ВІСНИК ПРИАЗОВСЬКОГО ДЕРЖАВНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ
2026р. Серія: Технічні науки Вип. 53. Т. 1
p-ISSN: 2225-6733; e-ISSN: 2519-271X

УДК 004.8:004.4:004.056.5:004.64 DOI: 10.31498/2225-6733.53.1.2026.359767

ІНТЕГРАЦІЯ POLICY LAYER У NL2SQL-СИСТЕМИ ДЛЯ ЗАПОБІГАННЯ ДОСТУПУ ДО ЧУТЛИВИХ ДАНИХ

Морозова А.І. *канд. техн. наук, доцент, Харківський національний університет радіоелектроніки, м. Харків, ORCID: https://orcid.org/0000-0002-7082-4115, e-mail: anna.morozova@nure.ua;*

Новоселова А.С. *магістр, Харківський національний університет радіоелектроніки, м. Харків, ORCID: https://orcid.org/0009-0001-2854-5569, e-mail: anastasiia.novoselova@nure.ua;*

Петрова Р.В. *канд. техн. наук, доцент, Харківський національний університет радіоелектроніки, м. Харків, ORCID: https://orcid.org/0000-0001-5886-8943, e-mail: roksana.petrova@nure.ua*

У статті розглянуто проблему відсутності механізмів контролю доступу в сучасних NL2SQL-системах, які зосереджуються переважно на підвищенні точності генерації SQL-запитів мовними моделями та не враховують ризиків витоку конфіденційних даних. Проаналізовано наявні підходи, що демонструють високу якість перетворення природної мови на SQL-запит, однак не забезпечують перевірку ролей користувачів, політик доступу та чутливості атрибутів. Запропоновано вдосконалену архітектуру NL2SQL-системи з інтегрованим Policy Layer, що являє собою проміжний сервіс, який виконує синтаксичний аналіз запиту, оцінює його ризик, порівнює зі встановленими політиками безпеки та за необхідності автоматично переписує SQL-запит або блокує його виконання. Рішення базується на побудові абстрактного синтаксичного дерева та формальному аналізі структури SQL-запиту, що дозволяє запобігати доступу до чутливих даних незалежно від коректності чи точності роботи мовної моделі.

Ключові слова: *NL2SQL; LLM; велика мовна модель; SQL; Policy Layer; оцінка ризику.*

Вступ

У сучасних інформаційних системах спостерігається стрімке зростання обсягу даних, що супроводжується потребою в зручних механізмах доступу до них. Цей розвиток спричинив появу та активне поширення систем перетворення природньої мови (NL) в SQL (NL2SQL), які дозволяють користувачам формулювати запити до баз даних звичайними фразами без спеціальних знань у галузі програмування чи реляційної алгебри. Завдяки інтеграції моделей машинного навчання та великих мовних моделей (LLM), NL2SQL-підходи стали особливо популярними в аналітичних системах, корпоративних інструментах, чат-ботах, користувацьких інтерфейсах та інтелектуальних помічниках [1].

Проте разом із підвищенням доступності даних, виникла низка проблем, пов'язаних із безпекою даних. Більшість сучасних NL2SQL-систем зосереджуються на точності синтаксичного й семантичного перетворення, але не враховують контекст користувача: його роль, рівень доступу, дозволені операції, а також політики безпеки, прийняті в організації [2]. За відсутності механізмів контролю, сформовані запити можуть розкривати конфіденційну інформацію, ініціювати небезпечні операції чи обходити системи контролю доступу. Отже, виникає потреба в архітектурі, яка доповнюватиме NL2SQL-системи спеціальним шаром безпеки (Policy Layer), у якому будуть зосереджені механізми перевірки політик безпеки, фільтрацію та безпечне переписування згенерованих SQL-запитів перед їх виконанням.

Аналіз останніх досліджень та публікацій

В останні роки спостерігається значне зростання наукового інтересу до проблеми перетворення природньої мови в SQL (Text-to-SQL, NL2SQL). Значна частина робіт спрямована на підвищення точності генерації SQL-запитів за допомогою великих мовних моделей та адаптивних навчальних методик. Проте, незважаючи на відчутний прогрес у якості трансляції NL в SQL, аналіз сучасних публікацій показує, що питання перевірки політик доступу, контролю прав користувача та блокування небезпечних SQL-конструкцій майже не розглядаються. Найбільший фокус досліджень направлений на якість генерації вихідного SQL, обробки складних структур запитів, зменшення помилок у JOIN, удосконалення семантичної відповідності тощо.

У роботі «HITSQL: Human-In-The-Loop Techniques for Enhancing Text-to-SQL Query Generation with Large Language Models» запропоновано модель, спрямовану на покращення точності генерації SQL у корпоративних середовищах на основі прогресивного активного навчання [3]. Автори наголошують на необхідності мінімізувати кількість помилок у трансляції запитів для бізнес-платформи SAP Business One. Методика передбачає поступове збагачення навчальних даних, поєднання експертного зворотного зв'язку та адаптивного донавчання, що забезпечило високий рівень точності – до 97% на незалежних тестових наборах. У статті розглядаються компоненти оцінки виконання SQL, методи порівняння компонентів запиту та семантичні евристики, проте у фокусі перебуває виключно підвищення точності генерації, а не перевірка прав

доступу, не обмеження SQL-операцій і не формування середовища безпечного виконання транслірованих запитів. Автори не описують ані елементів контрольного шару між NL-моделлю і СУБД, ані механізмів, здатних запобігти витокам конфіденційної інформації у випадку коректної, але небезпечної для підприємства генерації SQL.

Подібним чином, у дослідженні «AID-SQL: Adaptive In-Context Learning of Text-to-SQL with Difficulty-Aware Instruction and Retrieval-Augmented Generation» розглянуто адаптивний механізм in-context learning, який дозволяє підвищити ефективність Text-to-SQL трансляції за допомогою класифікації складності запитів, ланцюгових стратегій міркування (Chain of Thought) та retrieval-augmented generation [4]. Автори пропонують інноваційний підхід до вибору прикладів few-shot навчання шляхом оцінки їхньої релевантності за спеціалізованою моделлю ранжування. Усі ці методики спрямовані на покращення розуміння великими мовними моделями семантики природних запитань і синтаксису SQL. Однак, попри значний технічний внесок у сферу адаптивного навчання для NL2SQL, дослідження не розглядає аспекти безпеки, пов'язані з виконанням згенерованих запитів, та не пропонує механізмів перевірки прав доступу, контролю чутливості даних або блокування потенційно небезпечних SQL-операцій.

У статті «A Closed-Domain Natural Language Database Query System by LLMs» автори частково торкаються тематики приватності даних у системах, інтегрованих із великими мовними моделями [5]. У даному матеріалі пропонується реалізація локально розгорнутої системи запитів до структурованих даних, що дозволяє уникнути передачі корпоративних даних у зовнішні LLM-сервіси. Таким чином підвищується рівень приватності та інформаційної безпеки у порівнянні з хмарними рішеннями. Проте навіть у цьому випадку приватність розглядається лише у контексті інфраструктурного розгортання, тобто мінімізації ризиків витоку через зовнішні API. Питання внутрішніх політик доступу, перевірки ролей користувача та контролю за змістом самих згенерованих SQL-команд у статті не розкрито. Система забезпечує конфіденційність отриманих даних, однак не містить механізму аналізу та фільтрації SQL-запитів, що дозволяють або забороняють конкретні дії відповідно до регламентів підприємства.

Окремий підхід до забезпечення безпечної генерації SQL-запиту пропонується у статті «How to Safely Use LLMs for Text-to-SQL with Stored Procedures», у якій автор описує можливість реалізації функції перевірки політик доступу безпосередньо на LLM [6]. Основна ідея полягає в тому, що LLM не лише транслює природню мову в SQL, а й повинна одразу формувати «безпечний» запит, який відповідає заздалегідь визначеним правилам. У статті наголошується, що замість використання додаткової логіки в застосунку або окремого валідаційного шару, модель має бути навчена або сконфігурована таким чином, щоб ніколи не генерувати SQL, який порушує встановлені вимоги. Автори пропонують застосування збережених процедур, шаблонних промптів і вбудованих правил у самій LLM для обмеження типів операцій та дозволених таблиць.

Попри практичну цінність цього підходу, він має суттєве обмеження: контроль безпеки повністю делегується моделі, яка за своєю природою є стохастичною і може помилятися. Лінгвістичні та генеративні помилки LLM, а також її схильність до «галюцинацій», можуть призвести до формування некоректних або небезпечних SQL-запитів навіть за наявності інструкцій [7]. Це означає, що такий підхід не забезпечує гарантованого виконання політик, оскільки відсутній незалежний валідатор, який би аналізував згенерований SQL до його передачі в базу даних. Таким чином, хоча ідея інкорпорування політик у LLM є перспективною для зниження кількості помилок, вона не може повністю замінити архітектурний Policy Layer, який забезпечує формальний, визначальний і передбачуваний рівень контролю безпеки.

Таким чином, після аналізу вищеперелічених робіт, можна зробити висновок, що більшість Text-to-SQL досліджень концентруються на:

– підвищенні точності генерації SQL;

– семантичній відповідності між NL-запитом і SQL-структурою;

– оптимізації навчальних наборів, інструкцій та політик LLM;

– зменшенні помилок у складних сценаріях (JOIN, вкладені запити, агрегації).

Навіть у роботах, де порушується питання приватності або безпеки, основний фокус зводиться до інфраструктурних аспектів розгортання або до спроб навчити саму LLM генерувати «безпечні» запити, не застосовуючи незалежного механізму формального контролю.

Зважаючи на проведений аналіз, можна виокремити актуальна дослідницьку нішу, що сфокусована на архітектурному проєктуванні та реалізації Policy Layer. Ключове призначення цього компонента полягає в перевірці згенерованого NL2SQL-системою SQL-запиту на відповідність політикам безпеки компанії.

Висновки, отримані в результаті огляду наукової літератури, свідчать про нестачу уваги до систематичного запобігання несанкціонованого доступу до даних через згенеровані SQL-запити, що підтверджує наукову новизну досліджень у сфері розробки Policy Layer для NL2SQL-систем.

Мета статті

Метою дослідження є архітектурне проєктування та реалізація компонента Policy Layer у складі NL2SQL-системи, що забезпечує детермінований і формальний контроль безпеки шляхом синтаксичного аналізу згенерованого SQL-запиту, комплексної оцінки ризику (на основі чутливості атрибутів,

p-ISSN: 2225-6733; e-ISSN: 2519-271X

небезпечності операцій та рівня розкриття даних) та, за необхідності, автоматичного переписування запитів для гарантованої відповідності корпоративним політикам доступу до даних.

Постановка проблеми

Нехай розглядається компанія з розробки програмного забезпечення, у якій працюють співробітники різних посадових рівнів: розробники, тестувальники, менеджери проєктів, аналітики, а також працівники фінансового відділу, зокрема бухгалтер, який має офіційний доступ до інформації про заробітні плати та фінансові операції. Усі інші співробітники, включно з технічним персоналом, не повинні мати доступу до конфіденційних фінансових даних, оскільки ці відомості регулюються внутрішніми політиками безпеки та юридичними нормами щодо захисту персональних даних.

Розглянемо сценарій використання системи NL2SQL на вищеописаному підприємстві. Співробітник компанії, який займає посаду розробника програмного забезпечення (ПЗ), формує запит вигляду: «Покажи заробітні плати всіх співробітників компанії». NL2SQL-модель опрацьовує цей запит і генерує синтаксично коректний SQL-вираз, який може мати вигляд: SELECT full_name, salary FROM employees. Якщо використовувана NL2SQL система не має механізму перевірки політик, такий запит буде виконано, і користувач фактично отримає доступ до конфіденційної інформації, яку бачити не повинен. Проблема ускладнюється тим, що модель, згенерувавши SQL, не враховує ні рівень доступу користувача, ні списки дозволених таблиць, ні обмеження на чутливі поля. Вона працює виключно з текстом і статистичними закономірностями, а не з політиками безпеки чи ролями, визначеними в організації. Навіть якщо сформований SQL є синтаксично і семантично правильним, він може становити загрозу для безпеки. Це можна формалізувати через інтегральну оцінку ризику SQL-запиту:

$$R = w_1 * S + w_2 * O + w_3 * E, \quad (1)$$

де R – інтегральна оцінка ризику SQL-запиту;

S – ступінь чутливості атрибутів (Sensitivity);

O – небезпечність виконуваної операції (Operation Severity);

E – рівень потенційного розкриття даних (Exposure Level);

w_1, w_2, w_3 – вагові коефіцієнти, що визначають важливість кожної складової.

У розрахунку ризику параметр S зростає, коли SQL-запит звертається до чутливих даних, таких як заробітна плата (salary) чи персональні дані співробітників, такі як платіжні реквізити, домашня адреса тощо, тобто до даних, що підлягають посиленому захисту. Значення параметра O збільшується у випадках, коли операція, яку виконує запит, є потенційно небезпечною: наприклад, команда DELETE або UPDATE без умови WHERE становить підвищений ризик неконтрольованої модифікації даних. Параметр E характеризує рівень можливого розкриття інформації та набуває більшого значення, якщо запит повертає великі обсяги даних, наприклад, не містить обмеження LIMIT або охоплює всю таблицю. Вагові коефіцієнти w_1, w_2, w_3 визначаються політикою безпеки організації й задають відносну важливість кожного з вказаних факторів у загальній оцінці ризику запиту.

У випадку із запитом про заробітні плати ризик є високим, тобто: R(Query) >> 0, але NL2SQL-модель не завжди здатна це оцінити. Вона працює виключно з текстом і статистичними закономірностями, не маючи уявлення про роль користувача, дозволені таблиці, обмеження на чутливі поля чи внутрішні політики безпеки підприємства. У результаті навіть «коректний» із технічної точки зору SQL-запит може порушувати вимоги безпеки та призводити до витоку даних.

Таким чином, основна проблема полягає в тому, що NL2SQL-моделі залишаються «сліпими» до контексту користувача та корпоративної політики безпеки, а отже, можуть неявно порушувати правила доступу до даних. Це вимагає розробки додаткового захисного рівня – Policy Layer, який аналізує згенерований SQL, виявляє небезпечні конструкції, забороняє несанкціоновані операції та забезпечує відповідність запиту політикам підприємства.

Виклад основного матеріалу

У загальному вигляді робота NL2SQL-системи полягає в автоматичному перетворенні NL запиту користувача на SQL-запит, який може бути виконаний у реляційній базі даних. На першому етапі користувач формує запит у текстовій формі, наприклад: «Покажи заробітні плати всіх співробітників». Цей запит надходить у застосунок, який виступає проміжною ланкою між користувачем, моделлю та базою даних. Далі застосунок передає отриманий текст великій мовній моделі (LLM), формуючи відповідну підказку (prompt) для перетворення природної мови на SQL. Модель аналізує текст, виконує структурну та семантичну інтерпретацію запиту і на основі цього генерує SQL-команду, наприклад: SELECT full_name, salary FROM employees.

Процес перетворення запиту включає кілька послідовних етапів обробки. Модель спершу розпізнає ключові елементи NL запиту – такі як поняття «заробітні плати» та «співробітники» – і зіставляє їх із відповідними об’єктами бази даних, тобто таблицями та колонками, у цьому випадку із таблицею employees та полем salary. Після цього LLM визначає структуру потрібного SQL-запиту: встановлює, що користувач прагне прочитати дані, а не модифікувати їх, і формує оператор SELECT. У типовій NL2SQL-системі ці етапи охоплюють розпізнавання наміру, виявлення сутностей і їх прив’язку до схем бази даних (schema linking), що й дозволяє отримати готовий SQL-запит [8].

На останньому етапі застосунок приймає SQL-запит, надісланий моделлю, виконує його у базі даних

p-ISSN: 2225-6733; e-ISSN: 2519-271X

і повертає відповідь користувачу. Таким чином, система приховує для користувача механізм взаємодії з базою даних: він вводить звичайне текстове питання, а система автоматично обробляє його, транслює у SQL та повертає результат. Поданий на рисунку 1, процес демонструє вищеописаний цикл перетворення NL у SQL-запит: від запиту користувача, до генерації SQL-команди моделлю, і, у кінці, до одержання результатів виконання без необхідності ручного написання запитів.

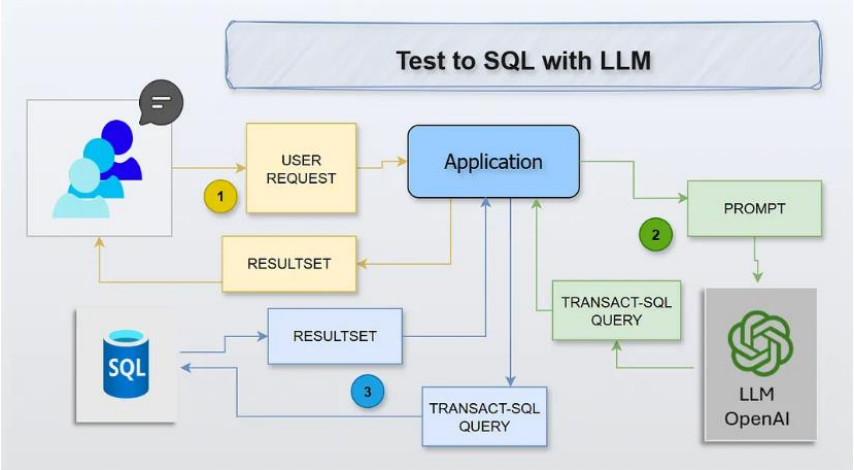

Рис. 1 – Загальна архітектура системи NL2SQL [6]

З рис. 1 видно, що у стандартних архітектурах систем перетворення природної мови в SQL, відсутній будь-який проміжний механізм контролю згенерованого запиту перед його виконанням у базі даних. Запит, згенерований великою мовною моделлю, передається застосунком безпосередньо до СУБД, де виконується без додаткової перевірки на відповідність політикам доступу, ролям користувачів або чутливості даних. Така схема орієнтована на автоматизацію та зручність, але не враховує ризики, пов'язані з некоректною або небезпечною генерацією SQL-команд. Відсутність Policy Layer означає, що безпека цілком залежить від якості та передбачуваності моделі, що створює значні загрози в корпоративних середовищах, де доступ до даних суворо регламентується [9].

З огляду на те, що типовим NL2SQL-системам бракує механізмів контролю безпеки згенерованих SQL-запитів, виникає потреба у впровадженні спеціального архітектурного прошарку – Policy Layer. Існує кілька варіантів його технічної реалізації, які відрізняються розташуванням у системі, рівнем доступу до інформації та можливістю впливати на структуру SQL. Першим підходом є створення Policy Layer у вигляді зовнішнього SQL Proxy Firewall, розміщеного між NL2SQL-моделлю та базою даних [10]. У цьому випадку всі запити проходять через незалежний фільтр, що аналізує синтаксис і структуру SQL, перевіряє відповідність ролям користувачів та корпоративним правилам доступу. Такий варіант забезпечує високий рівень безпеки, оскільки перевірка здійснюється ізольовано від логіки основної системи, проте він потребує складнішої інфраструктури та може впливати на продуктивність у масштабованих середовищах.

Іншим напрямом є інтеграція Policy Layer безпосередньо в NL2SQL-сервіс як внутрішнього модуля. У цьому випадку механізм контролю працює зі згенерованими SQL-запитами до їх передачі в базу даних, що дозволяє не лише блокувати небезпечні операції, але й переписувати SQL для приведення його у відповідність до політик. Такий підхід забезпечує найбільшу гнучкість, оскільки може використовувати інформацію про роль користувача, контекст запиту та специфіку даних, а також зберігає можливість динамічної адаптації політик без змін у СУБД. Третій можливий підхід передбачає переміщення функцій Policy Layer безпосередньо на рівень бази даних, інтегруючи їх через механізми плагінів або розширень. Подібні рішення вже існують у комерційних СУБД, де системи firewall-типу контролюють SQL, що надходить до БД, блокуючи запити, які не відповідають затвердженим політикам. Перевага цього підходу полягає у тому, що саме БД є останнім і найкритичнішим місцем контролю доступу, однак реалізація залежить від можливостей конкретної СУБД і може бути менш універсальною [11].

Окремий сучасний підхід передбачає спробу «вбудувати» політики безпеки безпосередньо у LLM шляхом формування спеціальних промптів або створення шаблонів, які повинні обмежувати генерацію SQL відповідно до визначених правил. Такий метод дозволяє зменшити кількість небезпечних запитів, оскільки модель заздалегідь отримує інструкції не використовувати певні операції або таблиці. Проте його принципова слабкість полягає в тому, що відповідальність за дотримання політик переноситься на стохастичний алгоритм, який не гарантує детермінованої поведінки. LLM може помилятися, іноді ігнорувати інструкції або формувати небезпечні запити через

p-ISSN: 2225-6733; e-ISSN: 2519-271X

«галюцинації», що робить цей варіант найменш надійним у критичних корпоративних умовах.

Усі наведені підходи демонструють різний рівень безпеки, контрольованості та інтеграційної складності, що дозволяє обрати архітектуру відповідно до вимог конкретного підприємства. Узагальнене порівняння основних характеристик (Таблиця 1) показує відмінності між варіантами реалізації: незалежність від LLM, здатність фільтрувати небезпечні операції, можливість переписування SQL, а також відповідність політикам доступу.

Таблиця 1

Порівняння архітектур реалізації Policy Layer

Критерій порівняння	SQL Proxy Firewall	Внутрішній модуль у NL2SQL системі	Плагін у СУБД	Налаштування політик безпеки в LLM
Контроль ролей і доступів	+	+	+	± (не гарантується)
Контроль чутливих колонок	+	+	+	± (не гарантується)
Блокування небезпечних операцій	+	+	+	± (не гарантується)
Можливість переписування SQL	–	+	–	–
Залежність від конкретної СУБД	–	–	+	–
Простота інтеграції	–	+	–	+
Надійність і передбачуваність	+	+	–	–
Продуктивність	±	+	+	+
Захист незалежний від якості LLM	+	+	+	–

У результаті порівняння архітектур реалізації Policy Layer можна зробити висновок, що технічно найбільш гнучким та найбільш збалансованим підходом вважається внутрішній Policy Layer, оскільки він поєднує достатню гнучкість інтеграції з високим рівнем контрольованості. Найвищий рівень гарантій безпеки забезпечує інтеграція Policy Layer на рівні СУБД, адже саме база даних є останньою точкою прийняття рішення щодо виконання запитів. Зовнішній SQL Proxy Firewall пропонує додаткову ізоляцію й підвищений захист, але вимагає складнішої інфраструктури, що може обмежувати його практичну застосовність у масштабованих середовищах. Натомість підхід, заснований на вбудуванні політик у LLM, може використовуватися як допоміжний інструмент для зменшення кількості небезпечних SQL-запитів, проте не може розглядатися як самостійний і надійний механізм, оскільки залежить від стохастичної природи моделей та їхньої потенційної непередбачуваності.

Висновки, отримані за результатами порівняння, свідчать, що ефективне впровадження політик доступу в NL2SQL-системах потребує незалежного, детермінованого компонента контролю, який не покладається на ймовірнісну поведінку мовної моделі. Саме тому найбільш перспективним і практично придатним є підхід із реалізацією внутрішнього Policy Layer або спеціалізованого модуля на рівні СУБД. Такі рішення здатні гарантувати виконання корпоративних політик незалежно від того, як саме сформувала SQL-вираз сама LLM.

У даній роботі розглядається впровадження внутрішнього Policy Layer, тобто проміжного програмного компонента, який інтегрується безпосередньо в застосунок, що працює з NL2SQL-моделлю. У запропонованому рішенні Policy Layer функціонує як окремий сервіс, написаний на Java 17, що отримує згенерований SQL-запит, перетворює його у внутрішню структуру, перевіряє на відповідність корпоративним політикам, оцінює рівень ризику та за необхідності переписує або блокує запит. Після цього застосунок отримує безпечний SQL і лише тоді передає його до бази даних. Таким чином, цей компонент виступає самостійним захисним шаром, незалежним від LLM, і забезпечує детермінований, формальний контроль над SQL-операціями.

Центральною точкою входу внутрішнього Policy Layer є клас PolicyLayerFacade, який реалізує однойменний шаблон проєктування Facade. Використання зазначеного патерну забезпечує уніфікований інтерфейс до складної підсистеми, приховуючи від зовнішніх компонентів, зокрема, NL2SQL-служби або бізнес-логіки застосунку, внутрішню роботу Policy Layer [12]. Завдяки цьому клієнтські модулі взаємодіють лише з одним методом даного класу, не маючи потреби керувати послідовністю викликів таких компонентів, як модуль AST-парсингу, валідатор доступу, детектор чутливих полів, механізм переписування запитів або журнал дій.

У метод process() класу PolicyLayerFacade передається згенерований LLM SQL-запит та інформація про користувача, який ініціював виконання даного запиту. Інформація про користувача включає в себе його

p-ISSN: 2225-6733; e-ISSN: 2519-271X

ідентифікатор та посаду, яку він займає в поточний момент часу. Далі виконується послідовний виклик усіх необхідних підкомпонентів та повертає безпечний SQL або сигналізує про блокування запиту.

Клас SqlAstParser відповідає за перетворення SQL-запиту у внутрішню структуру SqlAstNode. Для парсингу отриманого SQL-запиту використовується бібліотека JSQLParser. JSqlParser аналізує SQL-запит та перетворює його в ієрархію Java-класів [13]. Згенерованою ієрархією можна переміщуватися за допомогою шаблону відвідувача (Visitor). JSqlParser не обмежується однією базою даних, але забезпечує підтримку багатьох СУБД, таких як Oracle, SqlServer, MySQL, PostgreSQL тощо. У результаті роботи парсера формується абстрактне синтаксичне дерево (AST), яке являє собою внутрішню модель SQL-запиту, яка відображає його логічну структуру. AST містить інформацію про тип операції (SELECT, UPDATE, DELETE тощо), перелік таблиць та колонок, наявність операцій JOIN, структуру умов WHERE, використані агрегатні функції та інші елементи запиту. На відміну від текстового подання, AST забезпечує можливість аналізувати запит на рівні логічних конструкцій і семантичних зв'язків, що є критично важливим для даної задачі. Саме на основі AST формується спрощена, але контрольована внутрішня модель SqlAstNode, яка містить лише ті елементи, що необхідні Policy Layer: список таблиць і колонок, їх чутливість, структуру умов, наявність SELECT *, оцінку можливого обсягу вибірки та інші характеристики, що використовуються в механізмі оцінки ризику та переписування запитів (рис. 2).

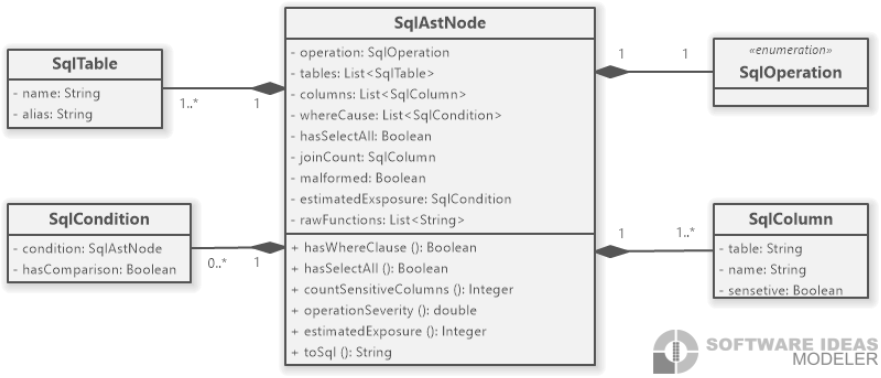

Рис. 2 – UML діаграма класу SqlAstParser та пов'язаних із ним класів

На основі отриманого AST виконується валідація SQL-запиту відповідно до політик безпеки компанії. Для цього розроблено клас AccessPolicyValidator із методом validate, який реалізує логіку зіставлення структури запиту з політиками доступу. У роботі передбачається, що політики доступу зберігаються у вигляді YAML-файлів, що містять перелік ролей, дозволених таблиць і колонок, а також рівні чутливості. YAML-формат вигідний тим, що є зрозумілим для адміністраторів і легко розширюваним [14]. Зокрема, можна описати політики доступу відповідно до посад: junior developer, middle, senior, QA, молодший бухгалтер тощо. Для збереження політик використовується клас PolicyRepository. Валідація будується на перевірці того, чи має користувач право читати відповідні таблиці/колонки й виконувати потрібні операції. Метод validate отримує на вхід UserContext та структуру SQL-дерева. Якщо користувач не має права читати значення колонки salary, валідатор фіксує порушення й передає його далі до модуля оцінки ризику.

Після перевірки доступів викликається метод detectSensitiveFields() класу AccessPolicyValidator, який визначає наявність у запиті колонок із підвищеною чутливістю. Логіка роботи цього методу базується на аналізі абстрактного синтаксичного дерева (SqlAstNode) та зіставленні колонок, присутніх у частині із SELECT або у фільтрах запиту, із внутрішнім реєстром чутливих полів. Для цього використовується окремий клас-репозиторій SensitiveFieldRegistry, який містить перелік таблиць і колонок, позначених як чутливі відповідно до політик безпеки підприємства. Якщо під час аналізу виявлено, що SQL-запит звертається до таких полів, цей факт фіксується у структурі AST, а запит позначається як потенційно ризиковий ще до етапу обчислення інтегральної оцінки ризику. Це дозволяє Policy Layer реагувати на такі ситуації на ранніх стадіях, або посилюючи вагові коефіцієнти під час обчислення ризику, або ініціюючи переписування запиту для виключення чутливих атрибутів, або ж здійснюючи негайне блокування запиту у випадку порушення критичних політик доступу.

Інтегральна оцінка ризику ризику SQL-запиту, яка розраховується за формулою (1), розраховується у методі calcilateRisk() класу RiskCalculator. На вхід передаються результати роботи попередніх модулів, а на виході метод повертає значення від 0 до 1, де 0 інтерпретується як повністю безпечний запит, що не становить ризиків витоку або модифікації даних, а 1 – як максимально небезпечна операція, що може призвести до суттєвих порушень політик доступу, розкриття

p-ISSN: 2225-6733; e-ISSN: 2519-271X

чутливої інформації або руйнівних змін у базі даних. Запити, що перевищують порогове значення, автоматично блокуються або передаються у модуль переписування.

Для зниження кількості заблокованих запитів у сервіс додано модуль Query Rewriter – компонент, що автоматично модифікує SQL-запити, замінюючи у них потенційно небезпечні конструкції, не змінюючи при цьому їхньої логічної семантики. Основна мета цього модуля полягає у тому, щоб замість повного блокування запиту забезпечити його безпечне виконання у межах політик доступу. До типових механізмів переписування належать: заміна SELECT * на перелік дозволених колонок, видалення або корекція чутливих полів у SELECT-списку, додавання умов обмеження (наприклад, LIMIT), а також додавання службових фільтрів у запити без WHERE з метою запобігання повному скануванню всієї таблиці. У програмній реалізації це представлено класом SqlRewriter, який працює безпосередньо з абстрактним синтаксичним деревом (AST), отриманим після розбору SQL-парсером JSQLParser.

Для автоматичного переписування SQL у модулі застосовуються вбудовані механізми JSQLParser, які забезпечують повноцінну роботу з абстрактним синтаксичним деревом [15]. Зокрема, для аналізу та модифікації SELECT-запитів використовується метод Select.getSelectBody(), що дозволяє отримати основну структуру запиту, тоді як обробка списку вибраних колонок здійснюється через PlainSelect.getSelectItems() та PlainSelect.setSelectItems(), що дає змогу замінювати конструкцію SELECT * на конкретні дозволені поля. Робота з умовами фільтрації реалізується через PlainSelect.getWhere() та PlainSelect.setWhere(), що забезпечує можливість додавати або змінювати умову WHERE. Аналогічно, доступ до операцій JOIN виконується через PlainSelect.getJoins() та PlainSelect.setJoins(). Для контролю обсягу повернених даних може використовуватися об’єкт Limit разом із методом setLimit(), що дозволяє встановлювати обмеження на кількість рядків. Поглиблені трансформації AST виконуються завдяки механізмам шаблону відвідувача, зокрема ExpressionVisitor та SelectVisitor, які дозволяють знаходити й змінювати будь-які вузли дерева на довільному рівні вкладеності. За потреби прямого редагування структур таблиць або колонок використовуються методи Column.setColumnName() та Table.setName(), тоді як для вибіркової модифікації елементів SELECT-списку застосовується SelectItemVisitor. Завершальне формування оновленого SQL-коду забезпечує ExpressionDeParser, який будує коректний синтаксично узгоджений SQL-рядок на основі модифікованого AST без ризику порушення синтаксичної цілісності запиту.

Останнім компонентом є Audit & Logging, де кожна операція Policy Layer фіксується для подальшого моніторингу. Для цього існує клас AuditLogger, який веде історію виконаних запитів, заблокованих операцій та значень ризиків. Журнал аудиту може бути використаний для подальшої аналітики або інтеграції з SIEM-системами.

Фінальна діаграма класів наведена нижче (рис. 3).

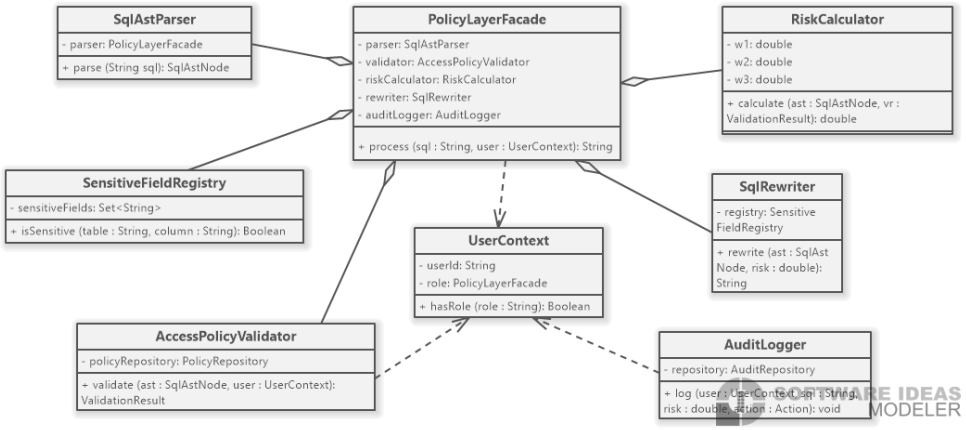

Рис. 3 – Фінальна діаграма класів для реалізації Policy Layer

Після побудови UML-діаграми та опису ключових компонентів запропонованої підсистеми, можна перейти до інтеграції отриманого сервісу, який являє собою Policy Layer, у типовий цикл обробки NL у SQL-запит. Базова схема роботи NL2SQL-рішень передбачає безпосередню передачу згенерованого LLM SQL-виразу до СУБД, після чого результат повертається користувачу (рис. 1). У запропонованій архітектурі ця схема розширюється шляхом додавання окремого модуля – Policy Layer, який розміщується між застосунком і компонентом виконання SQL-запитів (рис. 4).

p-ISSN: 2225-6733; e-ISSN: 2519-271X

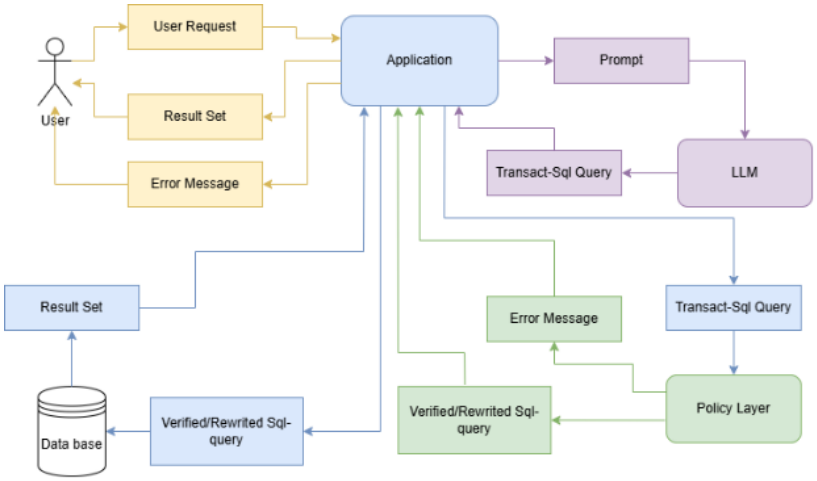

Рис. 4 – Архітектура системи NL2SQL із інтегрованим Policy Layer

У модифікованій архітектурі SQL-запит, сформований мовною моделлю, не надходить відразу до бази даних. Замість цього, застосунок перенаправляє його у Policy Layer, який згідно з визначеною UML-моделлю активує послідовність внутрішніх модулів: синтаксичний аналізатор, валідатор політик доступу, механізм оцінки ризику та, за потреби, модуль переписування. Кожний із цих компонентів виконує окрему частину загальної процедури контролю, що дозволяє формально встановити допустимість запиту для конкретного користувача. Варто відмітити, що інтеграція Policy Layer призводить до розгалуження виконання SQL-запиту. Якщо запит повністю відповідає політикам доступу і має низьке значення інтегрального ризику, він передається у СУБД без змін – фактично повторюючи поведінку первинної архітектури. У випадку помірного ризику Policy Layer активує Query Rewriter, який модифікує SQL таким чином, щоб усунути небезпечні або надлишкові елементи, але зберегти загальну семантику запиту. Таким чином, користувач отримує коректний і безпечний результат навіть у випадку недосконалого запиту, згенерованого LLM. Якщо ж оцінка ризику є високою або запит явно порушує політики, Policy Layer повертає у застосунок повідомлення про заборону виконання.

У результаті інтеграції Policy Layer вихідна архітектура NL2SQL-системи перетворюється з пасивної моделі «LLM → SQL → БД» на систему із розмежуванням допустимих запитів, запитів, що потребують автоматичного переписування, та запитів, які підлягають блокуванню. Це дозволяє поєднати гнучкість роботи з мовними моделями із формальними вимогами безпеки, підвищуючи надійність інтеграції LLM у корпоративні інформаційні системи.

Висновки

У результаті проведеного дослідження було проаналізовано сучасні наукові та прикладні роботи, присвячені проблемам автоматизованого перетворення природної мови у SQL-запити. Аналіз показав, що більшість існуючих рішень орієнтуються переважно на підвищення точності генерації SQL-запитів, оптимізацію in-context навчання, покращення семантичної відповідності або адаптацію промптів та навчальних наборів для різних рівнів складності запитів. Водночас майже всі роботи не враховують етап контролю доступу: роль користувача, корпоративні політики безпеки, обмеження на чутливі атрибути та запобігання можливим ризикам витоку даних. Такий недолік є критичним для практичного використання NL2SQL-рішень у корпоративних середовищах.

Для усунення виявленої проблеми було запропоновано нову архітектуру NL2SQL-системи з інтегрованим Policy Layer – окремим проміжним сервісом, що виконує синтаксичний аналіз згенерованого SQL-запиту, перевірку політик доступу, виявлення чутливих полів, оцінку ризику та можливе автоматичне переписування запиту. Запити, що не відповідають політикам безпеки або мають високий рівень ризику, блокуються ще до етапу виконання в СУБД. Таким чином, запропоноване рішення забезпечує керований і захищений механізм виконання SQL-запитів, генерованих мовними моделями, та усуває ключовий недолік наявних підходів.

Отримані результати підтверджують актуальність розробки Policy Layer як обов'язкового компонента сучасних NL2SQL-систем і відкривають перспективні напрями подальших досліджень: удосконалення механізмів переписування SQL-запитів, адаптивне налаштування вагових коефіцієнтів ризику, а також інтеграція багаторівневих політик доступу, що враховують контекст користувача й історію виконання запитів.

Перелік використаних джерел

[1] Review of question answering technology based on Text to SQL / Z. Ning et al. *2021 IEEE International Conference on Power Electronics, Computer Applications (ICPECA)*, Shenyang, China, 22–24 January

2021. Pp. 143–146. DOI: https://doi.org/10.1109/ICPECA51329.2021.9362554

[2] Natural Language to SQL Queries: A Review / M. S. Baig et al. *International Journal of Innovations in Science & Technology*. 2022. Vol. 4, no. 1. Pp. 147–162.

[3] HITSQL: Human-In-The-Loop Techniques for Enhancing Text-to-SQL Query Generation with Large Language Models / D. Al-Turki et al. *2025 International Joint Conference on Neural Networks (IJCNN)*, Rome, Italy, 30 June 2025 - 05 July 2025. Pp. 1-8. DOI: https://doi.org/10.1109/IJCNN64981.2025.11227910.

[4] AID-SQL: Adaptive In-Context Learning of Text-to-SQL with Difficulty-Aware Instruction and Retrieval-Augmented Generation / X. Li et al. *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, Hong Kong, Hong Kong, 19–23 May 2025. Pp. 3945–3957. DOI: https://doi.org/10.1109/ICDE65448.2025.00294.

[5] Chen Y.-X., Huang M.-J., Chen H.-K. A Closed-Domain Natural Language Database Query System by LLMs. *2025 Seventh International Symposium on Computer, Consumer and Control (IS3C)*, Taichung, Taiwan, 27–30 June 2025. Pp. 1–3. DOI: https://doi.org/10.1109/IS3C65361.2025.11131102.

[6] How to Safely Use LLMs for Text-to-SQL with Stored Procedures. Medium. URL: https://erincon01.medium.com/how-to-safely-use-llms-for-text-to-sql-with-stored-procedures-ba7540067f5f (дата звернення: 29.11.2025).

[7] Cognitive mirage: A Review of Hallucinations in Large Language Models / H. Ye et al. arXiv preprint. arXiv:2309.06794, 2023. DOI: https://doi.org/10.48550/arXiv.2309.06794.

[8] NL2SQL System Design Guide 2025. Medium. URL: https://medium.com/@adityamahakali/nl2sql-system-design-guide-2025-c517a00ae34d (дата звернення: 29.11.2025).

[9] Naidu Gundapaneni M. Security Practices in Database Access: A Technical Review. *Journal of Computer Science and Technology Studies*. 2025. Vol. 7, no. 8. Pp. 771-778. DOI: https://doi.org/10.32996/jcsts.2025.7.8.90.

[10] Khamdamov R. K., Kerimov K. F. Database protection based on web application firewall. *Journal of Automation and Information Sciences*. 2021. Vol. 1. Pp. 84–90. DOI: https://doi.org/10.34229/0572-2691-2021-1-7.

[11] Botros S., Tinley J. High Performance MySQL. O'Reilly Media, Incorporated, 2021. 365 p.

[12] Sierra K. Head First Design Patterns. O'Reilly Media, Incorporated, 2004. 692 p.

[13] Java SQL Parser Library. URL: https://jsqlparser.github.io/JSqlParser/ (дата звернення: 29.11.2025).

[14] What is YAML? Understanding the Basics, Syntax, and Use Cases. DataCamp. URL: https://www.datacamp.com/blog/what-is-yaml (дата звернення: 29.11.2025).

[15] Parse database query with JSQL Parser. Medium. URL: https://medium.com/@knoldus/parse-database-query-with-jsql-parser-16c708270433 (дата звернення: 29.11.2025).

INTEGRATION OF A POLICY LAYER INTO NL2SQL SYSTEMS TO PREVENT UNAUTHORIZED ACCESS TO SENSITIVE DATA

Morozova A.I. *PhD (Engineering), associate professor, Kharkiv National University of Radio Electronics, Kharkiv, ORCID: https://orcid.org/0000-0002-7082-4115, e-mail: anna.morozova@nure.ua;*

Novoselova A.S. *Master's degree candidate, Kharkiv National University of Radio Electronics, Kharkiv, ORCID: https://orcid.org/0009-0001-2854-5569, e-mail: anastasiia.novoselova@nure.ua;*

Petrova R.V. *PhD (Engineering), associate professor, Kharkiv National University of Radio Electronics, Kharkiv, ORCID: https://orcid.org/0000-0001-5886-8943, e-mail: roksana.petrova@nure.ua*

The article addresses the critical problem of the lack of robust access-control mechanisms in modern Natural Language to SQL (NL2SQL) systems. While these systems, powered by Large Language Models (LLMs), have demonstrated significant progress in the accuracy and efficiency of transforming natural language queries into syntactically and semantically correct SQL commands, they predominantly focus on linguistic quality and often fail to account for the security context. This critical oversight includes neglecting risks associated with user roles, defined corporate access policies, and the inherent sensitivity of the data attributes being queried. Consequently, a syntactically correct query, such as one requesting employee salaries, can inadvertently expose confidential corporate information if the user lacks the necessary permissions, posing a significant threat to data security. Existing research efforts primarily concentrate on enhancing LLM generation quality, optimizing training datasets, or attempting to «train» the stochastic LLM to generate «safe» queries, an approach that is inherently unreliable due to the non-deterministic nature and potential for «hallucinations» in LLMs. To mitigate these inherent security risks, an enhanced NL2SQL system architecture with an integrated Policy Layer is

*proposed. This Policy Layer is designed as an independent, intermediate service positioned between the application and the database management system, ensuring a deterministic control mechanism. The core functionality of this layer involves a sequential, formal process utilizing the generated SQL query as input: 1) Syntactic Parsing using tools like JSQLParser to convert the SQL into an Abstract Syntax Tree (AST); 2) Access Policy Validation by comparing the AST against pre-defined corporate policies (e.g., stored in YAML files) that map user roles to permitted tables and columns; 3) Sensitive Field Detection to flag the presence of high-sensitivity columns (e.g., salary, home_address); 4) Risk Evaluation; 5) Query Rewriting or Blocking. Based on the calculated risk score and policy validation results, the Policy Layer can either automatically modify the SQL query (e.g., replacing SELECT * with allowed columns, adding LIMIT clauses) via the Query Rewriter module to bring it into compliance, or it can outright block execution if the risk is high or critical policies are violated. This architecture transforms the NL2SQL workflow into a controlled system that provides formal, guaranteed policy adherence, reconciling the flexibility of LLM usage with strict corporate security requirements.*
***Keywords**: NL2SQL; LLM; large language model; SQL; Policy Layer; risk assessment.*

References

[1] Z. Ning, D. Zhang, L. Zhang, H. Yu, and F. Wan, "Review of question answering technology based on Text to SQL," in *Proc. of the 2021 IEEE Int. Conf. on Power Electronics, Computer Applications (ICPECA)*, Shenyang, China, January 22–24, 2021, pp. 143–146. **doi: 10.1109/ICPECA51329.2021.9362554**.

[2] M. S. Baig, A. Imran, A. U. Yasin, A. H. Butt, and M. I. Khan, "Natural Language to SQL Queries: A Review," *International Journal of Innovations in Science & Technology*, vol. 4, no. 1, pp. 147–162, 2022.

[3] D. Al-Turki, S. Basurra, M. Gaber, B. Attasi, and M. M. Abdelsamea, "HITSQL: Human-In-The-Loop Techniques for Enhancing Text-to-SQL Query Generation with Large Language Models," in *Proc. of the 2025 Int. Joint Conf. on Neural Networks (IJCNN)*, Rome, Italy, 30 June 2025 - 05 July 2025, pp. 1-8. **doi: 10.1109/IJCNN64981.2025.11227910**.

[4] X. Li, Q. Cai, Y. Shu, C. Guo, and B. Yang, "AID-SQL: Adaptive In-Context Learning of Text-to-SQL with Difficulty-Aware Instruction and Retrieval-Augmented Generation," in *Proc. of the 2025 IEEE 41st Int. Conf. on Data Engineering (ICDE)*, Hong Kong, Hong Kong, May 19–23, 2025, pp. 3945–3957. **doi: 10.1109/ICDE65448.2025.00294**.

[5] Y.-X. Chen, M.-J. Huang, and H.-K. Chen, "A Closed-Domain Natural Language Database Query System by LLMs," in *Proc. of the 2025 Seventh Int. Symposium on Computer, Consumer and Control (IS3C)*, Taichung, Taiwan, June 27–30, 2025, pp. 1–3. **doi: 10.1109/IS3C65361.2025.11131102**.

[6] "How to Safely Use LLMs for Text-to-SQL with Stored Procedures," *Medium*. [Online]. Available: https://erincon01.medium.com/how-to-safely-use-llms-for-text-to-sql-with-stored-procedures-ba7540067f5f. Accessed on: Nov. 29, 2025.

[7] H. Ye, T. Liu, A. Zhang, W. Hua, and W. Jia, "Cognitive mirage: A Review of Hallucinations in Large Language Models," *arXiv:2309.06794*, 2023. **doi: 10.48550/arXiv.2309.06794**.

[8] "NL2SQL System Design Guide 2025," *Medium*. [Online]. Available: https://medium.com/@adityamahakali/nl2sql-system-design-guide-2025-c517a00ae34d. Accessed on: Nov. 29, 2025.

[9] M. Naidu Gundapaneni, "Security Practices in Database Access: A Technical Review," *Journal of Computer Science and Technology Studies*, vol. 7, no. 8, pp. 771-778, 2025. **doi: 10.32996/jcsts.2025.7.8.90**.

[10] R. K. Khamdamov, and K. F. Kerimov, "Database protection based on web application firewall," *J Journal of Automation and Information Sciences*, vol. 1, pp. 84–90, 2021. **doi: https://doi.org/10.34229/0572-2691-2021-1-7**.

[11] S. Botros, and J. Tinley, *High Performance MySQL*. O'Reilly Media, Incorporated, 2021.

[12] K. Sierra, *Head First Design Patterns*. O'Reilly Media, Incorporated, 2004.

[13] "Java SQL Parser Library," *GitHub*. [Online]. Available: https://jsqlparser.github.io/JSqlParser/. Accessed on: Nov. 29, 2025.

[14] "What is YAML? Understanding the Basics, Syntax, and Use Cases," *DataCamp*. [Online]. Available: https://www.datacamp.com/blog/what-is-yaml. Accessed on: Nov. 29, 2025.

[15] "Parse database query with JSQL Parser," *Medium*. [Online]. Available: https://medium.com/@knoldus/parse-database-query-with-jsql-parser-16c708270433. Accessed on: Nov. 29, 2025.

Стаття надійшла 19.12.2025
Стаття прийнята 15.01.2026
Стаття опублікована 26.03.2026

Цитуйте цю статтю як: Морозова А. І., Новоселова А. С., Петрова Р. В. Інтеграція Policy Layer у NL2SQL-системи для запобігання доступу до чутливих даних. *Вісник Приазовського державного технічного університету. Серія: Технічні науки*. 2026. Вип. 53, том 1. С. 33–42. DOI: https://doi.org/10.31498/2225-6733.53.1.2026.359767.