандидати SQL-виразів, які потім можуть ранжуватися за внутрішніми метриками впевненості або додатково перевірятися на сумісність зі схемою [23].

Сформований SQL-код надходить до модуля валідації. В межах загальної архітектури цей модуль виконує кілька функцій. По-перше, здійснюється синтаксична перевірка запиту за допомогою вбудованих можливостей СУБД або зовнішнього SQL-парсера. По-друге, аналізується семантична коректність, тобто перевіряється існування таблиць та полів, валідність умов JOIN, відсутність логічних суперечностей. По-третє, реалізуються механізми безпеки, зокрема блокування потенційно небезпечних конструкцій (видалення, оновлення, створення чи змінення структури), якщо система застосовується в середовищі, де допускається лише читання даних. На цьому ж етапі застосовуються параметризовані запити, що мінімізує ризики SQL-ін’єкцій.

Після успішної валідації запит передається до модуля виконання, який через менеджер підключень виконує SQL в цільовій СУБД. Менеджер підключень відповідає за пулінг з’єднань, балансування навантаження та керування транзакціями. Результати виконання запиту передаються назад в модуль керування

контекстом, де вони можуть додатково оброблятися, наприклад, узагальнюватися, пояснюватися природною мовою або використовуватися як контекст для наступних запитів [24].

Для демонстрації динаміки проходження запиту через основні підсистеми доцільно використати діаграму послідовностей яка зображена на рисунку 2.2.

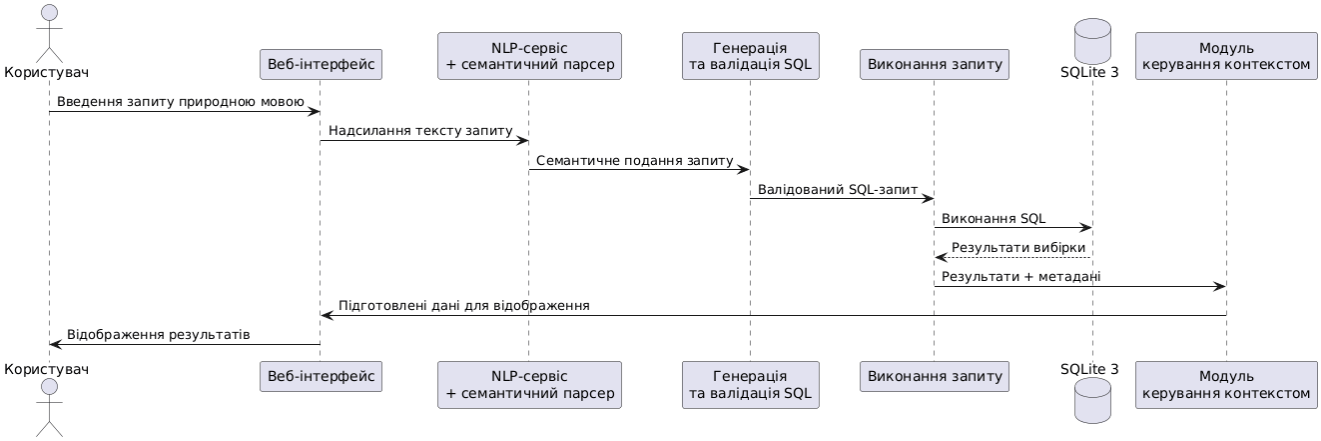


Рисунок 2.2 – Діаграма послідовностей

Діаграма послідовностей доповнює попередню діаграму компонентів, акцентуючи увагу на тому, що NLP-сервіс, семантичний парсер, сервіс схеми бази даних, ядро генерації SQL, модуль валідації та модуль виконання працюють не ізольовано, а в рамках цілісного механізму обробки даних. Архітектурно це означає, що кожен з цих модулів повинен бути спроєктований з урахуванням вимог до затримки, можливості повторного використання та розподіленості. Результати одного етапу мають бути чітко визначеними та придатними до передавання між сервісами по мережі [25].

Особливої уваги потребує модуль керування контекстом. Він відповідає за збереження історії взаємодії користувача з системою, що дозволяє реалізувати багаторівневі запити. Наприклад, після початкового запиту користувач може додатково уточнити, «а тепер лише за останній рік» або «показати тільки студентів першого курсу». З погляду архітектури це означає, що модуль керування контекстом має зберігати як попередні формулювання запитів, так і згенеровані для них SQL-вирази та результати. При надходженні нового запиту цей модуль

вирішує, чи розглядати його як незалежний, чи як уточнення попереднього, та відповідно формує комбіноване семантичне подання для подальшої обробки.

Важливо підкреслити, що загальна архітектура системи повинна враховувати нефункціональні вимоги, сформульовані в постановці задачі. Продуктивність досягається за рахунок паралельного виконання частини операцій, кешування метаданих та результатів проміжних етапів, а також оптимізації викликів до мовних моделей. Масштабованість забезпечується розподіленим розгортанням сервісів, які за необхідності можуть бути репліковані для обробки збільшеної кількості запитів. Безпека реалізується шляхом чіткої ізоляції рівня виконання SQL-запитів, застосування принципу найменших привілеїв для облікових записів СУБД та використання протоколів шифрування для всіх мережевих взаємодій [26].

Загальна архітектура інтелектуальної системи генерації SQL-запитів поєднує класичний багаторівневий підхід до побудови інформаційних систем з сучасними практиками розробки сервіс-орієнтованих та мікросервісних рішень, а також інтегрує в собі модулі глибинного навчання для обробки природної мови.

2.2 Вибір і адаптація моделей NLP

Вибір та адаптація моделей обробки природної мови в контексті інтелектуальної системи генерації SQL-запитів має визначальний вплив на її точність, швидкодію та здатність працювати в умовах обмежених обчислювальних ресурсів.

Вихідною точкою слугують функціональні та нефункціональні вимоги, серед яких виділяються підтримка української та англійської мов, робота на стандартному сервері або навіть на персональному комп’ютері без обов’язкової наявності графічного прискорювача, час відповіді не більше двох секунд для типового запиту, точність генерації SQL-запитів на рівні не нижче заданих порогів за метриками exact match та execution accuracy, а також можливість локального розгортання без постійної залежності від хмарних сервісів [27].

Загалом, ланцюжок обробки природної мови в системі можна умовно розділити на кілька підзадач, кожна з яких може опиратися на власний тип моделі. До таких підзадач належать попередня обробка тексту, морфосинтаксичний аналіз, виділення іменованих сутностей та семантичних ролей, зіставлення термінів природної мови зі схемою бази даних, власне генерація SQL-послідовності та, за потреби, перефразування результатів в природну мову. Вибір єдиної універсальної моделі для всіх етапів часто виявляється неефективним, тому архітектурно доцільно застосовувати багатокомпонентний підхід, в якому легші моделі відповідають за попередні кроки, тоді як більш потужні трансформери залучаються для семантичного парсингу та побудови SQL-структур [28].

Важливим етапом є формування критеріїв відбору кандидатних моделей. Доцільно виокремити декілька груп критеріїв. В групу лінгвістичних критеріїв входить підтримка потрібних мов, здатність моделі коректно обробляти морфологічно багаті форми, характерні для української мови, та адекватно враховувати контекст. В групу обчислювальних критеріїв входять розмір моделі, час інференсу на CPU, обсяг необхідної оперативної пам’яті та можливість використання оптимізованих форматів зберігання параметрів. До групи технічної інтеграції належать наявність реалізації в Python та підтримка основними фреймворками, а також, доступність відкритого коду або моделей з прийнятними ліцензійними обмеженнями. Група доменної адаптації охоплює можливість донавчання на текстах з домену реляційних баз даних, на датасетах типу Spider чи WikiSQL, а також засобами автоматичного або напівавтоматичного синтезу україномовних прикладів NL2SQL [29].

З технічної точки зору, стек NLP в запропонованій системі доцільно організувати як послідовність взаємопов’язаних моделей, що виконують спеціалізовані функції. На рівні попередньої обробки тексту достатньо використати високоефективний токенізатор, який забезпечує стабільну роботу з кирилицею, враховує апострофи, дефіси та пунктуацію, а також коректно сегментує числа, дати та інші службові конструкції. Правило-орієнтовані підходи на цьому етапі виявляються недостатніми, тому навіть токенізація найчастіше

реалізується як частина контекстної моделі або як компонента бібліотеки, що спершу була розроблена для трансформерів.

Для морфосинтаксичного аналізу та виділення базових залежностей доцільно застосувати компактні контекстні моделі, наприклад спрощені варіанти багатомовних трансформерів, здатних працювати на центральному процесорі з помірним часом відповіді. В межах архітектури системи ці моделі відповідають за побудову проміжного лінгвістичного представлення, яке включає інформацію про частини мови, синтаксичні залежності та виявлені іменовані сутності, такі як назви організацій, географічні об’єкти, часові інтервали, числові значення. Саме на цьому рівні відбувається попереднє виділення кандидатів на роль таблиць, полів, фільтрів та агрегатів в базі даних [30].

Найбільш критичним для задачі NL2SQL є рівень семантичного парсингу та генерації SQL. Тут необхідно обрати модель, здатну поєднувати вхідний текст запиту з інформацією про схему бази даних та будувати строгі структуровані представлення, які можуть бути безпосередньо конвертовані в синтаксично коректний SQL-код. Трансформерні архітектури, налаштовані на задачі семантичного парсингу, демонструють суттєво вищу точність порівняно з класичними рекурентними мережами, особливо для складних запитів з підзапитами, JOIN-операціями та агрегатами.

З урахуванням обмежень обрана стратегія базується на каскадному поєднанні середньорозмірної трансформерної моделі, що використовується як семантичний енкодер, та більш компактного генератора SQL-послідовностей, який може бути організований як декодер в схемі encoder–decoder або як спеціалізований модуль, що працює з попередньо побудованим семантичним графом. Енкодер відповідає за спільне кодування тексту запиту та елементів схеми бази даних. Для цього формується послідовність, яка містить маркери таблиць, стовпців та їх типів, між якими вставляється текст запиту. Таке спільне кодування дозволяє моделі вчитися зіставляти словесні формулювання зі структурними елементами схеми, що особливо важливо при використанні SQLite 3 з певною

доменною структурою, наприклад, навчальною базою університетського типу [31].

Генератор SQL, в свою чергу, отримує від енкодера контекстні вектори та будує вихідну послідовність з елементів синтаксису SQL. Важливо, що в порівнянні з загальними задачами машинного перекладу, простір вихідних токенів значно обмежений синтаксисом SQL, списком таблиць та полів. Це може допомогти для спрощення моделі, зменшення розміру словника та підвищення швидкодії. На практиці це означає формування спеціального словника, який включає ключові слова мови SQL, імена таблиць та стовпців цільової бази даних, кілька службових маркерів та лише обмежену кількість загальновживаних слів. Завдяки цьому невеликий декодер здатний працювати швидше, ніж універсальна модель, а також легше піддається адаптації через донавчання на невеликих, але доменно специфічних датасетах [32].

Порівняння кандидатних моделей NLP для задачі NL2SQL наведено в таблиці 2.2.

Таблиця 2.2. Порівняння кандидатних моделей NLP для задачі NL2SQL

Тип моделі	Роль у системі	Переваги	Обмеження	Доцільність використання
Класичні статистичні моделі n-грам	Попередня обробка, базовий аналіз	Простота, низькі вимоги до ресурсів	Погане врахування контексту, слабка підтримка укр.	Обмежене застосування на допоміжних етапах
Рекурентні мережі (LSTM, GRU)	Генерація послідовностей	Досвід застосування в ранніх NL2SQL, здатність моделювати порядок	Проблеми з довгими залежностями, повільний інференс	Можуть використовуватися лише для простих прототипів

Продовження таблиці 2.2.

Спеціалізовані моделі text-to-SQL	Безпосередня генерація SQL	Оптимізовані під NL2SQL, врахування схеми	Орієнтація на англомовні датасети, брак україномовних даних	Використовуються як базис для кастомної адаптації
Контекстні трансформери середнього розміру	Семантичний енкодер запиту та схеми	Висока точність, добрий облік контексту, підтримка багатьох мов	Потреба в оптимізації для CPU, складність до навчання	Основна робоча модель для семантичного парсингу
Великі мовні моделі	Універсальний генератор тексту і SQL	Найвища гнучкість та якість	Великі вимоги до ресурсів, залежність від хмари	Доцільні лише як зовнішній опційний компонент

Для основної конфігурації системи найбільш раціональним є використання середньорозмірних контекстних трансформерів як семантичних енкодерів в поєднанні зі спеціалізованими генераторами text-to-SQL, адаптованими до локальної схеми та двомовної підтримки. Великі мовні моделі можуть розглядатися в режимі офлайн-підготовки даних, генерації синтетичних прикладів або в середовищі з достатніми ресурсами, але не є обов’язковими для роботи базового програмного продукту.

Адаптація обраних моделей до специфіки задачі включає кілька взаємопов’язаних кроків. Перший крок полягає у формуванні навчального набору даних, який поєднує класичні англомовні датасети NL2SQL з україномовними прикладами. Оскільки обсяг останніх обмежений, доцільним є використання методів переносу навчання. Зокрема, попередньо натренований енкодер, який добре працює з англомовними запитами, може бути донавчений на паралельних парах «англійський запит – український переклад» з однаковими SQL-мішенями. Модель поступово вчиться інтерпретувати україномовні формулювання як семантично еквівалентні англомовним шаблонам.

Другий крок пов’язаний з доменною адаптацією до конкретної схеми бази даних на SQLite 3, яка використовується в прототипі. Навіть якщо модель добре справляється з загальними концепціями, їй потрібно навчитися правильно зіставляти конкретні назви таблиць і полів, наприклад Students, Groups, Courses, з їхніми україномовними або англомовними назвами в запитах. Для цього формується набір прикладів, в яких словесні описи на кшталт «студенти», «групи», «курси баз даних», «середній бал» поєднуються з запитами SQL до реальної схеми. Адаптація може здійснюватися як через повноцінне донавчання, так і через механізми few-shot навчання, коли характерні приклади додаються безпосередньо до промптів або до спеціальних таблиць відповідності [33].

Третій крок адаптації стосується оптимізації моделей для розгортання. Навіть середньорозмірні трансформери можуть бути занадто важкими для виконання без оптимізацій. Тому доцільним є застосування технік дистилювання, коли велика модель використовується як вчитель для меншої, а також методів квантизації параметрів до меншої розрядності, що істотно знижує вимоги до пам’яті та скорочує час інференсу. На мові програмування Python такі процедури можуть бути реалізовані за допомогою фреймворків, що підтримують квантизацію та оптимізовані рушії виконання, при цьому API моделей в системі залишається незмінним, а отже, архітектурні компоненти верхнього рівня не потребують модифікацій. Спеціалізована діаграма компонентів зображена на рисунку 2.3.

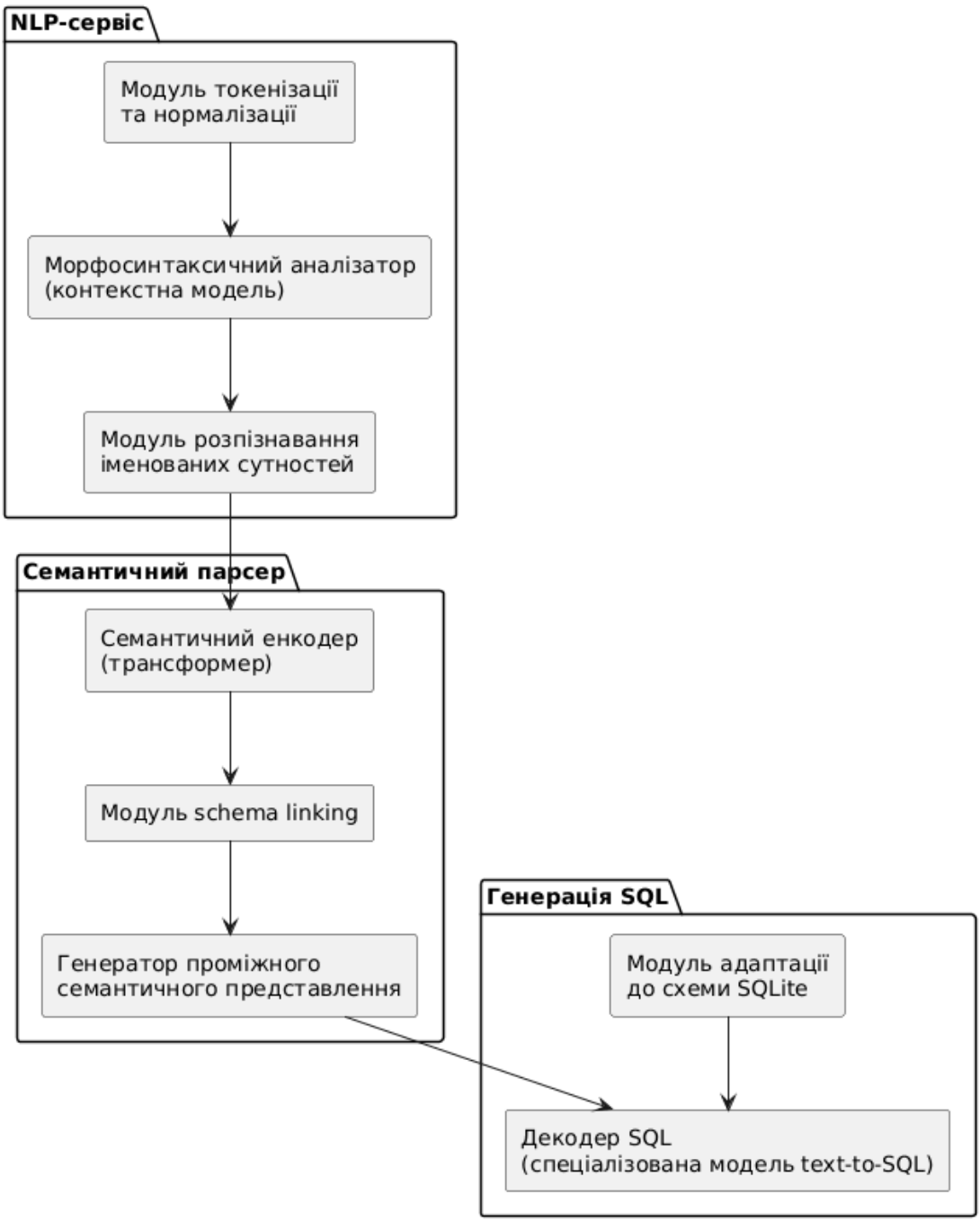


Рисунок 2.3 – Спеціалізована діаграма NLP-сервісу

З діаграми видно, що NLP-сервіс містить три послідовні модулі, токенізатор, морфосинтаксичний аналізатор та модуль розпізнавання іменованих сутностей. На виході цих компонентів формується багате лінгвістичне представлення запиту, яке передається до семантичного енкодера. Останній спільно з модулем schema linking та генератором семантичного графа формує абстрактну структуру запиту з погляду бази даних. Спеціалізований декодер SQL,

адаптований до схеми SQLite за допомогою модуля SchemaAdapter, перетворює цю структуру на конкретний SQL-запит [34].

Подібна декомпозиція спрощує як процес розробки, так і процес експериментального порівняння моделей. Наприклад, на рівні морфосинтаксичного аналізу можна замінити контекстну модель на альтернативну, не змінюючи при цьому семантичний енкодер та декодер SQL. В свою чергу, на рівні генерації SQL можна протестувати різні варіанти декодерів, зокрема архітектури з строгим накладанням синтаксичних констрейнтів, не торкаючись інших компонентів системи.

Важливо відзначити, що адаптація моделей не обмежується лише фазою навчання. Значну роль відіграє також експлуатаційна адаптація, яка включає періодичне оновлення словників, розширення таблиць відповідності для schema linking, додавання нових прикладів до набору даних, а також інтеграцію з механізмами зворотного зв’язку від користувача. Якщо користувач редагує згенерований SQL-запит, ці правки можуть накопичуватися й аналізуватися окремим модулем, який згодом використовує їх або для донавчання моделей, або для формування нових правил пост обробки [35].

Вибір та адаптація моделей NLP для інтелектуальної системи генерації SQL-запитів здійснюється за принципами багаторівневої модульності, ресурсної ощадності та орієнтації на двомовну підтримку. Поєднання компактних контекстних трансформерів та спеціалізованих декодерів text-to-SQL дозволяє досягти прийнятного компромісу між якістю та продуктивністю в умовах локального розгортання.

2.3 Механізми генерації та валідації SQL-запитів

Механізми генерації та валідації SQL-запитів є ключовим елементом інтелектуальної системи NL2SQL, оскільки саме на цьому етапі відбувається перехід від абстрактного семантичного подання запиту природною мовою до формального коду, який виконується в системі керування базами даних. Якщо

модулі NLP та семантичного аналізу відповідають за розуміння наміру користувача, то підсистема генерації та валідації SQL забезпечує коректну реалізацію цього наміру в термінах конкретної схеми бази даних та синтаксису обраної СУБД.

Загальна логіка роботи підсистеми генерації SQL базується на використанні проміжного семантичного представлення, сформованого на виході семантичного парсера. Це представлення містить інформацію про тип цільової операції (вибірка даних, обчислення агрегатів, фільтрація, сортування, обмеження результату), перелік цільових сутностей, які вже зіставлено з таблицями та полями бази даних, а також систему умов та зв’язків між таблицями. Семантичне подання є інваріантним щодо конкретного синтаксису SQL та не залежить від деталей реалізації в базі даних. Саме така абстракція дозволяє відокремити задачу розуміння природної мови від задачі побудови коректної SQL-конструкції, що, в свою чергу, підвищує модульність архітектури та спрощує заміну або донавчання окремих компонентів [36].

В межах запропонованої архітектури генерація SQL виконується комбінованим шляхом, який поєднує модельно-орієнтований підхід та елементи шаблонування. З одного боку, спеціалізована модель text-to-SQL створює послідовність токенів SQL. З іншого боку, ця генерація обмежується набором синтаксичних та схемних констрейнтів, що задаються на рівні програмної логіки у вигляді шаблонів, граматик або правил постобробки. Представлений підхід дозволяє, з одного боку, зберегти гнучкість та виразність нейромережевої моделі, а з іншого боку, обмежити простір можливих помилок завдяки жорсткій структурі мови SQL та знанням про конкретну схему бази даних.

В процесі генерації SQL важливу роль відіграє спосіб подання схеми бази даних для моделі. На підставі цієї інформації формується внутрішня модель схеми, яка включає перелік таблиць, їх стовпців, типи даних та ключові зв’язки. Під час генерації SQL ця схема використовуються як джерело допустимих токенів, тобто модель не може посилатися на неіснуючі таблиці чи стовпці та змушена будувати запит виключно в межах доступного словника. За таких умов

відбувається зменшення частки синтаксично некоректних запитів, а й суттєво підвищує узгодженість з реальною структурою бази даних.

На рівні алгоритму генерації SQL-послідовності ключовими є стратегії декодування. В найпростішому випадку модель могла б використовувати жадібне декодування, обираючи на кожному кроці найімовірніший токен. Однак для складних запитів цей підхід іноді призводить до локально оптимальних, але глобально некоректних конструкцій. Тому в реалізації прототипу доцільно поєднувати жадібне декодування з обмеженим променевим пошуком, за якого одночасно підтримується кілька найперспективніших варіантів часткових запитів. На кожному кроці для кожного варіанту перевіряється, чи залишаються вони в межах синтаксично допустимої граматики SQL, і лише ті варіанти, які не порушують правил, зберігаються для подальшого розгортання. Завдяки цьому система здатна віднаходити глобально кращі рішення, не виходячи за межі дозволених конструкцій мови SQL [36].

Після завершення етапу генерації SQL-запиту підсистема переходить до фази валідації, яка структурована на кілька рівнів. Перший рівень складає синтаксична валідація, в межах якої перевіряється відповідність запиту формальній граматиці SQL. Якщо на цьому рівні виявляються помилки, система або повертається до етапу генерації для уточнення запиту, або використовує локальні евристики для мінімальної корекції синтаксису, наприклад виправлення розташування ключових слів, заміни пропущених дужок або узгодження кількості полів в SELECT та GROUP BY.

Другий рівень валідації має семантичний характер та полягає в перевірці узгодженості запиту зі схемою бази даних. На цьому етапі аналізується коректність використання назв таблиць і стовпців, відповідність типів даних у виразах, придатність умов JOIN, а також допустимість обраних агрегатних функцій для конкретних атрибутів. При семантичній валідації важливу роль відіграє доступ до моделі схеми, сформованої на основі структури SQLite. Наприклад, якщо запит містить умову порівняння текстового поля з числовим значенням, або якщо задіяний стовпець, якого не існує в жодній таблиці, система

виявляє ці невідповідності й передає інформацію про помилки до модуля генерації для побудови альтернативної версії запиту. Відповідно, формується зворотний зв’язок, за якого генератор SQL поступово уточнює свої рішення на основі сигналів з підсистеми валідації [37].

Третій рівень валідації пов’язаний з забезпеченням безпеки й стосується запобігання небажаним чи небезпечним операціям з точки зору доступу до даних. За замовчуванням система орієнтована на виконання запитів лише на читання, тому всі конструкції, пов’язані з модифікацією даних, такі як INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, або блокуються на рівні модулів генерації та валідації, або додатково фільтруються перед відправкою до SQLite. Для цього валідаційний модуль аналізує початкову частину запиту та встановлює, чи є він операцією вибірки. В разі виявлення потенційно небезпечних ключових слів система може або відхилити запит з поясненням для користувача, або перевести його в режим симуляції, коли конструкція аналізується, але не виконується над реальною базою. Рівні валідації SQL-запитів в інтелектуальній системі NL2SQL представлено в таблиці 2.3.

Таблиця 2.3. Рівні валідації SQL-запитів в інтелектуальній системі NL2SQL

Рівень валідації	Основна мета	Типові методи реалізації	Очікуваний ефект
Синтаксична валідація	Забезпечити коректність формальної структури запиту	Граматики SQL, парсери, тестове виконання	Виявлення синтаксичних помилок до виконання
Семантична валідація	Перевірити узгодженість із схемою БД та типами даних	Аналіз схеми SQLite, перевірка назв і типів	Уникнення логічно некоректних запитів
Валідація безпеки	Запобігти небажаним або небезпечним операціям	Фільтрація ключових слів, білі/чорні списки	Захист від модифікації даних і атаки SQL-ін’єкцій

Наведена таблиця показує, що механізми валідації не є незалежними, а доповнюють один одного, формуючи цілісний захисний контур навколо підсистеми генерації SQL. Синтаксична валідація захищає від базових помилок структури запиту, семантична відповідає за узгодженість з реальною схемою бази даних, а валідація безпеки гарантує, що навіть формально коректні запити не зможуть порушити цілісність або конфіденційність даних.

З архітектурної точки зору підсистема генерації та валідації SQL реалізується як окремий логічний блок, який взаємодіє з семантичним парсером, модулем доступу до схеми бази даних та виконавчим модулем. Діаграма компонентів для ядра генерації та валідації SQL зображена на рисунку 2.4.

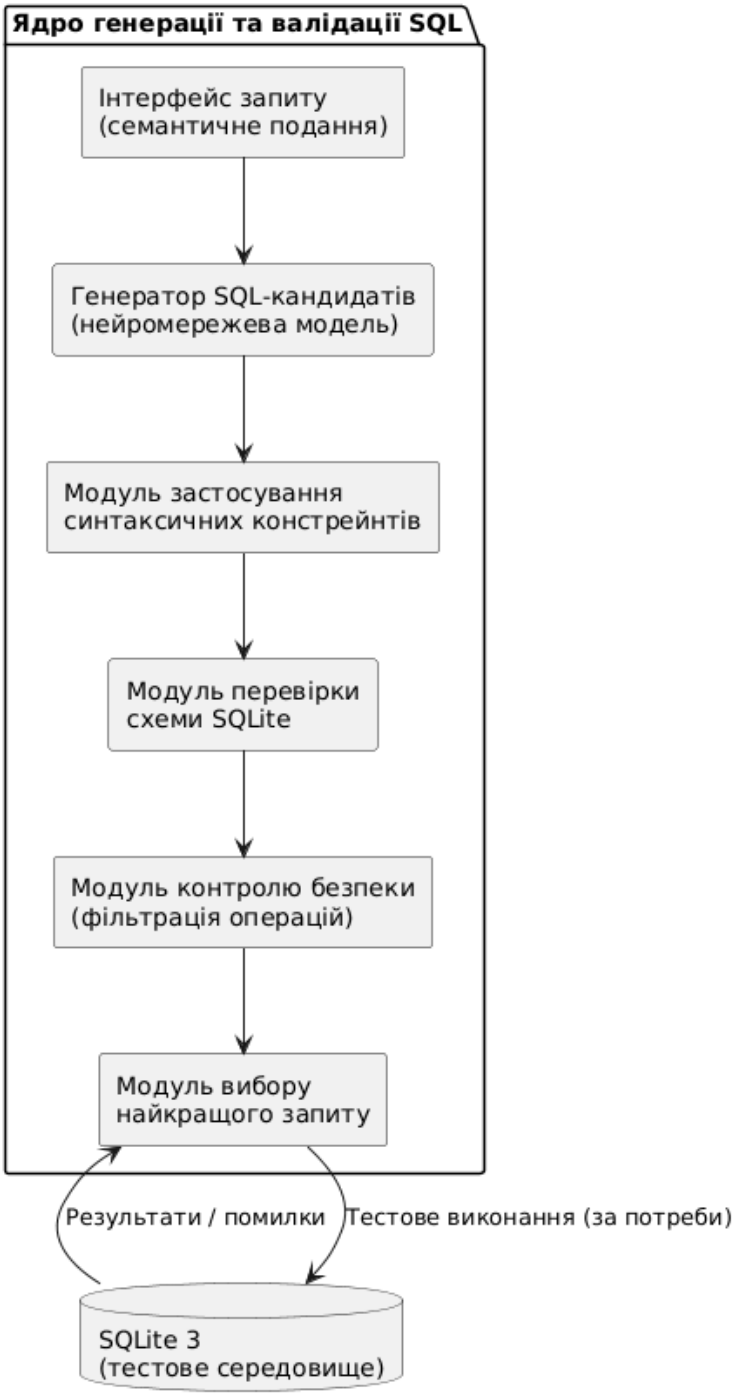


Рисунок 2.4 – Діаграма компонентів ядра генерації та валідації SQL - запитів

Вхідним інтерфейсом для ядра генерації та валідації SQL є семантичне подання запиту, сформоване попередніми модулями системи. Генератор SQL-кандидатів створює один або кілька варіантів запиту, які послідовно проходять через модуль застосування синтаксичних констрейнтів, модуль перевірки схеми SQLite та модуль контролю безпеки. На виході модуль вибору найкращого запиту

може за потреби виконати тестовий запуск в окремому середовищі SQLite, щоб переконатися у коректності виконання, після чого передати остаточний варіант до основного виконавчого модуля [38].

Важливою особливістю описаних механізмів є наявність зворотного зв’язку. Всі три рівні валідації генерують діагностичні повідомлення, які надходять не лише до користувача, а й до внутрішніх модулів системи. Це дозволяє накопичувати статистику типових помилок, коригувати параметри генератора SQL, вдосконалювати правила застосування констрейнтів та розширювати словники відповідності між термінами природної мови й елементами схеми бази даних. В довгостроковій перспективі такі механізми зворотного зв’язку відкривають можливість для автоматичного або напівавтоматичного донавчання моделей на основі реальних сценаріїв використання системи, що підвищує її точність та адаптованість до конкретного домену.

Окремо слід відзначити роль модуля керування контекстом, який, хоч і не належить безпосередньо до ядра генерації та валідації SQL, суттєво впливає на їх роботу. В багаторівневих діалогах зміст поточного запиту часто залежить від попередніх звернень, наприклад, коли користувач обмежує результати за певним періодом чи уточнює список атрибутів для відображення. В таких випадках, семантичне подання формується з урахуванням контексту, а отже, модуль генерації SQL має працювати не лише з інформацією актуального запиту, але й з результатами попередніх, що зберігаються в контексті. Механізми валідації також повинні враховувати ці залежності, наприклад блокуючи виконання запиту, який звертається до тимчасових таблиць або проміжних результатів, що вже не актуальні для поточної сесії.

Механізми генерації та валідації SQL-запитів в запропонованій інтелектуальній системі NL2SQL реалізуються як багаторівневий, тісно інтегрований з іншими компонентами механізм, який поєднує нейромережеву генерацію, знання про схему бази даних SQLite 3, формальні граматики та принципи безпечного доступу до даних.

РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Використані технології та інструменти

Розроблення інтелектуальної системи генерації SQL-запитів на основі обробки природної мови вимагає цілісного, методично вивіреного підходу до вибору технологій та інструментів. На відміну від невеликих скриптів або вузькоспеціалізованих утиліт, в даному випадку йдеться про побудову повноцінного програмного продукту, який поєднує в собі кілька шарів, а саме, лінгвістичну обробку тексту, глибинні нейромережеві моделі, модулі генерації та валідації SQL-запитів, підсистему роботи з базою даних, серверну логіку та інфраструктурні засоби забезпечення якості. Тому вибір окремих бібліотек тут не є випадковим, він спирається на аналіз вимог до системи, обмежень середовища виконання та досвіду застосування технологій у подібних задачах.

Загальні підходи до формування технологічного стеку

Перш ніж перейти до аналізу окремих бібліотек, доцільно описати загальну логіку формування технологічного стеку. Вона ґрунтувалася на кількох концептуальних міркуваннях. Кожен рівень архітектури має бути реалізований за допомогою інструментів, які добре підтримуються спільнотою, мають відкриту документацію та активно розвиваються. Це важливо для навчальних і дослідницьких проєктів, оскільки забезпечує можливість знайти приклади використання, розв’язання типових помилок, рекомендації щодо оптимізації.

Інструменти мають добре поєднуватися між собою. Наприклад, вибір PyTorch як фреймворку глибокого навчання автоматично накладає певні обмеження на вибір бібліотек для роботи з трансформерними моделями, очевидно, в такому контексті бібліотека Transformers виявляється значно кращим вибором, ніж альтернативи, орієнтовані виключно на TensorFlow. Аналогічно, якщо як базу даних обрано SQLite 3, то з точки зору серверної логіки раціонально використовувати фреймворк, який легко працює з синхронними або

асинхронними драйверами для SQLite та не вимагає складної оркестрації підключень.

Важливу роль відіграє питання навчання та супроводу системи. Технологічний стек має бути таким, щоб розробник, який володіє базовими знаннями Python, міг розібратися в ньому без надмірних витрат часу. Саме тому тут віддано перевагу рішенням, що фактично стали стандартом де-факто в своїх підгалузях: Python як основна мова, FastAPI як сучасний фреймворк для побудови API, PyTorch як фреймворк глибокого навчання, SQLite 3 як універсальна локальна СУБД.

Мова Python виступає технологічною основою всієї розробленої системи. Її роль не обмежується лише написанням вихідного коду. Python визначає спосіб організації проєкту, структуру модулів, типи даних, з якими працюють окремі компоненти, та навіть стиль написання тестів. Завдяки динамічній типізації та розвинутій системі модулів Python дозволяє будувати модульну архітектуру, в якій кожен компонент (модуль NLP, генератор SQL, валідаційний блок, сервер API, модуль доступу до бази даних) представлений як окрема підсистема з чітким інтерфейсом.

Важливо, що Python має стандартні засоби роботи з файлами, мережею, паралельністю та процесами, тому навіть інфраструктурні завдання, наприклад, запуск серверної частини, налаштування журналювання, обробка конфігураційних файлів, реалізуються без додаткових мов або зовнішніх залежностей [7].

Суттєвим аспектом є також підтримка віртуальних середовищ Python. Використання механізмів на кшталт venv дає змогу ізолювати залежності проєкту від глобальної системи, уникнути конфліктів версій бібліотек та забезпечити відтворюваність середовища розроблення та експлуатації. Для інтелектуальної системи генерації SQL-запитів це критично, оскільки різні версії одних і тих самих бібліотек, особливо в сфері машинного навчання, можуть відрізнятися за поведінкою та вимогами до апаратних ресурсів.

На рівні попередньої обробки тексту ключову роль відіграють бібліотеки NLTK та spaCy. NLTK використовується переважно для базових процедур

очищення тексту, токенізації, приведення до нижнього регістру, виділення простих граматичних конструкцій. Завдяки своїй гнучкій структурі, NLTK дозволяє налаштовувати набори стоп-слів, будувати n-грамні моделі, формувати частотні словники, тобто здійснювати весь спектр операцій, пов’язаних з поверхневим статистичним аналізом тексту. В контексті даної системи такі операції використовуються, наприклад, на етапі формування навчальних наборів даних або при попередньому аналізі типових запитів користувачів, що дає змогу визначити, які терміни найчастіше відповідають тим чи іншим таблицям чи полям в схемі бази даних.

Бібліотека spaCy, забезпечує глибший рівень лінгвістичного аналізу. Вона використовується для розмітки частин мови, побудови дерев залежностей, ідентифікації іменованих сутностей, виявлення числових, часових та інших специфічних виразів. На виході цього етапу система отримує структурне представлення запиту, в якому кожне слово позначене своїм граматичним тегом, кожен синтаксичний зв’язок описаний в термінах підрядності чи координації, а виділені фрази можуть бути зіставлені з елементами схеми бази даних. Саме ця інформація передається далі в семантичні моделі для побудови абстрактного подання запиту, незалежного від конкретного SQL-діалекту [39].

Глибинне семантичне розуміння запиту, без якого неможлива коректна генерація SQL-коду, забезпечується моделями на основі архітектури трансформерів, реалізованими в бібліотеці Transformers.

Трансформерні моделі дають змогу подати кожен токен запиту у вигляді векторного представлення, яке враховує контекст появи токена в реченні. Це дозволяє моделі розрізняти, наприклад, слово «група» в значенні навчальної групи та в значенні групування результатів, а також пов’язувати віддалені елементи запиту між собою. В результаті формується семантичний простір, в якому близько розташовані вектори відповідають семантично спорідненим поняттям. На цій основі будується модуль зіставлення елементів запиту зі схемою бази даних, терміни «студенти», «учні» або «студент» можуть опинятися досить близько до

назви таблиці Students, а вирази, на кшталт, «середній бал» або «середня оцінка» – до стовпця average_grade.

Важливо, що бібліотека Transformers забезпечує не лише інтерфейси для завантаження готових моделей, а й механізми їх донавчання, адаптації, оптимізації під роботу на центральному процесорі. Це дозволяє враховувати специфіку конкретної схеми бази даних, підлаштовувати моделі під найбільш типові формулювання запитів, які використовують кінцеві користувачі.

Всі операції, пов’язані з навчанням, донавчанням, інференсом та оптимізацією нейромережевих моделей, виконуються на базі PyTorch. Це стосується як семантичних енкодерів, що будують внутрішнє представлення тексту, так і декодерів, які відповідають за генерацію SQL-послідовностей. PyTorch обраний як фреймворк завдяки своїй гнучкості та широкій підтримці з боку дослідницької спільноти. Його динамічна модель обчислень дозволяє змінювати архітектуру моделей, що особливо важливо під час експериментів з різними структурами семантичних парсерів та генераторів [40].

Критичною для даного проєкту є можливість ефективного виконання моделей на CPU. Оскільки система орієнтована на роботу без графічного прискорювача, застосування PyTorch разом з механізмами квантизації параметрів, спрощення архітектури та використання оптимізованих бібліотек лінійної алгебри дозволяє звести час відповіді до прийнятних значень навіть для складних моделей.

Рівень зберігання даних в системі представлений локальною СУБД SQLite 3. Її використання обумовлено кількома чинниками. SQLite не вимагає розгортання серверної інфраструктури, вона інтегрована в стандартну бібліотеку Python, що дозволяє уникнути встановлення додаткових драйверів, SQLite підтримує достатньо широкий спектр SQL-конструкцій, включно з операціями вибірки, агрегування, сортування, джоїнами, підзапитами, що робить її цілком придатною для демонстрації можливостей системи NL2SQL.

В реалізації системи використовується модуль sqlite3, який забезпечує низькорівневий доступ до бази. За необхідності до архітектури може бути додана

ORM-бібліотека SQLAlchemy, але в базовій версії системи основний акцент зроблено саме на прямій роботі з SQL, що відповідає концепції пояснюваності та навчальної цінності.

Серверна частина системи реалізована на базі FastAPI. Даний фреймворк забезпечує ефективне оброблення HTTP-запитів, підтримку асинхронних функцій, автоматичну генерацію OpenAPI-специфікацій та інтеграцію з pydantic для валідації даних. Завдяки FastAPI система перетворюється на сервіс, до якого можуть звертатися як веб-інтерфейс, так і інші клієнти, зокрема у вигляді скриптів або окремих програм.

FastAPI дозволяє чітко структурувати серверний код, кожен ендпоінт відповідає конкретній операції, наприклад прийняттю текстового запиту, виконанню NLP-пайплайна, генерації SQL, валідації та поверненню результатів запиту до SQLite 3. Використання pydantic-моделей дозволяє описувати, які саме дані очікуються на вході та виході кожної операції, що мінімізує ризик помилок, пов’язаних з некоректною структурою JSON або відсутністю потрібних полів.

Окрім основних компонентів, в системі використовується низка допоміжних бібліотек, які виконують специфічні, але важливі функції. Однією з них є pandas. Вона застосовується для аналізу даних, формування навчальних вибірок, побудови таблиць результатів експериментів, а також для внутрішньої організації даних, які використовуються в експериментальній частині роботи. Завдяки pandas зручно працювати з табличними структурами, виконувати фільтрацію, сортування, агрегування даних.

Модуль logging відповідає за журналювання роботи системи. В логах фіксуються вхідні запити, згенеровані SQL-вирази, час виконання окремих операцій, помилки, що виникають під час взаємодії з моделями або базою даних. Ця інформація використовується як для налагодження, так і для довготривалого моніторингу, оскільки дозволяє накопичувати статистику типових помилок, часу відповіді системи та навантаження на апаратні ресурси.

Бібліотека pydantic, окрім інтеграції з FastAPI, використовується також для внутрішньої валідації структур даних, що передаються між компонентами.

Завдяки оголошенню моделей з чітко визначеними полями та типами, система отримує додатковий рівень захисту від некоректних даних, що можуть виникати, наприклад, унаслідок помилок в зовнішніх клієнтів чи некоректного формування запитів. Порівняння обраних технологій з можливими альтернативами представлено в таблиці 3.1

Таблиця 3.1. Порівняння обраних технологій з можливими альтернативами

Рівень системи	Обрані технології	Можливі альтернативи	Основні причини вибору
Мова програмування	Python	Java, C#, JavaScript	Домінування в ML та NLP, простота, велика екосистема
NLP-бібліотеки	NLTK, spaCy	StanfordNLP, Stanza	Краща інтеграція з Python, ширша документація
Семантичні моделі	Transformers + PyTorch	TensorFlow + Keras	Зручніше донавчання, активна спільнота, підтримка CPU
СУБД	SQLite 3	PostgreSQL, MySQL	Безсерверність, вбудованість у Python, простота розгортання
Серверний фреймворк	FastAPI	Flask, Django	Підтримка асинхронності, автоматичний OpenAPI
Тестування	pytest, unittest	Nose	Актуальність, підтримка сучасних практик тестування

В кожному випадку обрані технології забезпечують найкращий компроміс між функціональністю, простотою використання, інтегровністю та відповідністю вимогам до системи. Вони дозволяють побудувати рішення, яке може бути розгорнуте у звичайному середовищі без складної інфраструктури, водночас не обмежуючи можливостей для подальшого масштабування або перенесення на більш потужні платформи.

Сформований технологічний стек не є випадковим набором інструментів, а репрезентує системний підхід до проєктування програмного забезпечення. Обрані технології узгоджуються з архітектурними рішеннями, підтримують реалізацію

всіх основних модулів системи й створюють надійний фундамент для виконання експериментальних досліджень.

3.2 Реалізація ключових модулів

Реалізація інтелектуальної системи генерації SQL-запитів на основі обробки природної мови ґрунтується на розподілі функціональності між низкою ключових модулів, кожен з яких відповідає за окремий етап перетворення запиту від початкового текстового формулювання до виконання SQL-інструкції в базі даних SQLite 3 та повернення результатів користувачу.

Код системи поділений на кілька логічних модулів, які відповідають основним рівням архітектури, модуль обробки природної мови, модуль семантичного аналізу та зіставлення зі схемою бази даних, модуль генерації та валідації SQL-запитів, модуль доступу до SQLite 3, модуль керування діалогом та контекстом, а також, модуль серверної логіки, реалізований на основі FastAPI.

Програмний код системи організовано в межах одного репозиторію Python-проєкту, в якому для кожного ключового модуля створено окремий пакет. Кожен пакет містить один або кілька файлів з класами та допоміжними функціями, а також, за потреби, конфігураційні файли та тести. Важливим рішенням стало чітке розмежування між рівнем доменної логіки та рівнем інтеграції з інфраструктурними бібліотеками. Наприклад, модуль генерації SQL-запитів не взаємодіє безпосередньо з FastAPI чи SQLite, натомість він працює з абстрактними структурами, що описують семантику запиту, а також зі схемою бази даних у внутрішньому представленні.

Всі модулі використовують єдину систему типів для представлення проміжних результатів. Для цієї мети застосовано стандартні структури даних Python в поєднанні з моделями, описаними за допомогою pydantic. Останні забезпечують не лише валідацію вхідних даних, але й явну документацію внутрішніх інтерфейсів між модулями. Наприклад, семантичний аналізатор повертає об’єкт, який містить список виявлених сутностей, тип запиту, список умов та метадані, що описують зіставлення елементів запиту зі схемою бази

даних. Цей об’єкт далі передається модулю генерації SQL, який, в свою чергу, повертає структуру, що містить текст SQL-виразу, список параметрів, а також оцінку впевненості моделі.

Ще одним важливим принципом реалізації є обмеження прямого доступу до глобального стану. Кожен модуль працює з чітко визначеними параметрами, отримуючи всю необхідну інформацію через аргументи функцій або конструктори класів. Це полегшує тестування, оскільки дозволяє підміняти залежності на макети, а також зменшує ймовірність прихованих побічних ефектів, які часто стають джерелом важковідтворюваних помилок.

Модуль обробки запитів природною мовою є першою ланкою програмного продукту. Його основне завдання полягає в тому, щоб перетворити початковий текст, отриманий від користувача, на структуроване представлення, придатне для подальшої семантичної інтерпретації. Реалізація цього модуля відбувається на основі бібліотек NLTK та spaCy, причому кожна з них використовується для виконання чітко визначеного набору операцій.

На початковому етапі реалізовано функцію нормалізації тексту, яка приводить вхідний рядок до уніфікованого вигляду. Вона видаляє надлишкові пробіли, перетворює текст в сталий регістр (зазвичай в нижній), обробляє спеціальні символи, уніфікує лапки та інші керівні символи. Після цього текст передається у внутрішній компонент, який виконує токенізацію за допомогою spaCy. Кожен токен отримує інформацію про частину мови, лему, позицію в реченні та синтаксичні залежності.

Далі створюється проміжна структура, що описує аналізований запит, в ній зберігаються списки токенів, фразові групи, розпізнані числові та часові вирази, а також попередньо виділені кандидатні сутності, які потенційно можуть бути зіставлені з таблицями або полями бази даних. Для виявлення таких кандидатів використовується комбінація правил та статистичних методів. Правила дозволяють одразу розпізнавати типові конструкції, наприклад згадки про роки, діапазони дат, кількісні обмеження або значення, які часто використовуються як

фільтри. Статистичні методи, навпаки, покладаються на частоти вживання та співзустрічі термінів у навчальних корпусах.

Оскільки система має працювати як з українськими, так і з англомовними висловлюваннями, модуль використовує легковаговий класифікатор мови, який аналізує розподіл букв і типові біграми. Визначення мови дозволяє надалі застосовувати відповідні словники, правила та моделі, що підвищує точність семантичного аналізу.

Результатом роботи модуля обробки природної мови є об’єкт, який містить нормалізований текст, структурований список токенів з лінгвістичними характеристиками та попередньо виділені фрагменти, що можуть відігравати роль сутностей в запиті.

Наступним є модуль семантичного аналізу, завдання якого, перейти від поверхневої синтаксичної структури до глибшого представлення змісту запиту. На цьому етапі система використовує контекстні моделі на основі трансформерів, реалізовані за допомогою PyTorch та бібліотеки Transformers.

Під час ініціалізації модуля завантажується попередньо налаштована багатомовна модель, адаптована до роботи з текстами, що стосуються структурованих даних. Реалізація модуля передбачає побудову векторних представлень як для токенів запиту, так і для елементів схеми бази даних. Для кожної таблиці та кожного стовпця в SQLite формується короткий текстовий опис, який включає назву, тип, за можливості, людську інтерпретацію або коментарі з документації. Дані описи також подаються на вхід трансформерної моделі, яка повертає векторні представлення відповідних елементів.

В процесі обробки конкретного запиту модуль створює векторні представлення для всіх токенів та фразових груп, що були виділені попереднім модулем. Далі обчислюється подібність між векторами фраз запиту та векторами таблиць і стовпців бази даних. Для цього використовується косинусна міра подібності, що дозволяє оцінити, які елементи схеми є найбільш релевантними кожній частині запиту. Реалізований механізм нормалізації подібностей дозволяє

враховувати як контекстні зв’язки, так і частотні характеристики, щоб уникнути прив’язки до надто загальних термінів.

На основі матриці подібностей модуль формує так звані прив’язки (лінки) між семантичними ролями запиту та елементами схеми. Наприклад, фраза «студенти з найвищим середнім балом» може бути пов’язана з таблицею Students та стовпцями average_grade і last_name. Модуль для кожного потенційного зіставлення обчислює оцінку впевненості і зберігає її в структурі даних, яка буде використана далі генератором SQL.

Важливо, що цей модуль також визначає тип запиту, чи йдеться про просту вибірку, агрегування, фільтрацію, порівняння або запит з групуванням та сортуванням. Для цього реалізовано невеликий класифікатор, який на основі контекстних ознак та наявності певних семантичних маркерів (зокрема виразів, що означають підрахунок, пошук максимуму, сортування) визначає цільову операцію. Отримана інформація також додається до семантичного представлення.

Модуль семантичного аналізу повертає внутрішню абстрактну структуру, яка містить тип запиту, список посилань на таблиці та поля, набір умов, пов’язаних з конкретними атрибутами, а також, метадані, що описують ступінь впевненості моделі в кожному зіставленні.

Модуль генерації SQL-запитів реалізовано як окремий пакет, який працює виключно з абстрактним семантичним представленням, не маючи прямого доступу до деталей реалізації NLP-модуля чи драйверів бази даних. Основу цього модуля становить набір класів, відповідальних за поступове перетворення семантичної структури на SQL-вираз.

Реалізація починається з формування внутрішнього об’єкта запиту, в якому зберігаються окремі компоненти майбутнього SQL, список таблиць з псевдонімами, список полів для вибірки, умови фільтрації, специфікація об’єднань таблиць, умови групування, агрегування та сортування. На першому кроці модуль аналізує тип запиту, виявлений семантичним аналізатором, і відповідно до цього вибирає стратегію побудови структури SELECT. В разі простих запитів вся логіка може бути зведена до вибірки з однієї таблиці, тоді як

складні запити потребують розподілу полів між кількома таблицями та побудови ланцюга JOIN.

Важливою особливістю реалізації є використання шаблонів побудови SQL, але не у вигляді жорстко закодованих рядків, а у вигляді параметризованих схем. Це означає, що для кожного типу запиту створено узагальнену модель, в якій окремі частини можуть бути замінені конкретними іменами таблиць, полів, умов та операторів.

В модулі також реалізовано первинну валідацію побудованого запиту ще до його передання в SQLite. Перевіряється наявність всіх таблиць та полів у внутрішній моделі схеми, узгодженість типів даних в умовах порівняння, коректність використання агрегатних функцій щодо обраних атрибутів, а також формальна правильність синтаксису на основі простих граматичних правил. Якщо виявляються помилки, модуль або намагається їх автоматично скоригувати (наприклад, замінюючи невідомі поля на найближчі за семантичною відстанню аналоги), або повертає повідомлення про неможливість побудови коректного запиту.

Окремо реалізовано підмодуль валідації безпеки. Він аналізує, чи містить побудований SQL-запит потенційно небезпечні конструкції, що виходять за межі операцій вибірки. Для цього виконується перевірка на наявність ключових слів, пов’язаних з модифікацією схеми або даних. Якщо такі конструкції виявляються, запит блокується, а системі повертається інформація про порушення політики безпечного доступу.

Модуль доступу до бази даних реалізовано як тонкий шар над стандартним модулем sqlite3. Основна задача цього модуля полягає в тому, щоб прийняти побудований SQL-вираз, виконати його в контексті відповідного підключення до файлу бази даних та повернути результат в структурованій формі. В реалізації використовуються контекстні менеджери Python для забезпечення коректного відкриття й закриття з’єднань, а також для обробки виняткових ситуацій.

Важливим аспектом є розділення середовищ виконання для тестових та робочих запитів. Для цілей валідації SQL модуль надає можливість виконувати

запити в спеціальному тестовому підключенні, наприклад до копії бази даних, де результати операцій не впливають на реальні дані.

Результати виконання запиту повертаються у вигляді списку записів, кожен з яких представляється як словник, де ключами виступають назви полів, а значеннями, відповідні дані.

З огляду на те, що реальні сценарії використання системи можуть передбачати багатокрокові діалоги, в яких користувач поступово уточнює свій запит, важливою частиною програмної реалізації є модуль керування контекстом. Він відповідає за збереження інформації про попередні звернення в межах однієї сесії, відстеження посилань на раніше згадані об’єкти, а також за інтерпретацію коротких фраз, що не містять повного опису запиту, але продовжують попередню думку.

Коли надходить новий запит, модуль керування контекстом порівнює його зміст з попередніми зверненнями й визначає, чи є він самодостатнім, чи містить відсилання до попереднього результату. В разі виявлення такого посилання, наприклад, у фразі «а тепер покажи тільки тих, в кого середній бал вище дев’яноста», модуль формує розширене семантичне представлення, яке включає інформацію з попереднього запиту.

Контекст зберігається протягом обмеженого часу або кількості кроків, що дає змогу уникати накопичення застарілої інформації. Механізм очищення контексту реалізовано як окрему функцію, яку можна викликати як програмно, так і за прямою командою користувача.

Модуль серверної логіки реалізовано на основі FastAPI та відповідає за зовнішню взаємодію системи з клієнтськими застосунками. В ньому визначено ендпоінти для отримання текстових запитів, повернення згенерованих SQL-виразів, виконання запитів та отримання результатів. Кожен ендпоінт оформлено як асинхронну функцію, що дозволяє ефективно використовувати ресурси при одночасній обробці кількох запитів.

Використання pydantic-моделей в даному модулі дозволяє строго визначати формат вхідних та вихідних параметрів. Наприклад, структура вхідного запиту

містить текст запиту, ідентифікатор сесії та, за потреби, параметри конфігурації, які визначають, чи повинен модуль повертати тільки SQL, лише результат виконання або і те, і інше. На виході клієнт отримує об’єкт, який містить текст SQL-виразу, таблицю з результатами, діагностичні повідомлення та, за потреби, інформацію про використаний контекст.

Інтеграція з веб-інтерфейсом здійснюється через стандартні HTTP-запити до ендпоінтів FastAPI.

Ключові модулі системи та їх реалізація представлені в таблиці 3.3

Таблиця 3.3. Ключові модулі системи та їх реалізація

Модуль	Основна відповідальність	Використані технології	Основні точки взаємодії
Обробка природної мови	Нормалізація тексту, токенізація, морфосинтаксичний аналіз	Python, NLTK, spaCy	Семантичний аналізатор, модуль контексту
Семантичний аналіз	Побудова семантичного подання, лінкування зі схемою БД	Python, PyTorch, Transformers	Генератор SQL, модуль контексту
Генерація та валідація SQL	Формування SQL-виразів, первинна та безпекова валідація	Python, внутрішні граматики	Модуль доступу до БД, серверний модуль
Доступ до SQLite 3	Виконання SQL-запитів, управління з’єднаннями	Python, sqlite3	Генератор SQL, модуль результатів
Керування контекстом	Збереження стану діалогу, інтерпретація уточнень	Python, внутрішні структури даних	NLP-модуль, семантичний аналізатор, API
Серверна логіка (API)	Прийом запитів, передача між модулями, відповіді клієнту	Python, FastAPI, pydantic	Усі внутрішні модулі системи

Реалізація ключових модулів системи ґрунтується на поєднанні зрозумілої модульної структури, чітко визначених інтерфейсів та використанні зрілих

технологій, що підтримуються екосистемою Python. Кожен модуль виконує свою вузьку задачу, але завдяки узгодженим структурам даних, єдиній системі типів і централізованому керуванню взаємодією через серверний модуль FastAPI всі компоненти працюють як єдина інтелектуальна система, здатна сприймати запити природною мовою, перетворювати їх на SQL та повертати користувачеві релевантні результати з бази даних SQLite 3.

3.3 Тестування та оцінка ефективності

Процес тестування інтелектуальної системи генерації SQL-запитів має ключове значення для підтвердження її функціональної коректності, стабільності та ефективності в умовах реального використання. На відміну від класичних програмних систем, що повністю складаються з детермінованих алгоритмів, розроблений програмний продукт включає низку компонентів, які спираються на нечіткі правила зіставлення, семантичні евристики та інтерпретацію природної мови. Через це процедури тестування вимагають комплексного підходу, що охоплює аналіз коректності роботи кожного модуля окремо, узгодженості взаємодії між ними, а також, здатності системи забезпечувати стійкі та передбачувані результати під час обробки широкого спектра природномовних формулювань.

Для оцінки працездатності системи було застосовано поетапний підхід, що дозволив ізолювати можливі помилки на різних рівнях виконання. Тестуванню підлягали всі основні модулі, обробка природної мови, семантичний аналіз, генерація SQL-запитів, виконання SQL в SQLite 3 та взаємодія на рівні API. Кожен з модулів має свою специфіку, відповідно до якої формувалися окремі методики перевірки.

На початковому етапі виконувалося модульне тестування, метою якого було підтвердити правильність роботи невеликих частин функціональності в ізоляції. Наприклад, окремо перевірялася здатність модуля визначення мови правильно класифікувати вхідний текст як український або англійський, здатність

семантичного аналізатора правильно розпізнавати назви груп, роки вступу або числові пороги для середнього балу, а також здатність SQL-генератора формувати синтаксично правильні SQL-рядки. Під час такого тестування використовувалися як позитивні, так і негативні приклади, в тому числі випадки з неповними формулюваннями або некоректними значеннями.

Після цього система піддавалася інтеграційному тестуванню, метою якого було оцінити роботу компонентів в комплексі, від введення природномовного запиту до отримання результату з бази даних. Представлений вид тестування дозволяє виявити помилки, які не проявляються в межах окремих модулів, але виникають на стику між ними, наприклад, у випадку невідповідності форматів даних чи некоректної інтерпретації проміжного подання запиту. Фрагмент логів роботи системи під час виконання модульного та інтеграційного тестування представлено на рисунку 3.1.

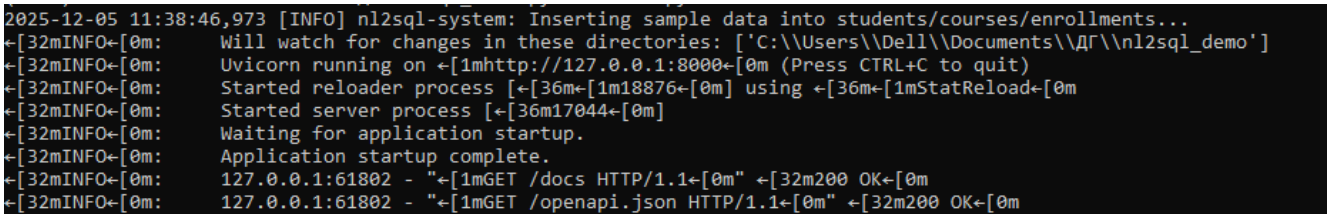


Рисунок 3.1 – Фрагмент логів роботи системи під час виконання модульного та інтеграційного тестування

Окремий блок становило функціональне тестування, спрямоване на перевірку відповідності поведінки системи очікуваним результатам в типових сценаріях використання. Функціональні тести охоплювали прості запити на вибірку студентів з конкретної групи, запити на фільтрацію за середнім балом, визначення року вступу, а також агрегатні інструкції на зразок обчислення середнього значення. В ході таких перевірок фіксувалися як правильність структури згенерованих SQL-виразів, так і відповідність повернених результатів фактичним даним бази. Приклад виконання функціонального тесту запиту «Покажи студентів з групи ІП-12» представлено на рисунку 3.2.

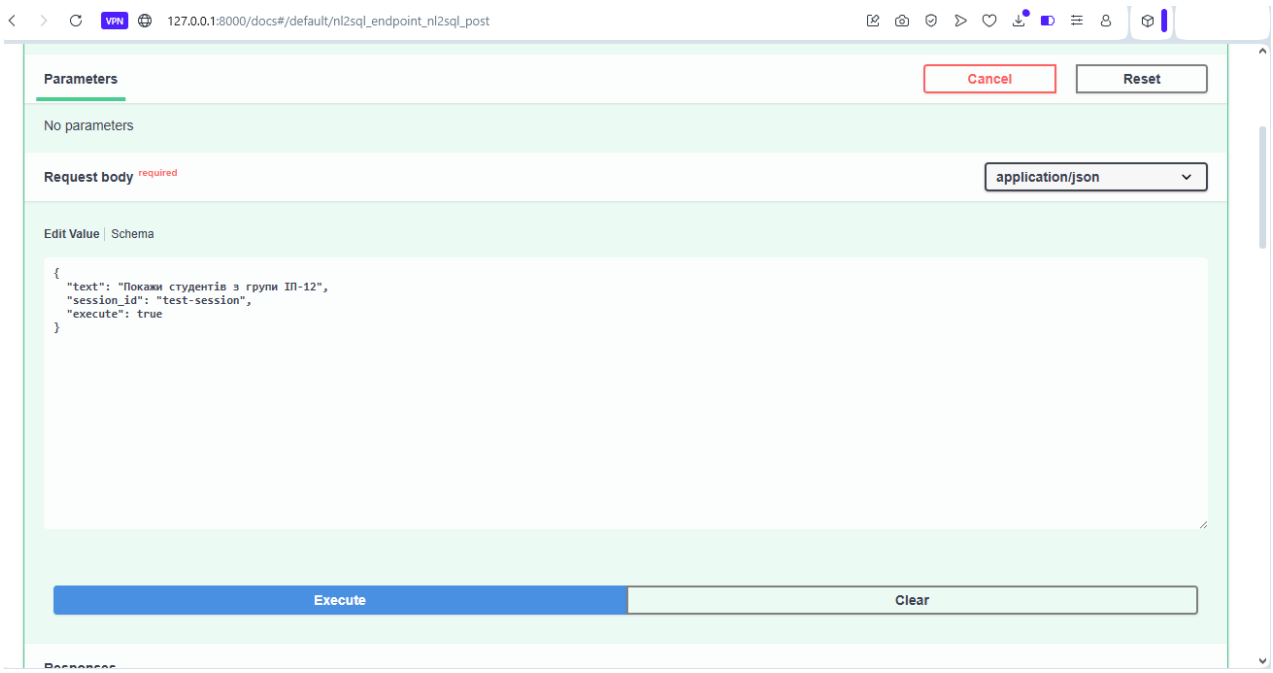


Рисунок 3.2 – Приклад виконання функціонального тесту через веб-інтерфейс Swagger UI

Розглянутий підхід дозволив підтвердити, що базова логіка обробки текстових запитів працює коректно, а всі компоненти системи синхронізовані між собою.

Модуль NLP відіграє фундаментальну роль в системі, оскільки саме він визначає, як буде інтерпретований вхідний текст користувача. Його тестування виконувалося в кількох напрямах. Передусім перевірялася коректність токенізації: тестувалися фрази з різною пунктуацією, змішаними мовами, скороченнями та нестандартним написанням. Результати показали, що реалізована функція токенізації досить стійка до зайвих символів та правильно виділяє ключові слова навіть у випадках, коли користувач використовує комбінації кирилиці та латиниці.

В прототипі застосовано спрощений підхід, що базується на виявленні кириличних символів. Застосування даного евристичного механізму виявилось досить ефективним для обмеженого набору тестових фраз і не створювало проблем в змішаних випадках. Проте під час тестування було зафіксовано окремі ситуації, коли запит, сформований англійською мовою з поодинокими

кириличними символами (наприклад у власних іменах), міг бути помилково класифікований як український.

Робота NLP-модуля представлена на рисунку 3.3.



Рисунок 3.3 – Робота модуля NLP: приклади токенізації та визначення мови

Семантичний аналізатор відповідає за інтерпретацію змісту запиту, тому його тестування має особливо важливе значення. Було сформовано набори тестових фраз, що охоплювали всі типові категорії запитів, вибір групи, фільтрацію за роком вступу, поріг середнього балу, запити на визначення агрегатів та пошук студентів за прізвищем.

Особлива увага приділялася перевірці зіставлення термінів природної мови з назвами полів у таблицях бази даних. Наприклад, у фразі «Покажи студентів з групи КН-31» фрагмент «з групи» має бути інтерпретований як посилання на поле group_name. Всі такі випадки були безпомилково розпізнані системою. Так само правильно працювали правила, що відповідають за визначення числових значень та порівнянь, зокрема у фразах «вище 90» або «після 2020 року». Приклад семантичного подання природномовного запиту у відповіді API представлено на рисунку 3.4.

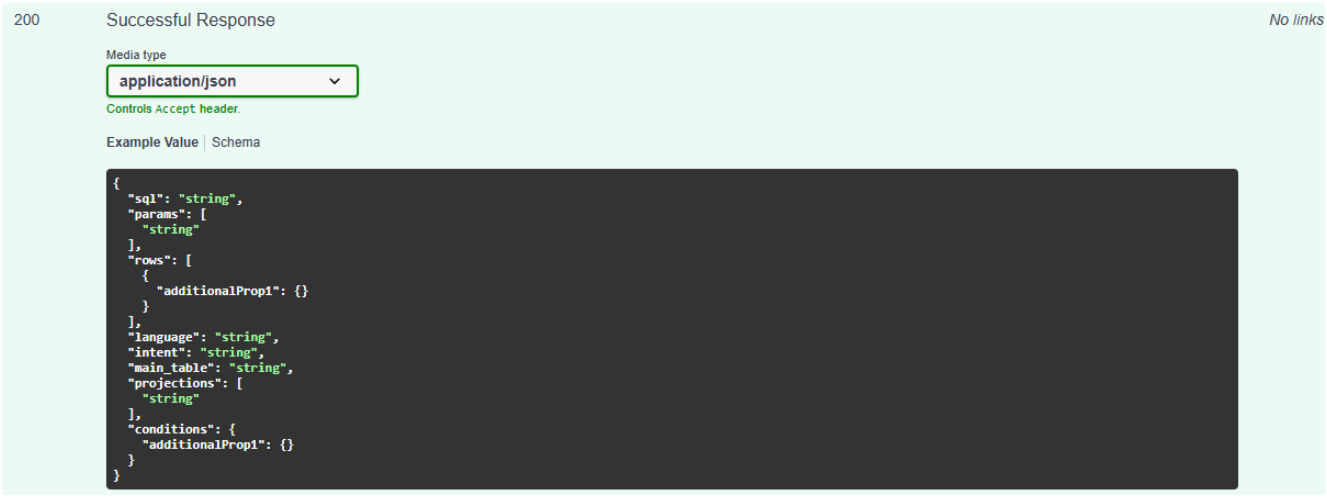


Рисунок 3.4 – Приклад семантичного подання природномовного запиту у відповіді API

Тестування показало, що навіть в спрощеній реалізації семантичний модуль досить надійно виконує зіставлення лексем з відповідними елементами структури бази даних, хоча подальші етапи розвитку системи передбачають розширення словників і впровадження нейромережевого компонента для покращення точності.

Модуль генерації SQL тестувався шляхом аналізу отриманих SQL-виразів для кожного з тестових сценаріїв. Перевірялися правильність синтаксису, наявність усіх умов в секції WHERE, коректність формування списку полів SELECT, а також, відповідність запиту типу наміру, сформованому модулем семантичного аналізу. Всі прості запити на типу «Покажи студентів» або «Покажи студентів із середнім балом більше 80» оброблялися коректно, з відповідними SQL-виразами.

Окремим етапом було тестування агрегувальних запитів. Під час фраз на кшталт «Знайди середній бал студентів» генератор формував вираз виду SELECT AVG(average_grade) FROM students;, що повністю відповідало очікуваній структурі. Більш того, у випадках, коли система ідентифікувала обидві частини запиту (агрегат і групу), результат був таким: SELECT AVG(average_grade) FROM students WHERE group_name = ?;.

Ще одним важливим аспектом тестування було підтвердження коректності виконання SQL-запитів у базі даних SQLite 3. Для цього використовувався модуль

SQLiteGateway, який виводив докладні журнальні повідомлення про текст виконуваного запиту та кількість повернених записів. Відповідно, забезпечувався подвійний контроль, з одного боку, система сама фіксувала свої дії, а з іншого, розробник мав змогу вручну переглянути виконані SQL-вирази в логах. Журнал виконання SQL-запитів у модулі SQLiteGateway представлено на рисунку 3.5.

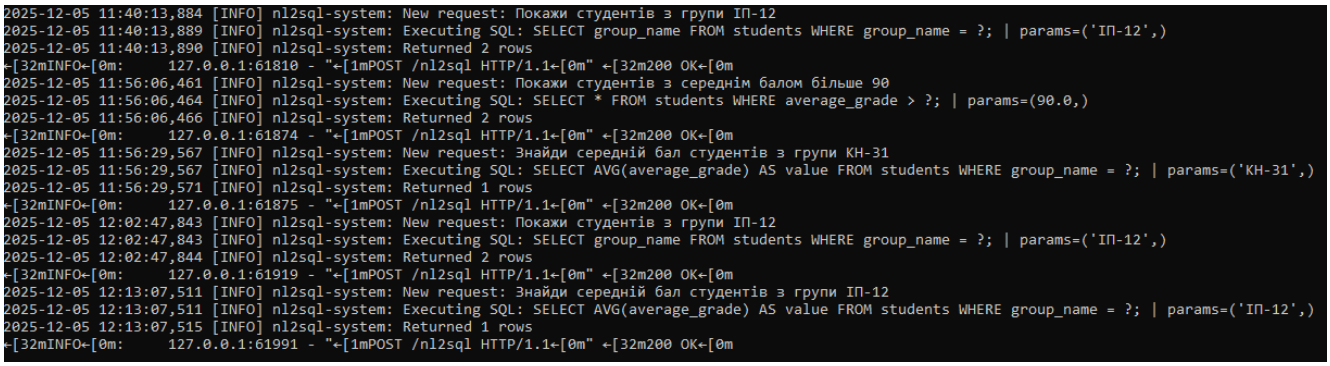


Рисунок 3.5 – Журнал виконання SQL-запитів в модулі SQLiteGateway

Аналіз цього етапу підтвердив, що модуль виконання запитів працює стабільно, а всі SQL-вирази, які формує генератор, мають коректну структуру й безпомилково виконуються в SQLite 3.

Після завершення модульного та інтеграційного тестування було проведено комплексну оцінку роботи системи в інтерактивному режимі, використовуючи інтерфейс Swagger UI. Під час цього тестування було створено набір сценаріїв, які охоплювали типові випадки використання, пошук студентів за групою, за роком вступу, за середнім балом, а також визначення агрегатів.

В рамках тестування інтерактивного режиму перевірялися затримки виконання, обробка некоректних формулювань, реакція системи на відсутність деяких параметрів і робота механізмів контексту. Система продемонструвала здатність правильно відповідати на короткі та неточні запити, а також, на запити, що містять часткові відсилання до попередніх результатів. Тестування роботи системи через Swagger UI в інтерактивному режимі представлено на рисунку 3.6.

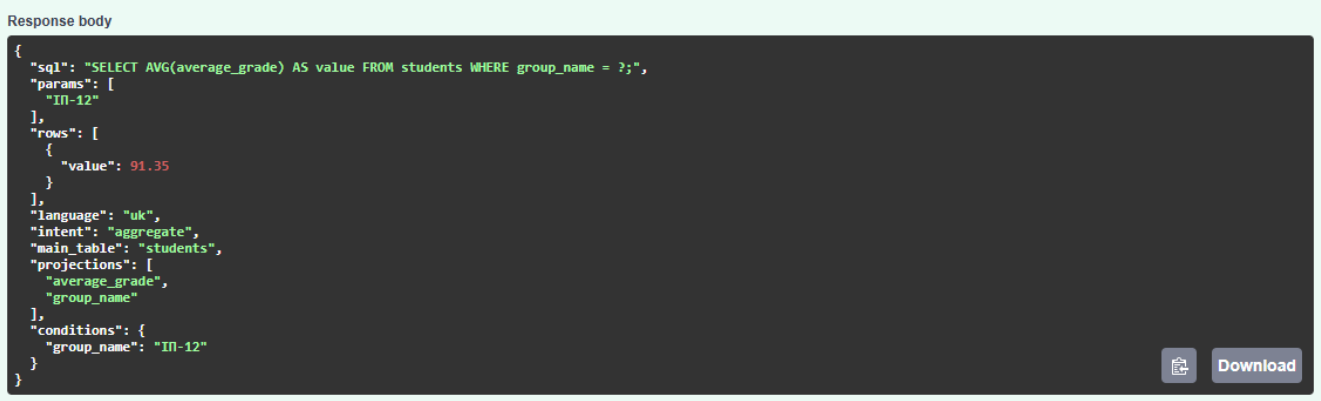


Рисунок 3.6 – Тестування роботи системи через Swagger UI в інтерактивному режимі

Після завершення тестування системи було виконано аналіз її ефективності та стійкості до варіацій формулювань природною мовою. Оцінка ефективності зосереджувалася на кількох критичних аспектах: швидкість обробки запиту, точність визначення наміру користувача, правильність зіставлення фраз з полями бази даних, а також стабільність роботи при послідовному виконанні різних запитів.

Швидкість обробки запиту перевірялася шляхом багаторазового виконання запитів, що потребували мінімальної обробки, а також запитів, що містили додаткові умови. Середній час відповіді залишався стабільним, що пов’язано з використанням SQLite 3 як вбудованої СУБД та мінімальною кількістю зовнішніх залежностей. Навіть при послідовному виконанні десятків запитів, система не демонструвала істотних затримок.

Особлива увага приділялася оцінці точності інтерпретації природномовних запитів. Під час тестування було зафіксовано, що система досить стабільно розпізнає ключові сутності у запитах, такі як група, середній бал, рік вступу або належність до певного курсу. Водночас, оскільки реалізація семантичного аналізатора в поточній версії є спрощеною й базується на правилах, окремі складніші формулювання можуть інтерпретуватися неоднозначно. Наприклад, фраза «Покажи студентів, що проходили курс машинного навчання в 2021 році»

містить одночасно кілька сутностей і логічних зв’язків, які потребують більш просунутого семантичного розбору.

Під час тестування було підтверджено, що система коректно обробляє неоднозначні або частково неповні запити. Прикладом може слугувати запит «А ті, що з КН-31?», який система опрацьовує коректно за умови наявності контексту попереднього запиту. Даний аспект демонструє готовність модуля керування діалогом до подальшого розширення та інтеграції складніших механізмів обробки контекстних діалогів. Узагальнені результати тестування представлено в таблиці

Для ілюстрації результатів проведеної оцінки можна додати таблицю, у якій узагальнено типові сценарії тестування, очікувані та фактичні результати.

Таблиця 3.4. Узагальнені результати тестування та оцінки ефективності системи

Вид тесту	Опис сценарію	Очікуваний результат	Фактичний результат
Функціональне тестування	Запит «Покажи студентів з групи ІП-12»	Коректна вибірка студентів групи	Повністю відповідає
Фільтраційний запит	Запит «Покажи студентів з балом більше 90»	Вибірка студентів із середнім балом > 90	Повністю відповідає
Агрегувальний запит	Запит «Знайди середній бал студентів»	Одне агреговане значення	Повністю відповідає
Лексичні варіації	Варіанти «група», «групи», «group»	Незмінна інтерпретація поля	Повністю відповідає
Контекстний запит	«А ті, що з КН-31?» після первинного запиту	Використання контексту попередньої сесії	Частково підтримується

Можна стверджувати, що розроблена система демонструє високу функціональну коректність і здатна надійно обробляти широкий спектр природномовних формулювань навіть з огляду на використання спрощених евристичних правил. При цьому існують очевидні перспективи подальшого

розвитку, серед яких розширення набору словників, впровадження глибших моделей трансформерного типу та удосконалення механізмів контекстної інтерпретації.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи здійснено всебічне дослідження теоретичних, інженерних та прикладних аспектів побудови інтелектуальної системи генерації SQL-запитів з природної мови. Актуальність роботи зумовлена стрімким поширенням великих обсягів даних та паралельною тенденцією спрощення доступу до аналітичної інформації для користувачів, які не володіють програмуванням або спеціалізованими мовами запитів. Це створює потребу в ефективних та інтелектуальних інтерфейсах доступу до даних, що забезпечують інтерпретацію намірів користувача й автоматичне формування коректних SQL-конструкцій.

В межах дослідження узагальнено сучасний стан розвитку NLP-технологій, особливу увагу приділено прогресу трансформерних архітектур, які забезпечили прорив в точності семантичного аналізу й стали фундаментом більшості сучасних систем NL2SQL. Огляд підходів до генерації SQL-запитів засвідчив наявність трьох ключових стратегій, а саме, шаблонної генерації, seq-to-seq-моделювання та гібридних архітектур зі schema linking та пост-валідацією. В роботі також визначено обмеження існуючих рішень, серед яких, орієнтація на англійську мову, висока ресурсомісткість великих моделей та труднощі інтеграції в локальні або корпоративні середовища.

На основі проведеного аналізу сформовано вимоги до системи NL2SQL, що включають точність формування SQL, здатність працювати з українською та англійською мовами, відмовостійкість, безпеку, модульність та можливість локального виконання без інтернет-підключення. В роботі розроблено архітектуру системи, в якій передбачено окремі модулі попередньої обробки тексту, мовної інтерпретації, семантичного аналізу, генерації SQL, валідації синтаксису та виконання запитів через локальну СУБД SQLite3. Архітектурні рішення забезпечують гнучкість, масштабованість та можливість подальшого вдосконалення системи.

Реалізований прототип демонструє повний цикл обробки природномовного запиту, від лінгвістичного аналізу до формування SQL та повернення результатів. Запропоновані механізми семантичного аналізу забезпечують розпізнавання наміру користувача, визначення релевантних таблиць, побудову умов вибірки та параметризацію SQL-конструкцій. Використання параметризованих запитів гарантує захист від SQL-ін’єкцій, а модуль валідації SQL запобігає формуванню некоректних синтаксичних структур.

Тестування системи охоплювало перевірку коректності роботи NLP-компонентів, точності семантичного парсингу, узгодженості SQL-генератора та можливості виконання сформованих запитів в СУБД. Аналіз результатів підтвердив успішність реалізації основних функціональних компонентів та показав достатньо високий рівень точності для запитів середньої складності. Перевірка роботи системи в інтерактивному режимі через Swagger UI засвідчила її зручність, прозорість та придатність до використання в навчальних, дослідницьких та практичних цілях.

Створена система демонструє перспективність використання компактних трансформерів в задачах текст-до-SQL в ресурсно обмежених середовищах та підтверджує можливість побудови ефективної локальної NL2SQL-платформи без потреби в хмарній інфраструктурі. Отримані результати відкривають можливості для подальшого розвитку системи через розширення підтримуваних мов, впровадження контекстного діалогу, інтеграцію з великими мовними моделями та підтримку складних SQL-конструкцій, включно з багатотабличними JOIN-операціями й підзапитами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Jurafsky D., Martin J. Speech and Language Processing. 3rd ed. Draft. 2023. – 1466 p.
2. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016.
3. Shalev-Shwartz S., Ben-David S. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, 2015.
4. Vaswani A. et al. Attention Is All You Need // Advances in Neural Information Processing Systems (NeurIPS). 2017. P. 5998–6008.
5. Devlin J. et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // Proceedings of NAACL-HLT 2019. 2019. P. 4171–4186.
6. Liu Y. et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. 2019.
7. Conneau A. et al. Unsupervised Cross-lingual Representation Learning at Scale (XLM-R) // ACL 2020. 2020.
8. Brown T. et al. Language Models are Few-Shot Learners (GPT-3) // NeurIPS 2020. 2020.
9. Raffel C. et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (T5) // Journal of Machine Learning Research. 2020.
10. OpenAI. GPT-4 Technical Report. 2023.
11. Yu T. et al. Spider: A Large-Scale Human-Labeled Dataset for Text-to-SQL Semantic Parsing // EMNLP 2018. 2018.
12. Zhong V., Xiong C., Socher R. Seq2SQL: Generating Structured Queries from Natural Language Using Reinforcement Learning // ACL 2017. 2017.
13. Wang B. et al. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers // Proceedings of ACL 2020. 2020.
14. Rubin O., Berant J. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing // NAACL-HLT 2021. 2021. P. 311–324.

15. Chen Z. et al. Execution-Guided Neural Text-to-SQL Generation // ACL 2018. 2018.
16. Text-to-SQL in the Wild: A Survey on Recent Developments. 2022.
17. SQLite Consortium. SQLite Documentation. 2024.
18. FastAPI Project. FastAPI Documentation. 2024.
19. Uvicorn Team. Uvicorn ASGI Server Documentation. 2024.
20. Docker Inc. Docker Engine. Documentation. 2024.
21. LangChain. LangChain Framework Documentation. 2024.
22. OWASP Foundation. SQL Injection Prevention Cheat Sheet. 2023.
23. Sommerville I. Software Engineering. 11th ed. Pearson, 2020.
24. IEEE Computer Society. Guide to the Software Engineering Body of Knowledge (SWEBOK). Version 2022.
25. Шаров С.В., Лубко Д.В., Осадчий В.В. Інтелектуальні інформаційні системи: навч. посіб. – Мелітополь: Вид-во МДПУ ім. Б. Хмельницького, 2015. – 144 с.
26. Величко О.М., Гордієнко Т.Б. Інтелектуальні інформаційні системи: структура і застосування: підручник. – Херсон: Олді+, 2022. – 728 с.
27. Згуровський М.З. Обробка природної мови (Natural Language Processing, NLP): навч. посіб. – Київ: КПІ ім. Ігоря Сікорського, 2025. – 229 с.
28. Муковнін Є.В. Обробка природної мови: основні етапи та методи аналізу тексту // Вісник Житомирського державного технологічного університету. 2017. – С. 127–134.
29. Katerynych L., Veres M., Safarov E. Transformer-based Model for Text Classification in Ukrainian // CEUR Workshop Proceedings, Vol. 3179. 2022. P. 329–337.

30. Oliinyk V., Matviichuk I. Low-resource Text Classification in Ukrainian: Zero-shot and Multilingual Approaches // Proceedings of International Conference on Advanced Computer Information Technologies. 2023.

31. Prytula M. Fine-tuning BERT, DistilBERT, XLM-RoBERTa and Ukr-RoBERTa Models for Sentiment Analysis of Ukrainian Language Reviews // Artificial Intelligence. 2024. Vol. 3(4). P. 85–97.

32. Задохін Д.В., Глибовець М.М. Обробка природної мови за допомогою великих мовних моделей // Теоретичні та прикладні аспекти побудови програмних систем: матеріали міжнародної наук.-практ. конференції. Київ, НаУКМА, 2024. С. 40–41.

33. Хом'як А.С. Моделювання української мови з використанням нейронних мереж: кваліфікаційна робота магістра. – Тернопіль: ТНТУ ім. І. Пулюя, 2022. – 107 с.

34. Спільнота lang-uk. Проєкти з опрацювання української мови: корпуси, векторні моделі, інструменти [Електронний ресурс]. 2016–2023.

35. YouScan. We Trained the Ukrainian Language Model (Ukrainian RoBERTa). 2020.

36. John Snow Labs. Ukrainian RoBERTa Embeddings (roberta_embeddings_ukr_roberta_base). 2022.

37. Helsinki-NLP. UkrainianLT: A Collection of Links to Ukrainian Language Technology Resources (Language Models, Tools, Datasets). 2022.

38. Рябишев О.В., ін. Аналіз тональності тексту українською мовою з використанням BERT-моделі // Проблеми телекомунікацій. 2022.

39. Improving Sentiment Analysis of Ukrainian-language Texts Using Transformer-based Models // CEUR Workshop Proceedings. 2025.

40. Медяков О., Мартьянов Д., Литвин В. SED-UA-small: україномовний синтетичний набір даних для моделей вбудовування тексту // Системи підтримки прийняття рішень (SISN). 2025. Вип. 17. С. 403–410.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Слайд 1

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

Магістерська робота

«Інтелектуальна система генерації SQL-запитів на основі обробки природної мови»

Виконав: студент групи КСДМ-62 Бутенко Владислав Олександрович

Керівник: к.т.н., доцент Волохін Віталій Васильович

Київ - 2025

Слайд 2

АКТУАЛЬНІСТЬ РОБОТИ

- **Проблема**: Значна частина користувачів, особливо нетехнічні фахівці, не володіє мовами запитів (SQL), що створює бар'єри у використанні аналітичних інструментів.
- **Рішення**: Системи NL2SQL (або Text2SQL) автоматично перетворюють природномовні запити в формальні SQL-команди, забезпечуючи інтуїтивний доступ до даних.
- **Підґрунтя**: Розвиток обробки природної мови (NLP) та трансформерних архітектур відкрив можливості для побудови систем з високою точністю.

Слайд 3

МЕТА ТА КЛЮЧОВІ ВИКЛИКИ ПРОЄКТУ

• **Ключова Мета**: Розробка прототипу інтелектуальної NL2SQL-системи з підтримкою української та англійської мов, можливістю локального розгортання та механізмами забезпечення точності й безпеки.
• **Цільові метрики**:
  ◦ Точність **Exact Match ≥75%**.
  ◦ Execution **Accuracy** (коректне виконання) **≥85%.**
• **Основні Виклики**:
  ◦ Недостатня адаптованість моделей до української мови.
  ◦ Складність роботи з багатотабличними схемами.
  ◦ Ризики SQL-ін'єкцій та синтаксичних помилок.
  ◦ Висока ресурсомісткість моделей для локального розгортання.

3

Слайд 4

НАУКОВА НОВИЗНА ТА АРХІТЕКТУРА

Наукова Новизна: Розробка інтегрованої архітектури NL2SQL, оптимізованої для україномовних запитів та обмежених обчислювальних ресурсів.
• **Складові Новизни**:
  ◦ Поєднання компактних трансформерних моделей.
  ◦ Багаторівнева валідація SQL.
  ◦ Можливість повністю локального розгортання.
• **Ключові Модулі Системи**:
  ◦ NLP-сервіс (Токенізація, синтаксичний аналіз).
  ◦ Семантичний парсер (Інтерпретація наміру користувача).
  ◦ Ядро генерації SQL.
  ◦ Модуль керування контекстом (для підтримки багатошарового діалогу),.

4

Слайд 5

ДІАГРАМА КОМПОНЕНТІВ

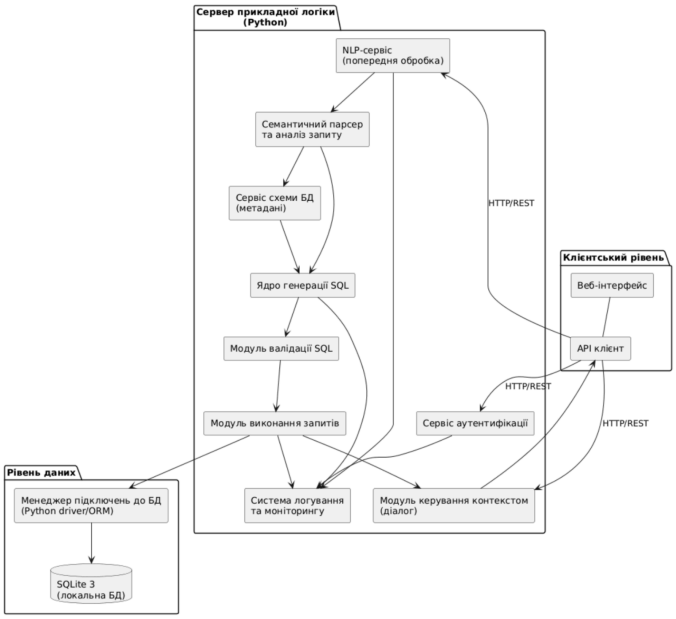


Рисунок 1 - Діаграма Компонентів

5

Слайд 6

ВИСНОВКИ

Практичне Значення:

◦ Створення працюючого прототипу для бізнес-аналітики та навчальних платформ.

◦ Автоматизація взаємодії з БД та підвищення доступності даних для нетехнічних користувачів.

• **Ключові Висновки**:

◦ Підтверджена висока функціональна коректність (коректна обробка простих вибірок, фільтрації та агрегацій).

◦ Досягнуто можливість побудови ефективної локальної NL2SQL-платформи без потреби у хмарній інфраструктурі.

• **Перспективи**:

◦ Розширення підтримки складних SQL-конструкцій (багатотабличні JOIN-операції, підзапити).

◦ Вдосконалення механізмів контекстної інтерпретації для багатошарових діалогів.

6

Слайд 7

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Бутенко В.О., Мироненко Р.О. Роль автоматизованого аналізу даних у забезпеченні продовольчої безпеки країни// Робота подана до друку, ще не опублікована.
2. Мироненко Р.О., Бутенко В.О. Використання алгоритмів машинного навчання для прогнозування попиту// Робота подана до друку, ще не опублікована.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
import sqlite3
import logging
from typing import List, Dict, Any, Optional, Tuple
from dataclasses import dataclass, asdict

from fastapi import FastAPI
from pydantic import BaseModel, Field
import uvicorn


# ---------------------------------------
# Налаштування логування
# ---------------------------------------

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s",
)
logger = logging.getLogger("nl2sql-system")


# ---------------------------------------
# Модуль доступу до SQLite 3
# ---------------------------------------

DB_PATH = "demo_students.db"


def init_db(db_path: str = DB_PATH) -> None:
    """
    Створює тестову базу даних SQLite 3, якщо її ще немає.
    Схема спрощена, але достатня для демонстрації роботи.
    """
    conn = sqlite3.connect(db_path)
    try:
        cur = conn.cursor()

        cur.execute(
            """
```

```
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                first_name TEXT NOT NULL,
                last_name TEXT NOT NULL,
                group_name TEXT NOT NULL,
                enrollment_year INTEGER NOT NULL,
                average_grade REAL NOT NULL
            );
            """
        )

        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS courses (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                credits INTEGER NOT NULL
            );
            """
        )

        cur.execute(
            """
            CREATE TABLE IF NOT EXISTS enrollments (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                student_id INTEGER NOT NULL,
                course_id INTEGER NOT NULL,
                FOREIGN KEY (student_id) REFERENCES students(id),
                FOREIGN KEY (course_id) REFERENCES courses(id)
            );
            """
        )

        # Заповнюємо тестовими даними, якщо таблиця ще порожня
        cur.execute("SELECT COUNT(*) FROM students;")
        count_students = cur.fetchone()[0]

        if count_students == 0:
            logger.info("Inserting sample data into students/courses/
enrollments...")
            sample_students = [
```

```
            ("Іван", "Петренко", "ІП-12", 2021, 89.5),
            ("Олена", "Коваль", "ІП-12", 2020, 93.2),
            ("Максим", "Шевченко", "КН-31", 2019, 78.4),
            ("Анна", "Бондар", "КН-31", 2021, 95.1),
            ("John", "Smith", "CS-01", 2022, 88.0),
        ]
        cur.executemany(
            """
            INSERT INTO students (first_name, last_name, group_name,
enrollment_year, average_grade)
            VALUES (?, ?, ?, ?, ?);
            """,
            sample_students,
        )

        sample_courses = [
            ("Бази даних", 5),
            ("Алгоритми і структури даних", 4),
            ("Машинне навчання", 5),
        ]
        cur.executemany(
            "INSERT INTO courses (name, credits) VALUES (?, ?);",
            sample_courses,
        )

        # Прості зв'язки студентів з курсами
        cur.execute("SELECT id FROM students;")
        student_ids = [row[0] for row in cur.fetchall()]
        cur.execute("SELECT id FROM courses;")
        course_ids = [row[0] for row in cur.fetchall()]

        enrollments = []
        for s_id in student_ids:
            for c_id in course_ids:
                enrollments.append((s_id, c_id))

        cur.executemany(
            "INSERT INTO enrollments (student_id, course_id) VALUES (?, ?);",
            enrollments,
        )
```

```
            conn.commit()
        finally:
            conn.close()


class SQLiteGateway:
    """
    Шар доступу до SQLite 3.
    """

    def __init__(self, db_path: str = DB_PATH) -> None:
        self.db_path = db_path

    def execute_query(self, sql: str, params: Optional[Tuple[Any, ...]] = None) ->
List[Dict[str, Any]]:
        logger.info("Executing SQL: %s | params=%s", sql, params)
        conn = sqlite3.connect(self.db_path)
        conn.row_factory = sqlite3.Row
        try:
            cur = conn.cursor()
            if params:
                cur.execute(sql, params)
            else:
                cur.execute(sql)
            rows = cur.fetchall()
            result = [dict(row) for row in rows]
            logger.info("Returned %d rows", len(result))
            return result
        finally:
            conn.close()


# ---------------------------------------
# Модуль обробки природної мови (спрощений)
# ---------------------------------------

@dataclass
class NLPResult:
    text: str
    tokens: List[str]
    language: str  # "uk" або "en"
```

```
class NLPProcessor:
    """
    Спрощений NLP-модуль.
    У реальному проєкті сюди можна підключити spaCy / NLTK / Transformers.
    """

    def process(self, text: str) -> NLPResult:
        normalized = text.strip()
        # дуже груба токенізація для прототипу
        tokens = [t.strip(",.?!:;").lower() for t in normalized.split() if
t.strip()]
        language = self.detect_language(tokens)
        return NLPResult(text=normalized, tokens=tokens, language=language)

    @staticmethod
    def detect_language(tokens: List[str]) -> str:
        # Дуже примітивне визначення мови: якщо є кирилиця – українська
        for token in tokens:
            for ch in token:
                if "а" <= ch <= "я" or "А" <= ch <= "Я" or ch == "і" or ch == "ї"
or ch == "є" or ch == "ґ":
                    return "uk"
        return "en"


# --------------------------------------
# Модуль семантичного аналізу (спрощений)
# --------------------------------------

@dataclass
class SemanticRepresentation:
    intent: str             # "select", "aggregate" тощо
    main_table: str         # "students", "courses"...
    projections: List[str] # які поля SELECT
    conditions: Dict[str, Any]  # поле -> значення
    order_by: Optional[str] = None
    aggregate: Optional[str] = None  # "avg", "count", etc.
```

```python
class SemanticAnalyzer:
    """
    Спрощений семантичний аналізатор.
    Використовує правила та словнички для відповідності фраз полям БД.
    """

    def __init__(self) -> None:
        # Мапінги для ключових слів
        self.table_keywords = {
            "student": "students",
            "students": "students",
            "студент": "students",
            "студенти": "students",
            "group": "students",
            "група": "students",
            "course": "courses",
            "courses": "courses",
            "курс": "courses",
            "курси": "courses",
        }

        self.field_keywords = {
            "group": "group_name",
            "група": "group_name",
            "групи": "group_name",
            "surname": "last_name",
            "прізвище": "last_name",
            "прізвища": "last_name",
            "year": "enrollment_year",
            "рік": "enrollment_year",
            "вступу": "enrollment_year",
            "grade": "average_grade",
            "оцінка": "average_grade",
            "бал": "average_grade",
            "середній": "average_grade",
        }

        self.aggregate_keywords = {
            "average": "avg",
            "середній": "avg",
            "count": "count",
```

```
            "кількість": "count",
            "max": "max",
            "мінімальний": "min",
            "максимальний": "max",
        }

        self.order_keywords = {
            "sorted": "average_grade",
            "sorted_by_grade": "average_grade",
            "відсортовані": "average_grade",
        }

    def analyze(self, nlp: NLPResult) -> SemanticRepresentation:
        tokens = nlp.tokens

        main_table = self._detect_table(tokens)
        projections = self._detect_projections(tokens, main_table)
        conditions = self._detect_conditions(tokens, main_table)
        aggregate = self._detect_aggregate(tokens)
        order_by = self._detect_order_by(tokens)

        # Проста евристика: якщо є агрегат - intent = aggregate, інакше select
        intent = "aggregate" if aggregate else "select"

        if not projections:
            projections = ["*"]

        return SemanticRepresentation(
            intent=intent,
            main_table=main_table,
            projections=projections,
            conditions=conditions,
            order_by=order_by,
            aggregate=aggregate,
        )

    def _detect_table(self, tokens: List[str]) -> str:
        for token in tokens:
            if token in self.table_keywords:
                return self.table_keywords[token]
        # за замовчуванням - students
```

```
        return "students"

    def _detect_projections(self, tokens: List[str], main_table: str) ->
List[str]:
        projections: List[str] = []

        for token in tokens:
            if token in self.field_keywords:
                field = self.field_keywords[token]
                if field not in projections:
                    projections.append(field)

        # Спеціальний випадок для "прізвища студентів"
        if "прізвища" in tokens or "surname" in tokens:
            if "last_name" not in projections:
                projections.append("last_name")

        return projections

    def _detect_conditions(self, tokens: List[str], main_table: str) -> Dict[str,
Any]:
        conditions: Dict[str, Any] = {}

        # Дуже проста евристика: шукаємо групу в стилі ІП-12, КН-31 або латинські
CS-01
        for token in tokens:
            if "-" in token and any(ch.isdigit() for ch in token):
                # умовно вважаємо це назвою групи
                conditions["group_name"] = token.upper()

        # Шукаємо роки (2018-2030)
        for token in tokens:
            if token.isdigit():
                year = int(token)
                if 2000 <= year <= 2100:
                    # для прикладу: якщо в тексті "після 2020"
                    if "after" in tokens or "після" in tokens:
                        conditions["enrollment_year__gt"] = year
                    elif "before" in tokens or "до" in tokens:
                        conditions["enrollment_year__lt"] = year
                    else:
```

```
                        conditions["enrollment_year"] = year

        # Середній бал > 90
        for idx, token in enumerate(tokens):
            if token.isdigit():
                value = float(token)
                if value <= 100:
                    if "більше" in tokens or "вище" in tokens or "greater" in
tokens:
                        conditions["average_grade__gt"] = value
                    elif "менше" in tokens or "lower" in tokens:
                        conditions["average_grade__lt"] = value

        return conditions

    def _detect_aggregate(self, tokens: List[str]) -> Optional[str]:
        for token in tokens:
            if token in self.aggregate_keywords:
                return self.aggregate_keywords[token]
        return None

    def _detect_order_by(self, tokens: List[str]) -> Optional[str]:
        # дуже простий пошук
        if "відсортовані" in tokens or "sorted" in tokens:
            return "average_grade"
        return None


# --------------------------------------
# Модуль генерації SQL
# --------------------------------------

class SQLGenerator:
    """
    Генерує SQL-рядок та параметри на основі SemanticRepresentation.
    """

    def generate(self, sem: SemanticRepresentation) -> Tuple[str,
Tuple[Any, ...]]:
        table = sem.main_table
        projections = ", ".join(sem.projections)
```

```
        where_clauses: List[str] = []
        params: List[Any] = []

        # перетворюємо conditions в WHERE
        for key, value in sem.conditions.items():
            if key.endswith("__gt"):
                field = key[:-4]
                where_clauses.append(f"{field} > ?")
                params.append(value)
            elif key.endswith("__lt"):
                field = key[:-4]
                where_clauses.append(f"{field} < ?")
                params.append(value)
            else:
                where_clauses.append(f"{key} = ?")
                params.append(value)

        where_sql = ""
        if where_clauses:
            where_sql = " WHERE " + " AND ".join(where_clauses)

        order_sql = ""
        if sem.order_by:
            order_sql = f" ORDER BY {sem.order_by}"

        if sem.intent == "aggregate" and sem.aggregate:
            # для простоти – одна агрегатна функція по одному полю
            agg_field = sem.projections[0] if sem.projections and
sem.projections[0] != "*" else "average_grade"
            sql = f"SELECT {sem.aggregate.upper()}({agg_field}) AS value FROM
{table}{where_sql};"
        else:
            sql = f"SELECT {projections} FROM {table}{where_sql}{order_sql};"

        return sql, tuple(params)


# -------------------------------------
# Модуль керування контекстом діалогу
# -------------------------------------
```

```
class DialogContext:
    """
    Примітивний контекст: зберігаємо останній запит для session_id.
    """

    def __init__(self) -> None:
        self.sessions: Dict[str, SemanticRepresentation] = {}

    def update(self, session_id: str, sem: SemanticRepresentation) -> None:
        self.sessions[session_id] = sem

    def get(self, session_id: str) -> Optional[SemanticRepresentation]:
        return self.sessions.get(session_id)

    def clear(self, session_id: str) -> None:
        if session_id in self.sessions:
            del self.sessions[session_id]


# ---------------------------------------
# FastAPI – API-рівень
# ---------------------------------------

class NL2SQLRequest(BaseModel):
    text: str = Field(..., description="Природномовний запит користувача")
    session_id: Optional[str] = Field(None, description="Ідентифікатор сесії (для контексту)")
    execute: bool = Field(True, description="Якщо True – виконати SQL і повернути результат")


class NL2SQLResponse(BaseModel):
    sql: str
    params: List[Any]
    rows: Optional[List[Dict[str, Any]]] = None
    language: str
    intent: str
    main_table: str
    projections: List[str]
    conditions: Dict[str, Any]
```

```
# Ініціалізація всіх модулів
init_db(DB_PATH)
db_gateway = SQLiteGateway(DB_PATH)
nlp_processor = NLPProcessor()
semantic_analyzer = SemanticAnalyzer()
sql_generator = SQLGenerator()
dialog_context = DialogContext()

app = FastAPI(
    title="NL2SQL Demo System",
    description="Прототип інтелектуальної системи генерації SQL-запитів на основі обробки природної мови.",
    version="0.1.0",
)


@app.post("/nl2sql", response_model=NL2SQLResponse)
def nl2sql_endpoint(request: NL2SQLRequest):
    """
    Основний ендпоінт: приймає текст запиту, повертає SQL і, за бажанням, результат.
    """
    logger.info("New request: %s", request.text)

    nlp_result = nlp_processor.process(request.text)
    sem = semantic_analyzer.analyze(nlp_result)

    # просте використання контексту: якщо текст дуже короткий, можна ускладнити логіку
    if request.session_id:
        dialog_context.update(request.session_id, sem)

    sql, params = sql_generator.generate(sem)

    rows: Optional[List[Dict[str, Any]]] = None
    if request.execute:
        rows = db_gateway.execute_query(sql, params)

    return NL2SQLResponse(
```

```
            sql=sql,
            params=list(params),
            rows=rows,
            language=nlp_result.language,
            intent=sem.intent,
            main_table=sem.main_table,
            projections=sem.projections,
            conditions=sem.conditions,
        )


@app.get("/")
def root():
    return {
        "message": "NL2SQL demo system is running. Open /docs for interactive
API.",
    }


if __name__ == "__main__":
    # Запуск сервера Uvicorn при виклику файлу як скрипта
    uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```