МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проектами

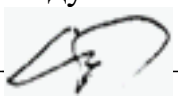
Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри

_  __ проф. Приходько С.Б.

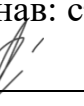
«___» _____ 2026 р.

КВАЛІФІКАЦІЙНАРОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: Розробка програмного забезпечення для відстеження змін в інформаційній системі

Виконав: студент групи 4157ст3

 Макодзьоб А. О.

(підпис, ПІБ)

Керівник роботи:

асистент кафедри ПЗАС, доктор філософії

(посада, науковий ступень вчене звання)

 Трухов А. С.

(підпис, ПІБ)

Миколаїв – 2026 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проектами
Кафедра програмного забезпечення автоматизованих систем
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ доц. Макарова Л.М.
(підпис)
«_07_» ___04___ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студенту Макодзьоб Анна Олександрівна
(Прізвище, ім'я, по батькові)

1. Тема роботи: Розробка програмного забезпечення для відстеження змін в інформаційній системі
Керівник роботи асистент кафедри ПЗАС, доктор філософії Трухов А. С
Затверджені наказом ректора №275-уч від 07.04.2026 р.
2. Термін подання роботи: 01.06.2026 р.
3. Вихідні дані по роботі: _____

4. Перелік питань, що належать до розробки (найменування розділів)
Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).

5. Перелік презентаційних матеріалів Титульний слайд; Мета та завдання роботи; Актуальність теми; Аналіз існуючих програмних рішень; Сценарії використання програмного забезпечення; Архітектура програмного забезпечення ChangeTracker; Компонентна структура застосунку; Проєктування бази даних; UML-діаграма класів; Реалізація програмного забезпечення; Демонстрація інтерфейсу: Dashboard, Data Sources, Change Log; Демонстрація інтерфейсу: Automation, Executions, Settings; Висновки; Заключний слайд

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 07.04.2026 р.

К А Л Е Н Д А Р Н И Й П Л А Н

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	30.03.2026	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	06.04.2026	ПП*
3.	Аналіз вимог до ПЗ	13.04.2026	ПП*
4.	Розробка архітектури ПЗ	20.04.2026	
5.	Розробка проекту ПЗ	04.05.2026	
6.	Реалізація та тестування ПЗ	11.05.2026	
7.	Підготовка розділу Результати розробки ПЗ	15.05.2026	
8.	Підготовка розділу з охорони праці	18.05.2026	ПП*
9.	Підготовка розділу ВИСНОВКИ	22.05.2026	
10.	Оформлення списку використаних джерел та додатків	25.05.2026	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	01.06.2026	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	05.06.2026	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	08.06.2026	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	15.06.2026	

* - за результатами переддипломної практики (ПП), яка була з 02.02.2026 до 22.03.2026 р.

Студент _____ Макодзьоб А. О.
(підпис) (ПІБ)

Керівник роботи _____ Трухов А. С
(підпис) (ПІБ)

АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці програмного забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture. Дана задача належить до практичних задач інженерії програмного забезпечення, характеризується комплексністю та невизначеністю умов, оскільки потребує організації взаємодії з базою даних, автоматичного отримання змін, збереження журналу подій та виконання дій у відповідь на визначені події.

У роботі проведено аналіз практичної задачі інженерії програмного забезпечення, розглянуто існуючі програмні рішення для аудиту, моніторингу та обробки змін у базах даних, визначено їхні переваги й недоліки. На основі проведеного аналізу обґрунтовано доцільність розробки власного програмного забезпечення, сформульовано постановку задачі та визначено основні вимоги до програмного продукту.

Розроблене програмне забезпечення ChangeTracker реалізовано як десктопний застосунок для операційної системи Windows із використанням мови програмування C#, платформи .NET 8 та технології WPF.

Результатом кваліфікаційної роботи є програмне забезпечення, яке забезпечує автоматизований моніторинг змін у базі даних, перегляд журналу подій, аналіз деталей змін та автоматичне реагування на визначені події шляхом виконання HTTP-запитів, запису інформації до файлу або запуску зовнішніх програм.

Кваліфікаційна робота викладена на 154 сторінках друкованого тексту, містить 13 рисунків, 7 таблиць, список використаних джерел із 14 найменувань та 5 додатків.

ABSTRACT

The qualification work is devoted to the development of software for tracking changes in an information system based on Change Data Capture technology. This task belongs to practical software engineering tasks and is characterized by complexity and uncertainty of conditions, as it requires interaction with a database, automatic change capturing, event logging, and execution of actions in response to specific events.

The work analyzes the practical software engineering task, examines existing software solutions for database auditing, monitoring, and change processing, and identifies their advantages and disadvantages. Based on the analysis, the feasibility of developing custom software is substantiated, the problem statement is formulated, and the main requirements for the software product are defined.

The developed ChangeTracker software is implemented as a Windows desktop application using the C# programming language, the .NET 8 platform, and WPF technology.

Within the scope of the work, the software architecture was developed, the database structure was designed, and modules for managing data sources, the change log, automation rules, execution history, and system settings were implemented. A background service was also developed to poll CDC tables, store captured events, match them with automation rules, and execute user-defined actions.

The result of the qualification work is software that provides automated monitoring of database changes, viewing of the event log, analysis of change details, and automatic response to specified events by executing HTTP requests, writing information to a file, or launching external applications.

The qualification work consists of 154 pages of printed text, contains 13 figures, 7 tables, a list of 14 references, and 5 appendices.

ЗМІСТ

ВСТУП

У сучасних інформаційних системах бази даних є одним із ключових компонентів, оскільки саме в них зберігається основна інформація про користувачів, документи, довідники, транзакції та інші об’єкти предметної області. У процесі функціонування таких систем постійно виконуються операції додавання, редагування та видалення даних. Ці зміни можуть впливати на роботу інших програмних модулів, зовнішніх сервісів, кешованих даних або аналітичних підсистем. Тому виникає потреба не лише у збереженні інформації про зміни, а й в організації автоматизованого моніторингу та реагування на події, що виникають у базі даних.

Відстеження змін в інформаційній системі є практичною задачею інженерії програмного забезпечення, що характеризується комплексністю та невизначеністю умов. Її комплексність полягає у необхідності поєднання декількох аспектів: взаємодії з базою даних, отримання змін за допомогою спеціалізованих механізмів СУБД, збереження журналу подій, відображення інформації користувачу, налаштування правил обробки та виконання автоматичних дій. Невизначеність умов пов’язана з тим, що структура цільових баз даних може відрізнятися, кількість змін може бути різною, правила реагування залежать від потреб конкретної інформаційної системи, а зовнішні сервіси, з якими взаємодіє програмне забезпечення, можуть мати різні вимоги до форматів запитів і способів обробки даних.

Одним із підходів до вирішення задачі відстеження змін є використання технології Change Data Capture (CDC), яка дозволяє фіксувати операції зміни даних у базі даних без необхідності внесення змін до прикладної логіки інформаційної системи. Використання CDC дає змогу отримувати інформацію про операції вставки, оновлення та видалення записів, що створює основу для побудови систем аудиту, моніторингу та автоматизованої обробки подій [1].

Існують програмні рішення, які частково розв’язують задачу аудиту та

моніторингу змін у базах даних. До таких рішень можна віднести ApexSQL Audit [2], Debezium [3] та Oracle GoldenGate [4]. ApexSQL Audit орієнтований на аудит і контроль активності в Microsoft SQL Server, однак є комерційним продуктом і переважно призначений для задач аналізу подій, а не для гнучкого виконання користувацьких дій після зміни даних. Debezium забезпечує потокове отримання змін із різних СУБД, однак для його повноцінного використання зазвичай потрібна складніша інфраструктура, зокрема Apache Kafka [5] та Kafka Connect [6]. Oracle GoldenGate є потужною корпоративною платформою для реплікації та інтеграції даних, але характеризується високою складністю впровадження, значною вартістю та орієнтацією на великі корпоративні середовища.

Аналіз існуючих аналогів показує, що вони або є надмірно складними для локального використання, або мають високу вартість, або не забезпечують простого механізму налаштування правил автоматичного реагування на події бази даних у межах одного десктопного застосунку. Саме тому доцільною є розробка власного програмного забезпечення, яке поєднує отримання змін на основі CDC, зручний графічний інтерфейс, журнал подій, систему правил автоматизації та можливість виконання дій у відповідь на визначені зміни.

Метою кваліфікаційної роботи є розробка програмного забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– виконати аналіз практичної задачі інженерії програмного забезпечення щодо відстеження змін в інформаційній системі;

– проаналізувати існуючі програмні рішення та обґрунтувати доцільність розробки власного програмного забезпечення;

– сформулювати постановку задачі на розробку програмного забезпечення для відстеження змін в інформаційній системі;

– провести аналіз вимог до програмного забезпечення для відстеження змін в інформаційній системі;

– розробити архітектурні рішення програмного забезпечення;

– виконати проєктування бази даних та користувацького інтерфейсу;

– реалізувати механізм отримання змін за допомогою технології Change Data Capture;

– реалізувати механізм автоматичного виконання дій на основі налаштованих правил обробки подій;

– провести тестування та випробування розробленого програмного забезпечення;

– підготувати необхідну програмну та технічну документацію.

Об’єктом кваліфікаційної роботи є процес розробки програмного забезпечення для відстеження змін в інформаційній системі.

1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВІДСТЕЖЕННЯ ЗМІН В ІНФОРМАЦІЙНІЙ СИСТЕМІ

1.1 Аналіз практичної задачі інженерії програмного забезпечення

У сучасних інформаційних системах база даних є одним із ключових компонентів, оскільки саме в ній зберігається основна інформація про користувачів, документи, довідники, транзакції та інші об’єкти предметної області. Під час функціонування системи постійно виконуються операції додавання, зміни та видалення даних. Кількість таких операцій може досягати тисяч або навіть десятків тисяч на добу залежно від масштабу інформаційної системи. Тому контроль змін даних є важливою задачею, що безпосередньо впливає на надійність, безпеку та керованість програмного забезпечення.

Одним із поширених підходів до контролю змін є ведення журналів аудиту. Аудит дозволяє визначити факт виконання операції, час її здійснення та зміни, які були внесені до даних. Наявність аудиту є важливою складовою супроводу програмного забезпечення, оскільки дозволяє аналізувати причини виникнення помилок, контролювати коректність роботи користувачів та забезпечувати відновлення послідовності подій під час розслідування інцидентів [7].

Разом із задачею аудиту дедалі більшої актуальності набуває задача моніторингу змін даних у режимі, наближеному до реального часу. У багатьох випадках недостатньо лише зберігати інформацію про зміни. Виникає необхідність автоматично виконувати певні дії після виникнення конкретної події. Наприклад, після додавання нового запису до таблиці модулів може виникнути потреба очистити кеш вебзастосунку, після зміни довідника – оновити дані у зовнішньому сервісі, а після створення нового документа – надіслати повідомлення до іншої інформаційної системи.

Традиційний підхід до реалізації подібної функціональності передбачає внесення додаткового програмного коду безпосередньо до прикладних систем або використання тригерів бази даних. Однак такий підхід має ряд недоліків. По-перше, він збільшує складність супроводу програмного забезпечення через необхідність внесення змін до вже існуючих компонентів системи. По-друге, логіка обробки подій розподіляється між різними модулями, що ускладнює її централізоване адміністрування. По-третє, розширення функціональності потребує додаткових витрат часу на модифікацію та тестування існуючого програмного коду.

Для вирішення зазначених проблем можуть використовуватися механізми відстеження змін на рівні системи управління базами даних. Однією з таких технологій є CDC, яка дозволяє автоматично фіксувати операції вставки, оновлення та видалення даних без необхідності втручання у бізнес-логіку інформаційної системи. Використання CDC забезпечує отримання достовірної інформації про всі зміни, що відбуваються в базі даних, а також створює основу для побудови централізованих систем моніторингу та аудиту.

Незважаючи на наявність механізмів відстеження змін на рівні СУБД, вони зазвичай не надають зручних засобів візуального аналізу отриманих даних та автоматичного виконання дій на основі визначених користувачем правил. Для перегляду інформації про зміни часто необхідно працювати безпосередньо із системними таблицями бази даних або використовувати спеціалізовані інструменти адміністрування, що потребує відповідної кваліфікації та не завжди є зручним для кінцевих користувачів.

У межах даної практичної задачі інженерії програмного забезпечення необхідно розробити програмне забезпечення, яке забезпечуватиме централізований моніторинг змін у базі даних, ведення журналу подій, пошук та фільтрацію інформації про зміни, а також автоматичне виконання налаштованих користувачем дій при виникненні визначених подій. Зокрема, система повинна надавати можливість налаштовувати правила реагування на операції вставки, оновлення або видалення записів у конкретних таблицях та виконувати

відповідні дії, такі як виклик зовнішнього API, запис інформації до файлу журналу або запуск зовнішнього застосунку.

Таким чином, розробка програмного забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture дозволить автоматизувати процес моніторингу даних, підвищити ефективність супроводу інформаційних систем, забезпечити централізований контроль змін та створити гнучкий механізм автоматичного реагування на події, що виникають у процесі функціонування бази даних.

1.2 Аналіз аналогічного існуючого ПЗ та обґрунтування розробки власного ПЗ

Одним із важливих етапів розробки програмного забезпечення є аналіз існуючих програмних рішень предметної області. Такий аналіз дозволяє дослідити сучасні підходи до розв'язання поставленої задачі, визначити їх сильні та слабкі сторони, а також обґрунтувати необхідність створення нового програмного продукту. У межах даної роботи розглядаються програмні рішення, що забезпечують аудит, моніторинг та обробку змін у базах даних. Основна увага приділяється можливостям відстеження змін, налаштуванню автоматизованих сценаріїв реагування, складності впровадження та зручності використання. В ході проведення аналізу було обрано програмні продукти ApexSQL Audit [2], Debezium [3] та Oracle GoldenGate [4].

Першим досліджуваним програмним рішенням є ApexSQL Audit. Це спеціалізоване програмне забезпечення для аудиту та моніторингу активності в базах даних Microsoft SQL Server. Основним призначенням системи є відстеження змін даних, контроль доступу користувачів до інформації, моніторинг операцій із базою даних та формування звітів щодо виконаних дій.

До переваг ApexSQL Audit належать широкі можливості аудиту подій, зручні засоби формування звітів, підтримка моніторингу змін у режимі реального часу та інтеграція з Microsoft SQL Server. Також система надає засоби пошуку та аналізу подій, що дозволяє швидко знаходити необхідну інформацію

про зміни в базі даних.

До недоліків ApexSQL Audit можна віднести його комерційний характер та високу вартість ліцензування. Крім того, програмне забезпечення орієнтоване переважно на аудит та аналіз подій, а не на автоматичне виконання користувацьких дій після виникнення змін. Для реалізації складних сценаріїв автоматизації необхідно використовувати додаткові інструменти або програмні засоби. Також функціональність системи є надлишковою для невеликих інформаційних систем та навчальних проєктів.

Другим проаналізованим програмним рішенням є Debezium. Це програмна платформа з відкритим вихідним кодом, призначена для відстеження змін у базах даних та передачі інформації про них до інших програмних систем. Debezium підтримує різні системи управління базами даних та широко використовується в архітектурах обробки подій і мікросервісних рішеннях.

До переваг Debezium належать підтримка різних СУБД, висока продуктивність, можливість отримання змін практично в режимі реального часу та інтеграція з сучасними платформами потокової обробки даних. Завдяки відкритому вихідному коду система активно розвивається та має широку спільноту користувачів.

Недоліками Debezium є складність налаштування та експлуатації. Для повноцінної роботи зазвичай необхідно використовувати додаткові компоненти, зокрема Apache Kafka та Kafka Connect. Відсутність готового графічного інтерфейсу для перегляду журналу змін і керування правилами реагування ускладнює використання системи кінцевими користувачами. Для невеликих організацій або навчальних проєктів така архітектура може бути надмірно складною та потребувати значних ресурсів для підтримки.

Третім досліджуваним програмним рішенням є Oracle GoldenGate. Це корпоративна платформа для реплікації даних, інтеграції інформаційних систем та відстеження змін у базах даних. Система забезпечує передачу інформації між різними серверами та програмними платформами з мінімальною затримкою.

До переваг Oracle GoldenGate належать висока продуктивність, надійність

роботи, підтримка великої кількості систем управління базами даних та можливість використання у масштабних корпоративних середовищах. Платформа забезпечує відстеження змін даних у режимі реального часу та має розвинуті механізми адміністрування.

До недоліків Oracle GoldenGate можна віднести високу вартість ліцензування, складність впровадження та налаштування, а також значні вимоги до кваліфікації персоналу. Більшість функціональних можливостей системи орієнтована на великі корпоративні інфраструктури та задачі реплікації даних, тоді як функціональність локального моніторингу змін і налаштування користувацьких правил реагування не є основним призначенням даного програмного забезпечення.

Проведений аналіз існуючих програмних рішень дозволяє зробити висновок, що всі вони мають такі спільні недоліки відповідно до теми роботи:

– висока складність налаштування та супроводу;

– значна вартість ліцензування окремих програмних продуктів;

– орієнтація переважно на корпоративне використання та адміністрування баз даних;

– відсутність простого механізму налаштування користувацьких правил автоматичного реагування на зміни;

– надлишковість функціональних можливостей для задач локального моніторингу змін в інформаційній системі;

– відсутність єдиного настільного застосунку, який одночасно забезпечує моніторинг змін, аудит подій, перегляд журналу та виконання автоматичних дій на основі визначених користувачем правил.

Наведена вище інформація обґрунтовує доцільність розробки власного програмного забезпечення для відстеження змін в інформаційній системі. Запропоноване рішення дозволить поєднати механізм отримання змін на основі технології Change Data Capture, зручний графічний інтерфейс користувача та систему автоматичного реагування на події без необхідності використання складних корпоративних платформ або додаткових програмних компонентів.

1.3 Постановка задачі на розробку ПЗ

Згідно з аналізом практичної задачі інженерії програмного забезпечення та результатами дослідження існуючих аналогів, для розв’язання поставленої задачі необхідно розробити програмне забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture. Програмне забезпечення повинно забезпечувати автоматичне отримання інформації про зміни в базі даних, ведення журналу подій, перегляд історії змін та виконання налаштованих користувачем дій при виникненні визначених подій.

Вимоги до складу виконуваних функцій

Програмне забезпечення для відстеження змін в інформаційній системі повинно забезпечити можливість виконання таких функцій:

– можливість додавати, корегувати, вилучати та переглядати інформацію про джерела даних, тобто наявність дій CRUD для даної сутності;

– можливість додавати, корегувати, вилучати та переглядати інформацію про правила автоматичної обробки подій, тобто наявність дій CRUD для даної сутності;

– можливість додавати, корегувати, вилучати та переглядати інформацію про дії, що виконуються при спрацюванні правил, тобто наявність дій CRUD для даної сутності;

– автоматичне отримання інформації про зміни даних із використанням технології Change Data Capture;

– перегляд журналу змін бази даних;

– перегляд детальної інформації про зміни записів;

– пошук та фільтрація подій за таблицею, типом операції та часовим проміжком;

– автоматичне виконання дій при виникненні визначених подій;

– перегляд історії виконання автоматичних дій;

– формування звітів про зміни в інформаційній системі.

Вимоги до організації вхідних та вихідних даних

– Для додавання інформації про джерело даних:

Вхідні дані — назва підключення, адреса сервера, назва бази даних, облікові дані для підключення.

Вихідні дані — новий запис про джерело даних у базі даних застосунку та системне повідомлення про успішне додавання.

– Для корегування інформації про джерело даних:

Вхідні дані — обране джерело даних та нові параметри підключення.

Вихідні дані — оновлений запис про джерело даних та системне повідомлення про успішне корегування.

– Для вилучення інформації про джерело даних:

Вхідні дані — обране джерело даних та підтвердження дії.

Вихідні дані — вилучений запис про джерело даних та системне повідомлення про успішне вилучення.

– Для відображення інформації про джерела даних:

Вхідні дані — відкриття вкладки «Джерела даних».

Вихідні дані — таблиця зі списком налаштованих підключень.

– Для додавання правила автоматичної обробки подій:

Вхідні дані — назва правила, таблиця бази даних, тип події та параметри дії.

Вихідні дані — новий запис про правило в базі даних застосунку та системне повідомлення про успішне додавання.

– Для корегування правила автоматичної обробки подій:

Вхідні дані — обране правило та нові параметри.

Вихідні дані — оновлений запис про правило та системне повідомлення

про успішне корегування.

– Для вилучення правила автоматичної обробки подій:

Вхідні дані — обране правило та підтвердження дії.

Вихідні дані — вилучений запис про правило та системне повідомлення про успішне вилучення.

– Для відображення правил автоматичної обробки подій:

Вхідні дані — відкриття вкладки «Правила».

Вихідні дані — таблиця зі списком правил та їх параметрами.

– Для перегляду журналу змін:

Вхідні дані — відкриття вкладки «Журнал змін».

Вихідні дані — список подій із зазначенням часу виникнення, таблиці та типу операції.

– Для перегляду детальної інформації про зміну:

Вхідні дані — вибір запису в журналі змін.

Вихідні дані — відображення змінених полів, попередніх та нових значень.

– Для виконання автоматичної дії:

Вхідні дані — зафіксована подія зміни даних та відповідне правило обробки.

Вихідні дані — результат виконання дії та запис в історії виконань.

Вимоги до складу та параметрів технічних засобів

Операційна система: Windows 10 або новіша.

Процесор: 4 ядра або більше із частотою не менше 2,5 ГГц.

Оперативна пам’ять: не менше 8 ГБ.

Жорсткий диск: не менше 500 МБ вільного простору для встановлення

програмного забезпечення та збереження службових даних.

Мережеве підключення: доступ до локальної мережі або мережі Інтернет для підключення до баз даних та виконання HTTP-запитів.

Система управління базами даних: Microsoft SQL Server з підтримкою технології Change Data Capture.

Отже, розроблене програмне забезпечення повинно забезпечити автоматизований моніторинг змін у базі даних, надання користувачеві зручного інтерфейсу для перегляду журналу подій, налаштування правил автоматичної обробки змін та виконання визначених дій у відповідь на виникнення подій. Це дозволить спростити супровід інформаційних систем, підвищити оперативність реагування на зміни та автоматизувати виконання типових процедур адміністрування.

Детальні вимоги до програмного забезпечення наведені у Додатку А – Технічне завдання.

2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВІДСТЕЖЕННЯ ЗМІН В ІНФОРМАЦІЙНІЙ СИСТЕМІ

2.1 Аналіз вимог до програмного забезпечення

Аналіз вимог до програмного забезпечення є одним із ключових етапів розробки, оскільки дозволяє визначити основні сценарії роботи системи, її користувачів, функціональні можливості та взаємодію між окремими компонентами. У межах даної кваліфікаційної роботи розробляється програмне забезпечення ChangeTracker, призначене для відстеження змін в інформаційній системі на основі технології Change Data Capture та автоматичного виконання дій у відповідь на визначені події.

Оскільки загальні вимоги до програмного забезпечення вже були сформульовані під час постановки задачі на розробку, у даному підрозділі основну увагу приділено аналізу варіантів використання системи. Це дозволяє уточнити, які дії виконують користувачі програмного забезпечення, які сценарії запускаються автоматично, а також які функції повинні бути реалізовані для забезпечення повного циклу роботи застосунку.

Програмне забезпечення ChangeTracker має три основні групи учасників взаємодії: користувач, адміністратор та фоновий сервіс. Користувач є основним актором системи. Він працює з графічним інтерфейсом застосунку, переглядає інформацію про зміни, налаштовує джерела даних, створює правила автоматичної обробки подій та аналізує результати виконання цих правил. Адміністратор відповідає за налаштування параметрів роботи системи, які впливають на загальну поведінку програмного забезпечення. Фоновий сервіс є автоматизованим внутрішнім актором, який виконує періодичне опитування CDC-таблиць, зберігає події, зіставляє їх із правилами та виконує налаштовані дії без прямої участі користувача.

Для формалізації функціональних вимог до системи було побудовано

діаграму варіантів використання (рис. 2.1). Вона відображає основні сценарії взаємодії користувача, адміністратора та фонового сервісу із системою ChangeTracker.

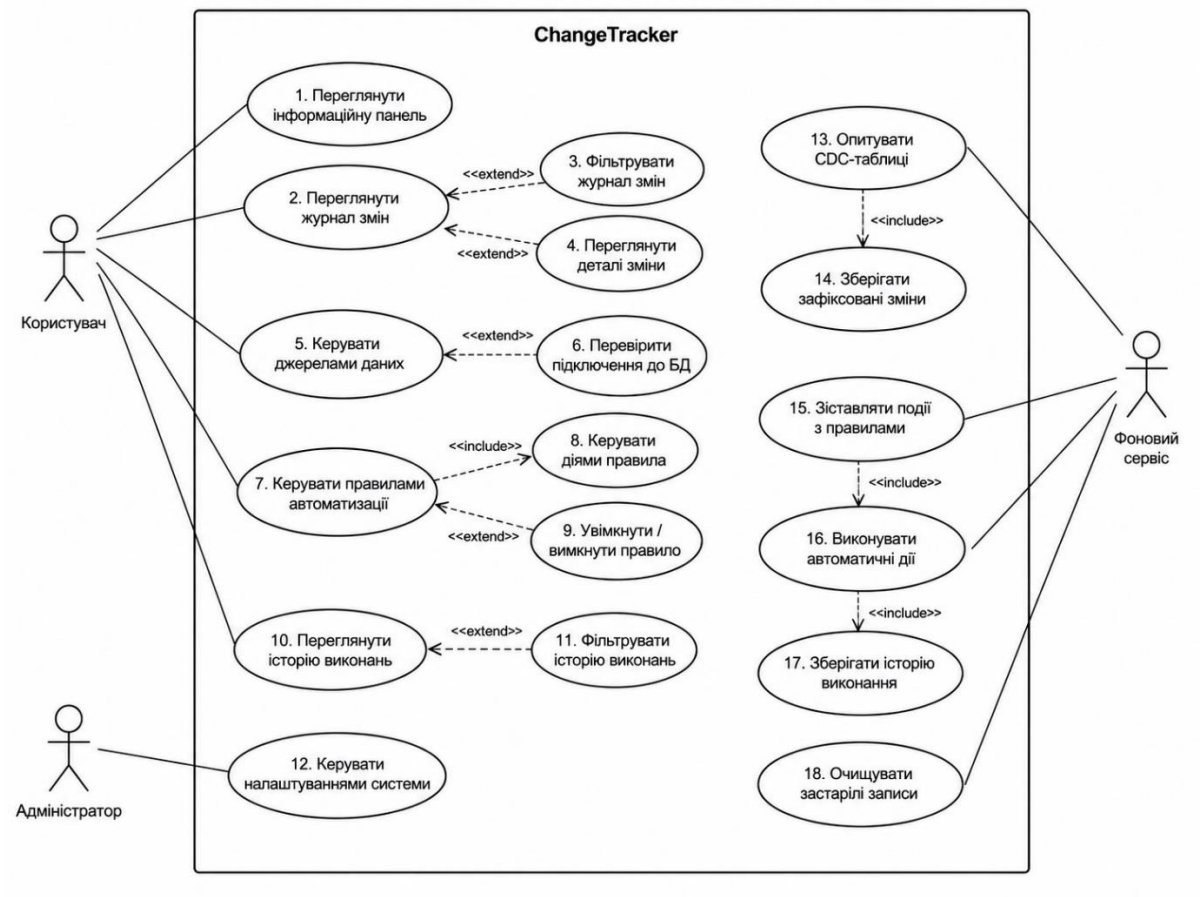


Рисунок 2.1 – Діаграма варіантів використання програмного забезпечення ChangeTracker

На діаграмі варіантів використання всі сценарії доцільно поділити на кілька логічних груп: моніторинг, журнал змін, керування джерелами даних, керування правилами автоматизації, історія виконань, налаштування та фонова обробка. Такий поділ відповідає структурі інтерфейсу програмного забезпечення та дозволяє простіше простежити зв’язок між функціональними вимогами, екранами застосунку та внутрішньою логікою системи.

Першою групою варіантів використання є сценарії моніторингу. До неї належать перегляд інформаційної панелі та оновлення статистичних даних. Під час відкриття інформаційної панелі користувач отримує узагальнену

інформацію про стан системи: загальну кількість зафіксованих подій, кількість подій за поточний день, кількість операцій вставки, оновлення та видалення, а також статистику активності змін за певний період. Сценарій перегляду інформаційної панелі включає внутрішній сценарій завантаження статистики, під час якого система отримує дані з локальної бази та передає їх у графічний інтерфейс. Окремо передбачено сценарій оновлення інформаційної панелі, коли користувач вручну ініціює повторне завантаження статистичних показників.

Другою важливою групою є робота з журналом змін. Цей сценарій є одним із основних у системі, оскільки саме журнал змін дозволяє користувачу переглядати всі події, отримані з CDC-таблиць. Під час відкриття журналу система завантажує першу сторінку подій, відсортованих за часом виникнення. Для кожної події відображаються дата та час, база даних, таблиця, тип операції, значення первинного ключа та статус обробки. Користувач може застосовувати фільтри за часовим проміжком, назвою таблиці та типом операції. У цьому випадку система повторно завантажує список подій уже з урахуванням заданих критеріїв. Також передбачено сценарії очищення фільтрів і навігації між сторінками журналу, що необхідно для роботи з великою кількістю записів.

Окремим сценарієм у межах журналу змін є перегляд деталей події. Після вибору конкретного запису система повинна відобразити детальну інформацію про змінені поля. Для кожного поля показуються його назва, попереднє значення та нове значення. Такий підхід дозволяє користувачу не лише побачити факт зміни, а й проаналізувати, які саме дані були змінені. Це особливо важливо для задач аудиту та контролю коректності роботи інформаційної системи.

Третьою групою варіантів використання є керування джерелами даних. Користувач повинен мати можливість переглядати список налаштованих підключень до баз даних, додавати нові джерела, редагувати існуючі та видаляти непотрібні. Під час додавання джерела даних користувач вводить назву підключення, сервер, базу даних та облікові дані. Після збереження ці дані записуються до локальної бази застосунку та використовуються фоновим сервісом для подальшого опитування CDC-таблиць. Сценарій редагування

дозволяє змінити параметри вже створеного підключення, а сценарій видалення передбачає підтвердження дії, щоб запобігти випадковому вилученню запису.

Важливим допоміжним сценарієм у групі керування джерелами даних є перевірка підключення. Користувач може виконати тестування з'єднання із базою даних, після чого система намагається підключитися до вказаного SQL Server і перевірити доступність бази даних. Результат перевірки відображається користувачу у вигляді повідомлення. Такий сценарій дозволяє виявити помилки в параметрах підключення ще до початку роботи фонового механізму моніторингу. Крім того, джерело даних може бути активним або неактивним. Якщо джерело вимкнене, фоновий сервіс не включає його до чергового циклу опитування, що дає змогу тимчасово призупиняти моніторинг без видалення налаштувань.

Четверта група варіантів використання пов'язана з керуванням правилами автоматизації. Це одна з ключових функціональних частин системи, оскільки вона забезпечує не лише фіксацію змін, а й автоматичне реагування на них. Користувач може переглядати список створених правил, створювати нові правила, редагувати наявні, видаляти їх, а також швидко вмикати або вимикати окреме правило. Кожне правило містить назву, ознаку активності, назву таблиці, тип події та набір дій, які повинні бути виконані після спрацювання правила.

Під час створення правила користувач вказує, для якої таблиці та для якого типу операції воно повинно застосовуватися. Типом події може бути INSERT, UPDATE, DELETE або ALL. Використання значення ALL дозволяє виконувати правило для будь-якого типу зміни в таблиці. У межах створення або редагування правила користувач також додає одну або декілька дій. До підтримуваних типів дій належать HTTP GET-запит, HTTP POST-запит, запис у файл та запуск зовнішньої програми. Для кожного типу дії задаються відповідні параметри: адреса API, заголовки, тіло запиту, тайм-аут, шлях до файлу, текстовий шаблон, шлях до виконуваного файлу або аргументи запуску.

Особливістю сценаріїв роботи з правилами є підтримка декількох дій в одному правилі. Це означає, що одна подія в базі даних може спричинити

послідовне виконання кількох операцій. Наприклад, після вставки нового запису в таблицю модулів система може викликати зовнішній API для очищення кешу, а потім записати інформацію про подію до текстового журналу. Такий підхід робить систему гнучкою та дозволяє адаптувати її до різних задач супроводу інформаційних систем.

Окремим сценарієм є вмикання або вимикання правила. Ця дія виконується без відкриття вікна редагування та дозволяє користувачу швидко призупинити виконання правила, не видаляючи його з системи. Якщо правило вимкнене, фоновий сервіс пропускає його під час зіставлення подій із правилами. Це зручно у випадках, коли певну автоматичну дію потрібно тимчасово відключити.

П’ята група варіантів використання стосується перегляду історії виконань. Після кожного виконання автоматичної дії система створює запис в історії. Користувач може переглядати ці записи, аналізувати статус виконання, тривалість дії та повідомлення про помилки. Для зручності передбачено фільтрацію історії за назвою правила, статусом виконання та часовим проміжком. Також підтримується посторінкова навігація, що дозволяє працювати з великою кількістю записів. Наявність історії виконань є важливою для контролю правильності роботи правил, виявлення помилок інтеграції із зовнішніми сервісами та аналізу стабільності системи.

Шоста група варіантів використання пов’язана з налаштуваннями системи. Ці сценарії виконує адміністратор. До них належать перегляд налаштувань, зміна інтервалу опитування CDC-таблиць, налаштування терміну зберігання журналів, вибір рівня логування та налаштування директорії експорту. Після зміни параметрів адміністратор зберігає налаштування, і вони записуються до локальної бази даних. Фоновий сервіс повинен враховувати оновлені параметри під час наступного циклу роботи. Наприклад, якщо змінено інтервал опитування, нове значення застосовується без необхідності перезапуску програми.

Окрему роль у системі відіграє фоновий сервіс. Він виконує сценарії, які не запускаються безпосередньо користувачем, але забезпечують основну логіку

роботи програмного забезпечення. Під час кожного циклу фоновий сервіс отримує список активних джерел даних, підключається до відповідних баз даних, опитує CDC-таблиці та зберігає нові події у локальній базі застосунку. Після цього сервіс отримує необроблені події, зіставляє їх із активними правилами та передає відповідні пари «подія – правило» до механізму виконання автоматичних дій.

Сценарій зіставлення подій із правилами передбачає перевірку назви таблиці та типу операції. Правило вважається відповідним, якщо його таблиця збігається з таблицею події або задана як універсальне значення, а тип події збігається з типом операції або має значення ALL. Такий механізм дозволяє створювати як вузькоспеціалізовані правила для конкретної таблиці та операції, так і загальні правила для кількох типів подій.

Після знаходження відповідних правил фоновий сервіс запускає сценарій виконання автоматичних дій. Дії виконуються у визначеному порядку. Перед виконанням система підставляє значення події у шаблонні параметри, які можуть використовуватися в URL, тілі HTTP-запиту, тексті файлу або аргументах зовнішньої програми. Після виконання кожної дії система створює запис в історії виконань, де фіксує статус, тривалість та повідомлення про помилку у разі невдалого завершення.

До фонових сценаріїв також належать позначення подій як оброблених і очищення застарілих записів. Після завершення обробки подія отримує статус обробленої, що запобігає повторному виконанню правил для тієї самої зміни. Очищення застарілих записів виконується відповідно до налаштованого терміну зберігання журналів. Це дозволяє обмежувати розмір локальної бази даних і підтримувати стабільну роботу застосунку протягом тривалого часу.

Для окремих сценаріїв використання передбачено спільні допоміжні механізми. Наприклад, під час видалення джерела даних або правила система повинна показувати діалог підтвердження, щоб уникнути випадкового видалення важливої інформації. Після успішного створення, редагування, видалення або збереження даних система повинна показувати коротке

повідомлення користувачу. Такі повідомлення підвищують зручність роботи із застосунком і дають користувачу зрозуміти, що дія була виконана успішно або завершилася з помилкою.

Таким чином, аналіз варіантів використання показує, що програмне забезпечення ChangeTracker повинно забезпечувати взаємодію користувача з основними модулями системи через графічний інтерфейс, а також виконувати значну частину операцій автоматично у фоновому режимі. Людина відповідає за налаштування джерел даних, правил та параметрів роботи системи, тоді як фоновий сервіс забезпечує безперервне отримання змін, їх обробку та виконання налаштованих дій. Такий розподіл сценаріїв використання дозволяє побудувати систему, яка поєднує функції моніторингу, аудиту та автоматизованого реагування на зміни в інформаційній системі.

2.2 Розробка архітектури програмного забезпечення

Після визначення основних вимог і сценаріїв використання було виконано розробку архітектури програмного забезпечення ChangeTracker. Архітектура програмного забезпечення визначає загальну структуру системи, склад її основних компонентів, спосіб їх взаємодії, розподіл відповідальності між модулями та принципи організації доступу до даних. Правильно обрана архітектура забезпечує можливість подальшого розширення системи, спрощує супровід програмного коду та підвищує надійність роботи застосунку.

Розроблюване програмне забезпечення реалізується як десктопний застосунок для операційної системи Windows. Такий підхід обрано з огляду на специфіку задачі, оскільки застосунок призначений для роботи технічного спеціаліста або адміністратора інформаційної системи, який виконує налаштування джерел даних, переглядає журнал змін і керує правилами автоматизації. Десктопний формат дозволяє забезпечити зручну роботу з локальною базою даних застосунку, файловою системою, зовнішніми програмами та мережевими ресурсами без необхідності розгортання окремого вебсервера.

Для реалізації програмного забезпечення обрано мову програмування C# та платформу .NET 8. Вибір цих технологій обумовлений їхньою придатністю для створення Windows-застосунків, підтримкою сучасних засобів асинхронного програмування, наявністю розвиненої екосистеми бібліотек, можливістю використання Entity Framework Core [8], Microsoft.Extensions.Hosting, механізму впровадження залежностей та фонових служб. Для побудови графічного інтерфейсу використовується технологія WPF [9], яка дозволяє створювати структуровані десктопні інтерфейси з підтримкою прив’язки даних, шаблонів відображення, стилів та команд.

Основним архітектурним підходом під час розробки застосунку обрано шаблон MVVM [10]. Його використання є доцільним для WPF-застосунків, оскільки він дозволяє відокремити графічний інтерфейс від логіки представлення та бізнес-логіки. У межах даного підходу XAML-представлення відповідають лише за відображення даних і взаємодію з користувачем [11], ViewModel-класи містять стан сторінок, команди та логіку взаємодії з сервісами, а сервіси та репозиторії виконують роботу із зовнішніми системами, базами даних і фоновою обробкою.

Архітектура програмного забезпечення ChangeTracker побудована за багаторівневим принципом і складається з чотирьох основних рівнів:

– рівень представлення;

– рівень моделей представлення;

– сервісний рівень;

– рівень доступу до даних.

Кожен рівень має чітко визначену відповідальність і взаємодіє лише з нижчим рівнем через інтерфейси (рис. 2.2). Такий підхід зменшує зв’язаність між компонентами, спрощує тестування окремих частин системи та дозволяє змінювати реалізацію окремого рівня без значного впливу на інші частини застосунку.

Рівень представлення містить WPF-вікна, сторінки та елементи керування, які відповідають за візуальне відображення даних і приймання дій користувача.

До цього рівня належать головне вікно застосунку, сторінки інформаційної панелі, джерел даних, журналу змін, правил автоматизації, історії виконань і налаштувань. Особливістю реалізації рівня представлення є те, що він не містить бізнес-логіки. Усі дії користувача передаються до відповідних ViewModel через команди та прив’язки даних.

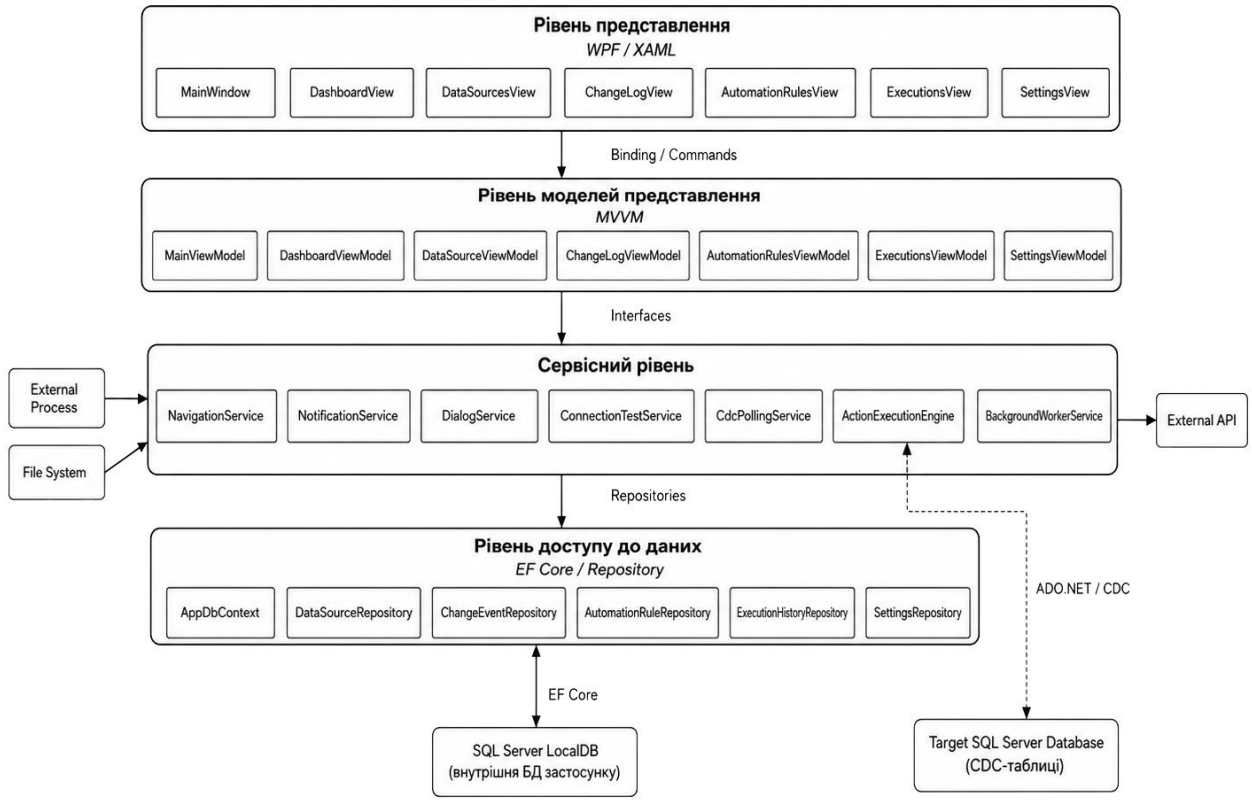


Рисунок 2.2 –Загальна багаторівнева архітектура програмного забезпечення

Головним елементом рівня представлення є головне вікно застосунку. Воно містить бічну навігаційну панель та область відображення активної сторінки. Перемикання сторінок виконується за допомогою прив’язки до поточної ViewModel. Для цього використовується механізм DataTemplate, який визначає, яке представлення повинно бути створене для конкретного типу ViewModel. Такий підхід дозволяє уникнути прямого створення сторінок у коді вікна та відповідає принципам MVVM.

Рівень моделей представлення містить ViewModel-класи, які виступають посередниками між графічним інтерфейсом і сервісним рівнем. Саме цей рівень відповідає за стан сторінок, обробку команд користувача, завантаження даних, фільтрацію, відкриття діалогових вікон та відображення повідомлень. Для реалізації ViewModel використовується бібліотека CommunityToolkit.Mvvm, яка надає базові класи для реалізації сповіщень про зміну властивостей і механізм автоматичної генерації команд.

Центральною ViewModel застосунку є MainViewModel. Вона відповідає за поточну сторінку, навігацію між розділами застосунку та відображення повідомлень користувачу. Окремі ViewModel створюються для кожної сторінки: DashboardViewModel, DataSourceViewModel, ChangeLogViewModel, AutomationRulesViewModel, ExecutionsViewModel та SettingsViewModel. Такий поділ дозволяє ізолювати логіку кожного екрана та спростити подальший супровід коду.

Сервісний рівень містить компоненти, які реалізують основну прикладну логіку системи, взаємодію із зовнішніми ресурсами та фонову обробку. До цього рівня належать сервіси опитування CDC-таблиць, виконання автоматичних дій, перевірки підключення до бази даних, навігації, повідомлень, діалогових вікон та фонової обробки. Усі сервіси використовуються через інтерфейси, що дозволяє відокремити контракти від конкретних реалізацій.

Одним із ключових компонентів сервісного рівня є CdcPollingService. Він відповідає за підключення до цільової бази даних SQL Server і отримання інформації про зміни з CDC-таблиць. Для роботи з CDC використовується ADO.NET, оскільки механізм Change Data Capture передбачає звернення до системних таблиць, функцій і динамічних імен CDC-таблиць, що незручно реалізовувати засобами Entity Framework Core. Сервіс отримує зміни для активних джерел даних, перетворює їх у внутрішні об’єкти ChangeEvent та ChangeDetail і передає для збереження у локальній базі застосунку.

Іншим важливим компонентом є ActionExecutionEngine. Його призначення полягає у виконанні дій, пов’язаних із правилами автоматизації. Після того як

подія була зіставлена з відповідним правилом, механізм виконання дій послідовно обробляє всі дії правила. Залежно від типу дії система може виконати HTTP GET-запит, HTTP POST-запит, записати інформацію до файлу або запустити зовнішню програму. Після виконання кожної дії результат зберігається в історії виконань.

Фонову роботу системи забезпечує BackgroundWorkerService. Він запускається разом із застосунком і циклічно виконує основні автоматизовані операції. У кожному циклі сервіс отримує активні джерела даних, виконує опитування CDC-таблиць, зберігає нові події, отримує необроблені записи, зіставляє їх із правилами, запускає механізм виконання дій, позначає події як оброблені та очищує застарілі записи. Завдяки фоновому сервісу користувач не повинен вручну запускати моніторинг змін, а основна логіка обробки виконується автоматично.

Рівень доступу до даних відповідає за збереження та отримання службових даних застосунку. Для цього використовується Entity Framework Core 8 і шаблон Repository. Усі запити до локальної бази даних інкапсульовані в репозиторіях, що дозволяє не використовувати DbContext безпосередньо у ViewModel або сервісах. Такий підхід забезпечує кращу структурованість коду та спрощує зміну способу зберігання даних у майбутньому.

У системі передбачено кілька репозиторіїв: DataSourceRepository, ChangeEventRepository, AutomationRuleRepository, ExecutionHistoryRepository та SettingsRepository. Вони відповідають відповідно за роботу з джерелами даних, журналом змін, правилами автоматизації, історією виконань і налаштуваннями системи. Кожен репозиторій реалізує відповідний інтерфейс, що дозволяє сервісному рівню працювати з абстракціями, а не з конкретними класами.

Архітектура застосунку передбачає використання двох різних баз даних. Перша база даних є внутрішньою службовою базою застосунку і розміщується у SQL Server LocalDB. У ній зберігаються налаштування джерел даних, журнал зафіксованих змін, деталі змін, правила автоматизації, дії правил, історія виконань і параметри системи. Друга база даних є цільовою базою

інформаційної системи, зміни в якій необхідно відстежувати. Вона розміщується на окремому екземплярі SQL Server і повинна мати увімкнений механізм Change Data Capture.

Поділ внутрішньої та цільової баз даних є важливим архітектурним рішенням. Внутрішня база використовується лише самим застосунком ChangeTracker і не впливає на структуру цільової інформаційної системи. Цільова база даних, у свою чергу, залишається джерелом змін, які отримуються через CDC. Такий підхід дозволяє не втручатися у бізнес-логіку основної інформаційної системи та зберігати всі службові дані моніторингу окремо.

Для кращого подання складу програмного забезпечення та взаємодії між його компонентами було розроблено компонентну діаграму (рис. 2.3). Вона відображає основні програмні компоненти системи, їхню належність до архітектурних рівнів і зовнішні ресурси, з якими взаємодіє застосунок.

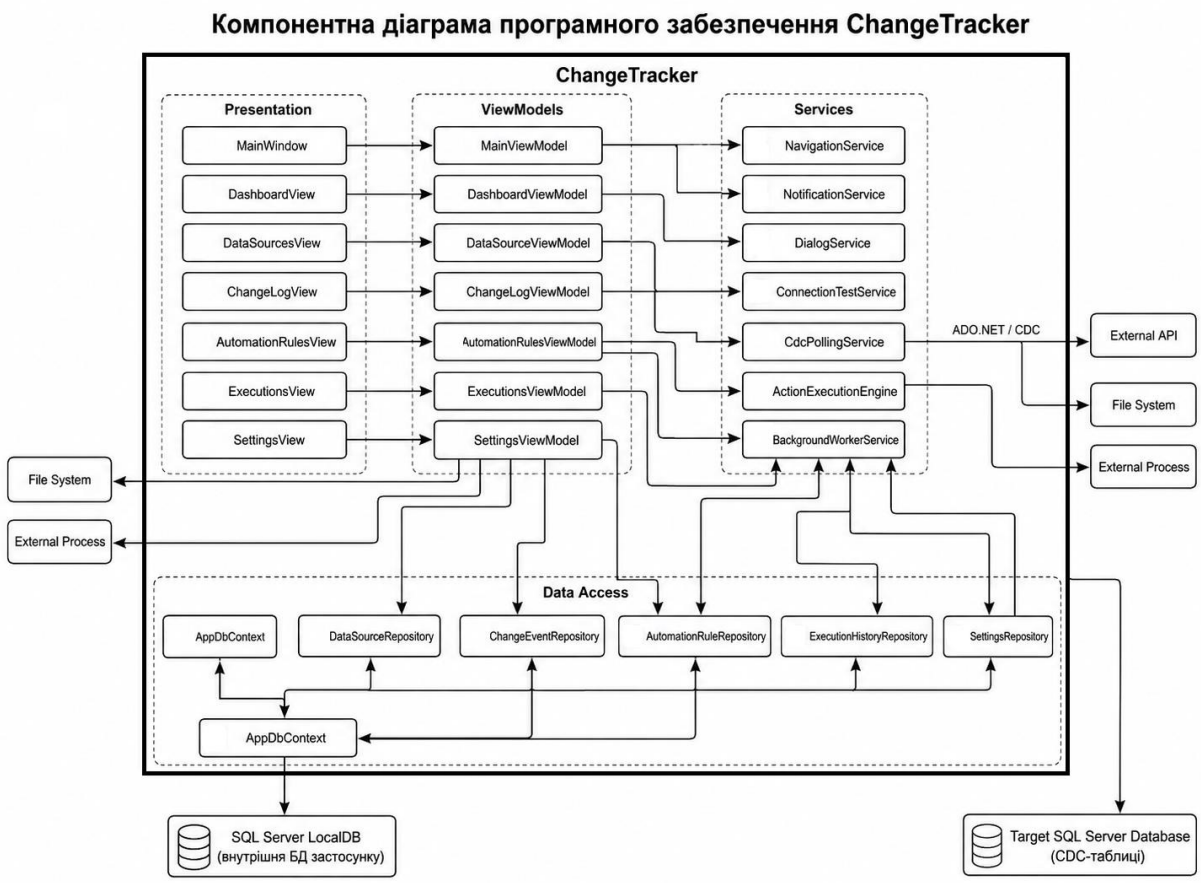


Рисунок 2.3 – Компонентна діаграма програмного забезпечення ChangeTracker

На компонентній діаграмі доцільно виділити такі основні компоненти: WPF Presentation, ViewModels, Services, Repositories, AppDbContext, SQL Server LocalDB, Target SQL Server Database, External API, File System та External Process. Рівень представлення взаємодіє з ViewModel, ViewModel звертаються до сервісів і репозиторіїв через інтерфейси, сервіси виконують прикладну логіку, а репозиторії забезпечують доступ до локальної бази даних. Компонент CdcPollingService окремо взаємодіє з цільовою базою даних SQL Server через ADO.NET, а ActionExecutionEngine може взаємодіяти із зовнішнім API, файловою системою або зовнішніми програмами.

2.3 Розробка проєкту програмного забезпечення

Після визначення вимог до програмного забезпечення та розробки його архітектури було виконано проєктування основних структурних елементів системи ChangeTracker. На цьому етапі було уточнено склад класів, сутностей бази даних, зв’язків між ними, логіку взаємодії модулів та структуру користувацького інтерфейсу. Проєктування програмного забезпечення виконувалося з використанням об’єктно-орієнтованого підходу, що є доцільним для застосунків, реалізованих мовою C# із використанням шаблону MVVM.

Основною метою проєктування є створення такої структури програмного забезпечення, яка забезпечує відповідність сформульованим вимогам, підтримує подальшу реалізацію основних функцій застосунку та дозволяє розширювати систему без значного ускладнення її внутрішньої логіки. Для програмного забезпечення ChangeTracker важливо було передбачити не лише зберігання інформації про зміни в базі даних, а й можливість налаштування правил автоматичної обробки подій, виконання різних типів дій, збереження історії виконань та перегляд усіх цих даних через графічний інтерфейс.

Проєкт програмного забезпечення охоплює кілька логічних частин: доменну модель, модель доступу до даних, сервісний рівень, рівень моделей представлення та структуру користувацького інтерфейсу. Такий поділ відповідає

архітектурі, розглянутій у попередньому підрозділі, і дозволяє представити систему на різних рівнях деталізації.

Для опису основних класів і зв’язків між ними було розроблено UML-діаграму класів (рис. 2.4). Вона відображає доменні сутності, об’єкти передавання даних, інтерфейси репозиторіїв, реалізації репозиторіїв, сервісні інтерфейси, сервісні класи, базові класи моделей представлення та ViewModel-класи окремих сторінок застосунку.

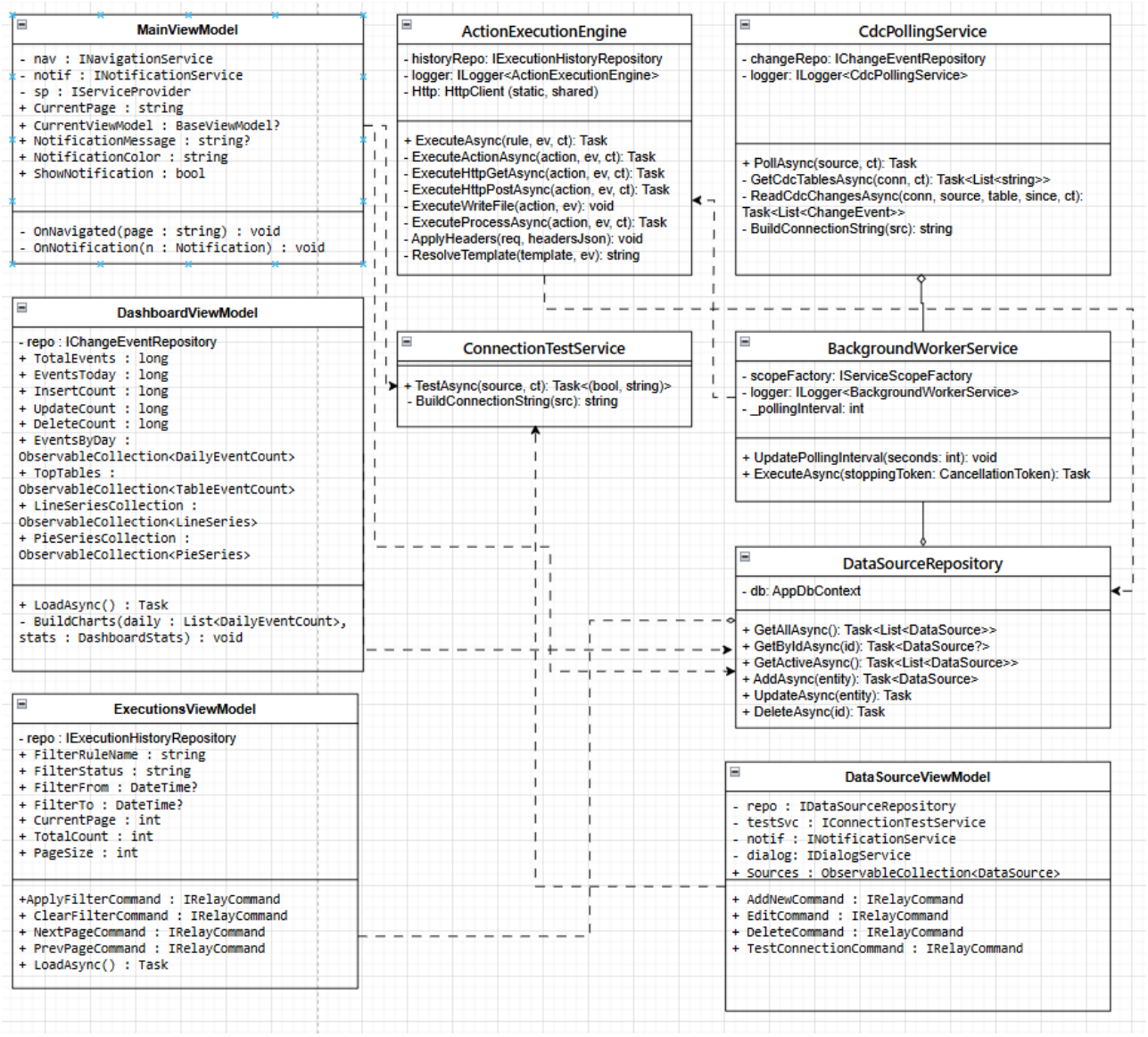


Рисунок 2.4 – UML-діаграма класів

Центральною частиною проєкту є доменна модель. Вона описує основні сутності предметної області, з якими працює програмне забезпечення. До

доменних сутностей належать DataSource, ChangeEvent, ChangeDetail, AutomationRule, RuleAction, ExecutionHistory та ApplicationSettings.

Сутність DataSource призначена для зберігання інформації про джерело даних, тобто про базу даних, зміни в якій необхідно відстежувати. Вона містить такі основні поля, як ідентифікатор, назва підключення, сервер, назва бази даних, ім’я користувача, пароль, ознака активності, дата створення та дата останнього опитування. Наявність ознаки активності дозволяє тимчасово виключати джерело даних із процесу моніторингу без його видалення із системи.

Сутність ChangeEvent призначена для зберігання інформації про зафіксовану зміну в цільовій базі даних. Вона містить ідентифікатор події, посилання на джерело даних, дату й час виникнення події, назву бази даних, назву таблиці, тип операції, значення первинного ключа та ознаку обробки. Саме ця сутність є основою журналу змін, який відображається користувачу в інтерфейсі застосунку.

Для деталізації події використовується сутність ChangeDetail. Вона пов’язана з ChangeEvent і містить інформацію про окреме змінене поле: назву колонки, попереднє значення та нове значення. Завдяки цьому система може показувати користувачу не лише сам факт зміни, а й конкретний зміст цієї зміни.

Сутність AutomationRule описує правило автоматичної обробки подій. Вона містить назву правила, ознаку активності, назву таблиці, тип події, ідентифікатор джерела даних і дату створення. Правило може застосовуватися до конкретної таблиці або до всіх таблиць, якщо використовується універсальне значення. Аналогічно тип події може бути конкретним, наприклад INSERT, UPDATE або DELETE, або універсальним, якщо правило повинно виконуватися для всіх типів операцій.

Для опису дій, які виконуються при спрацюванні правила, використовується сутність RuleAction. Одне правило може містити кілька дій, що виконуються у визначеному порядку. Для цього в сутності передбачено поле SortOrder. Також RuleAction містить тип дії та набір параметрів, необхідних для її виконання: URL, заголовки, тіло запиту, тайм-аут, шлях до файлу, текстовий

шаблон, шлях до виконуваного файлу та аргументи запуску. Наявність різних наборів параметрів в одній сутності дозволяє підтримувати декілька типів дій у межах єдиної моделі.

Сутність ExecutionHistory призначена для зберігання результатів виконання автоматичних дій. Вона містить час виконання, ідентифікатор правила, ідентифікатор події, тип дії, статус виконання, тривалість виконання та повідомлення про помилку. Ця сутність використовується для побудови журналу виконань і дозволяє аналізувати, які правила спрацювали успішно, а які завершилися помилкою.

Сутність ApplicationSettings містить загальні параметри роботи застосунку: інтервал опитування CDC-таблиць, термін зберігання журналів, рівень логування, директорію експорту та дату останнього оновлення налаштувань. Наявність окремої сутності для налаштувань дозволяє змінювати поведінку фонового сервісу без перезапуску програми.

Між доменними сутностями встановлено кілька основних зв’язків. DataSource пов’язана з ChangeEvent, оскільки одне джерело даних може мати багато зафіксованих подій. DataSource також пов’язана з AutomationRule, оскільки правила можуть створюватися для конкретного джерела даних. ChangeEvent пов’язана з ChangeDetail за принципом “один до багатьох”, оскільки одна подія може містити кілька змінених полів. AutomationRule пов’язана з RuleAction також за принципом “один до багатьох”, оскільки одне правило може мати кілька дій. ExecutionHistory пов’язана одночасно з ChangeEvent і AutomationRule, оскільки кожен запис історії виконання створюється внаслідок обробки певної події певним правилом.

Для зручності передавання агрегованих даних у системі використовуються окремі об’єкти передавання даних. До них належать DashboardStats, DailyEventCount і TableEventCount. DashboardStats використовується для зберігання загальної статистики інформаційної панелі: загальної кількості подій, кількості подій за поточний день, кількості операцій INSERT, UPDATE і DELETE. DailyEventCount використовується для відображення кількості подій

за днями, а TableEventCount — для визначення таблиць, у яких найчастіше відбувалися зміни.

Окрему частину проєкту становить рівень доступу до даних. Для роботи з внутрішньою базою даних застосунку використовується AppDbContext, який містить набори DbSet для всіх основних сутностей: DataSources, ChangeEvents, ChangeDetails, AutomationRules, RuleActions, ExecutionHistories та ApplicationSettings. Саме AppDbContext відповідає за відображення класів доменної моделі на таблиці внутрішньої бази даних.

Для ізоляції логіки доступу до даних від інших частин системи використовується шаблон Repository. Кожен репозиторій має відповідний інтерфейс, що визначає доступні операції. Наприклад, IDataSourceRepository визначає операції отримання всіх джерел даних, отримання активних джерел, додавання, оновлення та видалення запису. IChangeEventRepository містить методи для фільтрації подій, отримання кількості записів, завантаження події з деталями, отримання необроблених подій, додавання групи подій, позначення подій як оброблених, отримання статистики для інформаційної панелі та очищення застарілих записів.

IAutomationRuleRepository відповідає за роботу з правилами автоматизації та їхніми діями. Особливо важливим є метод отримання правил, які відповідають конкретній події. Він використовується фоновим сервісом під час зіставлення подій із правилами. IExecutionHistoryRepository забезпечує роботу з історією виконань, а ISettingsRepository — отримання та збереження загальних налаштувань застосунку.

Реалізації репозиторіїв працюють із AppDbContext і приховують деталі запитів до бази даних від ViewModel та сервісів. Завдяки цьому інші компоненти системи працюють не з конкретною реалізацією бази даних, а з абстрактними інтерфейсами. Такий підхід підвищує гнучкість проєкту та спрощує подальше тестування.

Для опису структури внутрішньої бази даних було розроблено схематичне представлення бази даних (рис. 2.5). Вона відображає таблиці, що відповідають

доменним сутностям, а також зв’язки між ними.

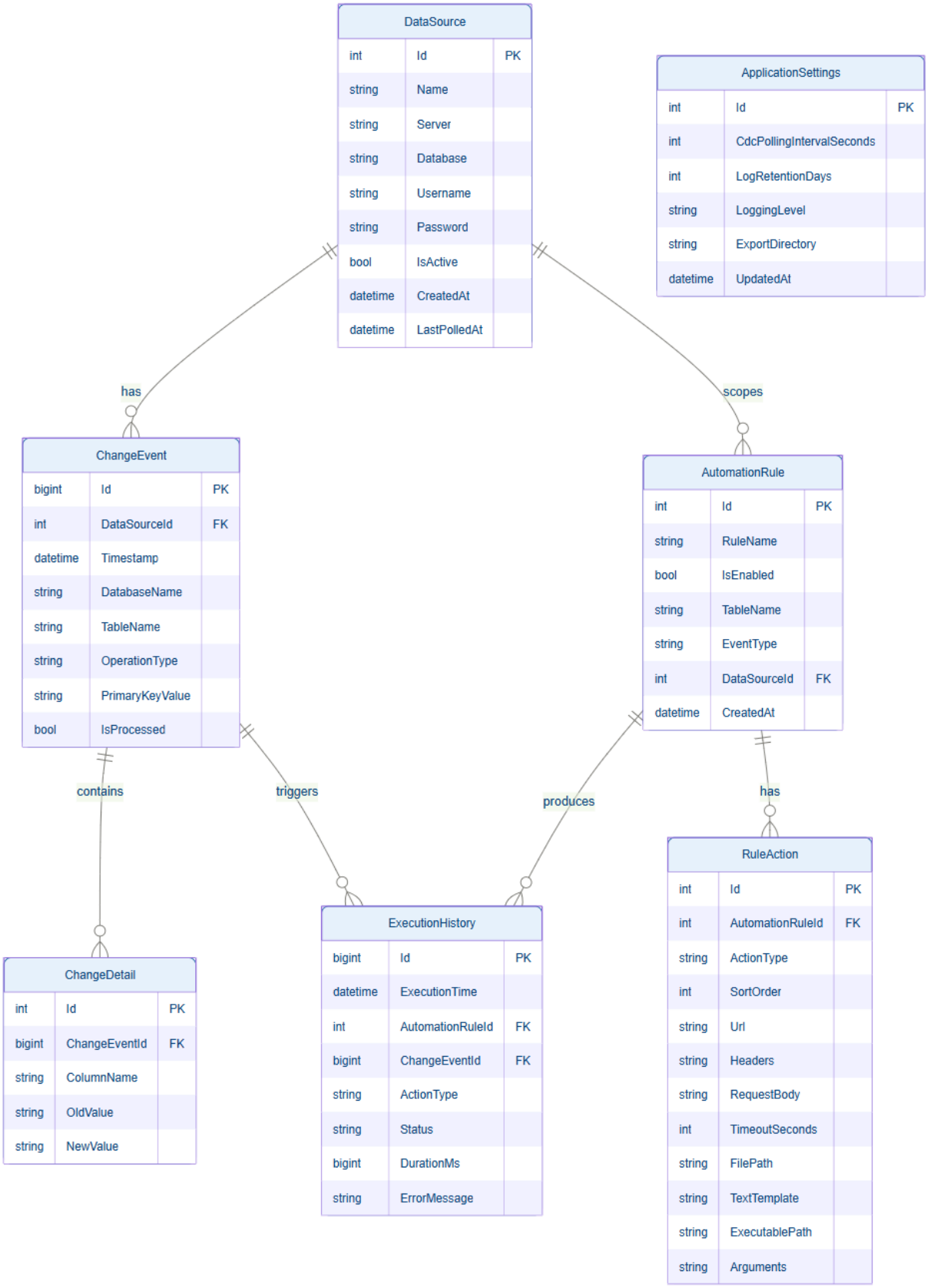


Рисунок 2.5 – Схематичне представлення структури бази даних програмного забезпечення ChangeTracker

Для детальнішого опису структури внутрішньої бази даних програмного забезпечення було сформовано специфікації таблиць (табл. 2.1 – 2.7). У них наведено назви полів, типи даних і призначення кожного поля. Внутрішня база даних застосунку містить таблиці для зберігання джерел даних, зафіксованих подій, деталей змін, правил автоматизації, дій правил, історії виконань і загальних налаштувань системи.

Основною таблицею для зберігання підключень є DataSources (табл. 2.1). У ній зберігаються параметри підключення до цільових баз даних, а також інформація про активність джерела та дату останнього опитування.

Таблиця 2.1 – Опис таблиці DataSources

Поле	Тип даних	Опис
Id	int	Первинний ключ таблиці, що автоматично збільшується.
Name	nvarchar(200)	Зручна для користувача назва підключення, наприклад «Production DB».
Server	nvarchar(500)	Ім’я хоста або IP-адреса SQL Server, за потреби разом із назвою екземпляра.
Database	nvarchar(200)	Назва цільової бази даних, зміни в якій необхідно відстежувати за допомогою CDC.
Username	nvarchar(200)	Ім’я користувача SQL Server. Порожнє значення означає використання Windows-автентифікації.
Password	nvarchar(500)	Пароль користувача SQL Server. Порожнє значення використовується для Windows-автентифікації.
IsActive	bit	Ознака активності джерела даних. Якщо значення дорівнює true, фоновий сервіс опитує це джерело під час кожного циклу.
CreatedAt	datetime2	Дата і час створення запису про підключення у форматі UTC.
LastPolledAt	datetime2	Дата і час останнього успішного опитування CDC-таблиць у форматі UTC. Поле може бути порожнім.

Таблиця ChangeEvents є основною таблицею журналу змін (табл. 2.2). У ній зберігається загальна інформація про кожну подію, отриману з CDC-таблиць цільової бази даних. Ці дані використовуються для відображення журналу змін, фільтрації подій, побудови статистики та запуску правил автоматизації.

Таблиця 2.2 – Опис таблиці ChangeEvents

Поле	Тип даних	Опис
Id	bigint	Первинний ключ таблиці, що автоматично збільшується. Тип bigint використовується з урахуванням можливої великої кількості подій.
DataSourceId	int	Зовнішній ключ, що посилається на джерело даних, у якому була зафіксована подія.
Timestamp	datetime2	Дата і час виявлення та збереження зміни фоновим сервісом у форматі UTC.
DatabaseName	nvarchar(200)	Назва бази даних, у якій відбулася зміна. Значення дублюється для зручного відображення в інтерфейсі.
TableName	nvarchar(200)	Назва таблиці, у якій було виконано зміну, наприклад Customers або Orders.
OperationType	nvarchar(20)	Тип операції зміни даних: INSERT, UPDATE або DELETE.
PrimaryKeyValue	nvarchar(500)	Текстове представлення значення первинного ключа зміненого запису.
IsProcessed	bit	Ознака того, що подія була зіставлена з правилами автоматизації та оброблена системою.

Таблиця ChangeDetails деталізує події, збережені в таблиці ChangeEvents (табл. 2.3). Вона дозволяє відобразити користувачу не лише факт зміни запису, а й конкретні поля, які були змінені, а також їх попередні та нові значення.

Таблиця 2.3 – Опис таблиці ChangeDetails

Поле	Тип даних	Опис
Id	int	Первинний ключ таблиці, що автоматично збільшується.
ChangeEventId	bigint	Зовнішній ключ, що посилається на батьківську подію ChangeEvent. Запис видаляється каскадно разом із подією.
ColumnName	nvarchar(200)	Назва колонки бази даних, значення якої було створено або змінено.
OldValue	nvarchar(max)	Текстове представлення значення поля до зміни. Для операцій INSERT може бути порожнім.
NewValue	nvarchar(max)	Текстове представлення значення поля після зміни. Для операцій DELETE може бути порожнім.

Таблиця AutomationRules (табл. 2.4) використовується для зберігання правил автоматичної обробки подій. Кожне правило пов’язане з джерелом даних

і визначає, для якої таблиці та якого типу операції воно повинно спрацьовувати. Таблиця ExecutionHistories зберігає результати виконання дій (табл. 2.5). Вона пов’язана з таблицями AutomationRules і ChangeEvents, що дозволяє визначити, яке правило було виконано, для якої події та з яким результатом.

Таблиця 2.4 – Опис таблиці AutomationRules

Поле	Тип даних	Опис
Id	int	Первинний ключ таблиці, що автоматично збільшується.
RuleName	nvarchar(200)	Зрозуміла для користувача назва правила, наприклад «Notify on new order».
IsEnabled	bit	Ознака активності правила. Якщо значення дорівнює false, правило пропускається під час обробки подій, але не видаляється із системи.
TableName	nvarchar(200)	Назва таблиці, для якої повинно спрацьовувати правило. Значення * використовується для всіх таблиць джерела даних.
EventType	nvarchar(20)	Тип операції, для якої застосовується правило: INSERT, UPDATE, DELETE або ALL.
DataSourceId	int	Зовнішній ключ, що обмежує дію правила подіями з конкретного джерела даних.
CreatedAt	datetime2	Дата і час створення правила у форматі UTC.

Таблиця 2.5 – Опис таблиці ExecutionHistories

Поле	Тип даних	Опис
Id	bigint	Первинний ключ таблиці, що автоматично збільшується. Тип bigint використовується з урахуванням можливої великої кількості записів історії.
ExecutionTime	datetime2	Дата і час початку виконання дії у форматі UTC.
AutomationRuleId	int	Зовнішній ключ, що посилається на правило, яке було спрацьоване.
ChangeEventId	bigint	Зовнішній ключ, що посилається на подію, яка спричинила виконання дії.
ActionType	nvarchar(50)	Дубльоване значення типу виконаної дії: HTTP_GET, HTTP_POST, WRITE_FILE або EXEC_PROCESS.
Status	nvarchar(20)	Результат виконання дії: Success або Failed.
DurationMs	bigint	Тривалість виконання дії в мілісекундах, виміряна за допомогою Stopwatch.
ErrorMessage	nvarchar(max)	Повідомлення про помилку у випадку, якщо виконання завершилося зі статусом Failed. Для успішного виконання поле може бути порожнім.

Таблиця RuleActions (табл 2.6) містить перелік дій, пов’язаних із правилами. Оскільки одне правило може мати декілька дій, між таблицями AutomationRules і RuleActions встановлено зв’язок “один до багатьох”. Це дозволяє створювати складні сценарії реагування на одну подію, наприклад виконати HTTP-запит і додатково записати інформацію у файл.

Таблиця 2.6 – Опис таблиці RuleActions

Поле	Тип даних	Опис
Id	int	Первинний ключ таблиці, що автоматично збільшується.
AutomationRuleId	int	Зовнішній ключ, що посилається на батьківське правило AutomationRule. Запис видаляється каскадно разом із правилом.
ActionType	nvarchar(50)	Тип дії, що виконується системою: HTTP_GET, HTTP_POST, WRITE_FILE або EXEC_PROCESS.
SortOrder	int	Порядок виконання дії у випадку, якщо правило містить кілька дій. Дії виконуються за зростанням цього значення.
Url	nvarchar(2000)	Цільова URL-адреса для HTTP_GET та HTTP_POST дій. Підтримує підстановку шаблонних змінних.
Headers	nvarchar(max)	JSON-об’єкт HTTP-заголовків, які потрібно додати до запиту, наприклад Authorization.
RequestBody	nvarchar(max)	JSON-тіло запиту для HTTP_POST. Підтримує підстановку шаблонних змінних.
TimeoutSeconds	int	Тайм-аут HTTP-запиту в секундах. Якщо значення не задане, використовується стандартне значення 30 секунд.
FilePath	nvarchar(1000)	Абсолютний шлях до файлу для дій типу WRITE_FILE.
TextTemplate	nvarchar(max)	Шаблон рядка, який додається до файлу. Підтримує підстановку шаблонних змінних.
ExecutablePath	nvarchar(1000)	Абсолютний шлях до виконуваного файлу для дій типу EXEC_PROCESS.
Arguments	nvarchar(max)	Аргументи командного рядка, які передаються процесу. Підтримують підстановку шаблонних змінних.

Таблиця ApplicationSettings (табл. 2.7) використовується для зберігання глобальних параметрів роботи програмного забезпечення. Оскільки налаштування зберігаються у внутрішній базі даних, фоновий сервіс може

враховувати змінені значення під час наступного циклу роботи без необхідності перезапуску застосунку.

Таблиця 2.7 – Опис таблиці ApplicationSettings

Поле	Тип даних	Опис
Id	int	Первинний ключ таблиці. Значення завжди дорівнює 1, оскільки таблиця містить єдиний запис із налаштуваннями системи.
CdcPollingIntervalSeconds	int	Інтервал опитування CDC-таблиць фоновим сервісом у секундах. Мінімальне значення становить 5 секунд, стандартне — 10 секунд.
LogRetentionDays	int	Кількість днів зберігання записів ChangeEvents і ExecutionHistories. Старіші записи автоматично видаляються під час циклу фонового сервісу.
LoggingLevel	nvarchar(50)	Мінімальний рівень логування Serilog: Verbose, Debug, Information, Warning або Error.
ExportDirectory	nvarchar(500)	Стандартний шлях у файловій системі, який використовується як коренева директорія для дій типу WRITE_FILE.
UpdatedAt	datetime2	Дата і час останнього збереження налаштувань у форматі UTC.

Під час проєктування бази даних було враховано необхідність ефективного пошуку та фільтрації записів. Оскільки журнал змін може містити велику кількість подій, для таблиці ChangeEvents доцільно використовувати індекси за часом події, назвою таблиці та типом операції. Це дозволяє прискорити пошук записів під час фільтрації журналу. Для історії виконань важливим є індекс за часом виконання, оскільки користувач найчастіше переглядає останні виконання або фільтрує записи за часовим проміжком.

У таблиці ChangeDetails і RuleActions використовується каскадне видалення, оскільки деталі зміни не мають сенсу без відповідної події, а дії не мають сенсу без правила. Для історії виконань доцільно використовувати обмежене видалення або збереження історичних даних, оскільки записи історії мають аналітичне значення навіть після зміни або видалення правила.

Сервісний рівень проєкту містить класи, які реалізують основну прикладну логіку програмного забезпечення. Ключовим сервісом є

CdcPollingService. Він відповідає за підключення до цільової бази даних SQL Server, визначення CDC-таблиць, читання нових змін і формування об’єктів ChangeEvent та ChangeDetail. На відміну від внутрішньої бази застосунку, цільова база даних читається не через AppDbContext, а напряму через ADO.NET, оскільки CDC-дані зберігаються у системних таблицях і потребують виконання спеціальних SQL-запитів.

ActionExecutionEngine відповідає за виконання дій, пов’язаних із правилами автоматизації. Він отримує правило та подію, після чого послідовно виконує всі дії правила. Перед виконанням дії система підставляє значення події в шаблонні параметри. Наприклад, у URL, тілі HTTP-запиту, текстовому шаблоні або аргументах запуску можуть використовуватися значення ідентифікатора події, назви таблиці, типу операції, часу події та первинного ключа. Після завершення дії результат зберігається в історії виконань.

ConnectionTestService призначений для перевірки підключення до бази даних. Він використовується під час налаштування джерел даних і дозволяє визначити, чи доступна база даних та чи увімкнено в ній механізм Change Data Capture. NavigationService реалізує перемикання між сторінками застосунку, NotificationService відповідає за показ повідомлень користувачу, а DialogService використовується для відображення діалогів підтвердження.

Фонову логіку системи реалізує BackgroundWorkerService. Він періодично виконує цикл обробки, який складається з кількох етапів: отримання активних джерел даних, опитування CDC-таблиць, збереження нових подій, отримання необроблених подій, пошук відповідних правил, виконання дій, збереження результатів виконання, позначення подій як оброблених та очищення застарілих записів. Такий сервіс забезпечує автоматичну роботу системи без необхідності ручного запуску обробки користувачем.

Рівень моделей представлення складається з ViewModel-класів, кожен із яких відповідає певній сторінці або діалоговому вікну застосунку. MainViewModel відповідає за поточну сторінку, навігацію та повідомлення користувачу. DashboardViewModel отримує статистичні дані для інформаційної

панелі. DataSourceViewModel забезпечує керування джерелами даних. ChangeLogViewModel реалізує перегляд журналу змін, фільтрацію та посторінкову навігацію. AutomationRulesViewModel відповідає за керування правилами автоматизації. ExecutionsViewModel забезпечує перегляд історії виконань, а SettingsViewModel — зміну параметрів роботи системи.

Крім сторінкових ViewModel, у проєкті передбачено ViewModel для діалогових вікон. DataSourceDialogViewModel використовується під час створення або редагування джерела даних. RuleDialogViewModel використовується під час створення або редагування правила автоматизації та списку пов’язаних із ним дій. Такий підхід дозволяє відокремити логіку діалогових вікон від основних сторінок застосунку.

Користувацький інтерфейс проєктується відповідно до структури основних сценаріїв використання. Головне вікно застосунку містить бічну навігаційну панель і область відображення активної сторінки. У навігаційній панелі розміщуються розділи Dashboard, Data Sources, Change Log, Automation Rules, Executions та Settings. Така структура відповідає основним функціональним групам системи та дозволяє користувачу швидко переходити між модулями.

Сторінка Dashboard призначена для відображення загальної статистики. На ній розміщуються інформаційні картки з кількісними показниками та графічні елементи для аналізу змін за часом і типами операцій. Сторінка Data Sources містить таблицю джерел даних, кнопки додавання, редагування, видалення та перевірки підключення. Сторінка Change Log містить фільтри, таблицю подій і панель деталей вибраної зміни. Сторінка Automation Rules призначена для керування правилами та перегляду пов’язаних дій. Сторінка Executions відображає історію виконання автоматичних дій, а Settings дозволяє змінювати параметри роботи застосунку.

Існує сценарій автоматичної обробки зміни, починаючи з опитування CDC-таблиць і завершуючи збереженням історії виконання. У цьому сценарії BackgroundWorkerService ініціює черговий цикл роботи та отримує список

активних джерел даних. Для кожного джерела викликається CdcPollingService, який підключається до цільової бази даних і читає CDC-таблиці. Після отримання змін сервіс формує об’єкти ChangeEvent і ChangeDetail та передає їх до репозиторію для збереження у внутрішній базі даних. Далі фоновий сервіс отримує необроблені події, для кожної події викликає AutomationRuleRepository і визначає список правил, які відповідають назві таблиці та типу операції.

Після знаходження відповідних правил BackgroundWorkerService передає подію та правило до ActionExecutionEngine. Механізм виконання дій послідовно обробляє всі RuleAction, пов’язані з правилом. Залежно від типу дії він може звернутися до зовнішнього API, записати дані у файл або запустити зовнішню програму. Після виконання кожної дії створюється запис ExecutionHistory. Після завершення обробки подія позначається як оброблена, щоб уникнути повторного виконання тих самих правил для тієї самої зміни.

Проєктування програмного забезпечення ChangeTracker дозволило визначити структуру системи на рівні класів, бази даних, сервісів, моделей представлення та користувацького інтерфейсу. Отриманий проєкт забезпечує відповідність поставленим вимогам, підтримує автоматичне отримання змін із бази даних, дає змогу створювати правила реагування на події та зберігати результати виконання дій. Розроблена структура є основою для подальшої реалізації програмного забезпечення.

2.4 Реалізація програмного забезпечення

Після розробки архітектури та проєкту програмного забезпечення було виконано реалізацію застосунку ChangeTracker. Реалізація здійснювалася мовою програмування C# з використанням платформи .NET 8 та технології WPF для побудови графічного інтерфейсу користувача. Для організації структури застосунку було використано шаблон MVVM, що дозволило розділити представлення, логіку інтерфейсу, сервісну логіку та доступ до даних.

Основною метою етапу реалізації було створення працездатного десктопного застосунку, який забезпечує підключення до баз даних Microsoft

SQL Server, отримання змін за допомогою технології Change Data Capture, збереження подій у внутрішній базі даних, перегляд журналу змін, налаштування правил автоматичного реагування та виконання відповідних дій.

Реалізація програмного забезпечення виконувалася з урахуванням багаторівневої архітектури, описаної в попередніх підрозділах. Програмний код було розподілено між кількома логічними групами: представлення, моделі представлення, доменні сутності, сервіси, репозиторії та контекст бази даних. Такий підхід дозволив уникнути змішування логіки інтерфейсу з логікою доступу до даних і забезпечив кращу супроводжуваність програмного коду.

Для запуску застосунку та конфігурації його основних компонентів використовується механізм Host.CreateDefaultBuilder, який надає можливість налаштувати впровадження залежностей, конфігурацію, логування та фонові служби. У контейнері залежностей реєструються ViewModel-класи, сервіси, репозиторії, контекст бази даних та фоновий сервіс. Завдяки цьому окремі компоненти застосунку не створюють залежності самостійно, а отримують їх через конструктори.

```
_host = Host.CreateDefaultBuilder()
    .UseSerilog()
    .ConfigureServices((ctx, services) =>
    {
        // DB
        services.AddDbContext<AppDbContext>(opts =>
            opts.UseSqlServer(config.GetConnectionString("LocalDatabase")),
            ServiceLifetime.Transient);

        // Repositories
        services.AddTransient<IDataSourceRepository, DataSourceRepository>();
        services.AddTransient<IChangeEventRepository, ChangeEventRepository>();
        services.AddTransient<IAutomationRuleRepository,
AutomationRuleRepository>();
        services.AddTransient<IExecutionHistoryRepository,
ExecutionHistoryRepository>();
        services.AddTransient<ISettingsRepository, SettingsRepository>();

        // Services
        services.AddTransient<ICdcPollingService, CdcPollingService>();
        services.AddTransient<IActionExecutionEngine, ActionExecutionEngine>();
        services.AddTransient<IConnectionTestService, ConnectionTestService>();
        services.AddHostedService<BackgroundWorkerService>();

        // Infrastructure
        services.AddSingleton<INavigationService, NavigationService>();
        services.AddSingleton<INotificationService, NotificationService>();
        services.AddSingleton<IDialogService, DialogService>();

        // ViewModels
        services.AddSingleton<MainViewModel>();
        services.AddTransient<DashboardViewModel>();
```

```
        services.AddTransient<DataSourceViewModel>();
        services.AddTransient<ChangeLogViewModel>();
        services.AddTransient<AutomationRulesViewModel>();
        services.AddTransient<ExecutionsViewModel>();
        services.AddTransient<SettingsViewModel>();
        services.AddTransient<DataSourceDialogViewModel>();
        services.AddTransient<RuleDialogViewModel>();

        // Views
        services.AddTransient<MainWindow>();
        services.AddTransient<DataSourceDialog>();
        services.AddTransient<RuleDialog>();
    })
    .Build();
```

Графічний інтерфейс користувача реалізовано засобами WPF. Основним вікном застосунку є MainWindow, яке містить бічну навігаційну панель і область для відображення активної сторінки. Перемикання між сторінками виконується за допомогою прив’язки до поточної моделі представлення. Для цього використовується властивість CurrentViewModel у MainViewModel, а у XAML визначаються шаблони відображення для відповідних ViewModel.

Реалізація навігації побудована таким чином, що представлення не створюють інші сторінки напряму. Замість цього використовується сервіс навігації, який повідомляє головну модель представлення про необхідність перейти до певного розділу. Після цього MainViewModel отримує відповідну ViewModel із контейнера залежностей і встановлює її як поточну.

```
<ContentControl Grid.Row="0" Content="{Binding CurrentViewModel}" Margin="0">
    <ContentControl.Resources>
        <DataTemplate DataType="{x:Type vm:DashboardViewModel}">
            <views:DashboardView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type vm:DataSourceViewModel}">
            <views:DataSourcesView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type vm:ChangeLogViewModel}">
            <views:ChangeLogView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type vm:AutomationRulesViewModel}">
            <views:AutomationRulesView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type vm:ExecutionsViewModel}">
            <views:ExecutionsView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type vm:SettingsViewModel}">
            <views:SettingsView/>
        </DataTemplate>
    </ContentControl.Resources>
</ContentControl>
```

Такий підхід відповідає шаблону MVVM, оскільки логіка перемикання сторінок знаходиться не у code-behind файлах, а в моделях представлення та

сервісах. Це спрощує тестування й дозволяє змінювати інтерфейс без зміни основної логіки застосунку.

Для реалізації моделей представлення використовується бібліотека CommunityToolkit.Mvvm. Вона дозволяє скоротити кількість шаблонного коду завдяки атрибутам для автоматичної генерації властивостей і команд. Наприклад, властивості, що відображаються в інтерфейсі, позначаються атрибутом [ObservableProperty], а методи, які повинні викликатися з інтерфейсу, — атрибутом [RelayCommand].

Одним із ключових елементів реалізації є механізм отримання змін із цільової бази даних. Для цього було створено сервіс CdcPollingService, який відповідає за підключення до SQL Server, визначення CDC-таблиць, читання нових записів і формування внутрішніх об’єктів ChangeEvent та ChangeDetail.

Для роботи з CDC використовується ADO.NET, а не Entity Framework Core. Це пояснюється тим, що CDC-таблиці мають системний характер, розміщуються у схемі cdc та потребують виконання спеціальних SQL-запитів до системних таблиць і функцій SQL Server. Entity Framework Core використовується для роботи з внутрішньою базою застосунку, тоді як цільова база даних читається напряму через SqlConnection і SqlCommand.

```
await using var conn = new SqlConnection(connStr);
await conn.OpenAsync(ct);

// Get all CDC-enabled tables
var tables = await GetCdcTablesAsync(conn, ct);
var events = new List<ChangeEvent>();

foreach (var table in tables)
{
    var since = source.LastPolledAt ?? DateTime.UtcNow.AddMinutes(-5);
    var tableEvents = await ReadCdcChangesAsync(conn, source, table, since, ct);
    events.AddRange(tableEvents);
}

if (events.Count > 0)
{
    await changeRepo.AddRangeAsync(events);
    logger.LogInformation("Captured {Count} CDC events from {Source}", events.Count,
source.Name);
}

source.LastPolledAt = DateTime.UtcNow;
```

Під час опитування джерела даних сервіс формує рядок підключення на основі параметрів, збережених у таблиці DataSources. Після успішного

підключення визначаються таблиці, для яких увімкнено Change Data Capture. Далі виконується читання змін, що виникли після останнього успішного опитування. Отримані записи перетворюються у внутрішній формат застосунку та зберігаються у локальній базі даних.

Операції CDC у SQL Server мають числові коди. Наприклад, операція видалення має код 1, вставка — код 2, а оновлення може мати окремі записи для стану до та після зміни. Під час реалізації ці коди перетворюються у зрозумілі для користувача значення INSERT, UPDATE та DELETE, які потім відображаються у журналі змін і використовуються під час зіставлення подій із правилами автоматизації.

Фонову обробку подій реалізовано у класі BackgroundWorkerService, який наслідує базовий клас BackgroundService. Цей сервіс запускається разом із застосунком і виконує циклічну обробку у фоновому режимі. У кожному циклі він отримує поточні налаштування, список активних джерел даних, запускає опитування CDC-таблиць, обробляє нові події та виконує налаштовані дії.

```
await using var scope = scopeFactory.CreateAsyncScope();

var settingsRepo = scope.ServiceProvider.GetRequiredService<ISettingsRepository>();
var settings = await settingsRepo.GetAsync();
_pollingInterval = settings.CdcPollingIntervalSeconds;

// Poll all active data sources
var dataSourceRepo =
scope.ServiceProvider.GetRequiredService<IDataSourceRepository>();
var cdcService = scope.ServiceProvider.GetRequiredService<ICdcPollingService>();
var sources = await dataSourceRepo.GetActiveAsync();

foreach (var source in sources)
    await cdcService.PollAsync(source, stoppingToken);

// Process unhandled change events
var changeRepo = scope.ServiceProvider.GetRequiredService<IChangeEventRepository>();
var ruleRepo =
scope.ServiceProvider.GetRequiredService<IAutomationRuleRepository>();
var engine = scope.ServiceProvider.GetRequiredService<IActionExecutionEngine>();

var unprocessed = await changeRepo.GetUnprocessedAsync();
var processedIds = new List<long>();

foreach (var ev in unprocessed)
{
    var rules = await ruleRepo.GetMatchingRulesAsync(ev.TableName,
ev.OperationType);
    foreach (var rule in rules)
        await engine.ExecuteAsync(rule, ev, stoppingToken);
    processedIds.Add(ev.Id);
}

if (processedIds.Count > 0)
    await changeRepo.MarkProcessedAsync(processedIds);
```

```
// Clean up old records
var cutoff = DateTime.UtcNow.AddDays(-settings.LogRetentionDays);
await changeRepo.DeleteOlderThanAsync(cutoff);

var histRepo =
scope.ServiceProvider.GetRequiredService<IExecutionHistoryRepository>();
await histRepo.DeleteOlderThanAsync(cutoff);
```

Важливою особливістю реалізації фонового сервісу є використання окремої області видимості залежностей для кожного циклу обробки. Це дозволяє створювати нові екземпляри репозиторіїв і контексту бази даних на кожній ітерації та коректно звільняти ресурси після завершення циклу. Такий підхід запобігає накопиченню об’єктів у трекері змін Entity Framework Core і підвищує стабільність роботи застосунку під час тривалого виконання.

Після отримання нових подій фоновий сервіс звертається до репозиторію правил автоматизації та визначає, які правила відповідають конкретній події. Зіставлення виконується за назвою таблиці та типом операції. Правило вважається відповідним, якщо його таблиця збігається з таблицею події або має універсальне значення *, а тип події збігається з типом операції або має значення ALL.

Для виконання дій, пов’язаних із правилами, реалізовано сервіс ActionExecutionEngine. Він отримує правило та подію, після чого послідовно виконує всі дії правила відповідно до їх порядку. Підтримуються такі типи дій: HTTP GET-запит, HTTP POST-запит, запис у файл і запуск зовнішньої програми.

Перед виконанням дії сервіс виконує підстановку шаблонних змінних. У параметрах дій можуть використовуватися спеціальні маркери, наприклад {{EventId}}, {{Table}}, {{Operation}}, {{Database}}, {{Timestamp}} та {{PrimaryKey}}. Під час виконання вони замінюються фактичними значеннями поточної події. Це дозволяє формувати динамічні URL-адреси, тіла HTTP-запитів, текстові повідомлення для файлів і аргументи запуску зовнішніх програм.

HTTP-запити виконуються з використанням HttpClient. Для запобігання надмірному створенню мережевих підключень використовується спільний екземпляр HttpClient. Для кожної дії може задаватися тайм-аут виконання. Якщо

запит завершується помилкою або повертає невдалий статус, результат виконання фіксується як помилка.

Дія запису у файл передбачає створення директорії, якщо вона ще не існує, та додавання сформованого текстового рядка до файлу. Дія запуску зовнішньої програми реалізується через ProcessStartInfo. При цьому система передає задані аргументи, очікує завершення процесу та аналізує код завершення. Якщо код завершення відрізняється від нуля, дія вважається невдалою.

Після виконання кожної дії створюється запис в історії виконань. У ньому зберігаються ідентифікатор правила, ідентифікатор події, тип дії, статус виконання, тривалість виконання та повідомлення про помилку, якщо вона виникла. Для вимірювання тривалості використовується клас Stopwatch.

Доступ до внутрішньої бази даних реалізовано через Entity Framework Core та шаблон Repository. Усі основні операції з даними винесено в окремі репозиторії. Це дозволяє ViewModel та сервісам працювати з інтерфейсами, не використовуючи AppDbContext напряму. Такий підхід підвищує структурованість коду та спрощує його подальше тестування.

У репозиторії подій реалізовано методи для фільтрації журналу змін, отримання загальної кількості записів, отримання події разом із деталями, додавання групи подій, отримання необроблених записів, позначення подій як оброблених і видалення застарілих записів. Для операцій масового оновлення та видалення можуть використовуватися можливості Entity Framework Core, які дозволяють виконувати SQL-команди без попереднього завантаження всіх сутностей у пам’ять. Репозиторій правил автоматизації забезпечує отримання правил разом із діями. Це важливо, оскільки правило не має практичного сенсу без пов’язаних із ним дій. Метод пошуку відповідних правил використовується фоновим сервісом під час обробки подій. Репозиторій історії виконань забезпечує завантаження записів із можливістю фільтрації за назвою правила, статусом і часовим проміжком.

Для перевірки підключення до цільової бази даних реалізовано ConnectionTestService. Він формує рядок підключення, намагається відкрити

з’єднання з SQL Server і перевіряє, чи увімкнено механізм Change Data Capture для вказаної бази даних. Результат повертається у вигляді повідомлення, яке відображається користувачу в інтерфейсі.

```
    public async Task<(bool Success, string Message)> TestAsync(DataSource source)
    {
        var connStr = BuildConnectionString(source);
        try
        {
            await using var conn = new SqlConnection(connStr);
            await conn.OpenAsync();

            // Check if CDC is enabled on the database
            await using var cmd = new SqlCommand(
                "SELECT is_cdc_enabled FROM sys.databases WHERE name = DB_NAME()",
conn);
            var cdcEnabled = (bool?)await cmd.ExecuteScalarAsync();
            var cdcMsg = cdcEnabled == true ? "CDC is enabled." : "CDC is NOT
enabled on this database.";

            return (true, $"Connection successful. SQL Server {conn.ServerVersion}.
{cdcMsg}");
        }
        catch (Exception ex)
        {
            return (false, $"Connection failed: {ex.Message}");
        }
    }
```

Окрему увагу під час реалізації було приділено обробці помилок. Помилки підключення до бази даних, виконання HTTP-запитів, доступу до файлів або запуску зовнішніх програм не повинні призводити до аварійного завершення роботи застосунку. У разі виникнення помилки вона або відображається користувачу, або записується до історії виконань і журналу логування.

Для логування використовується бібліотека Serilog. Вона дозволяє записувати інформаційні повідомлення, попередження та помилки у файлові журнали. Логування особливо важливе для аналізу роботи фонового сервісу, оскільки значна частина обробки виконується автоматично без прямої участі користувача.

У результаті реалізації було створено десктопний застосунок, який поєднує графічний інтерфейс користувача, механізм отримання змін із SQL Server CDC, внутрішню базу даних, систему правил автоматичного реагування, історію виконань і налаштування. Реалізація виконана відповідно до розробленої архітектури та проєкту програмного забезпечення, що забезпечує узгодженість між вимогами, проєктними рішеннями та програмним кодом.

3 РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У результаті виконання кваліфікаційної роботи було розроблено десктопне програмне забезпечення ChangeTracker, призначене для відстеження змін в інформаційній системі на основі технології Change Data Capture. Розроблений застосунок забезпечує підключення до баз даних Microsoft SQL Server, отримання інформації про зміни в таблицях, збереження подій у внутрішній базі даних, перегляд журналу змін, налаштування правил автоматичної обробки подій та перегляд історії виконання дій.

Програмне забезпечення реалізовано як Windows desktop-застосунок із графічним інтерфейсом користувача. Інтерфейс побудовано за принципом розділення функціональності на окремі сторінки. У лівій частині головного вікна розташовано навігаційну панель, яка містить основні розділи: Dashboard, Data Sources, Change Log, Automation, Executions та Settings. У правій частині вікна відображається вміст вибраного розділу.

Після запуску застосунку користувач потрапляє на інформаційну панель Dashboard (рис. 3.1). Вона призначена для швидкого перегляду загальної статистики зафіксованих подій. На сторінці відображається загальна кількість подій, кількість подій за поточний день, кількість операцій вставки, оновлення та видалення. Також передбачено області для графічного відображення подій за часом, розподілу подій за типами операцій та переліку таблиць, у яких найчастіше відбувалися зміни.

Сторінка Data Sources (рис 3.2) використовується для керування джерелами даних. У цьому розділі користувач може переглядати список налаштованих підключень до баз даних, додавати нове джерело, редагувати існуюче, видаляти непотрібне підключення та виконувати перевірку з’єднання. Для кожного джерела даних зберігаються назва підключення, сервер, база даних, облікові дані та ознака активності. Якщо джерело даних активне, фоновий сервіс

враховує його під час чергового циклу опитування CDC-таблиць.

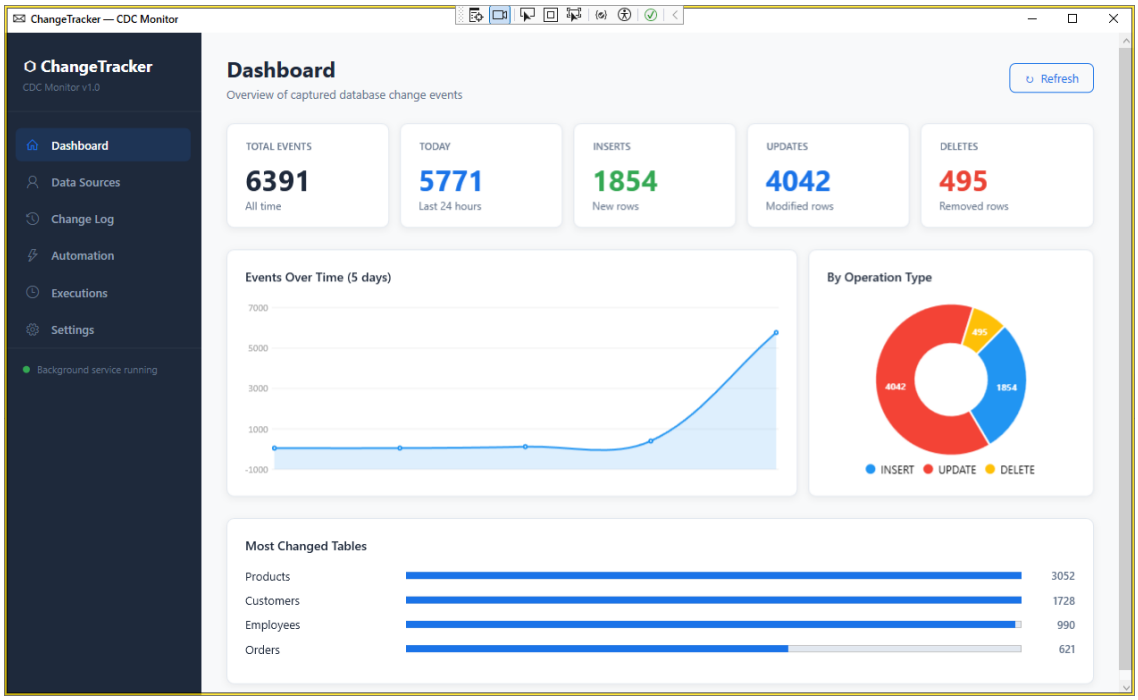


Рисунок 3.1 – Інформаційна панель програмного забезпечення ChangeTracker

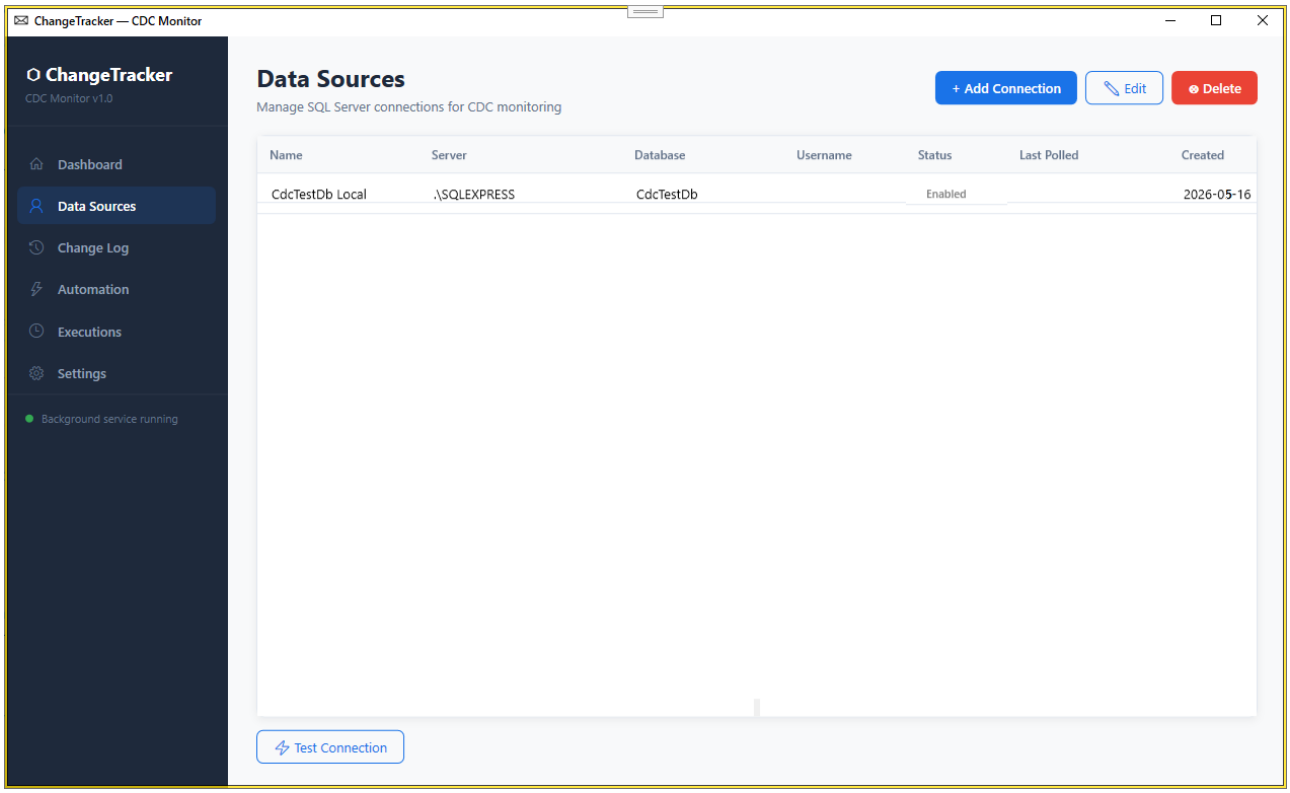


Рисунок 3.2 – Сторінка керування джерелами даних

Для створення або редагування підключення використовується окреме діалогове вікно (рис. 3.3). У ньому користувач задає параметри підключення до SQL Server та може перевірити коректність введених даних. Це дозволяє виявити помилки підключення ще до початку роботи механізму моніторингу.

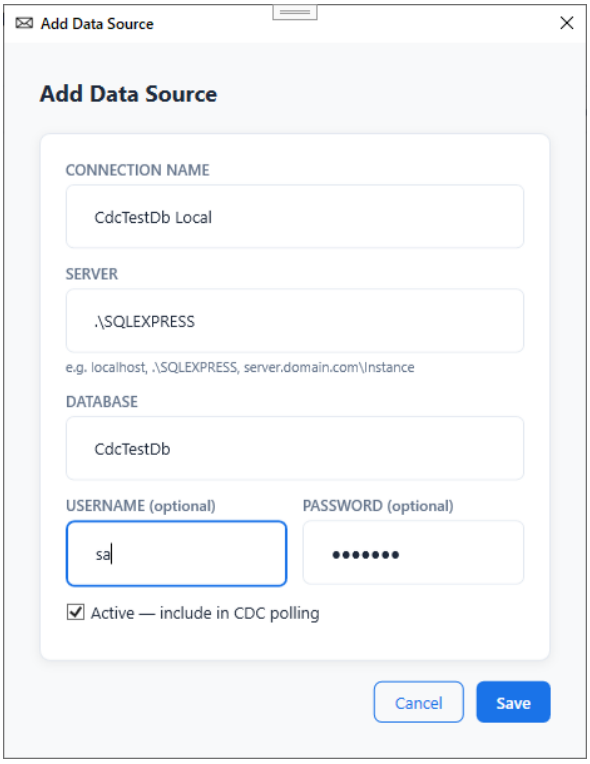


Рисунок 3.3 – Діалогове вікно створення джерела даних

Сторінка Change Log (рис. 3.4) призначена для перегляду журналу змін. У цьому розділі відображаються події, отримані з CDC-таблиць цільової бази даних. Для кожної події вказується дата та час, назва бази даних, назва таблиці, тип операції, значення первинного ключа та статус обробки. Користувач може фільтрувати записи журналу за часовим проміжком, назвою таблиці та типом операції. Це спрощує пошук потрібної інформації у випадку великої кількості зафіксованих змін.

Після вибору конкретної події користувач може переглянути її деталі. У панелі деталей відображається перелік змінених колонок, а також попередні та нові значення. Такий підхід дозволяє аналізувати не лише сам факт зміни запису, а й конкретний зміст цієї зміни.

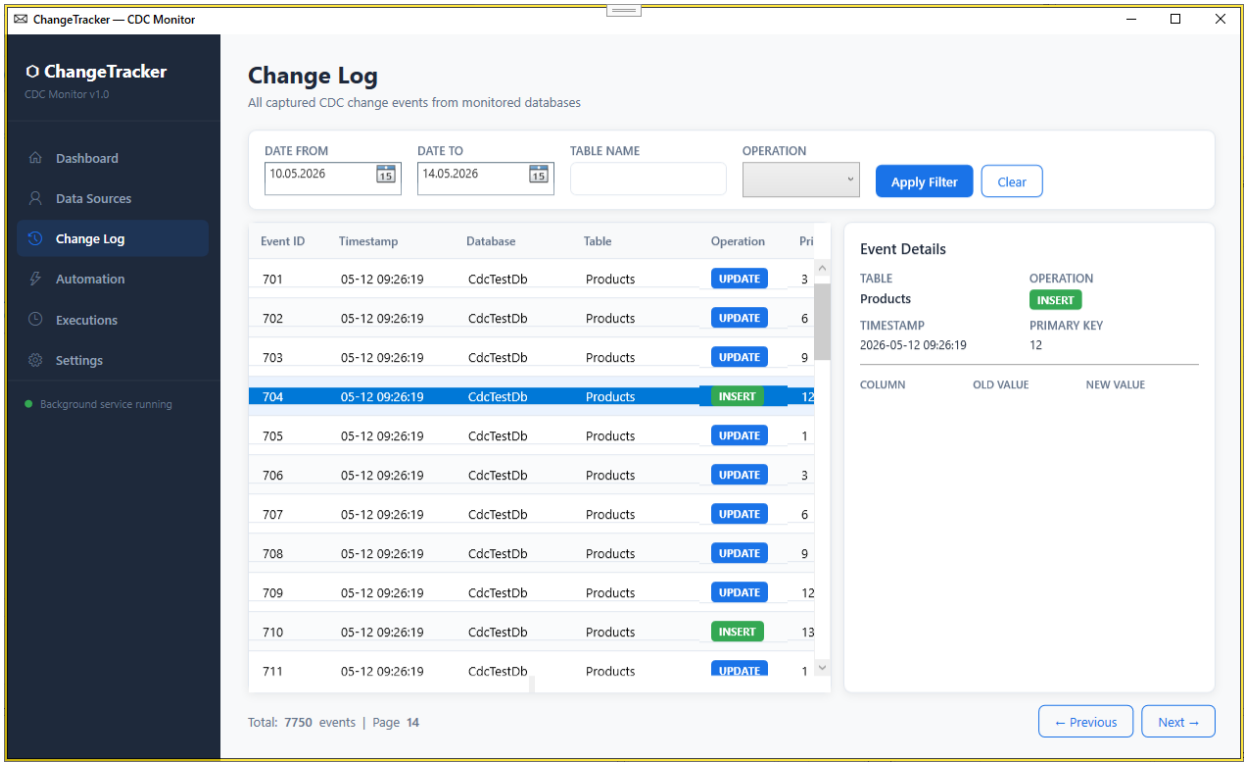


Рисунок 3.4 – Журнал змін інформаційної системи

Розділ Automation (рис 3.5) призначений для керування правилами автоматичної обробки подій. У ньому користувач може створювати нові (рис. 3.6) правила, редагувати існуючі, видаляти їх, а також вмикати або вимикати окремі правила без їх видалення. Кожне правило визначає таблицю, тип події та набір дій, які повинні бути виконані після появи відповідної зміни в базі даних.

Під час створення правила користувач може налаштувати одну або декілька дій. У програмному забезпеченні підтримуються такі типи дій: HTTP GET-запит, HTTP POST-запит, запис інформації до файлу та запуск зовнішньої програми. Для кожного типу дії передбачено відповідні параметри, зокрема URL-адресу, заголовки, тіло запиту, шлях до файлу, текстовий шаблон, шлях до виконуваного файлу та аргументи запуску.

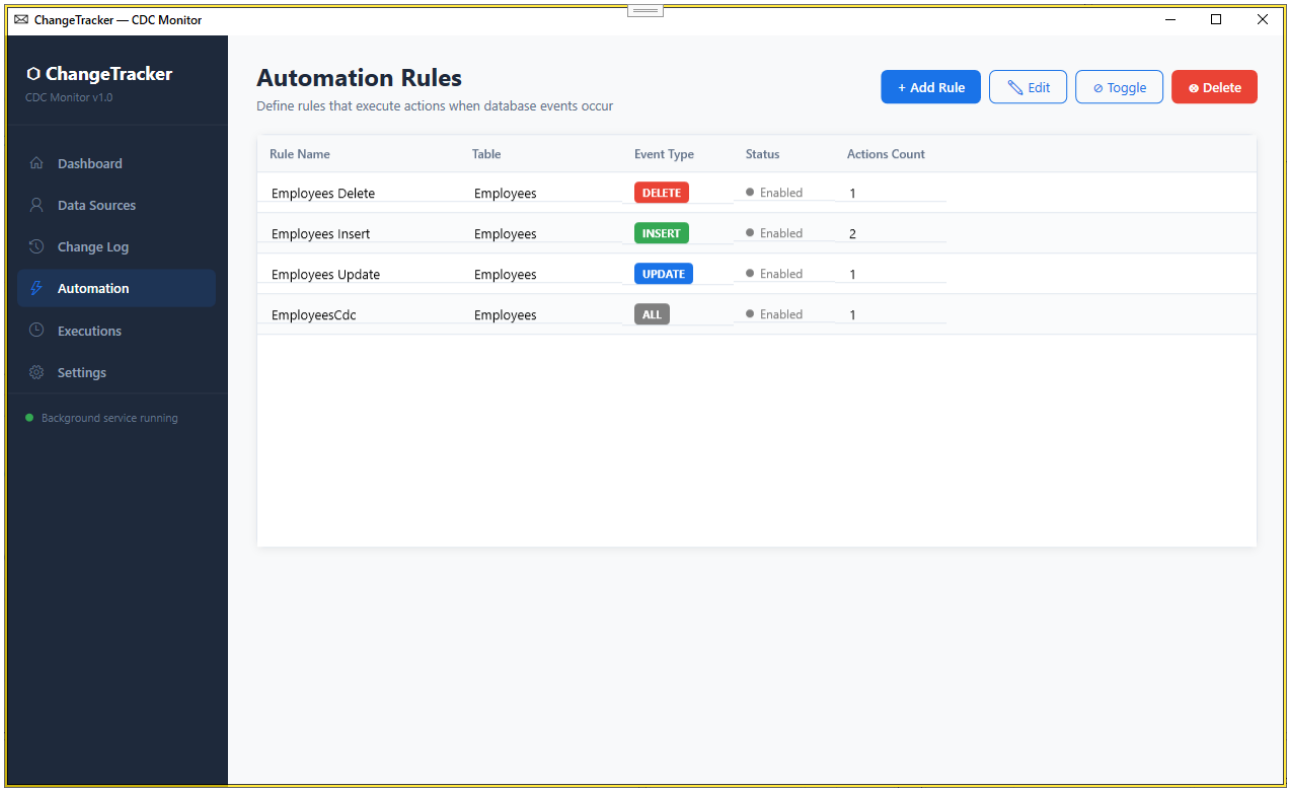


Рисунок 3.5 – Сторінка керування правилами автоматизації

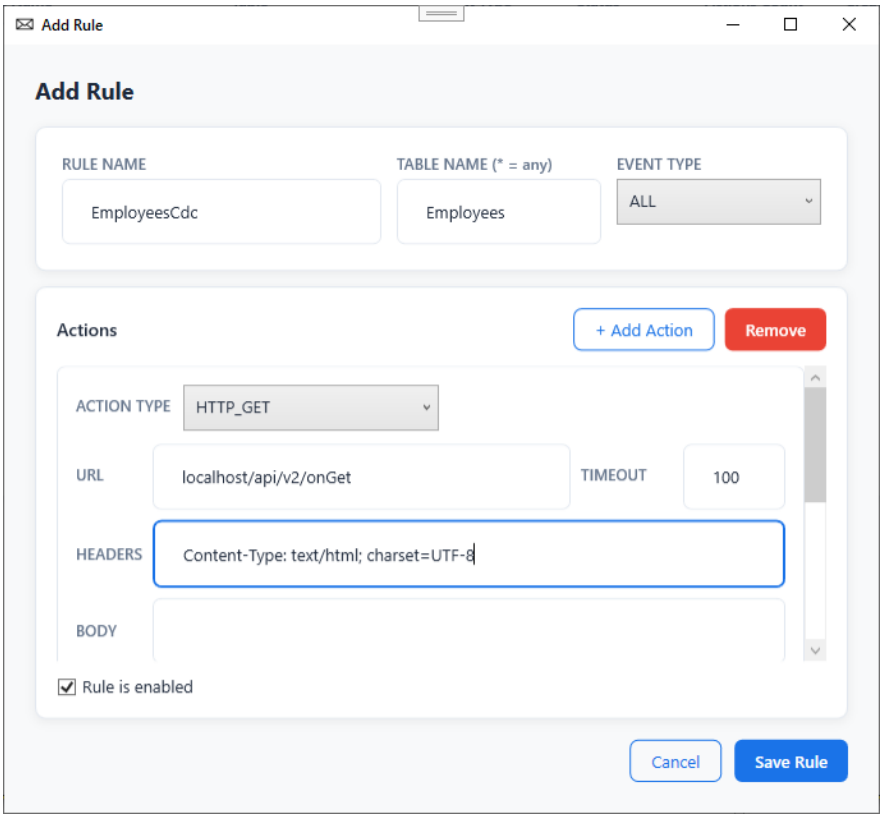


Рисунок 3.6 – Вікно створення правила автоматичної обробки подій

Розділ Executions (рис. 3.7) містить історію виконання автоматичних дій. У ньому користувач може переглянути, які правила були виконані, для яких подій вони спрацювали, який тип дії було виконано, скільки часу тривало виконання та чи завершилася дія успішно. У разі помилки в історії виконань зберігається відповідне повідомлення, що дозволяє швидко визначити причину невдалого виконання.

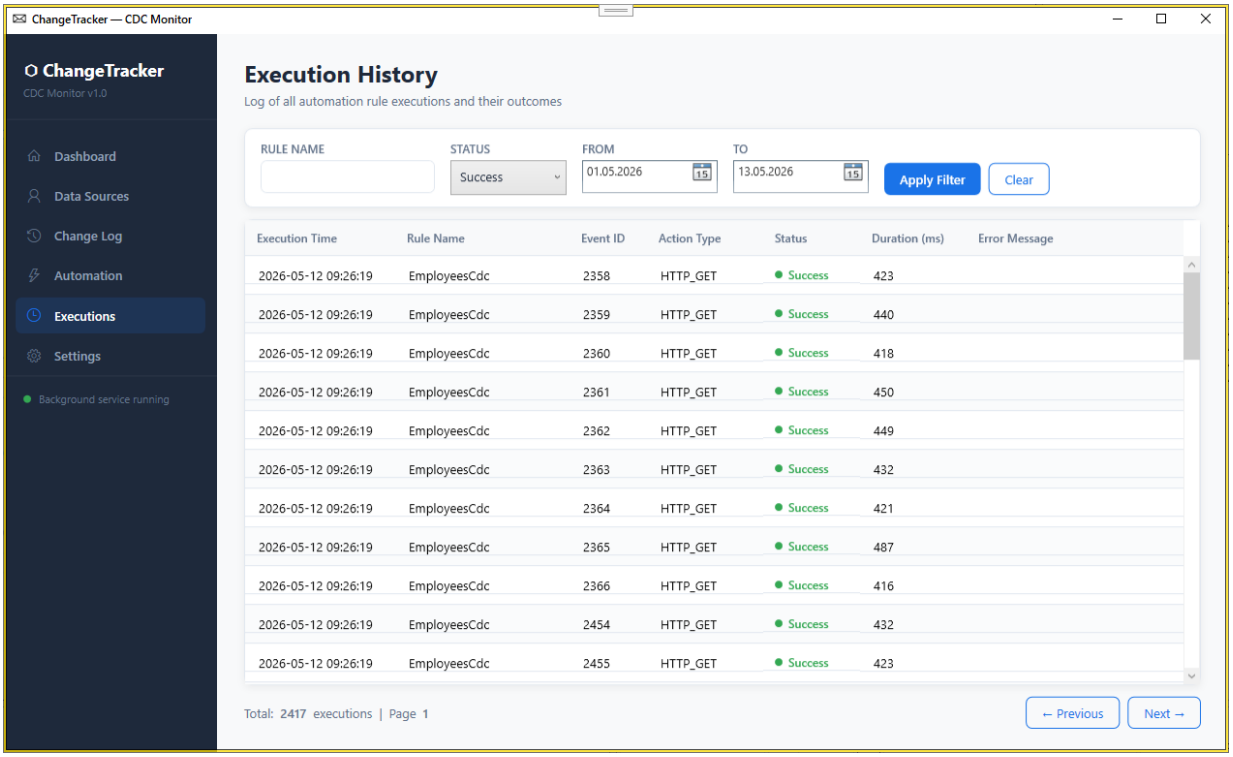


Рисунок 3.7 – Сторінка історії виконання автоматичних дій

Сторінка Settings (рис. 3.8) використовується для налаштування загальних параметрів роботи програмного забезпечення. У цьому розділі користувач може змінити інтервал опитування CDC-таблиць, термін зберігання журналів, рівень логування та директорію експорту. Збережені налаштування використовуються фоновим сервісом під час наступних циклів роботи.

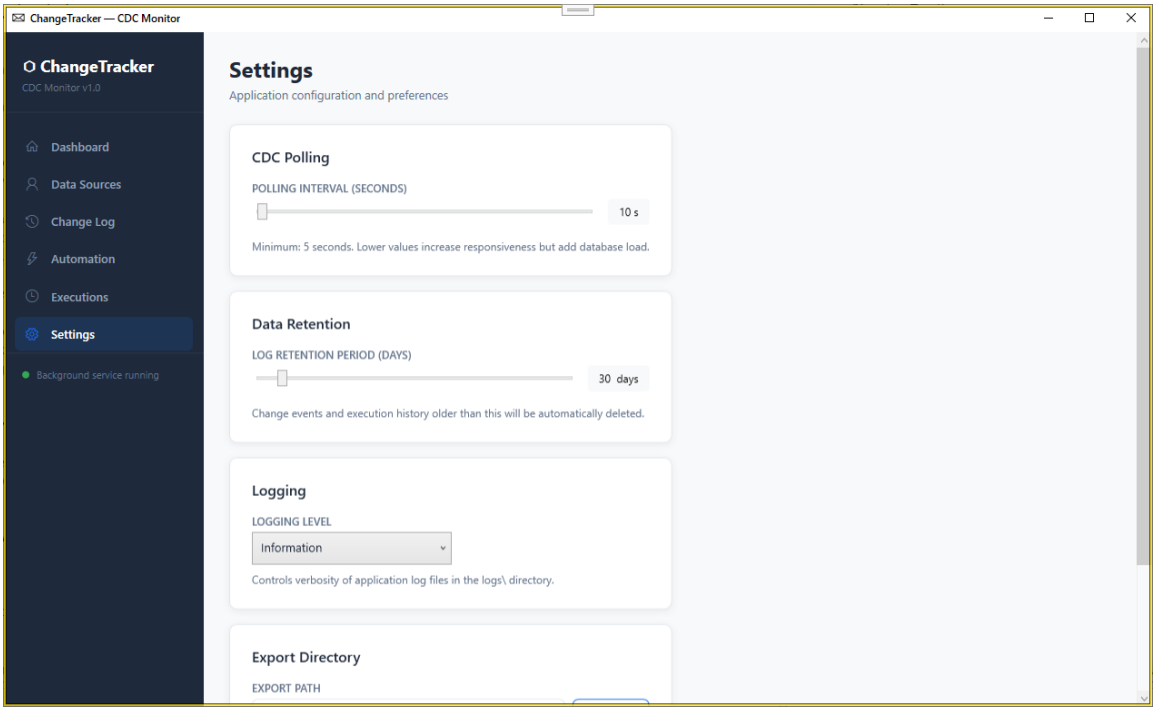


Рисунок 3.8 – Сторінка налаштувань програмного забезпечення

Розроблене програмне забезпечення забезпечує виконання основних функцій, визначених технічним завданням. Реалізовано керування джерелами даних, отримання змін із CDC-таблиць, збереження журналу подій, перегляд деталей змін, створення правил автоматичної обробки подій, виконання дій за правилами, збереження історії виконань та налаштування параметрів системи.

Під час розробки було реалізовано внутрішню базу даних застосунку, що містить таблиці для зберігання джерел даних, подій, деталей змін, правил автоматизації, дій правил, історії виконань і налаштувань. Також реалізовано фоновий сервіс, який автоматично виконує опитування CDC-таблиць, зберігає нові події, зіставляє їх із правилами та запускає відповідні дії.

Кількісні характеристики розробленого програмного забезпечення наведено нижче:

– кількість основних сторінок інтерфейсу — 6;

– кількість основних таблиць внутрішньої бази даних — 7;

– кількість підтримуваних типів автоматичних дій — 4;

– підтримувані типи подій бази даних — INSERT, UPDATE, DELETE;

– основна мова програмування — C#;

– платформа реалізації — .NET 8;

– технологія побудови інтерфейсу — WPF;

– система управління базами даних — Microsoft SQL Server.

Отримані результати підтверджують, що поставлену задачу розробки програмного забезпечення для відстеження змін в інформаційній системі було виконано. Розроблений застосунок дозволяє автоматизувати процес моніторингу змін у базі даних, забезпечує зручне відображення журналу подій та надає механізм автоматичного реагування на визначені користувачем події.

Технічне завдання на розробку програмного забезпечення для відстеження змін в інформаційній системі наведено в додатку А. Код програмного забезпечення представлено в додатку Б. Опис програмного забезпечення ChangeTracker наведено в додатку В. Для забезпечення зручності експлуатації програмного забезпечення було розроблено інструкцію користувача, яку наведено в додатку Г. Програму та методику випробувань програмного забезпечення представлено в додатку Д.

4 ОХОРОНА ПРАЦІ

4.1 Правові та організаційні основи охорони праці під час розробки програмного забезпечення

Охорона праці є важливою складовою організації трудової діяльності працівників у сфері інформаційних технологій та розробки програмного забезпечення. Незважаючи на те, що робота програміста не пов'язана з виконанням фізично важких або небезпечних виробничих операцій, тривале використання комп'ютерної техніки та перебування в робочому середовищі можуть негативно впливати на стан здоров'я працівника. Тому створення безпечних та комфортних умов праці є необхідною умовою ефективного виконання робіт із проєктування, розробки та тестування програмного забезпечення.

Основним нормативно-правовим документом, який регулює питання охорони праці в Україні, є Закон України «Про охорону праці» [12]. Відповідно до цього закону охорона праці визначається як система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних та лікувально-профілактичних заходів і засобів, спрямованих на збереження життя, здоров'я та працездатності людини у процесі трудової діяльності.

Відповідно до чинного законодавства роботодавець зобов'язаний забезпечити належні умови праці, створити безпечне робоче середовище, організувати навчання працівників з питань охорони праці та здійснювати контроль за дотриманням встановлених вимог безпеки. Працівник, у свою чергу, повинен дотримуватися вимог нормативних документів з охорони праці, правильно використовувати технічні засоби та виконувати правила безпечної роботи з комп'ютерним обладнанням.

Під час розробки програмного забезпечення основним засобом праці є персональний комп'ютер. У зв'язку з цим особливого значення набувають

вимоги щодо організації робочого місця користувача комп’ютерної техніки. Робоче місце повинно забезпечувати зручне розташування обладнання, достатній рівень освітлення, відповідний мікроклімат приміщення та можливість підтримання правильної робочої пози працівника протягом робочого дня.

Під час виконання робіт із розробки програмного забезпечення для відстеження змін в інформаційній системі працівник здійснює проєктування баз даних, написання програмного коду, налагодження програмних компонентів, тестування функціональності та аналіз результатів роботи системи. Значна частина робочого часу проходить за комп’ютером, що може призводити до перевтоми органів зору, підвищеного статичного навантаження на опорно-руховий апарат та психоемоційного напруження. Саме тому важливим завданням охорони праці є мінімізація впливу зазначених факторів шляхом дотримання санітарно-гігієнічних норм та вимог ергономіки.

Організація безпечних умов праці передбачає не лише забезпечення відповідного технічного оснащення робочого місця, а й дотримання встановленого режиму праці та відпочинку. Регламентовані перерви під час роботи за комп’ютером сприяють зменшенню навантаження на зір, покращують працездатність та дозволяють знизити ризик виникнення професійних захворювань.

Таким чином, правові та організаційні основи охорони праці під час розробки програмного забезпечення спрямовані на створення безпечних і комфортних умов роботи, збереження здоров’я працівників та забезпечення належного рівня продуктивності праці. Дотримання вимог чинного законодавства та нормативних документів з охорони праці є обов’язковою умовою під час виконання робіт із розробки програмного забезпечення для відстеження змін в інформаційній системі.

4.2 Аналіз умов праці розробника програмного забезпечення

Розробка програмного забезпечення для відстеження змін в інформаційній

системі виконується з використанням персонального комп’ютера у приміщенні офісного типу. Основними видами діяльності розробника є проєктування архітектури програмного забезпечення, розробка програмного коду, робота з базами даних, тестування функціональності та аналіз результатів роботи програмної системи. Значна частина робочого часу пов’язана з безперервною роботою за комп’ютером, що потребує врахування вимог охорони праці та виробничої санітарії.

Відповідно до Закону України «Про охорону праці» роботодавець зобов’язаний забезпечити працівникам безпечні та нешкідливі умови праці, а також створити належні умови для збереження їх здоров’я та працездатності. Під час організації робочого місця користувача персонального комп’ютера необхідно враховувати вимоги державних санітарних норм та правил, а також рекомендації щодо ергономічного облаштування робочого середовища.

Основним обладнанням робочого місця розробника програмного забезпечення є персональний комп’ютер, монітор, клавіатура, маніпулятор типу «миша», робочий стіл та крісло. Від правильності їх розташування залежить рівень комфорту працівника та ефективність виконання професійних обов’язків.

Під час виконання робіт із розробки програмного забезпечення на працівника можуть впливати такі шкідливі виробничі фактори:

– підвищене навантаження на органи зору внаслідок тривалої роботи з монітором;

– статичне навантаження на м’язи шиї, спини та плечового поясу;

– нервово-емоційне напруження, пов’язане з аналізом інформації та вирішенням складних логічних задач;

– недостатній або надмірний рівень освітлення робочого місця;

– можливий вплив шуму від комп’ютерного обладнання та допоміжних технічних засобів;

– ризик ураження електричним струмом у разі несправності електрообладнання.

Одним із найбільш суттєвих факторів є навантаження на органи зору. Під

час розробки програмного забезпечення працівник протягом тривалого часу концентрує увагу на текстовій інформації, програмному коді та елементах графічного інтерфейсу. Це може призводити до зорової втоми, сухості очей та зниження працездатності. Для зменшення негативного впливу необхідно забезпечити достатній рівень освітлення робочого місця та дотримуватися рекомендованого режиму праці та відпочинку.

Іншим важливим фактором є статичне навантаження на опорно-руховий апарат. Тривале перебування у сидячому положенні може спричиняти дискомфорт у ділянці шиї, спини та попереку. Для зниження ризику виникнення професійних захворювань необхідно використовувати ергономічні меблі та підтримувати правильну робочу позу.

Мікроклімат приміщення також впливає на самопочуття та працездатність працівника. Відповідно до вимог санітарних норм температура повітря у приміщенні повинна перебувати в межах 20–24 °C, відносна вологість повітря — 40–60 %, а швидкість руху повітря не повинна перевищувати встановлених нормативних значень. Дотримання зазначених параметрів забезпечує комфортні умови праці та сприяє підтриманню високої продуктивності праці.

Під час експлуатації комп’ютерної техніки необхідно також враховувати вимоги електробезпеки. Усе обладнання повинно бути технічно справним, підключеним через справні електромережі та захисні пристрої. Працівник повинен дотримуватися правил безпечної експлуатації електрообладнання та негайно повідомляти про виявлені несправності.

Отже, основними факторами, що впливають на умови праці розробника програмного забезпечення, є навантаження на органи зору, статичне навантаження на опорно-руховий апарат, психоемоційне напруження, параметри мікроклімату приміщення та електробезпека. Дотримання вимог законодавства України з охорони праці, санітарно-гігієнічних норм та правил організації робочого місця дозволяє мінімізувати негативний вплив зазначених факторів і забезпечити безпечні умови праці під час розробки програмного забезпечення для відстеження змін в інформаційній системі.

4.3 Заходи щодо забезпечення безпечних умов праці під час розробки програмного забезпечення

Під час розробки програмного забезпечення для відстеження змін в інформаційній системі працівник більшу частину робочого часу проводить за персональним комп’ютером. Такий вид діяльності належить до робіт із високим рівнем інтелектуального навантаження та тривалим використанням засобів обчислювальної техніки. Для забезпечення безпечних і комфортних умов праці необхідно реалізувати комплекс організаційних, технічних та санітарно-гігієнічних заходів, спрямованих на мінімізацію впливу шкідливих виробничих факторів.

Одним із найважливіших заходів є правильна організація робочого місця користувача персонального комп’ютера. Робочий стіл повинен забезпечувати достатній простір для розміщення монітора, клавіатури, маніпулятора типу «миша» та іншого необхідного обладнання. Поверхня столу має бути матовою, щоб запобігати появі відблисків та надмірному навантаженню на органи зору. Висота робочого столу повинна забезпечувати комфортне положення рук під час роботи.

Особливу увагу необхідно приділити вибору робочого крісла. Крісло повинно мати регулювання висоти сидіння та кута нахилу спинки, що дозволяє адаптувати його до антропометричних характеристик працівника. Під час роботи спина повинна спиратися на спинку крісла, а стопи ніг мають повністю розташовуватися на підлозі або спеціальній підставці. Таке положення сприяє зниженню навантаження на хребет та запобігає виникненню професійних захворювань опорно-рухового апарату.

Відповідно до рекомендацій ДСанПіН 3.3.2.007-98 [13] відстань від очей користувача до екрана монітора повинна становити приблизно 50–70 см. Верхня межа екрана повинна знаходитися на рівні очей або трохи нижче. Така організація робочого місця дозволяє зменшити навантаження на м’язи шиї та забезпечити комфортне сприйняття інформації з екрана.

Важливим фактором безпечної роботи є забезпечення належного освітлення приміщення. Недостатнє освітлення призводить до швидкої втоми органів зору, тоді як надмірне освітлення або наявність відблисків на поверхні монітора можуть викликати дискомфорт та погіршення концентрації уваги. Згідно з вимогами ДБН В.2.5-28:2018 [14] робочі приміщення повинні бути забезпечені природним та штучним освітленням. Для освітлення робочого місця доцільно використовувати світлодіодні світильники, які забезпечують рівномірний розподіл світла та характеризуються низьким енергоспоживанням.

Для запобігання негативному впливу на органи зору рекомендується застосовувати сучасні монітори з високою роздільною здатністю та технологіями зниження мерехтіння екрана. Крім того, необхідно налаштовувати оптимальні параметри яскравості та контрастності відповідно до умов освітлення приміщення. Під час тривалої роботи за комп’ютером доцільно використовувати правило «20-20-20», згідно з яким кожні 20 хвилин необхідно переводити погляд на об’єкт, розташований на відстані близько 20 футів (приблизно 6 метрів), протягом 20 секунд.

Для зменшення статичного навантаження на опорно-руховий апарат необхідно організувати раціональний режим праці та відпочинку. Тривале перебування у сидячому положенні може спричиняти втому, дискомфорт та розвиток професійних захворювань. Тому рекомендується робити короткі перерви кожні 45–60 хвилин роботи за комп’ютером. Під час перерв доцільно виконувати нескладні фізичні вправи для м’язів спини, шиї та плечового поясу, а також змінювати положення тіла.

Особливого значення набуває забезпечення оптимального мікроклімату в робочому приміщенні. Згідно з санітарними нормами температура повітря повинна перебувати в межах 20–24 °C, а відносна вологість — 40–60 %. Для підтримання зазначених параметрів можуть використовуватися системи вентиляції, кондиціонування та зволоження повітря. Комфортний мікроклімат позитивно впливає на працездатність працівника та сприяє зниженню рівня

втоми.

Під час експлуатації комп’ютерної техніки необхідно дотримуватися вимог електробезпеки. Усе обладнання повинно бути підключене до справної електромережі та відповідати вимогам технічної безпеки. Перед початком роботи необхідно перевіряти цілісність кабелів живлення та відсутність механічних пошкоджень обладнання. Забороняється використовувати несправні електричні пристрої або самостійно виконувати їх ремонт без відповідної кваліфікації.

Для захисту обладнання та працівника від можливих аварійних ситуацій доцільно використовувати джерела безперебійного живлення та мережеві фільтри. Це дозволяє зменшити ризик пошкодження комп’ютерної техніки внаслідок перепадів напруги та втрати даних під час аварійного відключення електроенергії.

Окрему увагу слід приділити заходам пожежної безпеки. Приміщення повинно бути обладнане первинними засобами пожежогасіння, зокрема вуглекислотними або порошковими вогнегасниками. Робочі місця повинні бути організовані таким чином, щоб забезпечити вільний доступ до евакуаційних виходів. Працівники повинні бути ознайомлені з правилами поведінки у разі виникнення пожежі або іншої надзвичайної ситуації.

Важливим напрямом забезпечення безпечних умов праці є зниження рівня психоемоційного навантаження. Розробка програмного забезпечення потребує високої концентрації уваги, аналізу великих обсягів інформації та прийняття складних технічних рішень. Для підтримання працездатності рекомендується дотримуватися раціонального режиму праці, планувати робочі завдання, уникати надмірного перевантаження та забезпечувати достатній час для відпочинку.

Таким чином, забезпечення безпечних умов праці під час розробки програмного забезпечення для відстеження змін в інформаційній системі потребує комплексного підходу, який включає ергономічну організацію робочого місця, дотримання вимог освітлення та мікроклімату, забезпечення

електричної та пожежної безпеки, а також організацію раціонального режиму праці та відпочинку. Реалізація зазначених заходів дозволяє мінімізувати вплив шкідливих факторів виробничого середовища, зберегти здоров’я працівників та підвищити ефективність їх професійної діяльності.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено програмне забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture.

Розроблене програмне забезпечення ChangeTracker реалізовано як десктопний застосунок для операційної системи Windows, що забезпечує підключення до баз даних Microsoft SQL Server, отримання інформації про зміни в таблицях, збереження подій у внутрішній базі даних, перегляд журналу змін та виконання автоматичних дій відповідно до налаштованих користувачем правил.

Під час виконання кваліфікаційної роботи, було виконано наступні завдання:

– виконано аналіз практичної задачі інженерії програмного забезпечення щодо відстеження змін в інформаційній системі;

– проаналізовано існуючі програмні рішення та обґрунтовано доцільність розробки власного програмного забезпечення;

– сформульовано постановку задачі на розробку програмного забезпечення для відстеження змін в інформаційній системі;

– проведено аналіз вимог до програмного забезпечення для відстеження змін в інформаційній системі;

– розроблено архітектурні рішення програмного забезпечення;

– виконано проєктування бази даних та користувацького інтерфейсу;

– реалізовано механізм отримання змін за допомогою технології Change Data Capture;

– реалізовано механізм автоматичного виконання дій на основі налаштованих правил обробки подій;

– проведено тестування та випробування розробленого програмного забезпечення;

– підготовлено необхідну програмну та технічну документацію.

Під час проєктування було визначено основні сутності предметної області: джерело даних, подію зміни, деталі зміни, правило автоматизації, дію правила, історію виконання та налаштування системи. Доступ до внутрішньої бази даних реалізовано з використанням Entity Framework Core і шаблону Repository.

У межах реалізації програмного забезпечення було створено графічний інтерфейс користувача з основними розділами Dashboard, Data Sources, Change Log, Automation, Executions та Settings.

Після реалізації було проведено тестування програмного забезпечення. Перевірено основні функціональні можливості: додавання, редагування та видалення джерел даних, перевірку підключення до SQL Server, подій із CDC-таблиць, перегляд журналу змін, фільтрацію подій, перегляд деталей зміни, створення правил автоматизації, виконання HTTP-дій, запису у файл, запуску зовнішнього процесу, перегляд історії виконань і зміну налаштувань системи. Результати тестування підтвердили працездатність застосунку та відповідність основним функціональним вимогам.

Отже, мету кваліфікаційної роботи досягнуто. У результаті було створено програмне забезпечення ChangeTracker, яке дозволяє автоматизувати процес відстеження змін в інформаційній системі та забезпечує можливість реагування на події бази даних без внесення змін до основної прикладної логіки інформаційної системи. Розроблений застосунок може бути використаний для аудиту змін, моніторингу активності в базах даних, інтеграції з зовнішніми сервісами та автоматизації типових дій, що виконуються після зміни даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is change data capture (CDC)? URL: https://learn.microsoft.com/en-gb/sql/relational-databases/track-changes/about-change-data-capture-sql-server?view=sql-server-ver17 (дата звернення: 15.03.2026).

2. SQL SERVER – Database Auditing and Compliance. URL: https://blog.sqlauthority.com/2016/07/12/sql-server-database-auditing-compliance/ (дата звернення: 15.03.2026).

3. Debezium notifications. URL: https://debezium.io/documentation/reference/3.6/configuration/notification.html (дата звернення: 12.03.2026).

4. GoldenGate Features. URL: https://www.oracle.com/integration/goldengate/features/ (дата звернення: 12.03.2026).

5. Apache Kafka Introduction. URL: https://kafka.apache.org/43/getting-started/introduction/ (дата звернення: 14.03.2026).

6. What is Kafka Connect? URL: https://dev.to/dunithdanushka/what-is-kafka-connect-3eae (дата звернення: 14.03.2026).

7. Microsoft SQL Server 2012 Bible / A. Jorgensen та ін. New Jersey : John Wiley & Sons, 2012. 1416 p.

8. Entity Framework Core. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 16.03.2026).

9. Windows Presentation Foundation Overview. URL: https://learn.microsoft.com/en-gb/dotnet/desktop/wpf/overview/?view=netframeworkdesktop-4.8 (дата звернення: 16.03.2026).

10. Model-View-ViewModel (MVVM). URL: https://learn.microsoft.com/en-gb/dotnet/architecture/maui/mvvm (дата звернення: 18.03.2026).

11. XAML overview. URL: https://learn.microsoft.com/en-

gb/dotnet/desktop/wpf/xaml/?view=netframeworkdesktop-4.8 (дата звернення: 22.03.2026).

12. Про охорону праці : Закон України від 14 жовтня 1992 року № 2694-XII. *Відомості Верховної Ради України*. 1992. № 49. Ст. 668.

13. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин : ДСанПіН 3.3.2.007-98 : затв. Постановою Головного державного санітарного лікаря України від 10.12.1998 р. № 7.

14. ДБН В.2.5-28:2018. Природне і штучне освітлення : [Чинний від 2019-03-01]. Київ : Мінрегіон України, 2018. 133 с.

ДОДАТОК А - Технічне завдання на розробку програмного забезпечення для відстеження змін в інформаційній системі

Вступ

Повна назва проєкту: «Розробка програмного забезпечення для відстеження змін в інформаційній системі». Сферою застосування програмного забезпечення є інформаційні системи, що використовують бази даних Microsoft SQL Server та потребують моніторингу змін даних, ведення журналу подій і автоматичного виконання дій у відповідь на визначені зміни.

1 Підстави для розробки

Підставою для розробки даного програмного забезпечення є наказ на затвердження тем кваліфікаційних робіт бакалаврів зі спеціальності 121 «Інженерія програмного забезпечення» в Національному університеті кораблебудування імені адмірала Макарова.

2 Призначення розробки

2.1 Експлуатаційне призначення

Експлуатаційне призначення програмного забезпечення для відстеження змін в інформаційній системі полягає в автоматизації процесів моніторингу змін у базі даних, ведення журналу подій та виконання налаштованих користувачем дій у відповідь на виникнення певних подій.

Програмне забезпечення призначене для використання технічними спеціалістами, адміністраторами баз даних, системними адміністраторами або відповідальними особами, які здійснюють контроль за станом інформаційної системи. Використання програмного забезпечення дозволяє зменшити обсяг ручної роботи під час аналізу змін у базі даних, підвищити оперативність

реагування на події та забезпечити централізоване зберігання інформації про зміни.

Застосунок може використовуватися в навчальних, дослідницьких або прикладних інформаційних системах, де необхідно відстежувати операції додавання, редагування та видалення записів у таблицях бази даних. Програмне забезпечення орієнтоване на роботу в середовищі Windows та реалізується як десктопний застосунок мовою програмування C#.

2.2 Функціональне призначення

Функціональне призначення програмного забезпечення полягає у забезпеченні користувача засобами для підключення до бази даних, отримання інформації про зміни за допомогою технології Change Data Capture, перегляду журналу змін, налаштування правил автоматичного реагування та аналізу результатів виконання таких правил.

Програмне забезпечення повинно забезпечувати виконання таких основних функцій:

– можливість додавати, корегувати, вилучати та переглядати інформацію про джерела даних, тобто наявність дій CRUD для даної сутності;

– можливість додавати, корегувати, вилучати та переглядати інформацію про правила автоматичної обробки подій, тобто наявність дій CRUD для даної сутності;

– можливість додавати, корегувати, вилучати та переглядати інформацію про дії, що виконуються при спрацюванні правил, тобто наявність дій CRUD для даної сутності;

– автоматичне отримання інформації про зміни даних із використанням технології Change Data Capture;

– перегляд журналу змін бази даних;

– перегляд детальної інформації про зміни записів;

– пошук та фільтрація подій за таблицею, типом операції та часовим проміжком;

– автоматичне виконання дій при виникненні визначених подій;

– перегляд історії виконання автоматичних дій;

– формування звітів про зміни в інформаційній системі.

3 Вимоги до програмного забезпечення

3.1 Вимоги до функціональних характеристик

Програмним забезпеченням буде користуватися технічний спеціаліст або адміністратор інформаційної системи. Основним сценарієм роботи користувача є налаштування підключення до бази даних, вибір таблиць для моніторингу, перегляд журналу змін і створення правил автоматичного реагування на події.

Для джерела даних повинні бути доступні такі операції:

– додавання джерела даних;

Вхідні дані: назва підключення, адреса сервера, назва бази даних, логін, пароль, ознака активності.

Результат: створення нового запису про джерело даних у базі даних застосунку.

– корегування джерела даних;

Вхідні дані: обраний запис джерела даних та нові значення параметрів підключення.

Результат: оновлення інформації про джерело даних.

– вилучення джерела даних;

Вхідні дані: обраний запис джерела даних та підтвердження дії.

Результат: вилучення запису про джерело даних.

– перегляд джерел даних;

Вхідні дані: відкриття вкладки «Джерела даних».

Результат: відображення списку налаштованих джерел даних у табличному вигляді.

Для журналу змін повинні бути доступні такі операції:

– перегляд списку змін;

Вхідні дані: відкриття вкладки «Журнал змін».

Результат: відображення списку подій із зазначенням часу, таблиці, типу операції та ідентифікатора запису.

– фільтрація журналу змін;

Вхідні дані: назва таблиці, тип операції, початкова та кінцева дата.

Результат: відображення списку подій, які відповідають заданим параметрам фільтрації.

– перегляд деталей зміни;

Вхідні дані: вибір запису в журналі змін.

Результат: відображення старих та нових значень полів зміненого запису.

Для правил автоматичної обробки подій повинні бути доступні такі операції:

– додавання правила;

Вхідні дані: назва правила, назва таблиці, тип події, ознака активності, параметри дії.

Результат: створення нового правила автоматичної обробки події.

– корегування правила;

Вхідні дані: обране правило та нові параметри.

Результат: оновлення параметрів правила.

– вилучення правила;

Вхідні дані: обране правило та підтвердження дії.

Результат: вилучення правила з бази даних застосунку.

– перегляд правил;

Вхідні дані: відкриття вкладки «Правила».

Результат: відображення списку налаштованих правил.

Для дій, які виконуються при спрацюванні правила, повинні підтримуватися такі типи:

– HTTP GET-запит;

– HTTP POST-запит;

– запис у файл;

– запуск зовнішньої програми.

Для HTTP-запиту вхідними даними є адреса API, метод запиту, заголовки, тіло запиту та тайм-аут виконання. Результатом є виконаний HTTP-запит і запис про результат його виконання в історії.

Для запису у файл вхідними даними є шлях до файлу та текстовий шаблон повідомлення. Результатом є додавання інформації про подію до вказаного файлу.

Для запуску зовнішньої програми вхідними даними є шлях до виконуваного файлу та аргументи запуску. Результатом є запуск програми або команди та збереження результату виконання в історії.

3.1.2 Вимоги до організації вхідних та вихідних даних

Вхідні дані програмного забезпечення вводяться користувачем через графічний інтерфейс десктопного застосунку. Для введення інформації повинні використовуватися текстові поля, випадаючі списки, перемикачі, прапорці, таблиці та діалогові вікна.

До основних вхідних даних належать:

– параметри підключення до бази даних;

– назви таблиць, зміни в яких необхідно відстежувати;

– типи подій, що підлягають моніторингу;

– параметри правил автоматичної обробки подій;

– параметри HTTP-запитів;

– шлях до файлу для запису подій;

– шлях до зовнішньої програми або команди;

– параметри фільтрації журналу змін;

– параметри формування звітів.

Частина вхідних даних повинна автоматично отримуватися із бази даних.

До таких даних належать інформація про зафіксовані зміни, назви таблиць, типи операцій, значення первинних ключів, старі та нові значення полів.

Вихідні дані повинні відображатися у графічному інтерфейсі користувача у вигляді таблиць, форм, повідомлень і діалогових вікон. Основними вихідними даними є:

– список джерел даних;

– список зафіксованих змін;

– детальна інформація про зміну;

– список правил автоматичної обробки подій;

– список дій, пов’язаних із правилами;

– історія виконання дій;

– повідомлення про успішне виконання операцій;

– повідомлення про помилки.

Усі службові дані застосунку повинні зберігатися у локальній або серверній базі даних. До таких даних належать налаштування підключень, журнал змін, правила автоматичної обробки подій, дії та історія їх виконання.

3.2 Вимоги до надійності

Програмне забезпечення повинно забезпечувати стабільну роботу під час виконання основних операцій, зокрема підключення до бази даних, отримання інформації про зміни, відображення журналу подій та виконання налаштованих дій.

Надійність програмного забезпечення повинна забезпечуватися такими засобами:

– перевіркою коректності введених користувачем даних;

– обробкою помилок підключення до бази даних;

– обробкою помилок виконання HTTP-запитів;

– обробкою помилок доступу до файлів;

– обробкою помилок запуску зовнішніх програм;

– збереженням інформації про помилки в журналі виконання;

– використанням асинхронного виконання тривалих операцій;

– запобіганням аварійному завершенню роботи застосунку при виникненні помилок;

– підтвердженням дій користувача перед вилученням записів.

У разі неможливості підключення до бази даних програмне забезпечення повинно повідомляти користувача про помилку та не завершувати роботу аварійно. У разі помилки виконання автоматичної дії інформація про помилку повинна бути збережена в історії виконання.

Програмне забезпечення повинно забезпечувати збереження даних журналу змін і результатів виконання правил для подальшого перегляду та аналізу.

3.3 Умови експлуатації

Програмне забезпечення призначене для експлуатації на персональному комп’ютері або ноутбуці під керуванням операційної системи Windows. Робота із застосунком виконується в офісних або навчальних умовах користувачем, який має базові навички роботи з комп’ютером та розуміння принципів роботи з базами даних.

Для коректної роботи програмного забезпечення необхідна наявність доступу до бази даних Microsoft SQL Server, у якій увімкнено або може бути налаштовано механізм Change Data Capture. Також необхідна наявність мережевого доступу до зовнішніх API, якщо користувач налаштовує правила з виконанням HTTP-запитів.

Робота програмного забезпечення повинна здійснюватися в нормальних умовах експлуатації комп’ютерної техніки. Необхідно забезпечити справність електроживлення, стабільне мережеве підключення та наявність достатнього обсягу вільного місця на диску для збереження службових даних і звітів.

Користувач повинен мати права доступу, достатні для підключення до бази даних, читання CDC-таблиць та виконання необхідних операцій моніторингу.

3.4 Вимоги до параметрів технічних засобів

Для роботи програмного забезпечення необхідна система з такими мінімальними характеристиками:

– операційна система: Windows 10 або новіша;

– процесор: двоядерний із тактовою частотою не менше 2,5 ГГц;

– оперативна пам’ять: не менше 8 ГБ;

– вільне місце на диску: не менше 500 МБ для встановлення програмного забезпечення та збереження службових даних;

– монітор: роздільна здатність не менше 1366×768 пікселів;

– мережеве підключення: доступ до локальної мережі або мережі Інтернет;

– пристрої введення.

3.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення повинно бути сумісним з операційними системами Windows 10 та Windows 11. Розробка програмного забезпечення повинна виконуватися мовою програмування C# з використанням платформи .NET.

Для реалізації графічного інтерфейсу користувача доцільно використовувати технологію WPF. Для доступу до даних може використовуватися Entity Framework Core або стандартні засоби роботи з Microsoft SQL Server.

Програмне забезпечення повинно забезпечувати взаємодію з Microsoft SQL Server, який підтримує технологію Change Data Capture. Для виконання зовнішніх дій програмне забезпечення повинно підтримувати HTTP-запити до API, запис у файли та запуск зовнішніх програм.

3.6 Вимоги до маркування та пакування

Програмний продукт має бути упакований в електронний архів. До складу архіву повинні входити виконувані файли програмного забезпечення, необхідні

конфігураційні файли, файл із коротким описом запуску програми та програмна документація.

4 Вимоги до програмної документації

Склад програмної документації повинен включати:

– технічне завдання;

– опис програми;

– програму та методику випробувань програмного забезпечення;

– інструкцію користувача;

– код програми.

5 Стадії та етапи роботи

Розробка програмного забезпечення для відстеження змін в інформаційній системі виконується поетапно. Основні етапи роботи наведено у таблиці А.1.

Таблиця А.1 – Етапи розробки програмного забезпечення

Номер	Назва етапів роботи	Термін виконання
1	Аналіз практичної задачі інженерії програмного забезпечення	20.02.2026
2	Аналіз вимог до програмного забезпечення	13.03.2026
3	Розробка архітектури програмного забезпечення	03.04.2026
4	Розробка проєкту програмного забезпечення	10.04.2026
5	Реалізація та тестування програмного забезпечення	08.05.2026

Порядок контролю і приймання

Контроль виконання робіт здійснюється керівником кваліфікаційної роботи на основі перевірки результатів, отриманих на кожному етапі розробки. Приймання програмного забезпечення здійснюється після завершення реалізації основних функціональних можливостей і проведення тестування.

Під час приймання програмного забезпечення перевіряється відповідність реалізованих функцій поставленим вимогам, коректність роботи графічного

інтерфейсу, стабільність виконання основних сценаріїв використання та відсутність критичних помилок.

Результатом приймання є підтвердження готовності програмного забезпечення до використання в межах кваліфікаційної роботи та подальшого тестування.

ДОДАТОК Б - Вихідні тексти програмних модулів

Converters.cs

```
public class BoolToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value is true ? Visibility.Visible : Visibility.Collapsed;
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        value is Visibility.Visible;
}

public class InverseBoolToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value is true ? Visibility.Collapsed : Visibility.Visible;
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        value is not Visibility.Visible;
}

public class NullToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value != null ? Visibility.Visible : Visibility.Collapsed;
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        throw new NotImplementedException();
}

public class OperationTypeToBrushConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value?.ToString() switch
        {
            "INSERT" => new SolidColorBrush(Color.FromRgb(52, 168,
83)),
            "UPDATE" => new SolidColorBrush(Color.FromRgb(26, 115,
232)),
            "DELETE" => new SolidColorBrush(Color.FromRgb(234, 67,
53)),
            _ => new SolidColorBrush(Colors.Gray)
        };
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        throw new NotImplementedException();
}
```

```
public class StatusToBrushConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value?.ToString() switch
        {
            "Success" => new SolidColorBrush(Color.FromRgb(52, 168,
83)),
            "Failed" => new SolidColorBrush(Color.FromRgb(234, 67,
53)),
            _ => new SolidColorBrush(Colors.Gray)
        };
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        throw new NotImplementedException();
}

public class BoolToEnabledTextConverter : IValueConverter
{
    public object Convert(object value, Type t, object p,
CultureInfo c) =>
        value is true ? "Enabled" : "Disabled";
    public object ConvertBack(object value, Type t, object p,
CultureInfo c) =>
        throw new NotImplementedException();
}
```

Repositories.cs

```
public interface IDataSourceRepository
{
    Task<List<DataSource>> GetAllAsync();
    Task<DataSource?> GetByIdAsync(int id);
    Task<List<DataSource>> GetActiveAsync();
    Task<DataSource> AddAsync(DataSource entity);
    Task UpdateAsync(DataSource entity);
    Task DeleteAsync(int id);
}

public interface IChangeEventRepository
{
    Task<List<ChangeEvent>> GetFilteredAsync(DateTime? from,
DateTime? to, string? table, string? operation, int page, int
pageSize);
    Task<int> GetCountAsync(DateTime? from, DateTime? to, string?
table, string? operation);
    Task<ChangeEvent?> GetWithDetailsAsync(long id);
    Task<List<ChangeEvent>> GetUnprocessedAsync();
    Task AddRangeAsync(IEnumerable<ChangeEvent> events);
    Task MarkProcessedAsync(IEnumerable<long> ids);
    Task<DashboardStats> GetDashboardStatsAsync();
    Task<List<DailyEventCount>> GetEventsByDayAsync(int days);
```

```
    Task<List<TableEventCount>> GetTopChangedTablesAsync(int top);
    Task DeleteOlderThanAsync(DateTime cutoff);
}

public interface IAutomationRuleRepository
{
    Task<List<AutomationRule>> GetAllWithActionsAsync();
    Task<AutomationRule?> GetWithActionsAsync(int id);
    Task<AutomationRule> AddAsync(AutomationRule entity);
    Task UpdateAsync(AutomationRule entity);
    Task DeleteAsync(int id);
    Task<List<AutomationRule>> GetMatchingRulesAsync(string
tableName, string operationType);
}

public interface IExecutionHistoryRepository
{
    Task<List<ExecutionHistory>> GetFilteredAsync(string?
ruleName, string? status, DateTime? from, DateTime? to, int page,
int pageSize);
    Task<int> GetCountAsync(string? ruleName, string? status,
DateTime? from, DateTime? to);
    Task AddAsync(ExecutionHistory entity);
    Task DeleteOlderThanAsync(DateTime cutoff);
}

public interface ISettingsRepository
{
    Task<ApplicationSettings> GetAsync();
    Task UpdateAsync(ApplicationSettings settings);
}

// ——— DTOs

public record DashboardStats(long Total, long Today, long Inserts,
long Updates, long Deletes);
public record DailyEventCount(DateTime Date, int Count);
public record TableEventCount(string TableName, int Count);

// ——— Implementations

public class DataSourceRepository(AppDbContext db) :
IDataSourceRepository
{
    public Task<List<DataSource>> GetAllAsync() =>
db.DataSources.OrderBy(x => x.Name).ToListAsync();
    public Task<DataSource?> GetByIdAsync(int id) =>
db.DataSources.FindAsync(id).AsTask();
    public Task<List<DataSource>> GetActiveAsync() =>
db.DataSources.Where(x => x.IsActive).ToListAsync();

    public async Task<DataSource> AddAsync(DataSource entity)
    {
```

```
        db.DataSources.Add(entity);
        await db.SaveChangesAsync();
        return entity;
    }

    public async Task UpdateAsync(DataSource entity)
    {
        db.DataSources.Update(entity);
        await db.SaveChangesAsync();
    }

    public async Task DeleteAsync(int id)
    {
        var e = await db.DataSources.FindAsync(id);
        if (e != null) { db.DataSources.Remove(e); await
db.SaveChangesAsync(); }
    }
}

public class ChangeEventRepository(AppDbContext db) :
IChangeEventRepository
{
    public async Task<List<ChangeEvent>>
GetFilteredAsync(DateTime? from, DateTime? to, string? table,
string? operation, int page, int pageSize)
    {
        var q = db.ChangeEvents.Include(x =>
x.DataSource).AsQueryable();
        if (from.HasValue) q = q.Where(x => x.Timestamp >=
from.Value);
        if (to.HasValue) q = q.Where(x => x.Timestamp <=
to.Value);
        if (!string.IsNullOrWhiteSpace(table)) q = q.Where(x =>
x.TableName.Contains(table));
        if (!string.IsNullOrWhiteSpace(operation)) q = q.Where(x
=> x.OperationType == operation);
        return await q.OrderByDescending(x =>
x.Timestamp).Skip((page - 1) *
pageSize).Take(pageSize).ToListAsync();
    }

    public async Task<int> GetCountAsync(DateTime? from, DateTime?
to, string? table, string? operation)
    {
        var q = db.ChangeEvents.AsQueryable();
        if (from.HasValue) q = q.Where(x => x.Timestamp >=
from.Value);
        if (to.HasValue) q = q.Where(x => x.Timestamp <=
to.Value);
        if (!string.IsNullOrWhiteSpace(table)) q = q.Where(x =>
x.TableName.Contains(table));
        if (!string.IsNullOrWhiteSpace(operation)) q = q.Where(x
=> x.OperationType == operation);
```

```
        return await q.CountAsync();
    }

    public Task<ChangeEvent?> GetWithDetailsAsync(long id) =>
        db.ChangeEvents.Include(x => x.Details).Include(x =>
x.DataSource).FirstOrDefaultAsync(x => x.Id == id);

    public Task<List<ChangeEvent>> GetUnprocessedAsync() =>
        db.ChangeEvents.Where(x => !x.IsProcessed).OrderBy(x =>
x.Timestamp).Take(200).ToListAsync();

    public async Task AddRangeAsync(IEnumerable<ChangeEvent>
events)
    {
        db.ChangeEvents.AddRange(events);
        await db.SaveChangesAsync();
    }

    public async Task MarkProcessedAsync(IEnumerable<long> ids)
    {
        await db.ChangeEvents.Where(x => ids.Contains(x.Id))
            .ExecuteUpdateAsync(s => s.SetProperty(x =>
x.IsProcessed, true));
    }

    public async Task<DashboardStats> GetDashboardStatsAsync()
    {
        var today = DateTime.UtcNow.Date;
        var total = await db.ChangeEvents.LongCountAsync();
        var todayCount = await db.ChangeEvents.LongCountAsync(x =>
x.Timestamp >= today);
        var inserts = await db.ChangeEvents.LongCountAsync(x =>
x.OperationType == "INSERT");
        var updates = await db.ChangeEvents.LongCountAsync(x =>
x.OperationType == "UPDATE");
        var deletes = await db.ChangeEvents.LongCountAsync(x =>
x.OperationType == "DELETE");
        return new DashboardStats(total, todayCount, inserts,
updates, deletes);
    }

    public async Task<List<DailyEventCount>>
GetEventsByDayAsync(int days)
    {
        var since = DateTime.UtcNow.Date.AddDays(-days);
        var raw = await db.ChangeEvents
            .Where(x => x.Timestamp >= since)
            .GroupBy(x => x.Timestamp.Date)
            .Select(g => new { Date = g.Key, Count = g.Count() })
            .OrderBy(x => x.Date)
            .ToListAsync();
        return raw.Select(x => new DailyEventCount(x.Date,
x.Count)).ToList();
```

```
    }

    public async Task<List<TableEventCount>>
GetTopChangedTablesAsync(int top)
    {
        var raw = await db.ChangeEvents
            .GroupBy(x => x.TableName)
            .Select(g => new { Table = g.Key, Count = g.Count() })
            .OrderByDescending(x => x.Count)
            .Take(top)
            .ToListAsync();
        return raw.Select(x => new TableEventCount(x.Table,
x.Count)).ToList();
    }

    public async Task DeleteOlderThanAsync(DateTime cutoff) =>
        await db.ChangeEvents.Where(x => x.Timestamp <
cutoff).ExecuteDeleteAsync();
}

public class AutomationRuleRepository(AppDbContext db) :
IAutomationRuleRepository
{
    public Task<List<AutomationRule>> GetAllWithActionsAsync() =>
        db.AutomationRules.Include(x => x.Actions).OrderBy(x =>
x.RuleName).ToListAsync();

    public Task<AutomationRule?> GetWithActionsAsync(int id) =>
        db.AutomationRules.Include(x =>
x.Actions).FirstOrDefaultAsync(x => x.Id == id);

    public async Task<AutomationRule> AddAsync(AutomationRule
entity)
    {
        db.AutomationRules.Add(entity);
        await db.SaveChangesAsync();
        return entity;
    }

    public async Task UpdateAsync(AutomationRule entity)
    {
        db.AutomationRules.Update(entity);
        await db.SaveChangesAsync();
    }

    public async Task DeleteAsync(int id)
    {
        var e = await db.AutomationRules.FindAsync(id);
        if (e != null) { db.AutomationRules.Remove(e); await
db.SaveChangesAsync(); }
    }

    public Task<List<AutomationRule>> GetMatchingRulesAsync(string
```

```
tableName, string operationType) =>
        db.AutomationRules.Include(x => x.Actions)
            .Where(x => x.IsEnabled &&
                        (x.TableName == "*" || x.TableName ==
tableName) &&
                        (x.EventType == "ALL" || x.EventType ==
operationType))
            .ToListAsync();
}

public class ExecutionHistoryRepository(AppDbContext db) :
IExecutionHistoryRepository
{
    public async Task<List<ExecutionHistory>>
GetFilteredAsync(string? ruleName, string? status, DateTime? from,
DateTime? to, int page, int pageSize)
    {
        var q = db.ExecutionHistories.Include(x =>
x.AutomationRule).Include(x => x.ChangeEvent).AsQueryable();
        if (!string.IsNullOrWhiteSpace(ruleName)) q = q.Where(x =>
x.AutomationRule.RuleName.Contains(ruleName));
        if (!string.IsNullOrWhiteSpace(status)) q = q.Where(x =>
x.Status == status);
        if (from.HasValue) q = q.Where(x => x.ExecutionTime >=
from.Value);
        if (to.HasValue) q = q.Where(x => x.ExecutionTime <=
to.Value);
        return await q.OrderByDescending(x =>
x.ExecutionTime).Skip((page - 1) *
pageSize).Take(pageSize).ToListAsync();
    }

    public async Task<int> GetCountAsync(string? ruleName, string?
status, DateTime? from, DateTime? to)
    {
        var q = db.ExecutionHistories.Include(x =>
x.AutomationRule).AsQueryable();
        if (!string.IsNullOrWhiteSpace(ruleName)) q = q.Where(x =>
x.AutomationRule.RuleName.Contains(ruleName));
        if (!string.IsNullOrWhiteSpace(status)) q = q.Where(x =>
x.Status == status);
        if (from.HasValue) q = q.Where(x => x.ExecutionTime >=
from.Value);
        if (to.HasValue) q = q.Where(x => x.ExecutionTime <=
to.Value);
        return await q.CountAsync();
    }

    public async Task AddAsync(ExecutionHistory entity)
    {
        db.ExecutionHistories.Add(entity);
        await db.SaveChangesAsync();
    }
```

```
    public async Task DeleteOlderThanAsync(DateTime cutoff) =>
        await db.ExecutionHistories.Where(x => x.ExecutionTime <
cutoff).ExecuteDeleteAsync();
}

public class SettingsRepository(AppDbContext db) :
ISettingsRepository
{
    public async Task<ApplicationSettings> GetAsync()
    {
        return await db.ApplicationSettings.FirstOrDefaultAsync()
               ?? new ApplicationSettings();
    }

    public async Task UpdateAsync(ApplicationSettings settings)
    {
        settings.UpdatedAt = DateTime.UtcNow;
        db.ApplicationSettings.Update(settings);
        await db.SaveChangesAsync();
    }
}
```

AppDbContext.cs

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }

    public DbSet<DataSource> DataSources => Set<DataSource>();
    public DbSet<ChangeEvent> ChangeEvents => Set<ChangeEvent>();
    public DbSet<ChangeDetail> ChangeDetails =>
Set<ChangeDetail>();
    public DbSet<AutomationRule> AutomationRules =>
Set<AutomationRule>();
    public DbSet<RuleAction> RuleActions => Set<RuleAction>();
    public DbSet<ExecutionHistory> ExecutionHistories =>
Set<ExecutionHistory>();
    public DbSet<ApplicationSettings> ApplicationSettings =>
Set<ApplicationSettings>();

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<DataSource>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x =>
x.Name).HasMaxLength(200).IsRequired();
            e.Property(x =>
```

```
x.Server).HasMaxLength(500).IsRequired();
            e.Property(x =>
x.Database).HasMaxLength(200).IsRequired();
            e.Property(x => x.Username).HasMaxLength(200);
            e.Property(x => x.Password).HasMaxLength(500);
        });

        modelBuilder.Entity<ChangeEvent>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x => x.Id).ValueGeneratedOnAdd();
            e.Property(x =>
x.DatabaseName).HasMaxLength(200).IsRequired();
            e.Property(x =>
x.TableName).HasMaxLength(200).IsRequired();
            e.Property(x =>
x.OperationType).HasMaxLength(20).IsRequired();
            e.Property(x => x.PrimaryKeyValue).HasMaxLength(500);
            e.HasOne(x => x.DataSource).WithMany(x =>
x.ChangeEvents)
                .HasForeignKey(x =>
x.DataSourceId).OnDelete(DeleteBehavior.Restrict);
            e.HasIndex(x => new { x.Timestamp, x.TableName,
x.OperationType });
        });

        modelBuilder.Entity<ChangeDetail>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x =>
x.ColumnName).HasMaxLength(200).IsRequired();
            e.HasOne(x => x.ChangeEvent).WithMany(x => x.Details)
                .HasForeignKey(x =>
x.ChangeEventId).OnDelete(DeleteBehavior.Cascade);
        });

        modelBuilder.Entity<AutomationRule>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x =>
x.RuleName).HasMaxLength(200).IsRequired();
            e.Property(x =>
x.TableName).HasMaxLength(200).IsRequired();
            e.Property(x =>
x.EventType).HasMaxLength(20).IsRequired();
        });

        modelBuilder.Entity<RuleAction>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x =>
x.ActionType).HasMaxLength(50).IsRequired();
            e.Property(x => x.Url).HasMaxLength(2000);
```

```
            e.Property(x => x.FilePath).HasMaxLength(1000);
            e.Property(x => x.ExecutablePath).HasMaxLength(1000);
            e.HasOne(x => x.AutomationRule).WithMany(x =>
x.Actions)
                .HasForeignKey(x =>
x.AutomationRuleId).OnDelete(DeleteBehavior.Cascade);
        });

        modelBuilder.Entity<ExecutionHistory>(e =>
        {
            e.HasKey(x => x.Id);
            e.Property(x => x.Id).ValueGeneratedOnAdd();
            e.Property(x =>
x.ActionType).HasMaxLength(50).IsRequired();
            e.Property(x =>
x.Status).HasMaxLength(20).IsRequired();
            e.HasOne(x => x.AutomationRule).WithMany(x =>
x.Executions)
                .HasForeignKey(x =>
x.AutomationRuleId).OnDelete(DeleteBehavior.Restrict);
            e.HasOne(x => x.ChangeEvent).WithMany(x =>
x.Executions)
                .HasForeignKey(x =>
x.ChangeEventId).OnDelete(DeleteBehavior.Restrict);
            e.HasIndex(x => x.ExecutionTime);
        });

        modelBuilder.Entity<ApplicationSettings>(e =>
        {
            e.HasKey(x => x.Id);
        });

        // Seed default settings
        modelBuilder.Entity<ApplicationSettings>().HasData(new
ApplicationSettings
        {
            Id = 1,
            CdcPollingIntervalSeconds = 10,
            LogRetentionDays = 30,
            LoggingLevel = "Information",
            ExportDirectory = "C:\\ChangeTrackerExports",
            UpdatedAt = new DateTime(2024, 1, 1)
        });
    }
}
```

Services.cs

```
// ─── Navigation
public interface INavigationService
{
    event Action<string>? Navigated;
    void Navigate(string page);
```

```
    string CurrentPage { get; }
}

public class NavigationService : INavigationService
{
    public event Action<string>? Navigated;
    public string CurrentPage { get; private set; } = "Dashboard";

    public void Navigate(string page)
    {
        CurrentPage = page;
        Navigated?.Invoke(page);
    }
}

// ─── Notifications
public interface INotificationService
{
    void ShowSuccess(string message);
    void ShowError(string message);
    void ShowInfo(string message);
    event Action<Notification>? NotificationRaised;
}

public record Notification(string Message, NotificationType Type,
DateTime At);
public enum NotificationType { Success, Error, Info }

public class NotificationService : INotificationService
{
    public event Action<Notification>? NotificationRaised;

    public void ShowSuccess(string message) => Raise(message,
NotificationType.Success);
    public void ShowError(string message) => Raise(message,
NotificationType.Error);
    public void ShowInfo(string message) => Raise(message,
NotificationType.Info);

    private void Raise(string msg, NotificationType type) =>
        Application.Current?.Dispatcher.Invoke(() =>
            NotificationRaised?.Invoke(new Notification(msg, type,
DateTime.Now)));
}

// ─── Dialog Service
public interface IDialogService
{
    bool Confirm(string message, string title = "Confirm");
}

public class DialogService : IDialogService
{
```

```
    public bool Confirm(string message, string title = "Confirm")
=>
        MessageBox.Show(message, title, MessageBoxButton.YesNo,
MessageBoxImage.Question) == MessageBoxResult.Yes;
}
```

Entities.cs

```
public class DataSource
{
    public int Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Server { get; set; } = string.Empty;
    public string Database { get; set; } = string.Empty;
    public string Username { get; set; } = string.Empty;
    public string Password { get; set; } = string.Empty;
    public bool IsActive { get; set; } = true;
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public DateTime? LastPolledAt { get; set; }
    public ICollection<ChangeEvent> ChangeEvents { get; set; } =
new List<ChangeEvent>();
}

public class ChangeEvent
{
    public long Id { get; set; }
    public int DataSourceId { get; set; }
    public DataSource DataSource { get; set; } = null!;
    public DateTime Timestamp { get; set; }
    public string DatabaseName { get; set; } = string.Empty;
    public string TableName { get; set; } = string.Empty;
    public string OperationType { get; set; } = string.Empty; //
INSERT, UPDATE, DELETE
    public string PrimaryKeyValue { get; set; } = string.Empty;
    public bool IsProcessed { get; set; }
    public ICollection<ChangeDetail> Details { get; set; } = new
List<ChangeDetail>();
    public ICollection<ExecutionHistory> Executions { get; set; }
= new List<ExecutionHistory>();
}

public class ChangeDetail
{
    public int Id { get; set; }
    public long ChangeEventId { get; set; }
    public ChangeEvent ChangeEvent { get; set; } = null!;
    public string ColumnName { get; set; } = string.Empty;
    public string? OldValue { get; set; }
    public string? NewValue { get; set; }
}

public class AutomationRule
{
```

```
    public int Id { get; set; }
    public string RuleName { get; set; } = string.Empty;
    public bool IsEnabled { get; set; } = true;
    public string TableName { get; set; } = string.Empty;
    public string EventType { get; set; } = string.Empty; //
INSERT, UPDATE, DELETE, ALL
    public int DataSourceId { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public ICollection<RuleAction> Actions { get; set; } = new
List<RuleAction>();
    public ICollection<ExecutionHistory> Executions { get; set; }
= new List<ExecutionHistory>();
}

public class RuleAction
{
    public int Id { get; set; }
    public int AutomationRuleId { get; set; }
    public AutomationRule AutomationRule { get; set; } = null!;
    public string ActionType { get; set; } = string.Empty; //
HTTP_GET, HTTP_POST, WRITE_FILE, EXEC_PROCESS
    public int SortOrder { get; set; }

    // HTTP action fields
    public string? Url { get; set; }
    public string? Headers { get; set; }
    public string? RequestBody { get; set; }
    public int? TimeoutSeconds { get; set; }

    // File action fields
    public string? FilePath { get; set; }
    public string? TextTemplate { get; set; }

    // Process action fields
    public string? ExecutablePath { get; set; }
    public string? Arguments { get; set; }
}

public class ExecutionHistory
{
    public long Id { get; set; }
    public DateTime ExecutionTime { get; set; } = DateTime.UtcNow;
    public int AutomationRuleId { get; set; }
    public AutomationRule AutomationRule { get; set; } = null!;
    public long ChangeEventId { get; set; }
    public ChangeEvent ChangeEvent { get; set; } = null!;
    public string ActionType { get; set; } = string.Empty;
    public string Status { get; set; } = string.Empty; // Success,
Failed
    public long DurationMs { get; set; }
    public string? ErrorMessage { get; set; }
}
```

```
public class ApplicationSettings
{
    public int Id { get; set; }
    public int CdcPollingIntervalSeconds { get; set; } = 10;
    public int LogRetentionDays { get; set; } = 30;
    public string LoggingLevel { get; set; } = "Information";
    public string ExportDirectory { get; set; } =
"C:\\ChangeTrackerExports";
    public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;
}
```

BackgroundWorkerService.cs

```
public class BackgroundWorkerService(
    IServiceScopeFactory scopeFactory,
    ILogger<BackgroundWorkerService> logger) : BackgroundService
{
    private int _pollingInterval = 10;

    public void UpdatePollingInterval(int seconds) =>
_pollingInterval = Math.Max(5, seconds);

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        logger.LogInformation("BackgroundWorkerService started");

        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                await using var scope =
scopeFactory.CreateAsyncScope();

                var settingsRepo =
scope.ServiceProvider.GetRequiredService<ISettingsRepository>();
                var settings = await settingsRepo.GetAsync();
                _pollingInterval =
settings.CdcPollingIntervalSeconds;

                // Poll all active data sources
                var dataSourceRepo =
scope.ServiceProvider.GetRequiredService<IDataSourceRepository>();
                var cdcService =
scope.ServiceProvider.GetRequiredService<ICdcPollingService>();
                var sources = await
dataSourceRepo.GetActiveAsync();

                foreach (var source in sources)
                    await cdcService.PollAsync(source,
stoppingToken);

                // Process unhandled change events
```

```
                var changeRepo =
scope.ServiceProvider.GetRequiredService<IChangeEventRepository>()
;
                var ruleRepo =
scope.ServiceProvider.GetRequiredService<IAutomationRuleRepository
>();
                var engine =
scope.ServiceProvider.GetRequiredService<IActionExecutionEngine>()
;

                var unprocessed = await
changeRepo.GetUnprocessedAsync();
                var processedIds = new List<long>();

                foreach (var ev in unprocessed)
                {
                    var rules = await
ruleRepo.GetMatchingRulesAsync(ev.TableName, ev.OperationType);
                    foreach (var rule in rules)
                        await engine.ExecuteAsync(rule, ev,
stoppingToken);
                    processedIds.Add(ev.Id);
                }

                if (processedIds.Count > 0)
                    await
changeRepo.MarkProcessedAsync(processedIds);

                // Clean up old records
                var cutoff = DateTime.UtcNow.AddDays(-
settings.LogRetentionDays);
                await changeRepo.DeleteOlderThanAsync(cutoff);

                var histRepo =
scope.ServiceProvider.GetRequiredService<IExecutionHistoryReposito
ry>();
                await histRepo.DeleteOlderThanAsync(cutoff);
            }
            catch (Exception ex) when
(!stoppingToken.IsCancellationRequested)
            {
                logger.LogError(ex, "Error in background worker
iteration");
            }

            await
Task.Delay(TimeSpan.FromSeconds(_pollingInterval), stoppingToken);
        }

        logger.LogInformation("BackgroundWorkerService stopped");
    }
}
```

CdcServices.cs

```
// ─── CDC Polling Service
public interface ICdcPollingService
{
    Task PollAsync(DataSource source, CancellationToken ct);
}

public class CdcPollingService(
    IChangeEventRepository changeRepo,
    ILogger<CdcPollingService> logger) : ICdcPollingService
{
    public async Task PollAsync(DataSource source,
CancellationToken ct)
    {
        var connStr = BuildConnectionString(source);
        try
        {
            await using var conn = new SqlConnection(connStr);
            await conn.OpenAsync(ct);

            // Get all CDC-enabled tables
            var tables = await GetCdcTablesAsync(conn, ct);
            var events = new List<ChangeEvent>();

            foreach (var table in tables)
            {
                var since = source.LastPolledAt ??
DateTime.UtcNow.AddMinutes(-5);
                var tableEvents = await ReadCdcChangesAsync(conn,
source, table, since, ct);
                events.AddRange(tableEvents);
            }

            if (events.Count > 0)
            {
                await changeRepo.AddRangeAsync(events);
                logger.LogInformation("Captured {Count} CDC events
from {Source}", events.Count, source.Name);
            }

            source.LastPolledAt = DateTime.UtcNow;
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error polling CDC from {Source}",
source.Name);
        }
    }

    private static async Task<List<string>>
GetCdcTablesAsync(SqlConnection conn, CancellationToken ct)
    {
```

```
        const string sql = @"
            SELECT OBJECT_NAME(source_object_id) AS TableName
            FROM sys.change_tracking_tables
            UNION
            SELECT DISTINCT PARSENAME(REPLACE(capture_instance,
'_', '.'), 1)
            FROM cdc.change_tables";

        var tables = new List<string>();
        try
        {
            await using var cmd = new SqlCommand(sql, conn);
            await using var reader = await
cmd.ExecuteReaderAsync(ct);
            while (await reader.ReadAsync(ct))
                tables.Add(reader.GetString(0));
        }
        catch
        {
            // Fall back to demo tables if CDC not configured
            tables.AddRange(["Customers", "Orders", "Products",
"Employees"]);
        }
        return tables;
    }

    private static async Task<List<ChangeEvent>>
ReadCdcChangesAsync(
        SqlConnection conn, DataSource source, string tableName,
DateTime since, CancellationToken ct)
    {
        var events = new List<ChangeEvent>();

        // Query CDC change table: cdc.dbo_<table>_CT
        var cdcTable = $"cdc.dbo_{tableName}_CT";
        var sql = $@"
            SELECT __$start_lsn, __$operation, __$update_mask, *
            FROM {cdcTable}
            WHERE sys.fn_cdc_map_lsn_to_time(__$start_lsn) >
@Since
            ORDER BY __$start_lsn";

        try
        {
            await using var cmd = new SqlCommand(sql, conn);
            cmd.Parameters.AddWithValue("@Since", since);
            await using var reader = await
cmd.ExecuteReaderAsync(ct);

            var schemaTable = reader.GetSchemaTable();
            var columnNames =
schemaTable?.Rows.Cast<System.Data.DataRow>()
                .Select(r => r["ColumnName"].ToString()!)
```

```
                .Where(c => !c.StartsWith("__$"))
                .ToList() ?? [];

            while (await reader.ReadAsync(ct))
            {
                var opCode = reader.GetInt32(1);
                var operation = opCode switch
                {
                    1 => "DELETE",
                    2 => "INSERT",
                    3 => "UPDATE", // before
                    4 => "UPDATE", // after
                    _ => "UNKNOWN"
                };

                if (opCode == 3) continue; // skip "before" update
rows

                var ev = new ChangeEvent
                {
                    DataSourceId = source.Id,
                    Timestamp = DateTime.UtcNow,
                    DatabaseName = source.Database,
                    TableName = tableName,
                    OperationType = operation,
                    PrimaryKeyValue = columnNames.FirstOrDefault()
is { } pk
                        ? reader[pk]?.ToString() ?? ""
                        : "",
                    IsProcessed = false
                };

                foreach (var col in columnNames)
                {
                    ev.Details.Add(new ChangeDetail
                    {
                        ColumnName = col,
                        NewValue = reader[col]?.ToString()
                    });
                }

                events.Add(ev);
            }
        }
        catch
        {
            // CDC table might not exist for this table; skip
silently
        }

        return events;
    }
```

```
    private static string BuildConnectionString(DataSource source)
    {
        var builder = new SqlConnectionStringBuilder
        {
            DataSource = source.Server,
            InitialCatalog = source.Database,
            ConnectTimeout = 15
        };
        if (string.IsNullOrWhiteSpace(source.Username))
            builder.IntegratedSecurity = true;
        else
        {
            builder.UserID = source.Username;
            builder.Password = source.Password;
        }
        return builder.ConnectionString;
    }
}

// ─── Action Execution Engine
public interface IActionExecutionEngine
{
    Task ExecuteAsync(AutomationRule rule, ChangeEvent ev,
CancellationToken ct);
}

public class ActionExecutionEngine(
    IExecutionHistoryRepository historyRepo,
    ILogger<ActionExecutionEngine> logger) :
IActionExecutionEngine
{
    private static readonly HttpClient Http = new() { Timeout =
TimeSpan.FromSeconds(30) };

    public async Task ExecuteAsync(AutomationRule rule,
ChangeEvent ev, CancellationToken ct)
    {
        foreach (var action in rule.Actions.OrderBy(a =>
a.SortOrder))
        {
            var sw = Stopwatch.StartNew();
            string status = "Success";
            string? error = null;

            try
            {
                await ExecuteActionAsync(action, ev, ct);
            }
            catch (Exception ex)
            {
                status = "Failed";
                error = ex.Message;
                logger.LogWarning(ex, "Action {Type} failed for
```

```
rule {Rule}", action.ActionType, rule.RuleName);
            }
            finally
            {
                sw.Stop();
            }

            await historyRepo.AddAsync(new ExecutionHistory
            {
                ExecutionTime = DateTime.UtcNow,
                AutomationRuleId = rule.Id,
                ChangeEventId = ev.Id,
                ActionType = action.ActionType,
                Status = status,
                DurationMs = sw.ElapsedMilliseconds,
                ErrorMessage = error
            });
        }
    }

    private async Task ExecuteActionAsync(RuleAction action,
ChangeEvent ev, CancellationToken ct)
    {
        switch (action.ActionType)
        {
            case "HTTP_GET":
                await ExecuteHttpGetAsync(action, ev, ct);
                break;
            case "HTTP_POST":
                await ExecuteHttpPostAsync(action, ev, ct);
                break;
            case "WRITE_FILE":
                ExecuteWriteFile(action, ev);
                break;
            case "EXEC_PROCESS":
                await ExecuteProcessAsync(action, ev, ct);
                break;
            default:
                throw new InvalidOperationException($"Unknown
action type: {action.ActionType}");
        }
    }

    private async Task ExecuteHttpGetAsync(RuleAction action,
ChangeEvent ev, CancellationToken ct)
    {
        var url = ResolveTemplate(action.Url ?? "", ev);
        using var request = new HttpRequestMessage(HttpMethod.Get,
url);
        ApplyHeaders(request, action.Headers);
        var timeout = action.TimeoutSeconds ?? 30;
        using var cts =
CancellationTokenSource.CreateLinkedTokenSource(ct);
```

```
        cts.CancelAfter(TimeSpan.FromSeconds(timeout));
        var response = await Http.SendAsync(request, cts.Token);
        response.EnsureSuccessStatusCode();
        logger.LogInformation("HTTP GET {Url} → {Status}", url,
response.StatusCode);
    }

    private async Task ExecuteHttpPostAsync(RuleAction action,
ChangeEvent ev, CancellationToken ct)
    {
        var url = ResolveTemplate(action.Url ?? "", ev);
        var body = ResolveTemplate(action.RequestBody ?? "", ev);
        using var request = new
HttpRequestMessage(HttpMethod.Post, url)
        {
            Content = new StringContent(body, Encoding.UTF8,
"application/json")
        };
        ApplyHeaders(request, action.Headers);
        var timeout = action.TimeoutSeconds ?? 30;
        using var cts =
CancellationTokenSource.CreateLinkedTokenSource(ct);
        cts.CancelAfter(TimeSpan.FromSeconds(timeout));
        var response = await Http.SendAsync(request, cts.Token);
        response.EnsureSuccessStatusCode();
        logger.LogInformation("HTTP POST {Url} → {Status}", url,
response.StatusCode);
    }

    private static void ExecuteWriteFile(RuleAction action,
ChangeEvent ev)
    {
        if (string.IsNullOrWhiteSpace(action.FilePath)) return;
        var dir = Path.GetDirectoryName(action.FilePath);
        if (!string.IsNullOrWhiteSpace(dir))
Directory.CreateDirectory(dir);
        var content = ResolveTemplate(action.TextTemplate ?? "",
ev);
        File.AppendAllText(action.FilePath, content +
Environment.NewLine);
    }

    private static async Task ExecuteProcessAsync(RuleAction
action, ChangeEvent ev, CancellationToken ct)
    {
        if (string.IsNullOrWhiteSpace(action.ExecutablePath))
return;
        var args = ResolveTemplate(action.Arguments ?? "", ev);
        var psi = new ProcessStartInfo(action.ExecutablePath,
args)
        {
            UseShellExecute = false,
            RedirectStandardOutput = true,
```

```
            CreateNoWindow = true
        };
        using var proc = Process.Start(psi) ?? throw new
InvalidOperationException("Failed to start process");
        await proc.WaitForExitAsync(ct);
        if (proc.ExitCode != 0)
            throw new InvalidOperationException($"Process exited
with code {proc.ExitCode}");
    }

    private static void ApplyHeaders(HttpRequestMessage req,
string? headersJson)
    {
        if (string.IsNullOrWhiteSpace(headersJson)) return;
        try
        {
            var dict =
Newtonsoft.Json.JsonConvert.DeserializeObject<Dictionary<string,
string>>(headersJson);
            if (dict == null) return;
            foreach (var (k, v) in dict)
                req.Headers.TryAddWithoutValidation(k, v);
        }
        catch { /* ignore malformed headers */ }
    }

    private static string ResolveTemplate(string template,
ChangeEvent ev) =>
        template
            .Replace("{{EventId}}", ev.Id.ToString())
            .Replace("{{Table}}", ev.TableName)
            .Replace("{{Operation}}", ev.OperationType)
            .Replace("{{Database}}", ev.DatabaseName)
            .Replace("{{Timestamp}}", ev.Timestamp.ToString("O"))
            .Replace("{{PrimaryKey}}", ev.PrimaryKeyValue);
}

// ─── Connection Test Service
public interface IConnectionTestService
{
    Task<(bool Success, string Message)> TestAsync(DataSource
source);
}

public class ConnectionTestService : IConnectionTestService
{
    public async Task<(bool Success, string Message)>
TestAsync(DataSource source)
    {
        var connStr = BuildConnectionString(source);
        try
        {
            await using var conn = new SqlConnection(connStr);
```

```
            await conn.OpenAsync();

            // Check if CDC is enabled on the database
            await using var cmd = new SqlCommand(
                "SELECT is_cdc_enabled FROM sys.databases WHERE
name = DB_NAME()", conn);
            var cdcEnabled = (bool?)await
cmd.ExecuteScalarAsync();
            var cdcMsg = cdcEnabled == true ? "CDC is enabled." :
"CDC is NOT enabled on this database.";

            return (true, $"Connection successful. SQL Server
{conn.ServerVersion}. {cdcMsg}");
        }
        catch (Exception ex)
        {
            return (false, $"Connection failed: {ex.Message}");
        }
    }

    private static string BuildConnectionString(DataSource source)
    {
        var builder = new SqlConnectionStringBuilder
        {
            DataSource = source.Server,
            InitialCatalog = source.Database,
            ConnectTimeout = 10
        };
        if (string.IsNullOrWhiteSpace(source.Username))
            builder.IntegratedSecurity = true;
        else
        {
            builder.UserID = source.Username;
            builder.Password = source.Password;
        }
        return builder.ConnectionString;
    }
}
```

ViewModels.cs

```
using ChangeTracker.Data.Repositories;
using ChangeTracker.Infrastructure;
using ChangeTracker.Models;
using ChangeTracker.Services;
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
using System.Collections.ObjectModel;
using System.Windows;

namespace ChangeTracker.ViewModels;

// ─── Base
```

```
public abstract partial class BaseViewModel : ObservableObject
{
    [ObservableProperty] private bool _isBusy;
    [ObservableProperty] private string _statusMessage =
string.Empty;
}

// ─── Shell / Main Window

public partial class MainViewModel : BaseViewModel
{
    private readonly INavigationService _nav;
    private readonly INotificationService _notif;

    [ObservableProperty] private string _currentPage =
"Dashboard";
    [ObservableProperty] private BaseViewModel? _currentViewModel;
    [ObservableProperty] private string? _notificationMessage;
    [ObservableProperty] private string _notificationColor =
"#27AE60";
    [ObservableProperty] private bool _showNotification;

    public MainViewModel(INavigationService nav,
INotificationService notif, IServiceProvider sp)
    {
        _nav = nav;
        _notif = notif;
        _sp = sp;

        _nav.Navigated += OnNavigated;
        _notif.NotificationRaised += OnNotification;

        Navigate("Dashboard");
    }

    private readonly IServiceProvider _sp;

    private void OnNavigated(string page)
    {
        CurrentPage = page;
        CurrentViewModel = page switch
        {
            "Dashboard" =>
(BaseViewModel)_sp.GetService(typeof(DashboardViewModel))!,
            "DataSources" =>
(BaseViewModel)_sp.GetService(typeof(DataSourceViewModel))!,
            "ChangeLog" =>
(BaseViewModel)_sp.GetService(typeof(ChangeLogViewModel))!,
            "AutomationRules" =>
(BaseViewModel)_sp.GetService(typeof(AutomationRulesViewModel))!,
            "Executions" =>
(BaseViewModel)_sp.GetService(typeof(ExecutionsViewModel))!,
            "Settings" =>
```

```
(BaseViewModel)_sp.GetService(typeof(SettingsViewModel))!,
            _ =>
(BaseViewModel)_sp.GetService(typeof(DashboardViewModel))!
        };
        if (CurrentViewModel is ILoadable l)
            _ = l.LoadAsync();
    }

    private async void OnNotification(Notification n)
    {
        NotificationColor = n.Type switch
        {
            NotificationType.Success => "#27AE60",
            NotificationType.Error => "#E74C3C",
            _ => "#2980B9"
        };
        NotificationMessage = n.Message;
        ShowNotification = true;
        await Task.Delay(3500);
        ShowNotification = false;
    }

    [RelayCommand]
    private void Navigate(string page) => _nav.Navigate(page);
}

public interface ILoadable { Task LoadAsync(); }

// ─── Dashboard ────

public partial class DashboardViewModel : BaseViewModel, ILoadable
{
    private readonly IChangeEventRepository _repo;

    [ObservableProperty] private long _totalEvents;
    [ObservableProperty] private long _eventsToday;
    [ObservableProperty] private long _insertCount;
    [ObservableProperty] private long _updateCount;
    [ObservableProperty] private long _deleteCount;

    public ObservableCollection<DailyEventCount> EventsByDay {
get; } = [];
    public ObservableCollection<TableEventCount> TopTables { get;
} = [];

    // Chart series for LiveCharts
    public ObservableCollection<LiveCharts.Wpf.LineSeries>
LineSeriesCollection { get; } = [];
    public ObservableCollection<LiveCharts.Wpf.PieSeries>
PieSeriesCollection { get; } = [];

    public DashboardViewModel(IChangeEventRepository repo)
    {
```

```
        _repo = repo;
    }

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            var stats = await _repo.GetDashboardStatsAsync();
            TotalEvents = stats.Total;
            EventsToday = stats.Today;
            InsertCount = stats.Inserts;
            UpdateCount = stats.Updates;
            DeleteCount = stats.Deletes;

            var daily = await _repo.GetEventsByDayAsync(14);
            EventsByDay.Clear();
            foreach (var d in daily) EventsByDay.Add(d);

            var tables = await _repo.GetTopChangedTablesAsync(10);
            TopTables.Clear();
            foreach (var t in tables) TopTables.Add(t);

            BuildCharts(daily, stats);
        }
        finally { IsBusy = false; }
    }

    private void BuildCharts(List<DailyEventCount> daily,
DashboardStats stats)
    {
        LineSeriesCollection.Clear();
        LineSeriesCollection.Add(new LiveCharts.Wpf.LineSeries
        {
            Title = "Events",
            Values = new
LiveCharts.ChartValues<int>(daily.Select(d => d.Count)),
            PointGeometrySize = 6,
            StrokeThickness = 2
        });

        PieSeriesCollection.Clear();
        if (stats.Inserts > 0)
            PieSeriesCollection.Add(new LiveCharts.Wpf.PieSeries {
Title = "INSERT", Values = new LiveCharts.ChartValues<long> {
stats.Inserts }, DataLabels = true });
        if (stats.Updates > 0)
            PieSeriesCollection.Add(new LiveCharts.Wpf.PieSeries {
Title = "UPDATE", Values = new LiveCharts.ChartValues<long> {
stats.Updates }, DataLabels = true });
        if (stats.Deletes > 0)
            PieSeriesCollection.Add(new LiveCharts.Wpf.PieSeries {
Title = "DELETE", Values = new LiveCharts.ChartValues<long> {
```

```
stats.Deletes }, DataLabels = true });
    }

    [RelayCommand]
    private async Task Refresh() => await LoadAsync();
}

// ─── Data Sources ─

public partial class DataSourceViewModel : BaseViewModel,
ILoadable
{
    private readonly IDataSourceRepository _repo;
    private readonly IConnectionTestService _testSvc;
    private readonly INotificationService _notif;
    private readonly IDialogService _dialog;

    [ObservableProperty] private DataSource? _selectedSource;
    [ObservableProperty] private string _testResult =
string.Empty;
    [ObservableProperty] private bool _showTestResult;

    public ObservableCollection<DataSource> Sources { get; } = [];

    public DataSourceViewModel(IDataSourceRepository repo,
IConnectionTestService testSvc,
        INotificationService notif, IDialogService dialog)
    {
        _repo = repo; _testSvc = testSvc; _notif = notif; _dialog
= dialog;
    }

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            var list = await _repo.GetAllAsync();
            Sources.Clear();
            foreach (var s in list) Sources.Add(s);
        }
        finally { IsBusy = false; }
    }

    [RelayCommand]
    private async Task TestConnection()
    {
        if (SelectedSource == null) return;
        IsBusy = true;
        try
        {
            var (ok, msg) = await
_testSvc.TestAsync(SelectedSource);
```

```
            TestResult = msg;
            ShowTestResult = true;
            if (ok) _notif.ShowSuccess("Connection successful");
            else _notif.ShowError("Connection failed");
        }
        finally { IsBusy = false; }
    }

    [RelayCommand]
    private async Task Delete()
    {
        if (SelectedSource == null) return;
        if (!_dialog.Confirm($"Delete data source
'{SelectedSource.Name}'?")) return;
        await _repo.DeleteAsync(SelectedSource.Id);
        Sources.Remove(SelectedSource);
        SelectedSource = null;
        _notif.ShowSuccess("Data source deleted");
    }

    // Add/Edit are handled via dialog window — see
DataSourceDialogViewModel
    [RelayCommand]
    private void AddNew()
    {
        var win =
App.GetService<Views.Dialogs.DataSourceDialog>();
        win.SetMode(null);
        if (win.ShowDialog() == true)
        {
            _ = LoadAsync();
            _notif.ShowSuccess("Data source added");
        }
    }

    [RelayCommand]
    private void Edit()
    {
        if (SelectedSource == null) return;
        var win =
App.GetService<Views.Dialogs.DataSourceDialog>();
        win.SetMode(SelectedSource);
        if (win.ShowDialog() == true)
        {
            _ = LoadAsync();
            _notif.ShowSuccess("Data source updated");
        }
    }
}

// ─── Change Log ───

public partial class ChangeLogViewModel : BaseViewModel, ILoadable
```

```
{
    private readonly IChangeEventRepository _repo;

    [ObservableProperty] private DateTime? _filterFrom;
    [ObservableProperty] private DateTime? _filterTo;
    [ObservableProperty] private string _filterTable =
string.Empty;
    [ObservableProperty] private string _filterOperation =
string.Empty;
    [ObservableProperty] private ChangeEvent? _selectedEvent;
    [ObservableProperty] private int _currentPage = 1;
    [ObservableProperty] private int _totalCount;

    public int PageSize { get; } = 50;
    public ObservableCollection<ChangeEvent> Events { get; } = [];
    public List<string> OperationTypes { get; } = ["", "INSERT",
"UPDATE", "DELETE"];

    public ChangeLogViewModel(IChangeEventRepository repo) =>
_repo = repo;

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            TotalCount = await _repo.GetCountAsync(FilterFrom,
FilterTo, FilterTable, FilterOperation);
            var items = await _repo.GetFilteredAsync(FilterFrom,
FilterTo, FilterTable, FilterOperation, CurrentPage, PageSize);
            Events.Clear();
            foreach (var e in items) Events.Add(e);
        }
        finally { IsBusy = false; }
    }

    [RelayCommand] private async Task ApplyFilter() { CurrentPage
= 1; await LoadAsync(); }
    [RelayCommand] private async Task ClearFilter()
    {
        FilterFrom = null; FilterTo = null; FilterTable = "";
FilterOperation = "";
        CurrentPage = 1;
        await LoadAsync();
    }

    [RelayCommand] private async Task NextPage() { if (CurrentPage
* PageSize < TotalCount) { CurrentPage++; await LoadAsync(); } }
    [RelayCommand] private async Task PrevPage() { if (CurrentPage
> 1) { CurrentPage--; await LoadAsync(); } }
}

// ─── Automation Rules ───
```

```
public partial class AutomationRulesViewModel : BaseViewModel,
ILoadable
{
    private readonly IAutomationRuleRepository _repo;
    private readonly INotificationService _notif;
    private readonly IDialogService _dialog;

    [ObservableProperty] private AutomationRule? _selectedRule;
    public ObservableCollection<AutomationRule> Rules { get; } =
[];

    public AutomationRulesViewModel(IAutomationRuleRepository
repo, INotificationService notif, IDialogService dialog)
    {
        _repo = repo; _notif = notif; _dialog = dialog;
    }

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            var list = await _repo.GetAllWithActionsAsync();
            Rules.Clear();
            foreach (var r in list) Rules.Add(r);
        }
        finally { IsBusy = false; }
    }

    [RelayCommand] private void AddNew()
    {
        var win = App.GetService<Views.Dialogs.RuleDialog>();
        win.SetMode(null);
        if (win.ShowDialog() == true) { _ = LoadAsync();
_notif.ShowSuccess("Rule created"); }
    }

    [RelayCommand] private void Edit()
    {
        if (SelectedRule == null) return;
        var win = App.GetService<Views.Dialogs.RuleDialog>();
        win.SetMode(SelectedRule);
        if (win.ShowDialog() == true) { _ = LoadAsync();
_notif.ShowSuccess("Rule updated"); }
    }

    [RelayCommand] private async Task Delete()
    {
        if (SelectedRule == null) return;
        if (!_dialog.Confirm($"Delete rule
'{SelectedRule.RuleName}'?")) return;
        await _repo.DeleteAsync(SelectedRule.Id);
```

```
        Rules.Remove(SelectedRule);
        SelectedRule = null;
        _notif.ShowSuccess("Rule deleted");
    }

    [RelayCommand] private async Task ToggleEnabled()
    {
        if (SelectedRule == null) return;
        SelectedRule.IsEnabled = !SelectedRule.IsEnabled;
        await _repo.UpdateAsync(SelectedRule);
        _notif.ShowSuccess(SelectedRule.IsEnabled ? "Rule enabled"
: "Rule disabled");
    }
}

// ─── Executions ───

public partial class ExecutionsViewModel : BaseViewModel,
ILoadable
{
    private readonly IExecutionHistoryRepository _repo;

    [ObservableProperty] private string _filterRuleName =
string.Empty;
    [ObservableProperty] private string _filterStatus =
string.Empty;
    [ObservableProperty] private DateTime? _filterFrom;
    [ObservableProperty] private DateTime? _filterTo;
    [ObservableProperty] private int _currentPage = 1;
    [ObservableProperty] private int _totalCount;

    public int PageSize { get; } = 50;
    public ObservableCollection<ExecutionHistory> Executions {
get; } = [];
    public List<string> StatusOptions { get; } = ["", "Success",
"Failed"];

    public ExecutionsViewModel(IExecutionHistoryRepository repo)
=> _repo = repo;

    public async Task LoadAsync()
    {
        IsBusy = true;
        try
        {
            TotalCount = await _repo.GetCountAsync(FilterRuleName,
FilterStatus, FilterFrom, FilterTo);
            var items = await
_repo.GetFilteredAsync(FilterRuleName, FilterStatus, FilterFrom,
FilterTo, CurrentPage, PageSize);
            Executions.Clear();
            foreach (var e in items) Executions.Add(e);
        }
```

```
        finally { IsBusy = false; }
    }

    [RelayCommand] private async Task ApplyFilter() { CurrentPage
= 1; await LoadAsync(); }
    [RelayCommand] private async Task ClearFilter()
    {
        FilterRuleName = ""; FilterStatus = ""; FilterFrom = null;
FilterTo = null;
        CurrentPage = 1;
        await LoadAsync();
    }
    [RelayCommand] private async Task NextPage() { if (CurrentPage
* PageSize < TotalCount) { CurrentPage++; await LoadAsync(); } }
    [RelayCommand] private async Task PrevPage() { if (CurrentPage
> 1) { CurrentPage--; await LoadAsync(); } }
}

// ─── Settings ─────

public partial class SettingsViewModel : BaseViewModel, ILoadable
{
    private readonly ISettingsRepository _repo;
    private readonly INotificationService _notif;
    private ApplicationSettings _settings = new();

    [ObservableProperty] private int _pollingInterval = 10;
    [ObservableProperty] private int _retentionDays = 30;
    [ObservableProperty] private string _loggingLevel =
"Information";
    [ObservableProperty] private string _exportDirectory =
"C:\\ChangeTrackerExports";

    public List<string> LoggingLevels { get; } = ["Verbose",
"Debug", "Information", "Warning", "Error"];

    public SettingsViewModel(ISettingsRepository repo,
INotificationService notif)
    {
        _repo = repo; _notif = notif;
    }

    public async Task LoadAsync()
    {
        _settings = await _repo.GetAsync();
        PollingInterval = _settings.CdcPollingIntervalSeconds;
        RetentionDays = _settings.LogRetentionDays;
        LoggingLevel = _settings.LoggingLevel;
        ExportDirectory = _settings.ExportDirectory;
    }

    [RelayCommand] private async Task Save()
    {
```

```
        _settings.CdcPollingIntervalSeconds = PollingInterval;
        _settings.LogRetentionDays = RetentionDays;
        _settings.LoggingLevel = LoggingLevel;
        _settings.ExportDirectory = ExportDirectory;
        await _repo.UpdateAsync(_settings);
        _notif.ShowSuccess("Settings saved successfully");
    }

    [RelayCommand] private void BrowseExportDir()
    {
        var dlg = new System.Windows.Forms.FolderBrowserDialog
        {
            Description = "Select export directory",
            SelectedPath = ExportDirectory
        };
        if (dlg.ShowDialog() ==
System.Windows.Forms.DialogResult.OK)
            ExportDirectory = dlg.SelectedPath;
    }
}

// ─── Dialog ViewModels ──

public partial class DataSourceDialogViewModel : BaseViewModel
{
    private readonly IDataSourceRepository _repo;
    private DataSource? _editing;

    [ObservableProperty] private string _name = string.Empty;
    [ObservableProperty] private string _server = string.Empty;
    [ObservableProperty] private string _database = string.Empty;
    [ObservableProperty] private string _username = string.Empty;
    [ObservableProperty] private string _password = string.Empty;
    [ObservableProperty] private bool _isActive = true;
    [ObservableProperty] private string _title = "Add Data
Source";

    public DataSourceDialogViewModel(IDataSourceRepository repo)
=> _repo = repo;

    public void Load(DataSource? source)
    {
        _editing = source;
        Title = source == null ? "Add Data Source" : "Edit Data
Source";
        if (source == null) return;
        Name = source.Name; Server = source.Server; Database =
source.Database;
        Username = source.Username; Password = source.Password;
IsActive = source.IsActive;
    }

    public async Task<bool> SaveAsync()
```

```
    {
        if (string.IsNullOrWhiteSpace(Name) ||
string.IsNullOrWhiteSpace(Server) ||
string.IsNullOrWhiteSpace(Database))
            return false;

        if (_editing == null)
        {
            await _repo.AddAsync(new DataSource
            {
                Name = Name, Server = Server, Database = Database,
                Username = Username, Password = Password, IsActive
= IsActive
            });
        }
        else
        {
            _editing.Name = Name; _editing.Server = Server;
_editing.Database = Database;
            _editing.Username = Username; _editing.Password =
Password; _editing.IsActive = IsActive;
            await _repo.UpdateAsync(_editing);
        }
        return true;
    }
}

public partial class RuleDialogViewModel : BaseViewModel
{
    private readonly IAutomationRuleRepository _repo;
    private AutomationRule? _editing;

    [ObservableProperty] private string _ruleName = string.Empty;
    [ObservableProperty] private bool _isEnabled = true;
    [ObservableProperty] private string _tableName = string.Empty;
    [ObservableProperty] private string _eventType = "ALL";
    [ObservableProperty] private string _title = "Add Rule";
    [ObservableProperty] private RuleAction? _selectedAction;

    public List<string> EventTypes { get; } = ["ALL", "INSERT",
"UPDATE", "DELETE"];
    public List<string> ActionTypes { get; } = ["HTTP_GET",
"HTTP_POST", "WRITE_FILE", "EXEC_PROCESS"];
    public ObservableCollection<RuleAction> Actions { get; } = [];

    public RuleDialogViewModel(IAutomationRuleRepository repo) =>
_repo = repo;

    public void Load(AutomationRule? rule)
    {
        _editing = rule;
        Title = rule == null ? "Add Rule" : "Edit Rule";
        if (rule == null) return;
```

```
        RuleName = rule.RuleName; IsEnabled = rule.IsEnabled;
        TableName = rule.TableName; EventType = rule.EventType;
        Actions.Clear();
        foreach (var a in rule.Actions.OrderBy(x => x.SortOrder))
Actions.Add(a);
    }

    [RelayCommand] private void AddAction()
    {
        Actions.Add(new RuleAction { ActionType = "HTTP_GET",
SortOrder = Actions.Count });
    }

    [RelayCommand] private void RemoveAction()
    {
        if (SelectedAction != null)
Actions.Remove(SelectedAction);
    }

    public async Task<bool> SaveAsync()
    {
        if (string.IsNullOrWhiteSpace(RuleName)) return false;

        if (_editing == null)
        {
            var rule = new AutomationRule
            {
                RuleName = RuleName, IsEnabled = IsEnabled,
                TableName = TableName, EventType = EventType
            };
            foreach (var (a, i) in Actions.Select((a, i) => (a,
i))) { a.SortOrder = i; rule.Actions.Add(a); }
            await _repo.AddAsync(rule);
        }
        else
        {
            _editing.RuleName = RuleName; _editing.IsEnabled =
IsEnabled;
            _editing.TableName = TableName; _editing.EventType =
EventType;
            _editing.Actions.Clear();
            foreach (var (a, i) in Actions.Select((a, i) => (a,
i))) { a.SortOrder = i; _editing.Actions.Add(a); }
            await _repo.UpdateAsync(_editing);
        }
        return true;
    }
}
```

DataSourceDialog.xaml

```
<Window x:Class="ChangeTracker.Views.Dialogs.DataSourceDialog"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding Title}" Height="610" Width="480"
        WindowStartupLocation="CenterOwner"
        ResizeMode="NoResize"
        Background="{StaticResource BackgroundBrush}"
        FontFamily="{StaticResource AppFont}">
    <Grid Margin="28">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <!-- Title -->
        <TextBlock Grid.Row="0" Text="{Binding Title}"
FontSize="18" FontWeight="Bold"
                   Foreground="{StaticResource TextPrimaryBrush}"
Margin="0,0,0,20"/>

        <!-- Form -->
        <Border Grid.Row="1" Style="{StaticResource CardStyle}"
Padding="20">
            <StackPanel>
                <Label Content="CONNECTION NAME"/>
                <TextBox Text="{Binding Name,
UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,12"/>

                <Label Content="SERVER"/>
                <TextBox Text="{Binding Server,
UpdateSourceTrigger=PropertyChanged}"
                         Margin="0,0,0,4"/>
                <TextBlock Text="e.g. localhost, .\SQLEXPRESS,
server.domain.com\Instance"
                           FontSize="11"
Foreground="{StaticResource TextSecondaryBrush}"
Margin="0,0,0,12"/>

                <Label Content="DATABASE"/>
                <TextBox Text="{Binding Database,
UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,12"/>

                <Grid Margin="0,0,0,12">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*"/>
                        <ColumnDefinition Width="12"/>
                        <ColumnDefinition Width="*"/>
                    </Grid.ColumnDefinitions>
                    <StackPanel Grid.Column="0">
                        <Label Content="USERNAME (optional)"/>
                        <TextBox Text="{Binding Username,
UpdateSourceTrigger=PropertyChanged}"/>
                    </StackPanel>
```

```
                    <StackPanel Grid.Column="2">
                        <Label Content="PASSWORD (optional)"/>
                        <PasswordBox x:Name="PwdBox"/>
                    </StackPanel>
                </Grid>

                <CheckBox Content="Active — include in CDC
polling" IsChecked="{Binding IsActive}"/>
            </StackPanel>
        </Border>

        <!-- Buttons -->
        <StackPanel Grid.Row="2" Orientation="Horizontal"
HorizontalAlignment="Right" Margin="0,16,0,0">
            <Button Content="Cancel" Style="{StaticResource
SecondaryButton}"
                    Click="Cancel_Click" Margin="0,0,10,0"/>
            <Button Content="Save" Style="{StaticResource
PrimaryButton}"
                    Click="Save_Click" IsDefault="True"/>
        </StackPanel>
    </Grid>
</Window>
```

RuleDialog.xaml

```
<Window x:Class="ChangeTracker.Views.Dialogs.RuleDialog"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding Title}" Height="640" Width="700"
        WindowStartupLocation="CenterOwner"
        Background="{StaticResource BackgroundBrush}"
        FontFamily="{StaticResource AppFont}">
    <Grid Margin="24">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <!-- Title -->
        <TextBlock Grid.Row="0" Text="{Binding Title}"
FontSize="18" FontWeight="Bold"
                   Foreground="{StaticResource TextPrimaryBrush}"
Margin="0,0,0,16"/>

        <!-- Rule fields -->
        <Border Grid.Row="1" Style="{StaticResource CardStyle}"
Padding="20" Margin="0,0,0,12">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
```

```
                    <ColumnDefinition Width="12"/>
                    <ColumnDefinition Width="160"/>
                    <ColumnDefinition Width="12"/>
                    <ColumnDefinition Width="160"/>
                </Grid.ColumnDefinitions>
                <StackPanel Grid.Column="0">
                    <Label Content="RULE NAME"/>
                    <TextBox Text="{Binding RuleName,
UpdateSourceTrigger=PropertyChanged}"/>
                </StackPanel>
                <StackPanel Grid.Column="2">
                    <Label Content="TABLE NAME (* = any)"/>
                    <TextBox Text="{Binding TableName,
UpdateSourceTrigger=PropertyChanged}"/>
                </StackPanel>
                <StackPanel Grid.Column="4">
                    <Label Content="EVENT TYPE"/>
                    <ComboBox SelectedValue="{Binding EventType}"
ItemsSource="{Binding EventTypes}"/>
                </StackPanel>
            </Grid>
        </Border>

        <!-- Actions list -->
        <Border Grid.Row="2" Style="{StaticResource CardStyle}"
Padding="16">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="Auto"/>
                </Grid.RowDefinitions>

                <Grid Grid.Row="0" Margin="0,0,0,12">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*"/>
                        <ColumnDefinition Width="Auto"/>
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="Actions" FontSize="14"
FontWeight="SemiBold"
                               Foreground="{StaticResource
TextPrimaryBrush}" VerticalAlignment="Center"/>
                    <StackPanel Grid.Column="1"
Orientation="Horizontal">
                        <Button Content="+ Add Action"
Style="{StaticResource SecondaryButton}"
                                Command="{Binding
AddActionCommand}" Margin="0,0,8,0"/>
                        <Button Content="Remove"
Style="{StaticResource DangerButton}"
                                Command="{Binding
RemoveActionCommand}"
                                IsEnabled="{Binding
```

```
SelectedAction, Converter={StaticResource NullToVisibility}}"/>
                    </StackPanel>
                </Grid>

                <ScrollViewer Grid.Row="1"
VerticalScrollBarVisibility="Auto">
                    <ItemsControl ItemsSource="{Binding Actions}">
                        <ItemsControl.ItemTemplate>
                            <DataTemplate>
                                <Border Margin="0,0,0,10"
Padding="14" CornerRadius="6"

BorderBrush="{StaticResource BorderBrush}" BorderThickness="1"
                                        Background="White">
                                    <StackPanel>
                                        <!-- Action type selector
-->
                                        <Grid Margin="0,0,0,10">

<Grid.ColumnDefinitions>
                                                <ColumnDefinition
Width="Auto"/>
                                                <ColumnDefinition
Width="200"/>

</Grid.ColumnDefinitions>
                                            <Label Content="ACTION
TYPE   " VerticalAlignment="Center"/>
                                            <ComboBox
Grid.Column="1"

SelectedValue="{Binding ActionType,
UpdateSourceTrigger=PropertyChanged}"

ItemsSource="{Binding DataContext.ActionTypes,

RelativeSource={RelativeSource AncestorType=Window}}"/>
                                        </Grid>

                                        <!-- HTTP fields -->
                                        <StackPanel
Visibility="{Binding ActionType,

Converter={StaticResource NullToVisibility}}">
                                            <Grid
Margin="0,0,0,8">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="60"/>

<ColumnDefinition Width="*"/>
```

```
<ColumnDefinition Width="80"/>

<ColumnDefinition Width="80"/>

</Grid.ColumnDefinitions>
                                                    <Label
Content="URL" VerticalAlignment="Center"/>
                                                    <TextBox
Grid.Column="1"

Text="{Binding Url, UpdateSourceTrigger=PropertyChanged}"

Margin="0,0,8,0"/>
                                                    <Label
Grid.Column="2" Content="TIMEOUT" VerticalAlignment="Center"/>
                                                    <TextBox
Grid.Column="3"

Text="{Binding TimeoutSeconds,
UpdateSourceTrigger=PropertyChanged}"/>
                                                </Grid>
                                                <Grid
Margin="0,0,0,8">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="60"/>

<ColumnDefinition Width="*"/>

</Grid.ColumnDefinitions>
                                                    <Label
Content="HEADERS" VerticalAlignment="Center" Margin="0,4,0,0"/>
                                                    <TextBox
Grid.Column="1"

Text="{Binding Headers, UpdateSourceTrigger=PropertyChanged}"/>
                                                </Grid>
                                                <Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="60"/>

<ColumnDefinition Width="*"/>

</Grid.ColumnDefinitions>
                                                    <Label
Content="BODY" VerticalAlignment="Center" Margin="0,4,0,0"/>
                                                    <TextBox
Grid.Column="1"

Text="{Binding RequestBody, UpdateSourceTrigger=PropertyChanged}"
```

```
/>
                                        </Grid>
                                    </StackPanel>

                                    <!-- File fields -->
                                    <Grid Margin="0,4,0,0">

<Grid.ColumnDefinitions>
                                            <ColumnDefinition
Width="80"/>
                                            <ColumnDefinition
Width="*"/>

</Grid.ColumnDefinitions>
                                        <Grid.RowDefinitions>
                                            <RowDefinition
Height="Auto"/>
                                            <RowDefinition
Height="Auto"/>
                                        </Grid.RowDefinitions>
                                        <Label Grid.Row="0"
Grid.Column="0" Content="FILE PATH" VerticalAlignment="Center"/>
                                        <TextBox Grid.Row="0"
Grid.Column="1"

Text="{Binding FilePath, UpdateSourceTrigger=PropertyChanged}"

Margin="0,0,0,6"/>
                                        <Label Grid.Row="1"
Grid.Column="0" Content="TEMPLATE" VerticalAlignment="Center"
Margin="0,4,0,0"/>
                                        <TextBox Grid.Row="1"
Grid.Column="1"

Text="{Binding TextTemplate,
UpdateSourceTrigger=PropertyChanged}"/>
                                    </Grid>

                                    <!-- Process fields -->
                                    <Grid Margin="0,4,0,0">

<Grid.ColumnDefinitions>
                                            <ColumnDefinition
Width="80"/>
                                            <ColumnDefinition
Width="*"/>

</Grid.ColumnDefinitions>
                                        <Grid.RowDefinitions>
                                            <RowDefinition
Height="Auto"/>
                                            <RowDefinition
Height="Auto"/>
```

```
                                                </Grid.RowDefinitions>
                                                <Label Grid.Row="0"
Grid.Column="0" Content="EXE PATH" VerticalAlignment="Center"/>
                                                <TextBox Grid.Row="0"
Grid.Column="1"

Text="{Binding ExecutablePath,
UpdateSourceTrigger=PropertyChanged}"

Margin="0,0,0,6"/>
                                                <Label Grid.Row="1"
Grid.Column="0" Content="ARGUMENTS" VerticalAlignment="Center"/>
                                                <TextBox Grid.Row="1"
Grid.Column="1"

Text="{Binding Arguments, UpdateSourceTrigger=PropertyChanged}"/>
                                            </Grid>

                                            <!-- Template tokens hint
-->
                                            <TextBlock
Margin="0,8,0,0" FontSize="11"

Foreground="{StaticResource TextSecondaryBrush}">
                                                Available tokens:
{{EventId}} {{Table}} {{Operation}} {{Database}} {{Timestamp}}
{{PrimaryKey}}
                                            </TextBlock>
                                        </StackPanel>
                                    </Border>
                                </DataTemplate>
                            </ItemsControl.ItemTemplate>
                        </ItemsControl>
                    </ScrollViewer>

                    <CheckBox Grid.Row="2" Content="Rule is enabled"
IsChecked="{Binding IsEnabled}" Margin="0,12,0,0"/>
                </Grid>
            </Border>

            <!-- Buttons -->
            <StackPanel Grid.Row="3" Orientation="Horizontal"
HorizontalAlignment="Right" Margin="0,16,0,0">
                <Button Content="Cancel" Style="{StaticResource
SecondaryButton}"
                        Click="Cancel_Click" Margin="0,0,10,0"/>
                <Button Content="Save Rule" Style="{StaticResource
PrimaryButton}"
                        Click="Save_Click" IsDefault="True"/>
            </StackPanel>
        </Grid>
</Window>
```

AutomationRulesView.xaml

```
<UserControl x:Class="ChangeTracker.Views.AutomationRulesView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Background="{StaticResource BackgroundBrush}">
    <Grid Margin="28,24">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!-- Header -->
        <Grid Grid.Row="0" Margin="0,0,0,20">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <StackPanel>
                <TextBlock Text="Automation Rules" FontSize="24"
FontWeight="Bold"
                           Foreground="{StaticResource
TextPrimaryBrush}"/>
                <TextBlock Text="Define rules that execute actions
when database events occur"
                           FontSize="13"
Foreground="{StaticResource TextSecondaryBrush}"
Margin="0,4,0,0"/>
            </StackPanel>
            <StackPanel Grid.Column="1" Orientation="Horizontal"
VerticalAlignment="Center">
                <Button Content="+ Add Rule"
Style="{StaticResource PrimaryButton}"
                        Command="{Binding AddNewCommand}"
Margin="0,0,8,0"/>
                <Button Content="✎ Edit" Style="{StaticResource
SecondaryButton}"
                        Command="{Binding EditCommand}"
Margin="0,0,8,0"
                        IsEnabled="{Binding SelectedRule,
Converter={StaticResource NullToVisibility}}"/>
                <Button Content="⊘ Toggle" Style="{StaticResource
SecondaryButton}"
                        Command="{Binding ToggleEnabledCommand}"
Margin="0,0,8,0"
                        IsEnabled="{Binding SelectedRule,
Converter={StaticResource NullToVisibility}}"/>
                <Button Content="⊗ Delete" Style="{StaticResource
DangerButton}"
                        Command="{Binding DeleteCommand}"
                        IsEnabled="{Binding SelectedRule,
```

```
Converter={StaticResource NullToVisibility}}"/>
            </StackPanel>
        </Grid>

        <!-- Grid + action detail split -->
        <Grid Grid.Row="1">
            <Grid.RowDefinitions>
                <RowDefinition Height="*"/>
                <RowDefinition Height="12"/>
                <RowDefinition Height="200"/>
            </Grid.RowDefinitions>

            <!-- Rules DataGrid -->
            <Border Grid.Row="0" Style="{StaticResource
CardStyle}">
                <DataGrid ItemsSource="{Binding Rules}"
                          SelectedItem="{Binding SelectedRule}"
                          IsReadOnly="True"
CanUserAddRows="False">
                    <DataGrid.Columns>
                        <DataGridTextColumn Header="Rule Name"
Binding="{Binding RuleName}"   Width="200"/>
                        <DataGridTextColumn Header="Table"
Binding="{Binding TableName}"  Width="160"/>
                        <DataGridTemplateColumn Header="Event
Type" Width="110">
                            <DataGridTemplateColumn.CellTemplate>
                                <DataTemplate>
                                    <Border CornerRadius="4"
Padding="8,3" HorizontalAlignment="Left"
                                            Background="{Binding
EventType, Converter={StaticResource OpToBrush}}">
                                        <TextBlock Text="{Binding
EventType}" FontSize="11"

FontWeight="Bold" Foreground="White"/>
                                    </Border>
                                </DataTemplate>
                            </DataGridTemplateColumn.CellTemplate>
                        </DataGridTemplateColumn>
                        <DataGridTemplateColumn Header="Status"
Width="100">
                            <DataGridTemplateColumn.CellTemplate>
                                <DataTemplate>
                                    <StackPanel
Orientation="Horizontal">
                                        <Ellipse Width="8"
Height="8" VerticalAlignment="Center"
                                                 Fill="{Binding
IsEnabled, Converter={StaticResource StatusToBrush}}"/>
                                        <TextBlock Text="{Binding
IsEnabled, Converter={StaticResource BoolToEnabledText}}"
                                                   FontSize="12"
```

```
Margin="6,0,0,0"

Foreground="{Binding IsEnabled, Converter={StaticResource
StatusToBrush}}"/>
                                    </StackPanel>
                                </DataTemplate>
                            </DataGridTemplateColumn.CellTemplate>
                        </DataGridTemplateColumn>
                        <DataGridTextColumn Header="Actions Count"
                                            Binding="{Binding
Actions.Count}" Width="110"/>
                        <DataGridTextColumn Header="Created"
                                            Binding="{Binding
CreatedAt, StringFormat={}{0:yyyy-MM-dd}}" Width="110"/>
                    </DataGrid.Columns>
                </DataGrid>
            </Border>

            <!-- Actions preview for selected rule -->
            <Border Grid.Row="2" Style="{StaticResource
CardStyle}" Padding="16"
                    Visibility="{Binding SelectedRule,
Converter={StaticResource NullToVisibility}}">
                <StackPanel>
                    <TextBlock Text="Actions for selected rule"
FontSize="13" FontWeight="SemiBold"
                               Foreground="{StaticResource
TextSecondaryBrush}" Margin="0,0,0,8"/>
                    <ScrollViewer
VerticalScrollBarVisibility="Auto">
                        <ItemsControl ItemsSource="{Binding
SelectedRule.Actions}">
                            <ItemsControl.ItemTemplate>
                                <DataTemplate>
                                    <Border Margin="0,0,0,6"
Padding="12,8" CornerRadius="6"
                                            Background="#F8F9FA"
BorderBrush="{StaticResource BorderBrush}"
                                            BorderThickness="1">
                                        <Grid>

<Grid.ColumnDefinitions>
                                                <ColumnDefinition
Width="Auto"/>
                                                <ColumnDefinition
Width="*"/>

</Grid.ColumnDefinitions>
                                            <Border
Grid.Column="0" CornerRadius="4" Padding="8,3"

Background="{StaticResource PrimaryBrush}" Margin="0,0,12,0">
                                                <TextBlock
```

```
Text="{Binding ActionType}" FontSize="11"

FontWeight="Bold" Foreground="White"/>
                                                    </Border>
                                                    <TextBlock
Grid.Column="1" VerticalAlignment="Center" FontSize="12"

Foreground="{StaticResource TextSecondaryBrush}">
                                                        <Run
Text="{Binding Url}"/>
                                                        <Run
Text="{Binding FilePath}"/>
                                                        <Run
Text="{Binding ExecutablePath}"/>
                                                    </TextBlock>
                                                </Grid>
                                            </Border>
                                        </DataTemplate>
                                    </ItemsControl.ItemTemplate>
                                </ItemsControl>
                            </ScrollViewer>
                        </StackPanel>
                    </Border>
                </Grid>
            </Grid>
        </UserControl>
```

ChangeLogView.xaml

```
<UserControl x:Class="ChangeTracker.Views.ChangeLogView"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Background="{StaticResource BackgroundBrush}">
    <Grid Margin="28,24">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <!-- Header -->
        <StackPanel Grid.Row="0" Margin="0,0,0,20">
            <TextBlock Text="Change Log" FontSize="24"
FontWeight="Bold"
                       Foreground="{StaticResource
TextPrimaryBrush}"/>
            <TextBlock Text="All captured CDC change events from
monitored databases"
                       FontSize="13" Foreground="{StaticResource
TextSecondaryBrush}" Margin="0,4,0,0"/>
        </StackPanel>
```

```
        <!-- Filter bar -->
        <Border Grid.Row="1" Style="{StaticResource CardStyle}"
Padding="16,12" Margin="0,0,0,12">
            <StackPanel Orientation="Horizontal">
                <StackPanel Margin="0,0,16,0">
                    <Label Content="DATE FROM"/>
                    <DatePicker SelectedDate="{Binding
FilterFrom}" Width="140" Height="34"/>
                </StackPanel>
                <StackPanel Margin="0,0,16,0">
                    <Label Content="DATE TO"/>
                    <DatePicker SelectedDate="{Binding FilterTo}"
Width="140" Height="34"/>
                </StackPanel>
                <StackPanel Margin="0,0,16,0">
                    <Label Content="TABLE NAME"/>
                    <TextBox Text="{Binding FilterTable,
UpdateSourceTrigger=PropertyChanged}"
                             Width="160" Height="34"/>
                </StackPanel>
                <StackPanel Margin="0,0,16,0">
                    <Label Content="OPERATION"/>
                    <ComboBox SelectedValue="{Binding
FilterOperation}"
                              ItemsSource="{Binding
OperationTypes}" Width="120"/>
                </StackPanel>
                <StackPanel Orientation="Horizontal"
VerticalAlignment="Bottom" Margin="0,0,0,0">
                    <Button Content="Apply Filter"
Style="{StaticResource PrimaryButton}"
                            Command="{Binding ApplyFilterCommand}"
Margin="0,0,8,0"/>
                    <Button Content="Clear" Style="{StaticResource
SecondaryButton}"
                            Command="{Binding
ClearFilterCommand}"/>
                </StackPanel>
            </StackPanel>
        </Border>

        <!-- Main content: grid + detail panel -->
        <Grid Grid.Row="2">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="12"/>
                <ColumnDefinition Width="380"
                    x:Name="DetailColumn"/>
            </Grid.ColumnDefinitions>

            <!-- Events grid -->
            <Border Grid.Column="0" Style="{StaticResource
```

```
CardStyle}">
                <DataGrid ItemsSource="{Binding Events}"
                          SelectedItem="{Binding SelectedEvent}"
                          IsReadOnly="True"
CanUserAddRows="False">
                    <DataGrid.Columns>
                        <DataGridTextColumn Header="Event ID"
Binding="{Binding Id}"          Width="80"/>
                        <DataGridTextColumn Header="Timestamp"
                                            Binding="{Binding
Timestamp, StringFormat={}{0:MM-dd HH:mm:ss}}"
                                            Width="130"/>
                        <DataGridTextColumn Header="Database"
Binding="{Binding DatabaseName}" Width="120"/>
                        <DataGridTextColumn Header="Table"
Binding="{Binding TableName}"    Width="130"/>
                        <DataGridTemplateColumn Header="Operation"
Width="90">
                            <DataGridTemplateColumn.CellTemplate>
                                <DataTemplate>
                                    <Border CornerRadius="4"
Padding="8,3" HorizontalAlignment="Left"
                                            Background="{Binding
OperationType, Converter={StaticResource OpToBrush}}">
                                        <TextBlock Text="{Binding
OperationType}" FontSize="11"

FontWeight="Bold" Foreground="White"/>
                                    </Border>
                                </DataTemplate>
                            </DataGridTemplateColumn.CellTemplate>
                        </DataGridTemplateColumn>
                        <DataGridTextColumn Header="Primary Key"
Binding="{Binding PrimaryKeyValue}" Width="110"/>
                        <DataGridTemplateColumn Header="Processed"
Width="90">
                            <DataGridTemplateColumn.CellTemplate>
                                <DataTemplate>
                                    <TextBlock Text="{Binding
IsProcessed, Converter={StaticResource BoolToEnabledText}}"
                                               FontSize="12"

Foreground="{Binding IsProcessed, Converter={StaticResource
StatusToBrush}}"/>
                                </DataTemplate>
                            </DataGridTemplateColumn.CellTemplate>
                        </DataGridTemplateColumn>
                    </DataGrid.Columns>
                </DataGrid>
            </Border>

            <!-- Detail panel -->
            <Border Grid.Column="2" Style="{StaticResource
```

```
CardStyle}" Padding="16"
                    Visibility="{Binding SelectedEvent,
Converter={StaticResource NullToVisibility}}">
                <StackPanel>
                    <TextBlock Text="Event Details" FontSize="15"
FontWeight="SemiBold"
                               Foreground="{StaticResource
TextPrimaryBrush}" Margin="0,0,0,12"/>

                    <!-- Meta info -->
                    <Grid Margin="0,0,0,12">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="*"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <Grid.RowDefinitions>
                            <RowDefinition Height="Auto"/>
                            <RowDefinition Height="Auto"/>
                        </Grid.RowDefinitions>
                        <StackPanel Grid.Row="0" Grid.Column="0"
Margin="0,0,8,8">
                            <Label Content="TABLE"/>
                            <TextBlock Text="{Binding
SelectedEvent.TableName}" FontSize="13"
                                       FontWeight="SemiBold"
Foreground="{StaticResource TextPrimaryBrush}"/>
                        </StackPanel>
                        <StackPanel Grid.Row="0" Grid.Column="1"
Margin="0,0,0,8">
                            <Label Content="OPERATION"/>
                            <Border CornerRadius="4" Padding="8,3"
HorizontalAlignment="Left"
                                    Background="{Binding
SelectedEvent.OperationType, Converter={StaticResource
OpToBrush}}">
                                <TextBlock Text="{Binding
SelectedEvent.OperationType}"
                                           FontSize="11"
FontWeight="Bold" Foreground="White"/>
                            </Border>
                        </StackPanel>
                        <StackPanel Grid.Row="1" Grid.Column="0"
Margin="0,0,8,0">
                            <Label Content="TIMESTAMP"/>
                            <TextBlock Text="{Binding
SelectedEvent.Timestamp, StringFormat={}{0:yyyy-MM-dd HH:mm:ss}}"
                                       FontSize="12"
Foreground="{StaticResource TextPrimaryBrush}"/>
                        </StackPanel>
                        <StackPanel Grid.Row="1" Grid.Column="1">
                            <Label Content="PRIMARY KEY"/>
                            <TextBlock Text="{Binding
SelectedEvent.PrimaryKeyValue}"
```

```
                                        FontSize="12"
Foreground="{StaticResource TextPrimaryBrush}"/>
                        </StackPanel>
                    </Grid>

                    <Separator Margin="0,0,0,12"/>

                    <!-- Column changes header -->
                    <Grid Margin="0,0,0,6">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="*"/>
                            <ColumnDefinition Width="*"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <TextBlock Grid.Column="0" Text="COLUMN"
FontSize="11" FontWeight="SemiBold"
                                   Foreground="{StaticResource
TextSecondaryBrush}"/>
                        <TextBlock Grid.Column="1" Text="OLD
VALUE" FontSize="11" FontWeight="SemiBold"
                                   Foreground="{StaticResource
TextSecondaryBrush}"/>
                        <TextBlock Grid.Column="2" Text="NEW
VALUE" FontSize="11" FontWeight="SemiBold"
                                   Foreground="{StaticResource
TextSecondaryBrush}"/>
                    </Grid>

                    <ScrollViewer MaxHeight="360"
VerticalScrollBarVisibility="Auto">
                        <ItemsControl ItemsSource="{Binding
SelectedEvent.Details}">
                            <ItemsControl.ItemTemplate>
                                <DataTemplate>
                                    <Grid Margin="0,0,0,6">
                                        <Grid.ColumnDefinitions>
                                            <ColumnDefinition
Width="*"/>
                                            <ColumnDefinition
Width="*"/>
                                            <ColumnDefinition
Width="*"/>
                                        </Grid.ColumnDefinitions>
                                        <TextBlock Grid.Column="0"
Text="{Binding ColumnName}"
                                                   FontSize="12"
Foreground="{StaticResource TextPrimaryBrush}"

FontWeight="SemiBold"/>
                                        <Border Grid.Column="1"
Background="#FFF3CD" CornerRadius="3"
                                                Padding="4,2"
Margin="4,0">
```

```
                                                <TextBlock
Text="{Binding OldValue}" FontSize="11"

Foreground="#856404" FontFamily="{StaticResource MonoFont}"

TextWrapping="Wrap"/>
                                            </Border>
                                            <Border Grid.Column="2"
Background="#D1FAE5" CornerRadius="3"
                                                    Padding="4,2"
Margin="4,0,0,0">
                                                <TextBlock
Text="{Binding NewValue}" FontSize="11"

Foreground="#065F46" FontFamily="{StaticResource MonoFont}"

TextWrapping="Wrap"/>
                                            </Border>
                                        </Grid>
                                    </DataTemplate>
                                </ItemsControl.ItemTemplate>
                            </ItemsControl>
                        </ScrollViewer>
                    </StackPanel>
                </Border>
            </Grid>

            <!-- Pagination -->
            <Border Grid.Row="3" Margin="0,12,0,0">
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*"/>
                        <ColumnDefinition Width="Auto"/>
                    </Grid.ColumnDefinitions>
                    <TextBlock VerticalAlignment="Center"
Foreground="{StaticResource TextSecondaryBrush}" FontSize="13">
                        <Run Text="Total: "/>
                        <Run Text="{Binding TotalCount}"
FontWeight="SemiBold"/>
                        <Run Text=" events  |  Page "/>
                        <Run Text="{Binding CurrentPage}"
FontWeight="SemiBold"/>
                    </TextBlock>
                    <StackPanel Grid.Column="1"
Orientation="Horizontal">
                        <Button Content="← Previous"
Style="{StaticResource SecondaryButton}"
                                Command="{Binding PrevPageCommand}"
Margin="0,0,8,0"/>
                        <Button Content="Next →"
Style="{StaticResource SecondaryButton}"
                                Command="{Binding NextPageCommand}"/>
                    </StackPanel>
```

```
            </Grid>
        </Border>
    </Grid>
</UserControl>
```

MainWindow.xaml

```
<Window x:Class="ChangeTracker.Views.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:ChangeTracker.ViewModels"
        xmlns:views="clr-namespace:ChangeTracker.Views"
        Title="ChangeTracker — CDC Monitor"
        Height="780" Width="1280"
        MinHeight="600" MinWidth="900"
        WindowStartupLocation="CenterScreen"
        Background="{StaticResource BackgroundBrush}"
        FontFamily="{StaticResource AppFont}">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="220"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <!-- ── Sidebar ── -->
        <Border Grid.Column="0" Background="{StaticResource
SidebarBrush}">
            <DockPanel>
                <!-- Logo -->
                <Border DockPanel.Dock="Top" Padding="20,24,20,20"
                        BorderBrush="#2D3748"
BorderThickness="0,0,0,1">
                    <StackPanel>
                        <TextBlock Text="⬡ ChangeTracker"
FontSize="18" FontWeight="Bold"
                                   Foreground="White"
Padding="0.5"/>
                        <TextBlock Text="CDC Monitor v1.0"
FontSize="11"
                                   Foreground="#64748B"
Margin="0,4,0,0"/>
                    </StackPanel>
                </Border>

                <!-- Nav Items -->
                <StackPanel DockPanel.Dock="Top"
Margin="12,16,12,0">
                    <views:NavButton Icon="&#xE80F;"
Label="Dashboard"    Page="Dashboard"    CurrentPage="{Binding
CurrentPage}"/>
                    <views:NavButton Icon="&#xE77B;" Label="Data
Sources"  Page="DataSources"  CurrentPage="{Binding
```

```
CurrentPage}"/>
                    <views:NavButton Icon="&#xE81C;" Label="Change
Log"    Page="ChangeLog"    CurrentPage="{Binding CurrentPage}"/>
                    <views:NavButton Icon="&#xE945;"
Label="Automation"    Page="AutomationRules" CurrentPage="{Binding
CurrentPage}"/>
                    <views:NavButton Icon="&#xE823;"
Label="Executions"    Page="Executions"   CurrentPage="{Binding
CurrentPage}"/>
                    <views:NavButton Icon="&#xE713;"
Label="Settings"      Page="Settings"     CurrentPage="{Binding
CurrentPage}"/>
                </StackPanel>

                <!-- Bottom status -->
                <Border DockPanel.Dock="Bottom" Padding="20,16"
BorderBrush="#2D3748" BorderThickness="0,1,0,0">
                    <StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <Ellipse Width="8" Height="8"
Fill="#34A853" VerticalAlignment="Center"/>
                            <TextBlock Text="Background service
running" FontSize="11"
                                       Foreground="#64748B"
Margin="8,0,0,0" VerticalAlignment="Center"/>
                        </StackPanel>
                    </StackPanel>
                </Border>
            </DockPanel>
        </Border>

        <!-- ── Content Area ── -->
        <Grid Grid.Column="1">
            <Grid.RowDefinitions>
                <RowDefinition Height="*"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>

            <!-- Page host -->
            <ContentControl Grid.Row="0" Content="{Binding
CurrentViewModel}" Margin="0">
                <ContentControl.Resources>
                    <DataTemplate DataType="{x:Type
vm:DashboardViewModel}">
                        <views:DashboardView/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type
vm:DataSourceViewModel}">
                        <views:DataSourcesView/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type
vm:ChangeLogViewModel}">
                        <views:ChangeLogView/>
```

```
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type
vm:AutomationRulesViewModel}">
                        <views:AutomationRulesView/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type
vm:ExecutionsViewModel}">
                        <views:ExecutionsView/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type
vm:SettingsViewModel}">
                        <views:SettingsView/>
                    </DataTemplate>
                </ContentControl.Resources>
            </ContentControl>

            <!-- Loading overlay -->
            <Border Grid.Row="0" Background="#80FFFFFF"
                    Visibility="{Binding CurrentViewModel.IsBusy,
Converter={StaticResource BoolToVisibility}}">
                <StackPanel HorizontalAlignment="Center"
VerticalAlignment="Center">
                    <TextBlock Text="Loading..." FontSize="16"
Foreground="{StaticResource PrimaryBrush}"
                               FontWeight="SemiBold"
HorizontalAlignment="Center"/>
                </StackPanel>
            </Border>

            <!-- Toast notification -->
            <Border Grid.Row="0" VerticalAlignment="Bottom"
HorizontalAlignment="Right"
                    Margin="0,0,24,24" CornerRadius="8"
Padding="16,12" MinWidth="280"
                    Visibility="{Binding ShowNotification,
Converter={StaticResource BoolToVisibility}}"
                    Panel.ZIndex="100">
                <Border.Background>
                    <SolidColorBrush Color="{Binding
NotificationColor}"/>
                </Border.Background>
                <Border.Effect>
                    <DropShadowEffect Color="#000" Opacity="0.2"
BlurRadius="16" ShadowDepth="4"/>
                </Border.Effect>
                <TextBlock Text="{Binding NotificationMessage}"
Foreground="White"
                           FontSize="13" FontWeight="SemiBold"/>
            </Border>
        </Grid>
    </Grid>
</Window>
```

ДОДАТОК В - Опис програмного забезпечення

Повна назва програмного забезпечення: ChangeTracker — програмне забезпечення для відстеження змін в інформаційній системі на основі технології Change Data Capture.

Програмне забезпечення ChangeTracker призначене для автоматизованого моніторингу змін у базах даних Microsoft SQL Server, збереження інформації про події зміни даних, перегляду журналу змін та виконання налаштованих користувачем дій у відповідь на визначені події.

Програмний продукт реалізовано як десктопний застосунок для операційної системи Windows. Для розробки використано мову програмування C#, платформу .NET 8 та технологію WPF для створення графічного інтерфейсу користувача. Архітектура програмного забезпечення побудована з використанням шаблону MVVM, що забезпечує розділення графічного інтерфейсу, логіки представлення, сервісної логіки та доступу до даних.

Для збереження службових даних застосунку використовується внутрішня база даних SQL Server LocalDB. У ній зберігаються налаштування джерел даних, журнал зафіксованих змін, деталі змін, правила автоматизації, дії правил, історія виконань та параметри роботи системи.

Для отримання інформації про зміни у цільових базах даних використовується технологія Change Data Capture. Вона дозволяє отримувати відомості про операції вставки, оновлення та видалення записів без необхідності внесення змін до прикладної логіки інформаційної системи.

Програмне забезпечення ChangeTracker призначене для виконання таких основних функцій:

– керування джерелами даних;

– перевірка підключення до бази даних Microsoft SQL Server;

– автоматичне отримання змін із CDC-таблиць;

– збереження зафіксованих подій у внутрішній базі даних застосунку;

– відображення журналу змін;

– перегляд детальної інформації про змінені поля;

– фільтрація журналу змін за датою, таблицею та типом операції;

– створення, редагування, видалення та перегляд правил автоматичної обробки подій;

– увімкнення та вимкнення окремих правил автоматизації;

– налаштування дій, які виконуються при спрацюванні правил;

– виконання HTTP GET-запитів;

– виконання HTTP POST-запитів;

– запис інформації про подію до файлу;

– запуск зовнішньої програми або команди;

– збереження історії виконання автоматичних дій;

– перегляд і фільтрація історії виконань;

– налаштування інтервалу опитування CDC-таблиць;

– налаштування терміну зберігання журналів;

– налаштування рівня логування;

– налаштування директорії експорту.

Основним користувачем програмного забезпечення є технічний спеціаліст, системний адміністратор або адміністратор бази даних, який відповідає за контроль змін в інформаційній системі та налаштування автоматичного реагування на ці зміни.

Логічна структура програмного забезпечення ChangeTracker побудована за багаторівневим принципом. Основними рівнями системи є:

– рівень представлення;

– рівень моделей представлення;

– сервісний рівень;

– рівень доступу до даних.

Рівень представлення реалізовано засобами WPF і містить XAML-представлення, які відповідають за відображення інтерфейсу користувача. До цього рівня належать головне вікно застосунку та сторінки Dashboard, Data Sources, Change Log, Automation, Executions і Settings. Представлення не містять

бізнес-логіки, а взаємодіють із моделями представлення через механізм прив’язки даних та команд.

Рівень моделей представлення реалізує логіку взаємодії між графічним інтерфейсом і сервісами. Кожна сторінка застосунку має відповідну ViewModel. Наприклад, DashboardViewModel відповідає за отримання статистики для інформаційної панелі, DataSourceViewModel — за керування джерелами даних, ChangeLogViewModel — за відображення та фільтрацію журналу змін, AutomationRulesViewModel — за роботу з правилами автоматизації, ExecutionsViewModel — за перегляд історії виконань, а SettingsViewModel — за налаштування параметрів системи.

Сервісний рівень містить компоненти, що реалізують основну прикладну логіку системи. До нього належать CdcPollingService, ActionExecutionEngine, ConnectionTestService, BackgroundWorkerService, NavigationService, NotificationService та DialogService.

CdcPollingService відповідає за отримання змін із цільової бази даних SQL Server. Він підключається до бази даних, у якій увімкнено Change Data Capture, читає CDC-таблиці та формує внутрішні об’єкти подій зміни. Для роботи з цільовою базою даних використовується ADO.NET, оскільки CDC-таблиці мають системний характер і потребують виконання спеціальних SQL-запитів.

ActionExecutionEngine відповідає за виконання дій, пов’язаних із правилами автоматизації. Він послідовно виконує всі дії правила, підтримує підстановку шаблонних змінних і зберігає результат кожної дії в історії виконань.

ConnectionTestService використовується для перевірки підключення до бази даних. Він дозволяє перевірити коректність параметрів підключення та визначити, чи доступна база даних для моніторингу.

BackgroundWorkerService виконує основну автоматизовану фонову обробку. Він запускається разом із застосунком і періодично виконує опитування активних джерел даних, зберігає нові події, зіставляє їх із правилами автоматизації, виконує налаштовані дії та очищує застарілі записи.

Рівень доступу до даних реалізовано з використанням Entity Framework Core та шаблону Repository. Усі операції з внутрішньою базою даних виконуються через репозиторії. Це дозволяє ізолювати логіку доступу до даних від інших частин програми.

Основними репозиторіями є:

– DataSourceRepository;

– ChangeEventRepository;

– AutomationRuleRepository;

– ExecutionHistoryRepository;

– SettingsRepository.

DataSourceRepository забезпечує роботу з джерелами даних. ChangeEventRepository відповідає за збереження, фільтрацію та отримання подій зміни. AutomationRuleRepository використовується для роботи з правилами автоматизації та пошуку правил, що відповідають конкретній події. ExecutionHistoryRepository зберігає та повертає інформацію про результати виконання дій. SettingsRepository забезпечує отримання та збереження загальних налаштувань системи.

Внутрішня база даних застосунку містить такі основні таблиці:

– DataSources;

– ChangeEvents;

– ChangeDetails;

– AutomationRules;

– RuleActions;

– ExecutionHistories;

– ApplicationSettings.

Для роботи програмного забезпечення ChangeTracker необхідні такі технічні та програмні засоби:

– операційна система Windows 10 або новіша;

– процесор із тактовою частотою не менше 2,5 ГГц;

– оперативна пам’ять не менше 8 ГБ;

– не менше 500 МБ вільного місця на диску;

– Microsoft SQL Server;

– .NET 8 Runtime;

– доступ до локальної мережі або мережі Інтернет.

Для розробки програмного забезпечення використовувалися такі технології та бібліотеки:

– C#;

– .NET 8;

– WPF;

– XAML;

– CommunityToolkit.Mvvm;

– Entity Framework Core;

– ADO.NET;

– Microsoft.Extensions.Hosting;

– Microsoft.Extensions.DependencyInjection;

– Serilog;

– SQL Server.

Програмне забезпечення працює з двома типами баз даних. Внутрішня база даних застосунку використовується для зберігання службової інформації ChangeTracker. Цільова база даних є базою інформаційної системи, зміни в якій необхідно відстежувати. Цільова база даних повинна підтримувати технологію Change Data Capture.

Запуск програмного забезпечення ChangeTracker виконується шляхом відкриття виконуваного файлу застосунку. Після запуску ініціалізується графічний інтерфейс, завантажуються налаштування, створюється контейнер залежностей, підключається внутрішня база даних і запускається фоновий сервіс обробки подій.

Фоновий сервіс починає роботу автоматично після запуску програми. Він періодично перевіряє активні джерела даних і виконує опитування CDC-таблиць. Інтервал опитування задається в налаштуваннях застосунку.

Для завершення роботи програми користувач закриває головне вікно. Після цього фоновий сервіс отримує сигнал завершення, припиняє виконання поточного циклу обробки та звільняє використовувані ресурси.

Програмне забезпечення ChangeTracker призначене для роботи з базами даних Microsoft SQL Server. Для коректного отримання змін у цільовій базі даних повинен бути увімкнений механізм Change Data Capture. Якщо CDC не налаштовано, застосунок не зможе отримувати інформацію про зміни у таблицях.

Також для роботи фонового сервісу необхідно, щоб користувач мав права доступу до відповідної бази даних і CDC-таблиць. У разі відсутності необхідних прав система не зможе виконати опитування або отримає помилку підключення.

Програмне забезпечення розраховане на використання в середовищі Windows і не призначене для запуску на інших операційних системах без додаткової адаптації. Для роботи застосунку також необхідна наявність .NET 8 Runtime та доступу до SQL Server.

ДОДАТОК Г - Інструкції програмісту, оператору й користувачеві

Інструкція програмісту

Програмне забезпечення ChangeTracker реалізовано як десктопний застосунок для операційної системи Windows. Основною мовою програмування є C#, платформою реалізації — .NET 8, а для побудови графічного інтерфейсу використано технологію WPF. Архітектура застосунку побудована з використанням шаблону MVVM, що дозволяє відокремити графічний інтерфейс користувача від логіки представлення, сервісної логіки та доступу до даних.

Для роботи з вихідним кодом програмісту необхідно мати встановлене середовище Visual Studio 2022 або новіше, .NET 8 SDK, Microsoft SQL Server або SQL Server Express, SQL Server LocalDB, а також засоби адміністрування баз даних, наприклад SQL Server Management Studio. Після відкриття проєкту необхідно відновити NuGet-пакети, зібрати рішення та переконатися, що у конфігураційному файлі задано коректний рядок підключення до внутрішньої бази даних застосунку.

Структура програмного забезпечення складається з кількох логічних частин. Представлення реалізовані у вигляді XAML-файлів і відповідають за відображення інтерфейсу користувача. Моделі представлення містять логіку взаємодії з інтерфейсом, команди та властивості, що прив’язуються до елементів вікон. Доменні моделі описують основні сутності системи: джерела даних, події змін, деталі змін, правила автоматизації, дії правил, історію виконань і налаштування. Сервіси реалізують прикладну логіку застосунку, зокрема опитування CDC-таблиць, виконання автоматичних дій, перевірку підключення до бази даних, навігацію та відображення повідомлень. Репозиторії відповідають за доступ до внутрішньої бази даних застосунку.

Доступ до внутрішньої бази даних реалізовано за допомогою Entity Framework Core. Для цього використовується клас AppDbContext, у якому описано набори сутностей і зв’язки між ними. Робота з цільовою базою даних, у якій увімкнено Change Data Capture, виконується окремо через ADO.NET. Це

пов’язано з тим, що CDC-таблиці мають системний характер і потребують виконання спеціальних SQL-запитів до таблиць схеми cdc.

Під час модифікації програмного забезпечення необхідно дотримуватися існуючої архітектури. Нові сторінки інтерфейсу слід створювати разом із відповідними ViewModel, а нову прикладну логіку доцільно розміщувати у сервісному рівні. Якщо потрібно додати новий тип автоматичної дії, необхідно розширити модель дії правила, додати обробник у ActionExecutionEngine, оновити інтерфейс створення правила та перевірити нову функціональність за допомогою тестових сценаріїв. Якщо змінюється структура внутрішньої бази даних, необхідно оновити відповідні доменні сутності, конфігурацію AppDbContext і створити міграцію Entity Framework Core.

Перед внесенням змін до вихідного коду рекомендується перевірити працездатність основних сценаріїв: запуск застосунку, підключення до бази даних, отримання CDC-подій, створення правила автоматизації, виконання дії та запис результату в історію виконань. Після модифікації коду необхідно повторно виконати тестування основних функцій, щоб переконатися, що зміни не порушили роботу інших частин системи.

Інструкція оператору

Оператор відповідає за встановлення, первинне налаштування та контроль працездатності програмного забезпечення ChangeTracker. Перед початком роботи необхідно переконатися, що комп’ютер відповідає мінімальним вимогам: встановлено операційну систему Windows 10 або новішу, .NET 8 Runtime, SQL Server LocalDB, а також забезпечено доступ до цільової бази даних Microsoft SQL Server, зміни в якій необхідно відстежувати.

Програмний продукт постачається у вигляді електронного архіву. Для встановлення необхідно розпакувати архів у вибрану директорію на комп’ютері користувача. У директорії повинні міститися виконуваний файл застосунку, конфігураційні файли, службові файли, а також за потреби папки для логів і експортованих результатів. Запуск програми виконується відкриттям виконуваного файлу ChangeTracker.

Після запуску відкривається головне вікно програмного забезпечення (рис. Г.1). У лівій частині вікна розміщено навігаційну панель із розділами Dashboard, Data Sources, Change Log, Automation, Executions та Settings. У нижній частині навігаційної панелі відображається стан фонового сервісу. Якщо фоновий сервіс працює, застосунок може автоматично виконувати опитування активних джерел даних.

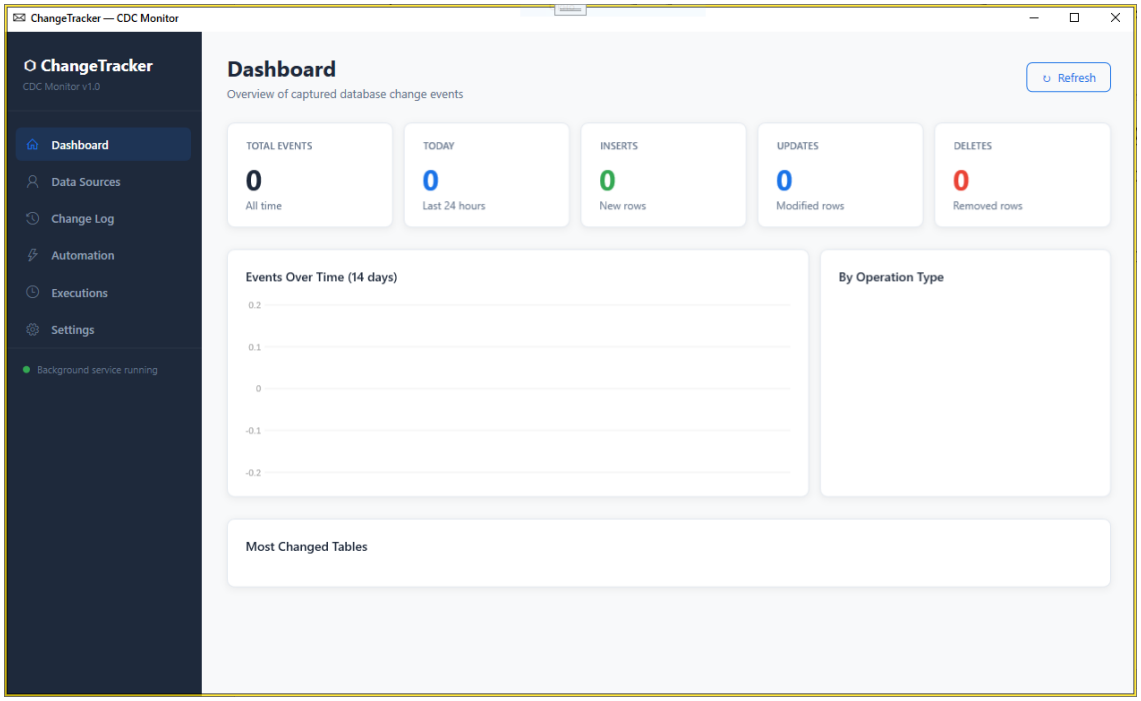


Рисунок Г.1 – Головне вікно програмного забезпечення

Першим етапом налаштування є додавання джерела даних (рис. Г.2). Для цього оператор переходить до розділу Data Sources і створює нове підключення. У формі необхідно вказати назву підключення, сервер SQL Server, назву бази даних, ім’я користувача, пароль і ознаку активності. Якщо для підключення використовується Windows-автентифікація, поля логіна й пароля можуть бути залишені порожніми відповідно до реалізованої логіки застосунку.

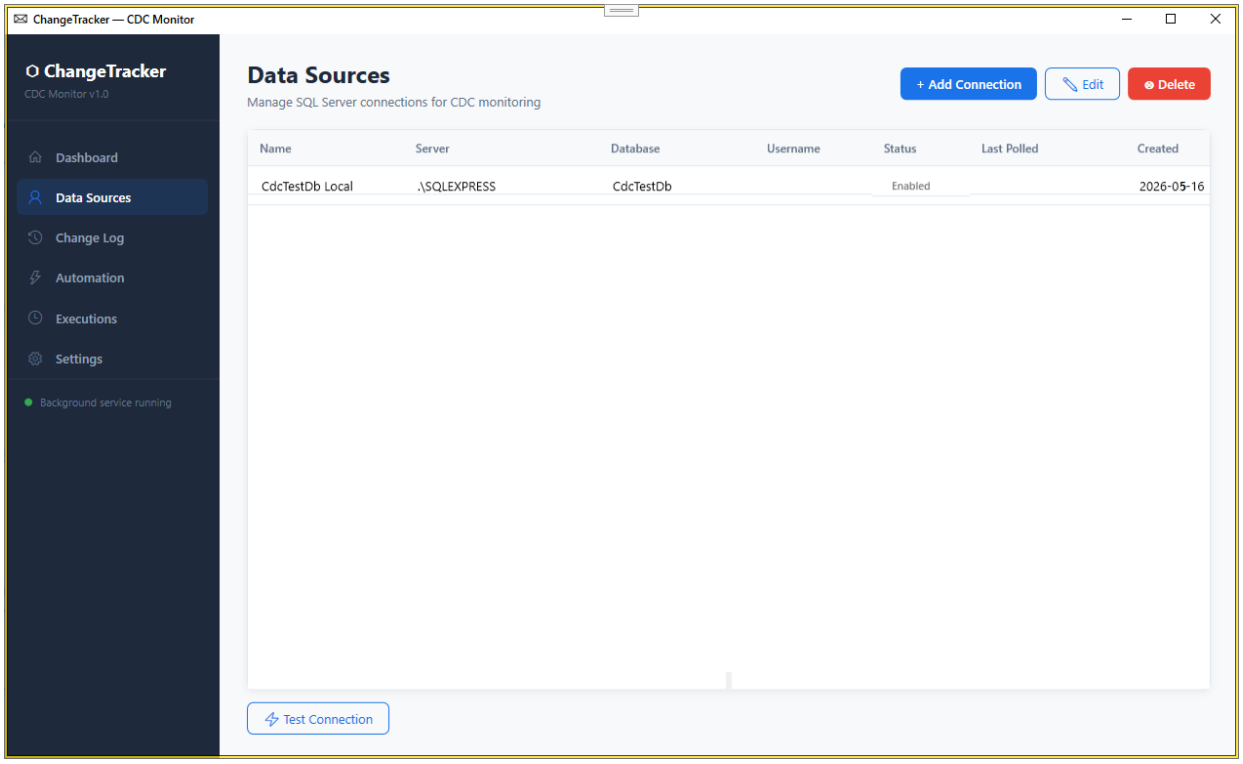


Рисунок Г.2 – Сторінка керування джерелами даних

Після введення параметрів підключення необхідно виконати перевірку з’єднання. Якщо параметри задано правильно, програма відобразить повідомлення про успішне підключення. Якщо підключення не встановлено, оператор повинен перевірити правильність назви сервера, назви бази даних, облікових даних, доступність SQL Server і права користувача. Також необхідно переконатися, що для цільової бази даних увімкнено механізм Change Data Capture.

Для коректної роботи ChangeTracker у цільовій базі даних повинен бути налаштований CDC. Оператор повинен перевірити, що Change Data Capture увімкнено на рівні бази даних і на рівні таблиць, зміни в яких потрібно відстежувати. Крім того, користувач, від імені якого виконується підключення, повинен мати права на читання необхідних CDC-таблиць. Якщо CDC не увімкнено, застосунок не зможе отримувати інформацію про вставку, оновлення та видалення записів.

Після налаштування джерела даних оператор може перейти до розділу Settings (рис. Г.3) і встановити загальні параметри роботи програми. У цьому

розділі задається інтервал опитування CDC-таблиць, термін зберігання журналів, рівень логування та директорія експорту. Інтервал опитування визначає, як часто фоновий сервіс перевірятиме наявність нових змін у цільових базах даних. Термін зберігання журналів визначає, через скільки днів застарілі події та записи історії виконань будуть видалятися автоматично. Директорія експорту використовується для файлових дій і збереження результатів роботи.

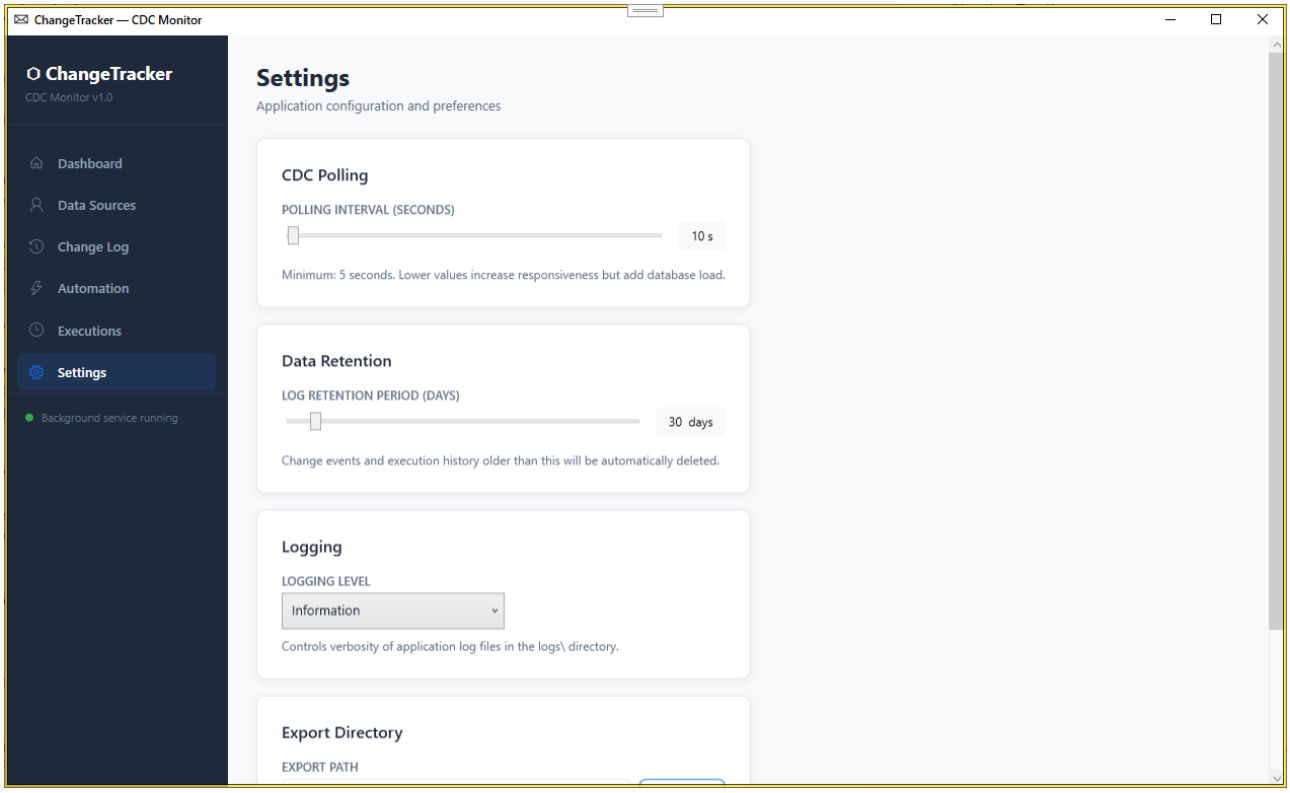


Рисунок Г.3 – Сторінка налаштувань програмного забезпечення

Під час експлуатації оператор повинен контролювати, чи працює фоновий сервіс, чи активні необхідні джерела даних і чи з’являються нові записи в журналі змін після виконання операцій у цільовій базі даних. Якщо події не з’являються, необхідно перевірити активність джерела даних, налаштування CDC, права доступу, стан SQL Server і службові журнали програми.

У разі виникнення помилок підключення, виконання HTTP-запитів, запису у файл або запуску зовнішніх програм інформацію про них можна переглянути в інтерфейсі застосунку або у файлах логів. Для аналізу помилок автоматичних дій потрібно перейти до розділу Executions, де зберігається історія виконань із

зазначенням статусу, тривалості та повідомлення про помилку.

Інструкція користувачеві

Користувач працює з програмним забезпеченням ChangeTracker через графічний інтерфейс. Основне призначення інтерфейсу — надати зручний доступ до статистики змін, журналу подій, джерел даних, правил автоматизації, історії виконань і налаштувань. Після запуску програми відкривається головне вікно, у якому зліва розташована навігаційна панель, а справа — робоча область вибраного розділу.

Початковою сторінкою є Dashboard. На ній відображається коротка зведена інформація про стан системи: загальна кількість подій, кількість подій за останній період, кількість операцій вставки, оновлення та видалення. Також на сторінці передбачено області для графіків, які показують активність змін за часом, розподіл подій за типами операцій і перелік таблиць із найбільшою кількістю змін. Для оновлення інформації користувач може натиснути кнопку Refresh.

Для роботи з підключеннями користувач переходить до розділу Data Sources. У цьому розділі відображається список усіх налаштованих джерел даних. Користувач може додати нове джерело, змінити параметри існуючого, видалити підключення або перевірити його працездатність. Якщо джерело даних позначене як активне, фоновий сервіс буде враховувати його під час кожного циклу опитування. Якщо джерело потрібно тимчасово не використовувати, достатньо вимкнути його активність без видалення запису.

Основним розділом для аналізу змін є Change Log. У ньому відображається журнал подій, отриманих із CDC-таблиць. Для кожної події показуються дата й час, назва бази даних, назва таблиці, тип операції, значення первинного ключа та статус обробки. Якщо в журналі міститься багато записів, користувач може застосувати фільтри за датою, назвою таблиці або типом операції. Це дозволяє швидко знайти потрібні події серед великого обсягу даних.

Для перегляду детальної інформації необхідно вибрати конкретний запис у журналі. Після вибору події в окремій області відображаються змінені поля,

попередні та нові значення. Такий перегляд дає змогу зрозуміти не лише те, що запис було змінено, а й які саме дані були змінені.

Для налаштування автоматичного реагування на події використовується розділ Automation. У ньому користувач може створювати правила, редагувати їх, видаляти, вмикати або вимикати. Правило визначає, для якої таблиці та якого типу операції потрібно виконувати певні дії. Наприклад, можна створити правило, яке після додавання нового запису до певної таблиці викликає зовнішній API або записує інформацію про подію до файлу.

Під час створення правила користувач задає назву правила, таблицю, тип події та список дій. Тип події може бути INSERT, UPDATE, DELETE або ALL. Значення ALL використовується тоді, коли правило має спрацьовувати для будь-якого типу зміни. Для одного правила можна додати кілька дій, які виконуватимуться послідовно.

Результати виконання автоматичних дій відображаються у розділі Executions. У цьому розділі користувач бачить, яке правило було виконано, для якої події, який тип дії запускався, чи завершилося виконання успішно та скільки часу воно тривало. Якщо дія завершилася помилкою, у відповідному полі буде показано повідомлення з описом проблеми. Для зручності роботи історію виконань можна фільтрувати за назвою правила, статусом і часовим проміжком.

Для завершення роботи із застосунком достатньо закрити головне вікно програми. Після цього фоновий сервіс припиняє роботу, а використовувані ресурси звільняються. Перед завершенням роботи бажано переконатися, що всі необхідні налаштування збережено, а важливі автоматичні дії виконалися успішно.

ДОДАТОК Д - Програма та методика випробувань ПЗ

Після реалізації основних функціональних модулів програмного забезпечення ChangeTracker було проведено тестування з метою перевірки відповідності розробленого застосунку сформульованим вимогам, виявлення помилок у роботі основних сценаріїв та підтвердження коректності взаємодії між компонентами системи.

Тестування програмного забезпечення є важливим етапом розробки, оскільки застосунок виконує не лише дії за запитом користувача, а й автоматичну фонову обробку подій. Для ChangeTracker особливо важливо перевірити правильність підключення до баз даних, отримання змін із CDC-таблиць, збереження журналу змін, зіставлення подій із правилами автоматизації, виконання дій та запис результатів в історію виконань.

Під час тестування було використано функціональний підхід, за якого перевіряється робота окремих функцій програмного забезпечення з погляду користувача та очікуваної поведінки системи. Також було виконано інтеграційне тестування окремих модулів, оскільки застосунок взаємодіє з Microsoft SQL Server, локальною базою даних, файловою системою та зовнішніми HTTP-сервісами.

Основними об’єктами тестування були:

– модуль керування джерелами даних;

– сервіс перевірки підключення до бази даних;

– механізм отримання змін із CDC-таблиць;

– журнал змін і перегляд деталей подій;

– модуль керування правилами автоматизації;

– механізм виконання автоматичних дій;

– історія виконання дій;

– модуль налаштувань;

– фоновий сервіс обробки подій.

Для проведення тестування було підготовлено тестову базу даних

Microsoft SQL Server, у якій увімкнено механізм Change Data Capture. У тестовій базі даних були створені таблиці, над якими виконувалися операції додавання, редагування та видалення записів. Це дозволило перевірити, чи коректно програмне забезпечення отримує CDC-події та перетворює їх у внутрішні записи журналу змін.

Тестування проводилося у середовищі Windows із використанням десктопного застосунку ChangeTracker. Внутрішня база даних застосунку використовувалася для зберігання джерел даних, журналу змін, деталей подій, правил автоматизації, дій правил, історії виконань і налаштувань системи. Для перевірки HTTP-дій використовувався тестовий API-запит, а для перевірки файлових дій — тестова директорія на локальному диску.

Під час тестування перевірялися як позитивні сценарії, коли користувач вводить коректні дані та система виконує очікувану дію, так і негативні сценарії, наприклад введення неправильних параметрів підключення, недоступність API або некоректний шлях до файлу. Це дозволило переконатися, що застосунок не завершує роботу аварійно, а повідомляє користувача про помилку або записує її в історію виконань. Основні тестові випадки наведено в таблиці Д.1.

Таблиця Д.1 – Тестові випадки програмного забезпечення ChangeTracker

№	**Назва тестового випадку**	**Вхідні дані / умови виконання**	**Очікуваний результат**	**Фактичний результат**
1	Додавання джерела даних	Користувач вводить назву підключення, сервер, назву бази даних та облікові дані	У системі створюється новий запис про джерело даних, він відображається у таблиці джерел	Виконано
2	Редагування джерела даних	Користувач змінює параметри існуючого джерела даних	Дані джерела оновлюються у внутрішній базі застосунку	Виконано
3	Видалення джерела даних	Користувач обирає джерело даних і підтверджує видалення	Запис видаляється зі списку джерел даних	Виконано

Продовження таблиці Д.1

№	Назва тестового випадку	Вхідні дані / умови виконання	Очікуваний результат	Фактичний результат
4	Перевірка коректного підключення до БД	Введено правильні параметри підключення до SQL Server	Система повідомляє про успішне підключення та доступність бази даних	Виконано
5	Перевірка некоректного підключення до БД	Введено неправильну адресу сервера або облікові дані	Система показує повідомлення про помилку підключення без аварійного завершення роботи	Виконано
6	Отримання INSERT-події	У тестову таблицю цільової БД додано новий запис	Після циклу опитування в журналі змін з’являється подія з типом INSERT	Виконано
7	Отримання UPDATE-події	У тестовій таблиці змінено існуючий запис	У журналі змін з’являється подія з типом UPDATE та деталями змінених полів	Виконано
8	Отримання DELETE-події	У тестовій таблиці видалено запис	У журналі змін з’являється подія з типом DELETE	Виконано
9	Перегляд деталей події	Користувач обирає запис у журналі змін	Відображається список змінених полів, старі та нові значення	Виконано
10	Фільтрація журналу змін за типом операції	Користувач обирає тип операції INSERT, UPDATE або DELETE	У журналі відображаються лише події вибраного типу	Виконано
11	Фільтрація журналу змін за таблицею	Користувач вводить назву таблиці	У журналі відображаються лише події, що належать до заданої таблиці	Виконано
12	Створення правила автоматизації	Користувач задає назву правила, таблицю, тип події та дію	Нове правило зберігається у системі та відображається у списку правил	Виконано
13	Вимкнення правила автоматизації	Користувач вимикає активне правило	Правило залишається у системі, але не виконується під час обробки подій	Виконано
14	Виконання HTTP GET-дії	Створено правило з дією HTTP_GET і коректною URL-адресою	Після появи відповідної події виконується HTTP GET-запит, результат записується в історію	Виконано

Кінець таблиці Д.1

№	Назва тестового випадку	Вхідні дані / умови виконання	Очікуваний результат	Фактичний результат
15	Виконання HTTP POST-дії	Створено правило з дією HTTP_POST і тілом запиту	Після спрацювання правила виконується HTTP POST-запит, результат фіксується в історії	Виконано
16	Виконання дії запису у файл	Створено правило з дією WRITE_FILE та шляхом до файлу	У файл додається сформований рядок із даними події	Виконано
17	Виконання дії запуску процесу	Створено правило з дією EXEC_PROCESS	Після спрацювання правила запускається вказаний зовнішній процес	Виконано
18	Обробка помилки HTTP-запиту	У правилі задано недоступну URL-адресу	Дія завершується зі статусом Failed, повідомлення про помилку зберігається в історії виконань	Виконано
19	Перегляд історії виконань	Користувач відкриває сторінку Executions	Відображається список виконаних дій із датою, правилом, статусом і тривалістю	Виконано
20	Фільтрація історії виконань	Користувач задає статус або назву правила	Відображаються лише записи історії, що відповідають заданим параметрам	Виконано
21	Зміна інтервалу опитування	Адміністратор змінює інтервал опитування у налаштуваннях	Нове значення зберігається та використовується фоновим сервісом у наступному циклі	Виконано
22	Очищення застарілих записів	У налаштуваннях задано термін зберігання журналів	Записи, старші за встановлений термін, видаляються під час фонового циклу	Виконано

Окремо було перевірено коректність роботи графічного інтерфейсу. Під час перевірки встановлено, що сторінки застосунку відкриваються через навігаційне меню, дані відображаються у відповідних таблицях, фільтри застосовуються без перезавантаження програми, а повідомлення про успішні або помилкові операції відображаються користувачу у зрозумілій формі.

Також було перевірено роботу фонового сервісу. Після запуску застосунку

сервіс автоматично виконує цикл опитування активних джерел даних. У разі появи нових записів у CDC-таблицях система зберігає відповідні події у внутрішній базі даних. Якщо для події існує активне правило, виконується відповідна дія, а результат записується в історію виконань. Після завершення обробки подія позначається як оброблена, що запобігає повторному виконанню правила для тієї самої події.

Під час тестування негативних сценаріїв було перевірено, що помилки не призводять до аварійного завершення роботи застосунку. У разі некоректних параметрів підключення користувач отримує повідомлення про помилку. У разі помилки виконання HTTP-запиту, запису у файл або запуску зовнішнього процесу відповідна інформація зберігається в історії виконань зі статусом Failed. Це дозволяє користувачу виявляти проблеми та аналізувати причини невдалого виконання автоматичних дій.

Результати тестування показали, що програмне забезпечення ChangeTracker виконує основні функції, визначені під час постановки задачі: забезпечує керування джерелами даних, отримує зміни з CDC-таблиць, відображає журнал змін, дозволяє переглядати деталі подій, підтримує правила автоматизації, виконує налаштовані дії та зберігає історію виконань. Виявлені під час тестування помилки були усунені, а робота основних сценаріїв підтверджена тестовими випадками.

Таким чином, проведене тестування підтвердило працездатність розробленого програмного забезпечення та його відповідність основним функціональним вимогам. Отримані результати дозволяють перейти до аналізу результатів розробки програмного забезпечення та оцінки ступеня реалізації поставлених завдань.