Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ДОСЛІДЖЕННЯ ПИТАННЯ ОБРАННЯ АРХІТЕКТРУРИ ВЕБЗАСТОСУНКУ НА ОСНОВІ АНАЛІЗУ ВИМОГ ПРИКЛАДНОЇ ЗАДАЧІ
(тема)

Виконав:
студент 2 курсу, групи ІНФм-23-1
Шелест В.А.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Яковлева О.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Шелесту Володимиру Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження питання обрання архітектури вебзастосунку на основі аналізу вимог прикладної задачі

затверджена наказом по університету від 25 листопада 2024 року № 1246Ст

2. Термін подання студентом роботи до екзаменаційної комісії 23 грудня 2024 р.

3. Вихідні дані до роботи архітектурні рішення, способи комунікації між вебсервісами, теоретичні відомості про вибір найкращого архітектурного рішення в залежності від вимог до програмного забезпечення, знання і навички написання вебсервісів.

4. Перелік питань, що потрібно опрацювати в роботі
1. Виокремлення критеріїв оцінки якості програмного забезпечення.
2. Огляд наявних архітектурних і технологічних рішень.
3. Порівняння варіантів технологічних рішень між собою.
4. Розробка алгоритму визначення найкращого архітектурного рішення базуючись на вимогах користувача.
5. Розробка програмного застосунку для зручного і швидкого виконання алгоритму.
6. Перевірка роботи алгоритму.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми обрання архітектурного рішення, постанова задачі, виокремлення критеріїв для оцінки якості архітектурного рішення, виокремлення технологічних рішень, аналіз результатів роботи алгоритму.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	25.11.2024	
2	Аналіз завдання, підбір літератури	26.11.24-27.11.24	
3	Аналіз літератури з досліджуваної проблеми	28.11.24-30.11.24	
4	Аналіз архітектурних і технологічних рішень	01.12.24-03.12.24	
5	Розробка алгоритму	04.12.24-11.12.24	
6	Програмна реалізація	12.12.24-19.12.24	
7	Оформлення пояснювальної записки	20.12.24-23.12.24	
8	Перевірка на плагіат	24.12.2024	
9	Рецензування	25.12.2024	
10	Підготовка презентації та доповіді	29.12.2024	
11	Занесення роботи в електронний архів	01.01.2025	
12	Попередній захист кваліфікаційної роботи	02.01.2025	

Дата видачі завдання 25 листопада 2024 р.

Студент _____
(підпис)

Керівник роботи _____ доц. Яковлева О.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 73 с., 5 табл., 36 рис., 1 дод., 34 джерела.

АРХІТЕКТУРНІ РІШЕННЯ, ЗАДАЧА БАГАТОКРИТЕРІАЛЬНОЇ ОПТИМІЗАЦІЇ, ISO/IEC 25010.

Об’єктом дослідження є питання вибору архітектури вебзастосунку на основі аналізу вимог прикладної задачі.

Метою дослідження є збирання критеріїв та формування їх в один алгоритм, який допомагатиме приймати рішення щодо вибору архітектури ПЗ.

Дослідження базується на задачі багатокритеріальної оптимізації та аналізі архітектурних і технологічних рішень. В дослідженні виокремлюються критерії необхідні для оцінки якості архітектурного рішення, встановлюються базові коефіцієнти, за допомогою яких проводиться вибір кращого технологічного рішення за виокремленими критеріями. Середовищем розробки було обрано Rider для імплементації серверу на мові програмування C# та Visual Studio Code для імплементації клієнта із використанням фреймворку Vue.

ARCHITECTURE SOLUTIONS, MULTI-CRITERION OPTIMIZATION PROBLEM, ISO/IEC 25010.

The object of the research is the issue of selecting a web application architecture based on the analysis of the requirements of an applied task.

The aim of the research is to gather criteria and integrate them into a single algorithm that will assist in making decisions regarding the selection of software architecture.

The research is based on the task of multi-criteria optimization and the analysis of architectural and technological solutions. The study identifies criteria necessary for assessing the quality of an architectural solution and establishes basic coefficients that are used to select the best technological solution according to the defined criteria.

The development environment chosen for the study includes Rider for implementing the server using the C# programming language and Visual Studio Code for implementing the client using the Vue framework.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ІТ – інформаційні технології

ПЗ – програмне забезпечення

LLM – Large Language Model (велика мовна модель)

ПК – персональний комп’ютер

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту)

XML – Extensible Markup Language (розширювана мова розмітки)

JSON – JavaScript Object Notation (нотація об’єктів JavaScript)

REST – Representational State Transfer (передача репрезентативного стану)

SOA – Service-Oriented Architecture (сервісно-орієнтована архітектура)

SOAP – Simple Object Access Protocol (простий протокол доступу до об’єктів)

SMTP – Simple Mail Transfer Protocol (простий протокол передачі пошти)

БД – база даних

CI/CD – Continuous Integration/Continuous Delivery (безперервна інтеграція/безперервна доставка)

ВСТУП

В сучасному світі інформаційних технологій (IT) архітектура у вебзастосунках відіграє ключову роль. Архітектори програмного забезпечення – є поважними професіоналами в сфері інформаційних технологій, які можуть полегшити процеси підтримки і розширення існуючого програмного забезпечення (ПЗ) і заощадити велику кількість грошей бізнесу.

Оцінити якість архітектури і обрати найкращу під конкретні вимоги є складною задачею, яка є такою для великої кількості програмістів, а представники бізнесу не бачать сенсу інвестувати ресурси у вирішення цього питання через нерозуміння наслідків і можливість перекласти відповідальність на програмістів. Для полегшення вирішення питання вибору архітектури для реалізації програмного забезпечення, метою цієї дипломної роботи є збирання критеріїв і розробка загального алгоритму для прийняття рішень для обрання архітектури при розробці програмного забезпечення, а також простого сервісу чату, який допоможе обрати архітектуру ставлячи допоміжні питання.

Алгоритм має за мінімальну кількість запитань надати опис та інструкції для реалізації архітектури програмного забезпечення враховуючи оцінки надані на питання поставлені алгоритмом. Застосунок має звертатись до LLM (Large Language Model) моделі для корегування результатів роботи алгоритму і для аналізу коментарів наданих користувачем.

Розробка даного застосунку має великий практичний і науковий інтерес, оскільки він може бути використаний як в інтересах бізнесу, так і для формування цілісних методичних вказівок про можливі типи архітектур вебзастосунків і нові способи використання мовних нейронних моделей.

1 СУЧАСНИЙ СТАН ПИТАННЯ ОБРАННЯ АРХІТЕКТУРИ ВЕБЗАСТОСУНКУ

1.1 Тенденції та виклики сучасного сектора інформаційних технологій

Останні декілька десятиліть дуже стрімко розвивається інтернет і вебсервіси, які його формують. Через дуже швидкий темп змін в вимогах користувачів до вебзастосунків, власники цих застосунків мають так само швидко адаптуватись під ці нові умови. Швидкий темп розвитку індустрії вимушує покращувати існуючи підходи, технології і техніку для підтримки всієї інфраструктури. Якщо ще на початку двохтисячних розробники програмного забезпечення переймались за кожен байт оперативної пам’яті, який може зайняти програма, та намагались придумати інноваційні методи для оптимізації роботи із потоками процесора, то зараз кожен персональний комп’ютер (ПК) звичайного користувача має від 8 гігабайт оперативної пам’яті і близько половини терабайта пам’яті для збереження файлів. І це пристрій користувача; сервери, на яких працюють вебзастосунки, мають на порядок кращі характеристики пристроїв, на яких працює сервіс.

Більшість підходів для оптимізації роботи програм розроблених на початку тисячоліття використовуються і зараз, але все одно здебільшого програмісти не рахують байт, який може бути не використаний. Натомість зараз вони переслідують іншу мету – зробити код легшим для підтримки наступниками. Наразі програмні застосунки створюються занадто великими, щоб їх було швидко і легко переписати. Це створює проблему того, що сервіс середнього або великого розміру стає занадто складним для розуміння і ще складнішим для підтримки. Із кожним місяцем кількість часу необхідного для ознайомлення із сервісом зростає на декілька днів. Враховуючи ситуацію із тим, що працівники в сфері IT стали частіше змінювати місце працевлаштування і проєкти, над якими вони працюють, то на швидко застарілі проєкти дуже складно знайти спеціалістів, а ще складніше їх

утримати. Звісно можна сповільнити процеси переходу вебзастосунку від технологічно прогресивного до застарілого, або й уникнути цієї проблеми взагалі приділенням часу на вибір архітектури на початкових стадіях побудови ПЗ.

1.2 Важливість архітектури у світі розробки програмного забезпечення

Із плином часу швидкість розвитку технологій зростає, тобто деякі з них стають застарілими занадто швидко. Наприклад, в сфері комп'ютерного зору [1-4], де дуже важлива якість і швидкість розпізнавання, так само швидко розвиваються і методи, які використовуються. Коли нейронні мережі не були настільки потужними, класичні методи комп'ютерного зору відігравали ключову роль в галузі. Саме їх впроваджували в різні системи, але з плином часу розвивались і самі методи, а також з'являлись нові та кращі, наприклад, із використанням нейронних мереж [5].

Впроваджувати такі технології в застосунок також стає викликом. Додавання в проєкт якоїсь технології як прямої залежності може бути дуже дорогою помилкою: в якийсь момент підтримка технології закінчиться, або з'явиться кращий варіант, на який варто буде перейти. Саме такі проблеми має вирішувати правильно підібране архітектурне рішення. Додавання прямих залежностей в проєкт може бути вигідним в короткостроковій перспективі, тому що відбудеться швидше і легше. Але для довгострокової перспективи така залежність може бути фатальною: замінити щось в проєкті набагато важче ніж додати або забрати. Дуже часто в таких ситуаціях приймається рішення все переробити і при цьому допускаються ті самі помилки. І такі перероблення коштують багато часу і ресурсів для імплементації. Такі важкі рішення дуже часто є навпаки корисними для довгострокової перспективи, але тільки якщо переосмислити вимоги застосунку і продумати вирішення проблем, які привели до цього важкого рішення.

В даних ситуаціях найкраще буде відокремити імплементації будь-яких деталей. Абстрагування того, що саме необхідно застосунку від технології, є важливим питанням, яке має ставити перед собою розробник додаючи якусь залежність в проєкт. Тобто наприклад при розробці системи для оцінки знань студентів із можливістю розпізнавання студента під час проходження тесту [6] саме розпізнавання обличчя винесене в окремий сервіс, тобто все може початись із імплементації якогось класичного методу розпізнавання обличчя [7], з часом перейти на підхід із поєднанням класичних підходів із нейронними мережами [8] і з часом стати ще чимось, уся логіка того, як працює розпізнавання інкапсульована в окремий сервіс і може бути легко змінена не вимагаючи змінювати спосіб комунікації із іншими сервісами.

Саме можливість розпізнати і відокремити абстракцію від деталі [9-12], провести кордон між зонами впливу модулів застосунку зробить проєкт довговічним, полегшить внесення будь-яких правок. Звісно, що такий підхід може сповільнити додавання якогось функціоналу, тому що буде вимагати додати абстракції і імплементації окремо і приділити час на те, щоб їх поєднати. Але будь-який застосунок, в який планується вносити зміни, отримає більше вигоди від більш довгого і продуманого процесу імплементації.

1.3 Еволюція архітектур вебзастосунків

При зародженні інтернету в 1990-тих і на початку 2000-них вебзастосунки були максимально простими і тому мали відповідну архітектуру – монолітну. Переваги цієї архітектури дуже сильно помітні на початкових стадіях написання проєкту: уся бізнес логіка, комунікація із базою даною та відображення сторінки користувачеві знаходились в одному місці і не мали жодного розділення.

Із плином часу багато програмістів почали помічати ці проблеми і намагатись їх вирішити, наприклад, розділенням застосунку на клієнт і сервер надаючи кожному конкретну мету існування, виконуючи SRP (Single Responsibility Principle) [13], окрім цього сам сервер теж розбивають на 3 рівні: рівень доступу до даних (data access layer), рівень бізнес логіки (business logic layer) та рівень презентації (presentation layer). Із цих рівнів було створено багато різних архітектур, наприклад, MVVM (Model View View-Model) (рис. 1.1) або MVC (Model View Controller) (рис. 1.2). Здебільшого їхня суть залишається незмінною від архітектури до архітектури: є шар відображення і є шар логіки, і між ними має існувати шар, який передає дані з одного стану в інший. Наприклад, передає дані для відображення, або реагує на дію користувача

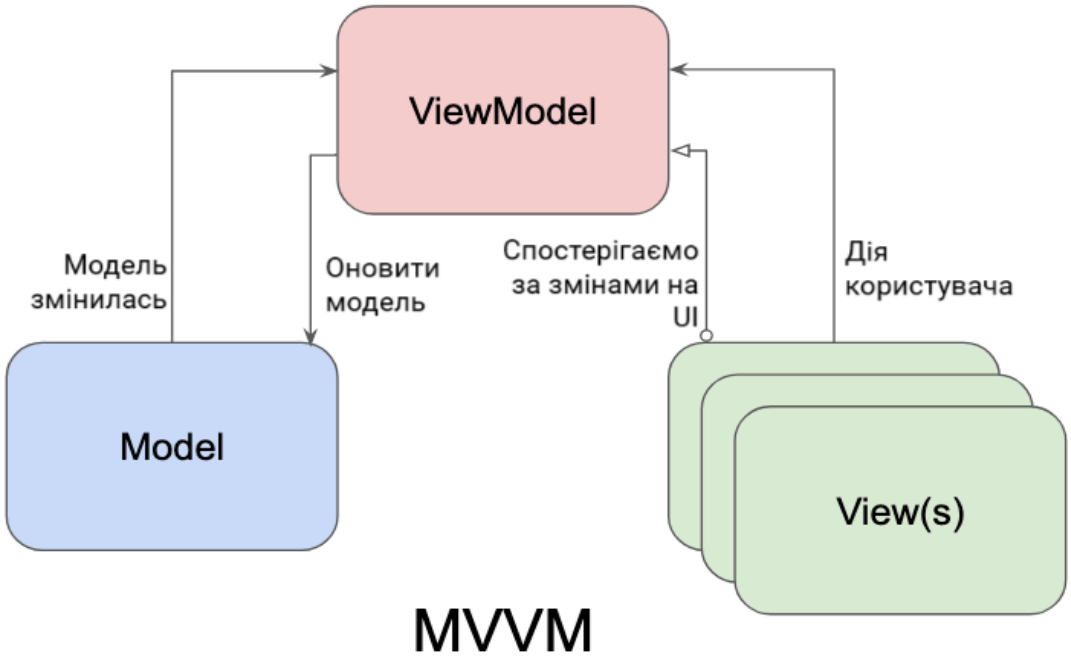

Рисунок 1.1 – Діаграма побудови архітектури MVVM [14]

В цей момент Рой Філдінг у своїй докторській дисертації [15] запропоновує ідею REST (Representational State Transfer) архітектури, діаграма побудови якого зображена на рисунку 1.3.

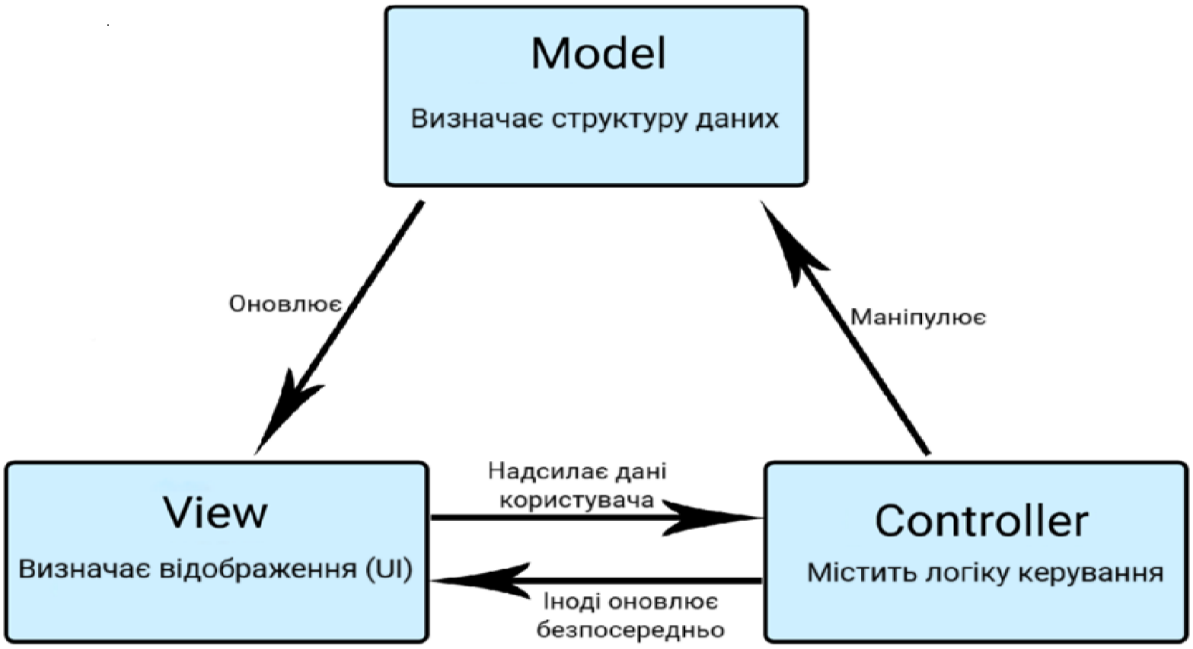

Рисунок 1.2 – Діаграма побудови архітектури MVC [16]

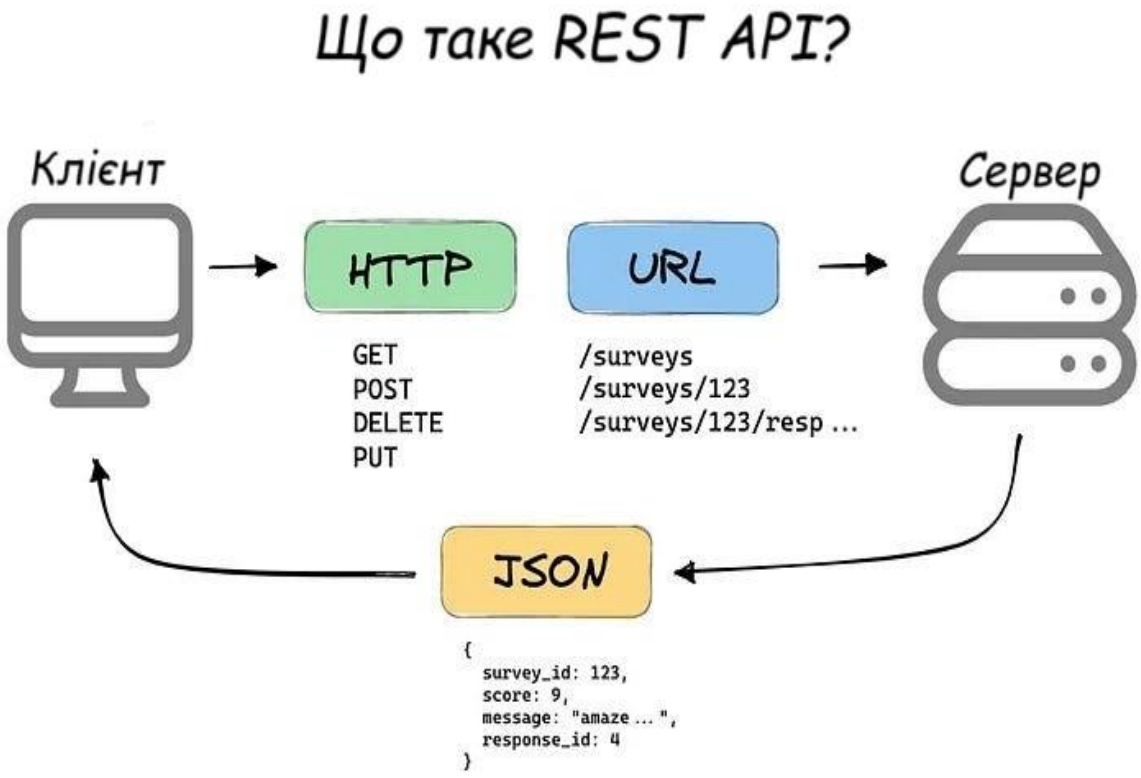

Рисунок 1.3 – Діаграма побудови REST API [17]

В основі цієї ідеї є клієнт серверна модель, де кожен з них є незалежним компонентом системи, сервер не має зберігати жодну інформацію про стан клієнта, а спілкування має відбуватись через HTTP у форматі XML за

стандартними HTTP методами: POST, PUT, DELETE і GET. Найголовнішою складовою даного застосунку на думку Роя Філдінга мала стати самоописність сервера, тобто сервер мав повертати не просто інформацію, яку просив клієнт, а також інформацію про те, як витягнути інформацію про якісь залежні речі.

Зараз REST – є одним із найрозповсюдженіших вибором архітектури у вебзастосунках, але він трохи відрізняється від того, як його задумував автор. REST так само про клієнт-серверну модель та комунікацію через HTTP запити, але із даними у форматі JSON, сервер може інколи зберігати інформацію про сесії користувача, тобто інформацію про стан клієнта, але найбільшою відмінністю є відсутність самоописності системи через складнощі її реалізації і підтримки, натомість програмісти відкритих систем намагаються вести документацію усіх ендпоінтів, які має сервер.

В 2000-них роках активно розвивалась SOA (Service-Oriented Architecture). Ця архітектура полягає в виокремлені незалежних сервісів, кожен з яких виконує певну бізнес-функцію і може взаємодіяти з іншими сервісами через стандартизовані інтерфейси, наприклад, SOAP [18] або REST. Ця архітектура надавала більшу гнучкість в тому, як можна було структурувати модулі і не тільки, але все одно кожен із сервісів міг перетворитись в складний моноліт, який буде складно підтримувати.

SOAP (Simple Object Access Protocol) – це протокол обміну структурованими повідомленнями в розподілених обчислювальних системах, який базується на XML. Він дозволяє програмам, написаним на різних мовах програмування та працюючим на різних платформах, взаємодіяти через мережу. SOAP підтримує різні транспортні протоколи, такі як HTTP, SMTP та інші, що робить його гнучким для використання в різних середовищах. Основними компонентами повідомлення SOAP є конверт (Envelope), заголовок (Header) та тіло (Body), що забезпечує чітку структуру для передачі даних. Діаграма побудови SOAP представлена на рисунку 1.4.

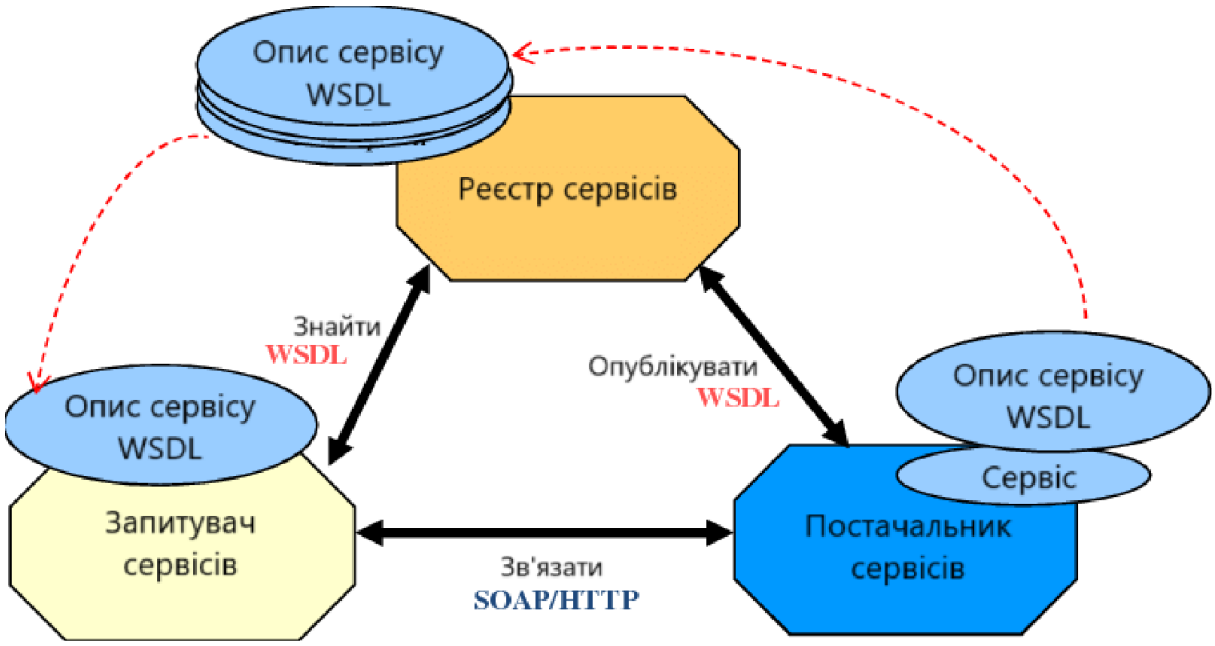

Рисунок 1.4 – Діаграма побудови SOAP [19]

В 2010-тих роках з’являється архітектура мікросервісів, яка зараз активно використовується для гігантських систем, наприклад, Netflix або Amazon. Принцип цієї архітектури полягає в тому, що застосунок має бути набором дуже маленьких сервісів, які виконують одну або декілька функцій. Найбільшої перевагою таких систем є їхня стабільна робота, кожен з мікросервісів може бути розташований в різних кінцях світу, а спілкуватись вони можуть між собою різними способами REST, gRPC (Google Remote Procedure Call) [20], GraphQL (Graph Query Language) [21]. Несправність якогось одного сервісу не призводить до того, що увесь застосунок перестає працювати. Звичайно, що найважливіші сервіси системи мають декілька реплік, які можуть запуститись на різних серверах в критичні моменти, тобто стабільність таких систем є надзвичайною. Але дану систему складно реалізувати, особливо, коли не має досвіду роботи із подібними системами. Також викликом для таких систем є їх вартість розгортання.

В 2012 році Facebook (на даний момент Meta) розробили вже згаданий GraphQL, а в 2015 відкрили його для усіх. Дана технологія є дуже потужною з точки зору того, які проблеми вона вирішує. Але вибір саме такого способу комунікації між клієнтом та сервером, або між сервісами, дуже сильно впливає

на те, який вигляд буде мати архітектура кожного окремого модулю, які технології будуть використовуватися тощо.

Хоча і перевага у вигляді можливості клієнта визначати, які саме дані необхідно повертати, – є дуже вагомою. Окрім цього перевагою може бути можливість дістати інформацію з різних ресурсів (рис. 1.5), якщо із REST API клієнту доведеться зробити 3 виклики серверу, якщо необхідна інформація про усі матчі зіграні командою і хто приймав участь в тих матчах, то в GraphQL це буде лише один виклик, в якому можна взяти тільки ту інформацію, яка необхідна саме в даному випадку.

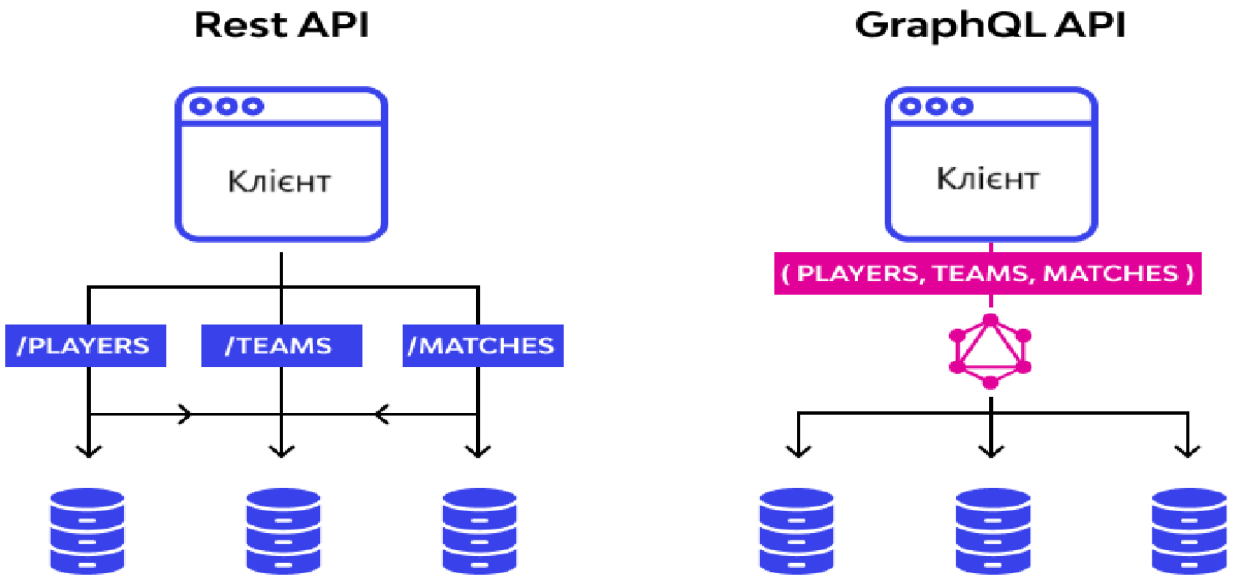

Рисунок 1.5 – Відмінність GraphQL API від REST API [22]

В 2016 році компанія Google випустила першу стабільну версію gRPC, нове покоління RPC технологій. Якщо в REST до серверу необхідно звертатись за різними URL, а в GraphQL прописувати запити або мутації, то в gRPC (рис 1.6) використовуються файли-контракти (proto файли), які визначають, які функції може опрацювати сервер, які аргументи приймає функція і який результат повертає. Також особливістю gRPC є його бінарна серіалізація і використання HTTP/2 для комунікації між клієнтом і сервером, що робить передачу даних швидшою і більш захищеною.

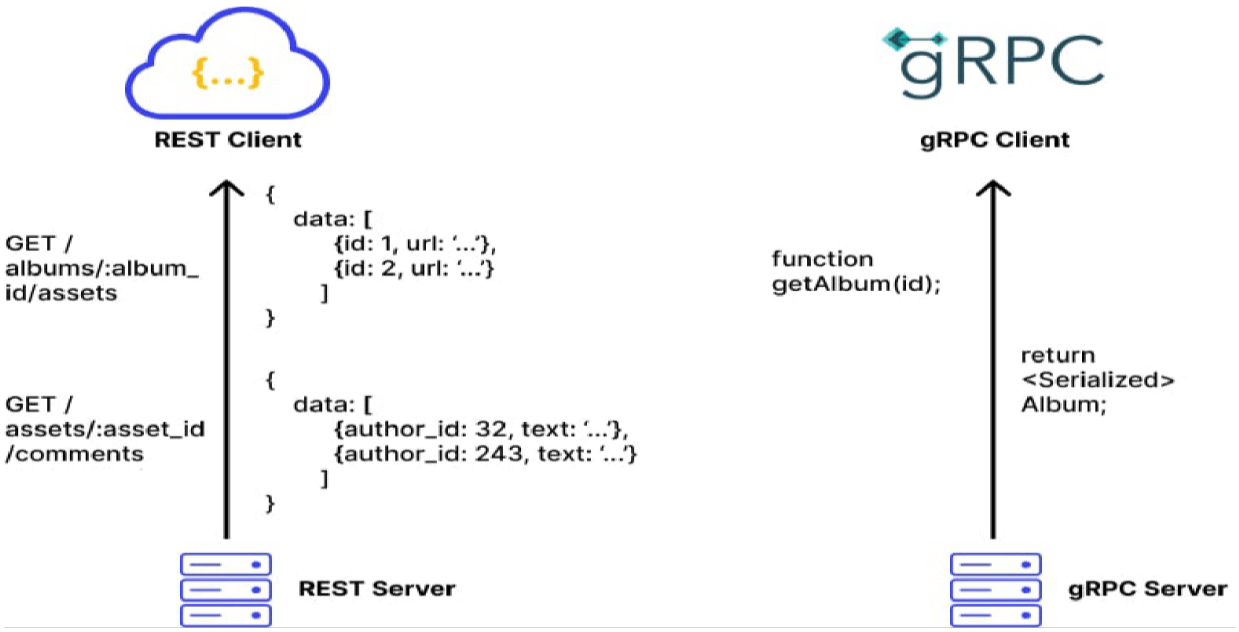

Рисунок 1.6 – Відмінність gRPC від REST [23]

Зараз активно розвиваються безсерверні (serverless) архітектури. Дана архітектура перекладає управління інфраструктурою на провайдерів хмар, наприклад, AWS (Amazon Web Services) від Amazon або Azure від Microsoft. На відміну від оренди серверу, за який платять помісячно однаково, хмарні сервіси оплачуються в залежності від того наскільки їхні потужності були використані. Окрім цього провайдер може зменшити кількість виділених ресурсів, якщо немає високого попиту на сервіс, або навпаки їх збільшити, якщо є великий наплив користувачів.

Також багато програмістів впроваджують підхід подієво-орієнтованого спілкування між сервісами.

В класичних підходах спілкування один сервіс звертається до іншого на пряму по різним протоколам комунікації і якщо приймаючий сервіс не відповідає, то повідомлення може загубитись. Подієво-орієнтований підхід використовує черги для спілкування, один із сервісів надсилає повідомлення в чергу і коли приймач буде вільний, то візьме повідомлення в обробку. Окрім цього це дозволяє запускати декілька реплік одного сервісу одночасно і паралельно обробляти різні запити.

Така велика кількість різних підходів в розробці вебзастосунків вимагає деяких стандартів для оцінювання якості розробленого вебзастосунку для розуміння проблем в даному сервісі, які існують, і які можуть з’явитись в наступному.

1.4 Стандарти оцінювання якості програмного забезпечення

Паралельно із розвитком архітектур були створені стандарти для оцінки якості: ISO/IEC 9126 [24] (перший міжнародний стандарт для оцінки якості ПЗ) або ISO/IEC 25010 [25] (сучасний стандарт для оцінки ПЗ).

Хоча стандарти і були видані із проміжком в 10 років, але ISO/IEC 25010 (рис. 1.7) є розширенням свого попередника. Він додає пункти про безпеку та сумісність, розширює деякі підпункти, а також робить орієнтацію на сучасні тренди.

Рисунок 1.7 – Графічний опис структури стандарту ISO/IEC 25010

ISO/IEC 25010 складається з 8 пунктів:

а) функціональна придатність (functional suitability) – відображає наскільки добре програмне забезпечення виконує свої функції відповідно до визначених вимог:

1) функціональна повнота (functional completeness) – чи покриває продукт всі необхідні функції;

2) функціональна коректність (functional correctness) – чи точно продукт виконує свої функції;

3) функціональна відповідальність (functional appropriateness) – чи ефективно продукт виконує свої функції;

б) продуктивність (performance efficiency) – оцінює, наскільки ефективно система виконує завдання щодо швидкості та використання ресурсів:

1) час відповіді (time behavior) – швидкість реакції системи на запити;

2) використання ресурсів (resource utilization) – кількість ресурсів, які використовуються для виконання завдань;

3) ємність (capacity) – максимальна кількість користувачів або запитів, які система може обробляти;

в) сумісність (compatibility) – наскільки добре програмне забезпечення взаємодіє з іншими системами:

1) співіснування (co-existence) – чи може система працювати в середовищі з іншими системами, не конфліктуючи з ними;

2) інтероперабельність (interoperability) – наскільки легко система може взаємодіяти з іншими системами та обмінюватися даними;

г) юзабіліті (usability) – оцінює, наскільки легко користувачам взаємодіяти з системою:

1) зрозумілість (learnability) – наскільки швидко користувачі можуть навчитися використовувати систему;

2) керованість (operability) – наскільки легко система дозволяє виконувати потрібні завдання;

3) захищеність від помилок (user error protection) – як добре система запобігає помилкам або допомагає користувачам виправляти їх;

ґ) надійність (reliability) – відображає стабільність і здатність системи працювати протягом тривалого часу без збоїв:

1) зрілість (maturity) – наскільки добре система працює без збоїв;

2) стійкість до збоїв (fault tolerance) – наскільки добре система може відновитися після збоїв;

3) відновлюваність (recoverability) – як швидко система може відновитися після збоїв;

д) безпека – захист інформації і даних системи від несанкціонованого доступу:

1) конфіденційність (confidentiality) – наскільки добре система захищає дані від несанкціонованого доступу;

2) цілісність (integrity) – чи зберігаються дані в незмінному вигляді;

3) аутентифікація (authentication) – наскільки надійно ідентифікуються користувачі;

е) підтримуваність (maintainability) – відображає, наскільки легко можна вносити зміни в систему:

1) модульність (modularity) – наскільки добре система поділена на незалежні частини;

2) здатність до аналізу (analyzability) – наскільки легко аналізувати систему для виявлення помилок або проблем;

3) можливість модифікації (modifiability) – наскільки легко вносити зміни в систему;

є) переносність (portability) – наскільки легко програмне забезпечення можна перенести на інші середовища або платформи:

1) адаптованість (adaptability) – наскільки легко система може бути модифікована для роботи в нових середовищах;

2) інсталяція (installability) – наскільки легко система може бути встановлена в новому середовищі;

3) замінність (replaceability) – наскільки легко один компонент системи можна замінити іншим.

ISO/IEC 25010 дозволяє комплексно оцінити якість ПЗ, орієнтуючись на різні аспекти функціональності, продуктивності, безпеки, підтримуваності та інших характеристик.

Цей стандарт допомагає визначити сильні та слабкі сторони архітектури ще на ранніх етапах розробки, що робить його цінним інструментом для вибору архітектурних рішень.

1.5 Технічні обмеження при виборі архітектури

Хоча технології зараз набагато потужніші і наявна техніка здатна покрити майже усі можливі варіанти обчислень, все ж таки зберігаються деякі обмеження при побудові ПЗ: знання, навички і досвід команди або фінансові обмеження. Також має враховуватись фактор реальності вимог до проєкту – немає сенсу планувати, як система витримає мільйони користувачів, якщо планується роботи лише на декілька тисяч.

Якщо проєкт тільки на стадії ідеї або планування і ще немає сформованої команди для реалізації сервісу, обмеження у вигляді знань команди є непотрібним, але якщо в компанії вже є сформована команда і є розуміння, що саме ці люди будуть розробляти проєкт, необхідно враховувати їх досвід при обранні стеку технологій для реалізації проєкту і складності самої архітектури. Очевидно, що якщо команда складається із розробників C# .Net вибір мови програмування не впаде на Java, але із зворотної сторони, якщо необхідно створити нейронну мережу, очевидним стає вибір Python. Це також стосується

і вибору підтримки власного серверу, або використання хмарних рішень і якщо вибір падає на друге, то є також багато різних варіантів: AWS, Azure, Google Cloud. І звісно це стосується і баз даних, не тільки якихось конкретних, а й типу баз даних SQL або NoSQL і зокрема тип БД серед NoSQL баз даних.

Дуже часто фінансові обмеження від замовника можуть ставати на заваді використання сучасніших технологій і стабільніших архітектур. Проєкти середньої величини можуть коштувати декілька тисяч доларів на місяць (тільки інфраструктура, без зарплат команді), що може бути занадто великою сумою для багатьох замовників. Якщо у замовника є сервер, на якому він/вона планує розгортати ПЗ, то немає сенсу шукати способи як інтегруватись із хмарними рішеннями, звичайно хороша архітектура включить в себе можливість легкого переходу з локального серверу в хмару, але на початкових етапах на перше місце необхідно ставити обставини, які є вже зараз. Окрім цього необхідно пам’ятати, що деяке програмне забезпечення, з яким можливо доведеться працювати не є безкоштовним при використанні в комерційних цілях, наприклад, бібліотека froala [26] в JavaScript для редагування тексту із дуже насиченим функціоналом.

Не менш важливим фактором, який має обмежувати архітектора, це реальність вимог і архітектури. Часто буває, що замовник хоче розробити систему, яка буде опрацьовувати таку ж саму кількість користувачів як Facebook, але при цьому застосунком будуть користуватись лише близьке коло друзів цього замовника. Не варто зазначати, наскільки великим марнотратством грошей і часу розробників може стати виконання такого замовлення за вимогами клієнта. Зворотною стороною монети можуть стати недосвідчені архітектори, які хочуть показати усі свої навички в межах одного проєкту, використати усі можливі сервіси, про які він/вона чув/чула, навіть якщо вони не підходять під умови проєкту. Якщо в першому випадку замовнику треба пояснювати, що сенсу немає в таких потужностях для даного проєкту, то в другому – замовнику має пощастити, щоб в команді був свідомий досвідчений програміст, який викаже свій протест даним методам.

1.6 Постановка задачі дослідження

Вибір архітектури – є дуже важким, кропітким і важливим питанням при створенні вебзастосунку. Правильний вибір архітектури в конкретній ситуації може заощадити багато грошей замовника та нервів програмістів, які будуть працювати із проєктом через декілька років. Існує стандарт для оцінювання архітектури ПЗ, але він не враховує особливості вимог і можливостей клієнта, хоча на його основі можна і треба робити фінальні рішення при виборі архітектури програмного застосунку.

Таким чином, питання обрання архітектури вебзастосунку на основі аналізу вимог прикладної задачі є актуальною темою, вирішення якої допоможе суттєво заощаджувати ресурси при розробці і підтримці вебсервісу.

Об’єктом дослідження є питання вибору архітектури вебзастосунку на основі аналізу вимог прикладної задачі.

Метою дослідження є збирання критеріїв та формування їх в один алгоритм, який допомагатиме приймати рішення щодо вибору архітектури ПЗ.

Для досягнення цієї мети були сформовані наступні задачі:

– розглянути різні варіанти архітектур, які існують на даний момент і виокремити ті, які є актуальними на даний момент;

– проаналізувати сучасний стандарт для оцінки ПЗ;

– відокремити критерії із стандарту ISO/IEC 25010, які можна застосувати на різних стадіях розробки вебзастосунку;

– сформувати алгоритм для вибору архітектури в залежності від вимог до проєкту;

– підготувати мовну модель для вміння працювати із розробленим алгоритмом і визначити список головних питань;

– розробити зручний застосунок, який допоможе приймати рішення з вибору архітектури ПЗ;

– провести тестування розробленого застосунку.

2 МАТЕМАТИЧНА МОДЕЛЬ АЛГОРИТМУ ДЛЯ ВИБОРУ АРХІТЕКТУРИ

2.1 Виокремлення критеріїв і їх аналіз

Хоч стандарт і надає велику кількість критеріїв, за якими можна оцінити якість конкретного архітектурного рішення, але не всі з них можна конвертувати в числові значення, а деякі не матимуть жодного значення для визначення кращого вибору архітектури.

«Функціональна придатність» є одним із пунктів, на які можна не зважати при виборі архітектури під час побудови ПЗ. Сам пункт є дуже важливим питанням, яке має ставити собі кожен програміст. Перш за все програма має виконувати поставлені перед нею задачу, а вже потім код має бути легко підтримуваним, має бути документування і тд. Але в момент коли здійснюється вибір архітектурного рішення для конкретної задачі, то вважається, що увесь функціонал буде реалізованим.

«Продуктивність» є одним із найважливіших пунктів при виборі саме архітектурного рішення. Правильніше буде сказати, що якщо користувачу необхідна велика ємність і/або швидка відповідь від застосунку, то цей пункт має бути найкритичнішим при виборі архітектури і скоріш за все в такому випадку рішення має бути прийнято в сторону мікросервісної архітектури розгорнутої в хмарі. Окрім того яку архітектуру обрати, важливість швидкої відповіді може вплинути на вибір мови програмування, хоча вже навіть зараз більшість мов програмування показують приблизно однакові результати продуктивності. Треба зважити на те, що якщо продуктивність для ПЗ має великий вплив, то необхідно взяти не просто критерій «продуктивність», а додати критерії «час відповіді», «використання ресурсів» та «ємність».

«Сумісність» є дуже суперечливим пунктом при виборі архітектури для вебзастосунку, тому що з однієї сторони вебзастосунок не має заважати або навіть якось співіснувати з іншими. Але це так. Наприклад, система, яка

займається синхронізацію продуктів між різними маркетплейсами впадає в залежність від існування і того як ці сервіси працюють, кожна зміна в них мусить змінюватись ПЗ для синхронізації. Окрім цього із ростом інтеграцій з платформами може зрости кількість змін необхідним для реалізації якогось нового функціоналу. «Сумісність» має 2 підпункти: «співіснування» і «інтерпорабельність». Перше як критерій може бути відкинуте у випадку, якщо сервер розміщується у хмарі, тому що усі проблеми із співіснуванням на себе бере провайдер. Але також у випадку коли сервісу не потрібно звертатись до налаштувань операційної системи чи чогось більш низькорівневого, даний пункт не повинен мати жодної ваги. «Інтерпорабельність» для даної задачі є більш цікавішим критерієм, якщо система залежить від інших, або якщо від системи залежатимуть інші. В першому випадку система має бути гнучкою при змінах в інших сервісах. В другому – бути стабільнішою для забезпечення стабільної роботи інших систем. Кожен з цих пунктів може бути важливим при написанні конкретних модулів, хоча і не є критичними при виборі архітектури. Окрім «інтерпорабельності», коли проєктуємий вебзастосунок має залежати від інших систем.

«Юзабіліті» – цей критерій є дуже важливою для успіху застосунку, але при виборі архітектури даний параметр не має грати жодної ролі. Так само як і «функціональна придатність», «юзабіліті» є ключовим критерієм хорошого застосунку, але при виборі архітектурного рішення, вважається, що продукт буде реалізованим зручним і зрозумілим для користувача.

«Надійність» в будь-якій системі як і «продуктивність» – є одним із найважливіших аспектів при побудові нового сервісу. В залежності від вимог один з них може бути більш важливим, при інших навпаки. Так само через високу критичність даного критерію необхідно розглядати кожен із його підпунктів окремо. «Зрілість» – на момент вибору найкращого архітектурного рішення цей пункт не має жодного значення, тому що сам проєкт ще на стадіях проєктування. «Відновлюваність» на системному рівні наразі не є дуже складною в реалізації і вона не може сильно вплинути на те, яка має бути

архітектура програмного забезпечення. Тобто або відновлення покладається на хмарного провайдера, або на операційну систему. Із розвитком мікросервісних архітектур стало складніше керувати великою кількістю цих сервісів, слідкувати за ними і тд. В 2014 році Google створили технологію, яка дозволяє вирішити цю проблему – Kubernetes [27]. Kubernetes – є диригентом усіх сервісів: дозволяє створити декілька реплік одного сервісу, знає як відновити цей сервіс після падіння, тощо. Надійність є одним із головним аспектів для бізнесу, тому що саме цей аспект вкаже на те, як часто і як довго ПЗ не буде приносити прибуток.

«Безпека» – є наразі одним із ключових аспектів будь-якого сервісу, один злам системи або витік даних до третіх осіб може вбити репутацію компанії. Базові концепції, наприклад, використання API ключів або шифрування чутливих даних користувачі, зазвичай не потребують обговорення при реалізації, а мають бути виконанні за замовчуванням. Ця деталь має бути необхідністю усюди. Але багато технологій мають свої нюанси безпеки, наприклад, те, в якому форматі передаються дані, або публічність і конфігурабельність способів шифрування цих даних. Найпростіше – це довірити безпеку даних своїх користувачів на якогось провайдера (AWS, Azure), тому налаштування безпеки на власних серверах вимагає багато ресурсів, як в працевлаштуванні відповідних спеціалістів, так і при закупівлі відповідного обладнання і програмного забезпечення.

Для усіх наступних (а також для вже працюючих над проєктом) програмістів критерій «підтримуванність» є найважливішим. Додавати новий функціонал або вносити правки в існуючу логіку мають різний ступінь тяжкості для добре структурованого коду і так званої «лапши», де все переплетено і нічого не зрозуміло. Важливо зазначити, що будь-який працюючий код можливо зрозуміти, але кількість часу необхідного для усвідомлення і розуміння алгоритму може сильно різнитись. Саме цей час грає ключову роль для замовника, якщо програміст за той самий час міг би зробити дві задачі замість однієї – це було б набагато краще. Тож фактор

підтримуванності ПЗ має бути так само важливий для замовника, як і для розробника. Але замість такого критерію як «підтримуванність», який окрім «можливості модифікації» включає в себе також «модульність» і «здатність до аналізу», має бути обраний більш зрозумілий «можливість модифікації». «Модульність» більш притаманний критерій саме для монолітного сервісу, де може взагалі не існувати будь-який розподіл на модулі, що звісно ускладнює будь-які нові зміни; в SOA кожен сервіс має бути як окремий модуль в системі. «Здатність до аналізу» може бути прописана не залежачи від типу архітектури або обраних технологій, логування і збір різних типів метрик може ставитись як на мікросервісні, так і монолітні застосунки. В свою чергу на складність модифікації може вплинути вибір архітектури/технології, саме тому це важливий критерій, який неможливо відкидати.

«Переносність» – не є найважливішим критерієм при більшості рішень, але він може бути одним з найкритичнішим, якщо застосунок має швидко розширюватись, або у бізнесу є план переходу з дешевого провайдера на дорожчий але при цьому надійніший для забезпечення більшої надійності. Є ситуації, коли даний критерій є важливим, але це дуже рідкі випадки.

Окрім критеріїв, які можна витягнути із стандарту, необхідно сказати про критерій «бюджету». Цей критерій може йти як регулятором важливості окремих критеріїв, таких як «продуктивність», «безпека» і інших, але через те, що він має вплив на вибір різних технологій, то найкраще буде виокремити його в окремий критерій.

Підсумовуючи усе вищесказане, можна виокремити наступні критерії: *продуктивність (час відповіді, використання ресурсів і ємність* – у випадку, якщо *продуктивність* є ключовим критерієм*), сумісність (*а саме *інтерпорабельність), надійність, безпека, складність модифікації, переносність, бюджет.*

2.2 Виокремлення технологічних рішень і їх аналіз

Визначивши критерії, за якими буде оцінюватися альтернатива архітектурного рішення, необхідно визначити альтернативи цих архітектурних рішень. Враховуючи аналіз еволюції архітектурних підходів необхідно виокремити 4 технічні рішення: архітектура, комунікація, інфраструктура та сховище даних.

2.2.1 Архітектура як технологічне рішення

Архітектура визначає архітектурний стиль при написанні програмного забезпечення: моноліт, SOA (Service-Oriented Architecture), мікросервіси або безсерверна архітектура, які є її основними компонентами. Вибір архітектури впливає на продуктивність, масштабованість, легкість підтримки та розвиток системи, а також на її відповідність бізнес-вимогам.

Моноліт в даному випадку розглядається як клієнт-серверний застосунок, де сервер представлений одним проєктом. Хоча існують рішення, які дозволяють поєднувати код клієнту і сервера, приховуючи багато деталей комунікації, як це реалізує Blazor [28], розроблений командою Microsoft Монолітна архітектура зазвичай вважається підходом, що є найпростішим для початкової розробки та підтримки малих і середніх проєктів. Однак, у великих проєктах, воно може стати важкою для масштабування та внесення змін через сильну взаємозалежність компонентів.

Навіть у рамках монолітного підходу можна впроваджувати модульність, створюючи окремі бібліотеки або функціональні модулі. Проте така модульність вважається рішенням на нижчому рівні архітектурного проєктування і не вирішує основних проблем, пов’язаних із масштабованістю чи складністю.

SOA – є наступним рівнем в розвитку архітектурних підходів, де система складається з кількох великих модулів або сервісів, які взаємодіють один з одним через стандартизовані протоколи, наприклад, HTTP чи SOAP. Ці сервіси працюють як незалежні системи, але забезпечують інтеграцію для досягнення загальної мети. Застосунок у рамках SOA може складатися з чотирьох-п’яти великих сервісів, кожен із яких виконує окрему логіку, наприклад, управління користувачами, обробка платежів чи аналітика. Основною перевагою SOA є незалежність модулів, що спрощує їхню підтримку, тестування та оновлення. Однак SOA не завжди є ефективним для розробки складних і динамічних систем.

Мікросервісна архітектура розвиває ідею SOA, але робить сервіси набагато меншими і більш спеціалізованими. Кожен мікросервіс зазвичай виконує одну або кілька функцій і може бути розроблений, розгорнутий і масштабований незалежно від інших сервісів. У великих системах кількість мікросервісів може досягати сотень або навіть тисяч, що дозволяє адаптувати систему до складних бізнес-вимог. Основна перевага мікросервісів полягає в їхній гнучкості та масштабованості, але їхня реалізація вимагає значних інвестицій у налаштуванні інфраструктури, моніторингу та управління.

Безсерверна архітектура є новітнім підходом, який суттєво змінює парадигму управління серверними ресурсами. Розробник фокусується на створенні окремих функцій, які виконуються в хмарі, а всі аспекти управління інфраструктурою бере на себе хмарний провайдер. Це знижує витрати на розробку та підтримку, забезпечує автоматичне масштабування та дозволяє значно прискорити процес розгортання. Одна безсерверна архітектура підходить не для всіх сценаріїв, і її використання може бути економічно невигідних для довготривалих задач або при високій інтенсивності використання.

2.2.2 Комунікація як технологічне рішення

Комунікація визначає спосіб спілкування між сервісами, або між клієнтом і сервером у випадку моноліту. Будуть розглянуті найпоширеніші способи комунікації: REST, GraphQL, gRPC та асинхронна комунікація (чергові системи).

REST є найбільш поширеним і простим способом взаємодії в веброзробці. Він базується на HTTP-протоколі та стандартних методах: GET, POST, PUT, DELETE. REST є одним із найлегших в імплементації способів комунікації між клієнтом і сервером. REST має широку підтримку і може бути реалізований використовуючи майже будь-яку мову програмування. Окрім цього REST підходить для написання простих систем із великою кількістю CRUD (Create Read Update Delete) сервісів.

GraphQL – це мова запитів і середовище виконання для API, яка дозволяє клієнтам запитувати тільки ті дані, які їм потрібні. GraphQL був створений, щоб вирішити проблему в REST, коли сервер повертає занадто багато або навпаки недостатньо інформації. Головною перевагою GraphQL є зменшення кількості даних, які передаються, і витягнення даних одним викликом, коли в REST для цього потрібно було зробити 2 або навіть 3 запити. Але такі переваги підвищують складність реалізації даної системи в динамічних системах із багатьма змінними.

gRPC – це сучасний фреймворк для віддалених викликів процедур, який має високу продуктивність завдяки бінарній передачі даних за протоколом HTTP/2, також має можливість потокової передачі даних. Дана технологія добре підходить для комунікації між сервісами всередині системи, а також має автоматичну генерацію коду на основі схеми, наданої через proto файли. Використання HTTP/2 є як і позитивним аспектом, так і негативним, тому що браузери не мають підтримки даної версії протоколу HTTP. Окрім цього проєкт впадає в високу залежність від інструментів Google Protocol Buffers [29]. А також вимагає достатньо знань в конфігуруванні gRPC.

Асинхронна комунікація реалізується через системи черг, наприклад, RabbitMQ [30], Apache Kafka [31] або Azure Service Bus [32]. Вона дозволяє компонентам системи працювати незалежно. Даний підхід підвищує надійність системи, тому що навіть у час коли сервіс є недоступним, повідомлення не буде втрачено, що робить дану систему підходящою для високонавантажених систем і обробки великих обсягів даних. Але така надійність і приносе проблеми пов’язані із великими затратами на розробку і підтримку, ускладнює діагностику і може мати невеликі затримки у відповідях.

2.2.3 Інфраструктура як технологічне рішення

Інфраструктура визначає де буде розміщена система, як вона буде розгорнута та яким чином буде забезпечуватися її робота. Основні варіанти: локальні сервери, віртуальні машини, контейнери і хмарні провайдери.

Локальні сервери це комп’ютери або мережа комп’ютерів, над якими власник має повний контроль в плані апаратного і програмного забезпечення. Такі сервери не вимагають регулярних витрат на оренду ресурсів, тільки затрати на спожиту електроенергію і затрати на їх обслуговування, що спрощує прогнозовані витрати. Це ж є і проблемою, тому що для розгортання системи на локальному сервері, цей сервіс необхідно придбати, що може коштувати чимало, також це буде вимагати багато коштів для заміни якихось складових системи, якщо вони з якихось причин вийдуть з ладу. Але найголовнішим мінусом такої інфраструктури є їхня складність в масштабуванні.

Хмарні провайдери ж надають в оренду свої сервери і особисто займаються доглядом за обладнанням та керуванням ресурсів, які необхідні застосунку, тобто хмарний провайдер може забрати деяке залізо, якщо застосунок не виконує майже ніяких обчислень і навпаки додати додаткові

ресурси для обчислень в пікові години. Окрім цього хмарний провайдер відповідає за автоматичне резервне копіювання та відновлення, а також надає доступ до оренди передових технологій і тільки з такою інфраструктурою стає можливим використання безсерверних архітектур.

Віртуальні машини представляє собою щось середнє між локальним сервером та хмарним провайдером. Це один ПК або їх група, які розташовані на хмарі, але вони не можуть бути автоматично масштабовані, це є просто комп’ютером на іншому кінці світу із віддаленим доступом до нього.

Контейнери, наприклад Docker [33], забезпечують легкий спосіб ізоляції застосунків та їхньої залежності. Вони можуть бути запущені як на локальному сервері, так і у хмарного провайдера, дуже легко налаштовуються в CI/CD процесах. Але при цьому вимагають великої кількості знань про контейнеризацію та потребують додаткові додаткові витрати на моніторинг та безпеку.

2.2.4 Сховище даних як технологічне рішення

Бази даних визначають спосіб зберігання, організації, доступу та маніпуляції даними. Їхній вибір впливає на продуктивність, масштабованість та простоту реалізації системи. Основними типами є SQL (Structured Query Language) та NoSQL (Not Only SQL).

SQL бази даних – це реляційні системи управління базами даних, які організовують дані у вигляді таблиці із чіткими схемами. Чіткі схеми дозволяють легко визначати зв’язки між таблицями. Для забезпечення надійності операцій в SQL базах даних існує підтримка транзакцій через механізми ACID (Atomicity, Consistency, Isolation, Durability). SQL-бази мають велику екосистему для аналізу даних, оптимізації запитів та роботи з великими обсягами даних, а також підходять для застосунків із чітко визначеними структурами даних, наприклад, ERP- або CRM-системи. З іншої сторони

вимоги до строгої схеми даних ускладнюють внесення змін у структуру без впливу на існуючі дані. Горизонтальне масштабування SQL бази даних є дуже складним випробуванням, тому здебільшого такі БД масштабують вертикально, збільшуючи потужності сервера на якому знаходиться база даних.

NoSQL бази даних – це нереляційні системи управління даними, які не обмежуються табличними структурами і можуть зберігати дані в різних форматах, таких як документи, графи, ключ-значення або колонки. Відсутність чіткої схеми дозволяє швидко змінювати структуру даних і дозволяє легко масштабувати систему горизонтально додаючи нові сервери до кластеру. Нереляційні бази даних оптимізовані для швидкого запису і читання даних і вони ідеально підходять для зберігання неструктурованих даних, такиї як мультимедійні файли чи лог-файли. Окрім цього вони були розроблені для обробки великих потоків неструктурованих даних (наприклад, соціальні мережі), для масштабованості динамічних і високонавантажених застосунків і для зберігання та обробки даних із нерегулярною структурою (наприклад, каталоги продуктів у маркетплейсах). Але відсутність чіткої схеми унеможливлює використання джойн-запитів, що ускладнює обробку взаємозалежних даних. Окрім цього відсутність стандартів ускладнює міграцію з однієї системи управління базами даних на іншу, а також деякі з них можуть підтримувати транзакції, деякі – ні.

2.3 Визначення базових коефіцієнтів

Кожне рішення має свої особливості того, для чого їх створили. Будь-яку технологію створюють з метою вирішити якусь проблему, саме тому всі технології є унікальними, мають свої особливості і найголовніше – переваги і недоліки один перед одним.

Кожна варіанта із технологічних рішень має свій коефіцієнт відносно інших варіантів відповідно до якогось критерію. Але не всі варіанти мають вплив на будь-який критерій. Наприклад, вибір способу комунікації між сервісами мало як впливає на те, який необхідний бюджет для реалізації даного рішення. Обрана комунікація може бути вагомою, якщо у замовника вже є сформована команда, яка не знайома із даною технологією, але даний нюанс буде опрацьований пізніше. У таблицях 2.1-2.4 представлені коефіцієнти якості технічного рішення відповідно до критерію.

Таблиця 2.1 – Коефіцієнти якості архітектури відповідно до критеріїв

Критерій	**Моноліт**	**SOA**	**Мікросервіси**	**Безсерверна**
Продуктивність	0,2	0,25	0,25	0,3
Час відповіді	0,3	0,22	0,18	0,3
Використання ресурсів	0,3	0,2	0,15	0,35
Ємність	0,1	0,2	0,35	0,35
Сумісність	0,35	0,35	0,2	0,1
Надійність	0,18	0,22	0,28	0,32
Безпека	0,18	0,22	0,3	0,3
Складність модифікації	0,4	0,3	0,15	0,15
Переносність	0,25	0,3	0,35	0,1

Таблиця 2.2 – Коефіцієнти якості способу комунікації відповідно до критеріїв

Критерій	**REST**	**GraphQL**	**gRPC**	**Черги**
Продуктивність	0,25	0,3	0,3	0,15
Час відповіді	0,25	0,3	0,3	0,15
Використання ресурсів	0,23	0,31	0,23	0,23
Ємність	0,2	0,2	0,2	0,4
Надійність	0,2	0,2	0,25	0,35

Таблиця 2.3 – Коефіцієнти якості інфраструктури відповідно до критеріїв

Критерій	**Локальний сервер**	**Хмарний провайдер**
Надійність	0,2	0,8
Безпека	0,3	0,7
Бюджет	0,7	0,3

Таблиця 2.4 – Коефіцієнти якості баз даних відповідно до критеріїв

Критерій	**SQL**	**NoSQL**
Продуктивність	0,5	0,5
Час відповіді	0,6	0,4
Використання ресурсів	0,4	0,6
Ємність	0,4	0,6
Складність модифікації	0,6	0,4

Кожен рядок в таблицях має в сумі давати одиницю. Тобто даний коефіцієнт показує відношення до інших варіантів технічного рішення в межах одного критерію.

Усі матриці важливості технічних рішень відповідно до критеріїв мають бути перемножені, тобто зібрані в єдину матрицю альтернатив і критеріїв. $\{T_1, T_2, \dots, T_n\}$ – множина технічних рішень, де n – загальна кількість рішень. Для кожного технічного рішення T_i $(i = 1, 2, \dots, n)$ має множину можливих варіантів рішення $\{V_{i1}, V_{i2}, \dots, V_{im_i}\}$, де m_i – кількість варіантів для рішення T_i. $\{C_1, C_2, \dots, C_k\}$ – множина критеріїв, за якими оцінюють варіанти технічних рішень, де k – загальна кількість критеріїв. Коефіцієнт варіанти є функцією $K(V_{ij}, C_l) \in \mathbb{R}$, де $0 \leq K(V_{ij}, C_l) \leq 1$. Коефіцієнт альтернативи розраховується за формулою (2.1).

$$A_z(C_l) = \sum_{i=1}^{n} \frac{K(V_{ij}C_l)}{n}, \tag{2.1}$$

де $z = 1,2, \dots, (n * m)$;

$j = z - m * (i - 1)$.

Окрім визначення коефіцієнтів якості технічних рішень, необхідно визначити вагу (важливість) критеріїв ($W(C_l) \in \mathbb{R}$, де $0 \leq W(C_l) \leq 1$ і $\sum_{l=1}^{k} W(C_l) = 1$) відповідно один до одного. Дані коефіцієнти в подальшого будуть корегуватись відповідно до вимог користувача, але навіть в базовій конфігурації неможливо сказати, що, наприклад, надійність буде настільки ж або менш важлива ніж продуктивність. Можливо для деяких замовників це

буде правдою, але для більшості надійність завжди буде переважати над продуктивністю. Ваги критеріїв представлені в таблиці А.1. Сума коефіцієнтів виходить за одиницю. Це пов’язано з тим, що деякі критерії не будуть залучені в обчислення, при наявності інших. Тобто якщо продуктивність є важливим критерієм для застосунку, то замість нього будуть використані три інших критерія: час відповіді, використання ресурсів і ємність. При фінальному застосуванні ваг критеріїв усі коефіцієнти ваги мають бути нормалізовані.

Таблиця 2.5 – Ваги критеріїв

Критерій	Вага
Продуктивність	0,2
Час відповіді	0,09
Використання ресурсів	0,02
Ємність	0,09
Сумісність	0,05
Надійність	0,2
Безпека	0,15
Складність модифікації	0,2
Переносність	0,05
Бюджет	0,15

2.4 Корегування коефіцієнтів відповідно до вимог замовника

Користувачеві ставиться декілька раундів запитань, із відповідей на які редагуються базові ваги і коефіцієнти.

Перший раунд запитань дає можливість переоцінити важливості критеріїв відповідно один до одного. В результаті даного раунду ваги критеріїв мають бути відкоректовані під конкретні вимоги користувача. Запитання мають наступний шаблон: «Наскільки важливим є {назва критерію} для Вашої системи?». Проситься оцінити лише критерії першого порядку: продуктивність, сумісність, надійність, безпека, складність модифікації, переносність та бюджет. Усі оцінки надаються в межах від 1 до 9,

але можуть використовуватись і інші методи оцінювання, які повинні нормалізуватись в межах від 0 до 2, де 1 означає незмінність базової ваги критерію. Якщо «продуктивність» має важливість від 7, то алгоритм повинен попросити оцінити й критерії другого порядку: час відповіді, використання ресурсів та ємність. Дані оцінки відповідно перемножуються із базовими вагами даючи правильніший розподіл важливості критеріїв для конкретної задачі.

Окрім оцінювання критеріїв користувачеві надаються уточнювальні запитання для корегування значимості альтернатив в залежності від досвіду команди. Наприклад, якщо в команди немає досвіду із gRPC, але є тільки досвід із REST, то результати до врахування досвіду команди зберігаються для опції «краще рішення, але команді потрібен час для донавчання», а самі коефіцієнти редагуються так, щоб саме технології, із якими знайома команда залишались найкращими. Це може бути як помноження усіх коефіцієнтів даної технології на 2 і зменшення інших вдвічі відповідно, так і будь-який інший механізм урегулювання. Після кожного такого запитання матриця альтернатив-критеріїв перераховується.

2.5 Корегування коефіцієнтів відповідно до сумісності технічних рішень

Одним з останніх етапів даного алгоритму є корегування коефіцієнтів альтернатив.

Занулення – для альтернатив, чиї технічні рішення не можуть бути реалізовані, наприклад, безсерверна архітектура із локальним сервером, тому що безсерверна архітектура перекладає проблему інфраструктури на хмарного провайдера за замовчуванням.

Для окремих критеріїв коефіцієнт може бути збільшений, якщо технічна зв’язка є потужним механізмом, наприклад, для надійності зв’язка мікросервісів із чергами є дуже потужним інструментом через легку

відновлюваність і стабільну роботу мікросервісів і через плюси асинхронної комунікації між сервісами. З іншої сторони, розглядаючи ту саму комбінацію, можна сказати, що її складніше реалізувати і вона також буде складнішою в підтримці, особливо якщо є якась проблема в комунікації між сервісами.

2.6 Вираховування найкращої альтернативи

Маючі коефіцієнти, які відповідають сумісності технологій і вимогам користувача, можна розраховувати коефіцієнт кожної альтернативи відповідно до усіх критеріїв (2.2).

$$x^* = \arg\max_{x \in X} \left\{ \sum_{l=1}^{k} \frac{1}{n} \sum_{i=1}^{n} K\left(V_{ij}, C_l\right) * W(C_l) \right\},$$
$$Score_z = \sum_{l=1}^{k} A_{zl}(C_l) * W(C_l). \quad (2.2)$$

Варто зауважити, що після виведення загальної матриці альтернатив-критеріїв (до корегування коефіцієнтів відповідно до вимог користувача, але із урахуванням сумісності технологій) не повинно бути найкращої альтернативи, тобто усі альтернативи мають мати однакові коефіцієнти для того, щоб унеможливити «улюблених» технологій алгоритму, які будуть видаватись для майже будь-яких вхідних даних.

Після того як усі фінальні коефіцієнти підраховані, обирається альтернатива із найбільшим коефіцієнтом, яка і стає архітектурним рішенням, яке буде порекомендоване користувачеві відповідно до результатів роботи алгоритму.

Якщо ж є декілька альтернатив, які мають однаково високі коефіцієнти найкращого вибору, то видаються два можливих варіанти архітектури із зазначенням, яка трохи краща.

2.7 Застосування мовних моделей для аналізування результатів

Після того як базовий алгоритм виконав свою роботу, усі дані про хід його виконання, отримані результати та проміжні обчислення можуть бути передані мовній моделі (LLM) для глибшого аналізу. Це дозволяє не лише підтвердити обґрунтованість вибору алгоритму, але й ідентифікувати можливі недоліки чи ризики у запропонованому рішенні. Мовна модель здатна надати детальні рекомендації стосовно реалізації обраного архітектурного підходу, враховуючи не тільки технічні аспекти, але й специфіку бізнес-вимог, ресурси, доступні для проєкту, та інші ключові фактори.

Визначивши найкращий варіант комбінації технічних рішень для конкретного користувача, мовна модель може взяти на себе роль інтерактивного асистента. Вона буде знати контекст вибору: чому саме ці технології обрані, які альтернативи було відхилено і з яких причин, а також як вони співвідносяться з вимогами проєкту. Такий підхід дозволяє зробити підтримку більш персоналізованою та релевантною.

У процесі реалізації архітектурного рішення асистент у вигляді LLM може допомогти вирішувати конкретні технічні проблеми, що виникають, надаючи рекомендації з оптимізації, налаштування інфраструктури, інтеграції компонентів чи обробки даних. Наприклад, мовна модель може запропонувати найкращі практики для налаштування мікросервісів, допомогти з вибором бібліотек або інструментів, створити приклади коду для інтеграції компонентів чи оптимізації запитів до баз даних.

Крім того, модель може виконувати функцію постійного моніторингу та адаптації архітектурного рішення до нових умов. У разі змін у вимогах користувача або виявлення нових технологічних можливостей, LLM здатна запропонувати зміни до існуючого плану та допомогти їх імплементувати. Це створює циклічний процес удосконалення, у якому мовна модель стає не лише інструментом підтримки, а й частиною стратегічного планування проєкту.

Таким чином, інтеграція роботи алгоритму та мовної моделі забезпечує більш глибокий аналіз, знижує ризики реалізації неправильного архітектурного рішення та підвищує ефективність розробки. Це дозволяє користувачу отримати не тільки найкращі технічні рекомендації, але й постійну підтримку у процесі реалізації, що значно спрощує роботу над складними проєктами.

3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ВИБОРУ АРХІТЕКТУРИ НА ОСНОВІ АНАЛІЗУ ВИМОГ ПРИКЛАДНОЇ ЗАДАЧІ

3.1 Розробка системи керування базовими вагами і коефіцієнтами

Змодельований алгоритм хоч і працює через динамічну систему ваг, але потребує базові коефіцієнти, щоб більш точну стартову точку для аналізування. Звісно дані коефіцієнти можна зберігати і в якомусь файлі не переймаючись за зручність їх зміни. Але на початкових стадіях тестування і адаптації ваг алгоритму їх доведеться часто змінювати, тому розробка зручного інтерфейсу є необхідністю, яка збереже чимало часу.

Система представляє собою вебзастосунок. Застосунок розроблений суто для адміністраторів, але саме через це необхідно мати процес авторизації користувача для забезпечення мінімального захисту від шкідливої маніпуляції даними. Окрім цього увесь застосунок представлений у двох темах: світла і темна. Усі скріншоти в дослідженні будуть наведенні із застосуванням світлої теми.

Логічно система ділиться на декілька моделей: технології та їх варіанти, критерії, альтернативи і регулятори. Останні – це або автоматичні регулятори ваг, наприклад, занулення альтернатив із несумісним набором технологій, або ж питання, які може задати саме алгоритм, щоб врівноважити ваги в одну чи іншу сторону. Окрім цього необхідно зазначити декілька додаткових моделей: чати, повідомлення, користувачі і ролі. На рисунку 3.1 зображена схема розподілу цих моделей за модулями і взаємозв’язків між цими моделями.

Тобто технологія має декілька варіант, критерії можуть мати підкритерії (які будуть використанні за конкретних умов), кожен з варіантів має свій коефіцієнт відповідно до специфічних критеріїв. Регулятори мають бути застосованими за конкретними критеріями, і за конкретним набором варіантів технологічних рішень. Чат має багато повідомлень, чат відноситься до конкретного користувача, а користувач в свою чергу має відповідну роль.

Кожен чат має контекст вимог користувача, який складається з важливості критеріїв та списку варіантів технологічних рішень, з якими знайома команда розробки.

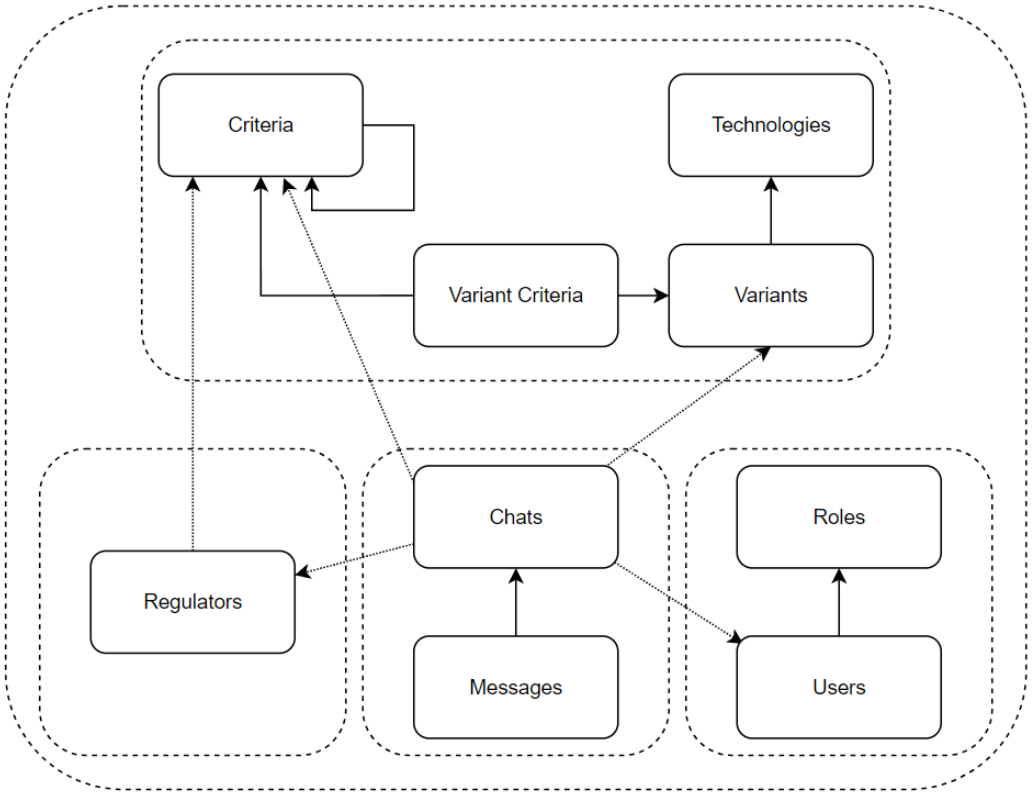

Рисунок 3.1 – Схема розподілу моделей системи за модулями і взаємозв’язки між моделями

3.1.1 Розробка інтерфейсу для керування критеріями та їх коефіцієнтами значимості між собою

Інтерфейс для керування критеріями (рис. 3.2) відображає усі існуючі критерії, коефіцієнт кожного з них відносно інших.

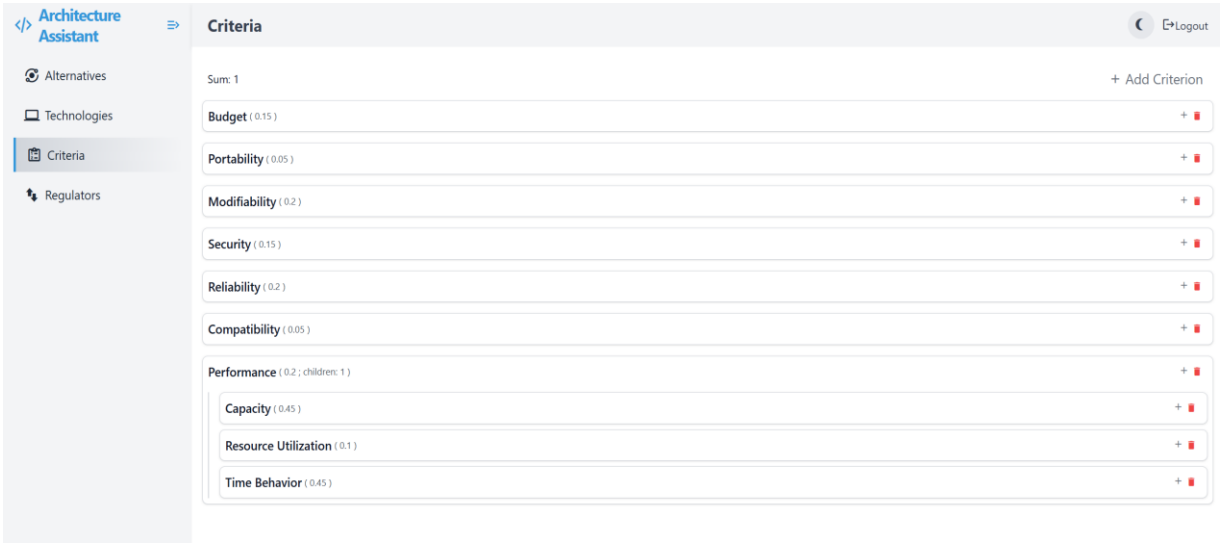

Рисунок 3.2 – Інтерфейс керування критеріями

Вгорі меню відображається сума всіх коефіцієнтів. Для дотримання балансу бажано тримати це значення рівним одиниці.

Також на прикладі критерію «Performance» можна помітити наявність критеріїв нижчого рівня, які будуть застосовані, тільки якщо важливість батьківського критерію користувач оцінив в 7 або вище балів за 10-бальною шкалою.

Сума коефіцієнтів всередині критеріїв нижчого рівня для конкретного батьківського критерію також має становити 1.

Існують кнопки як для додавання критеріїв, так і для їх видалення. При кліку на ім’я критерію відкривається меню його редагування (рис. 3.3). При створенні критерію відображається те саме модальне вікно тільки без введених даних.

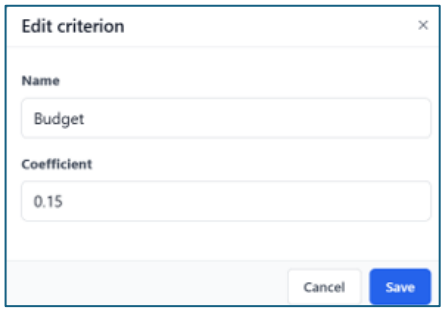

Рисунок 3.3 – Модальне вікно редагування критерію

3.1.2 Розробка інтерфейсу для керування технологічними рішеннями

Інтерфейс для керування можливими технологічними відображає ці рішення (рис. 3.4). Є можливість створювати нові, видаляти і редагувати існуючі.

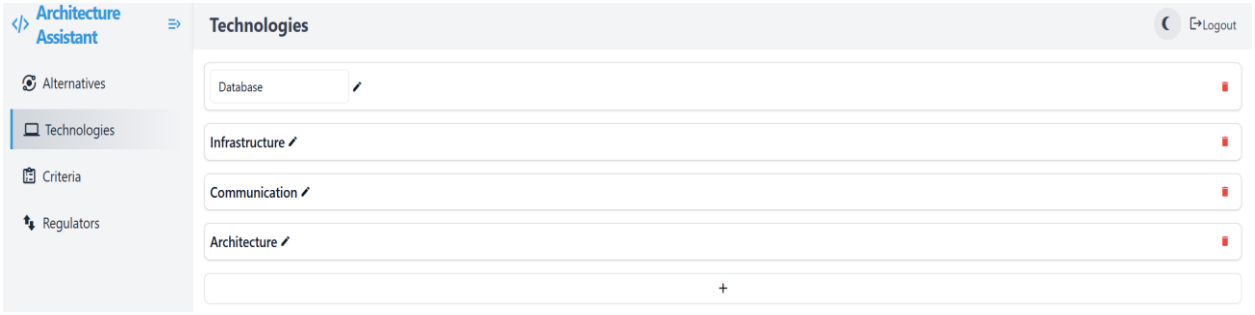

Рисунок 3.4 – Інтерфейс керування технологічними рішеннями

Приклад модального вікна додавання технологічного рішення представлений на рисунку 3.5.

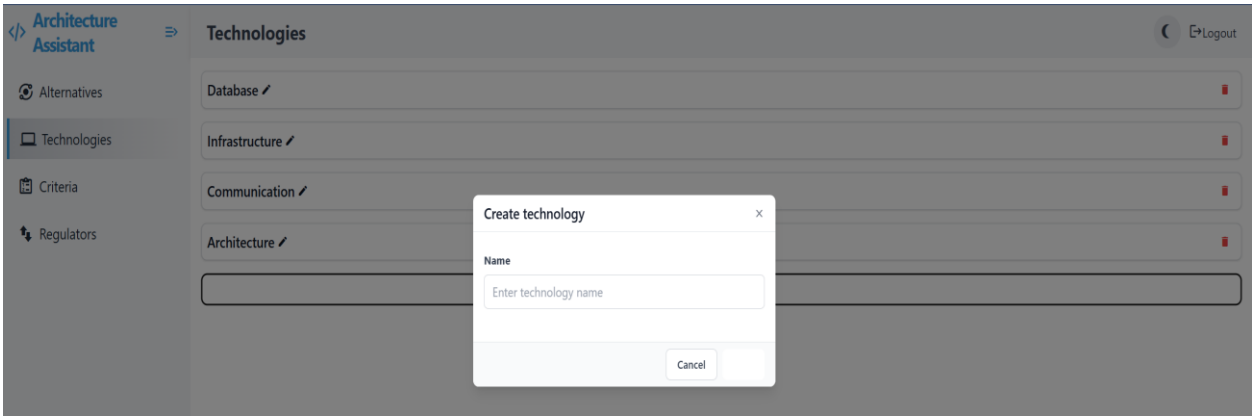

Рисунок 3.5 – Модальне вікно створення критерію

При кліку на назву технологічного рішення застосунок перенаправляє адміністратора до інтерфейсу керування варіантами технологічних рішень (рис. 3.6). Дана сторінка відображається у вигляді таблиці, в якій в колонках відображаються варіанти технологічного рішення, а в рядках критерії. На перетину варіанти технологічного рішення і критерію вказується

коефіцієнт вагомості цієї варіанти за цим коефіцієнтом відносно інших варіант. Тобто сума коефіцієнтів в рядку має бути рівною одному, при невідповідності цього значення адміністраторові буде показано відповідне повідомлення-помилка (рис. 3.7).

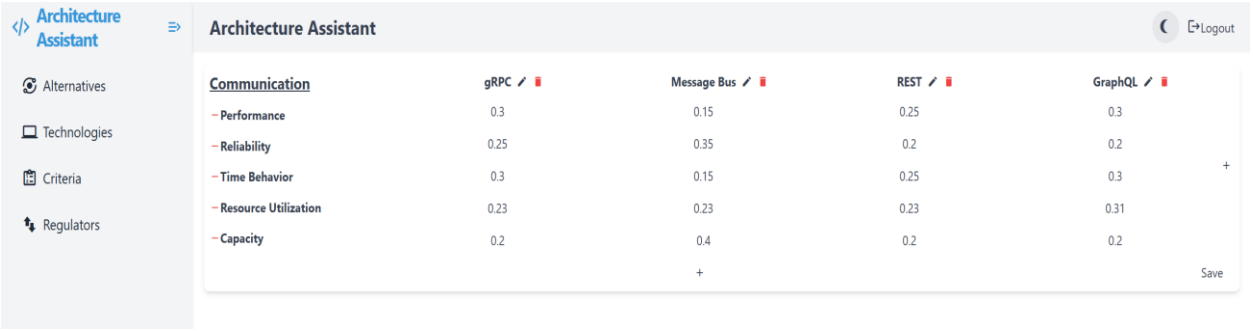

Рисунок 3.6 – Інтерфейс керування варіантами технологічних рішень

Рисунок 3.7 – Помилка «Сума коефіцієнтів має бути рівною одному»

Імена варіант технологічних рішень можливо редагувати, натиснувши на іконку олівця біля відповідної варіанти. Натиснувши на іконку смітника дозволяється видалити варіанту технологічного рішення.

Для того, щоб додати нову варіанту необхідно натиснути «плюс» з правого краю таблиці. Після натискання адміністраторові відображається модальне вікно (рис. 3.8) для додавання нової варіанти для даного технічного рішення, де застосунок просить ввести лише назву.

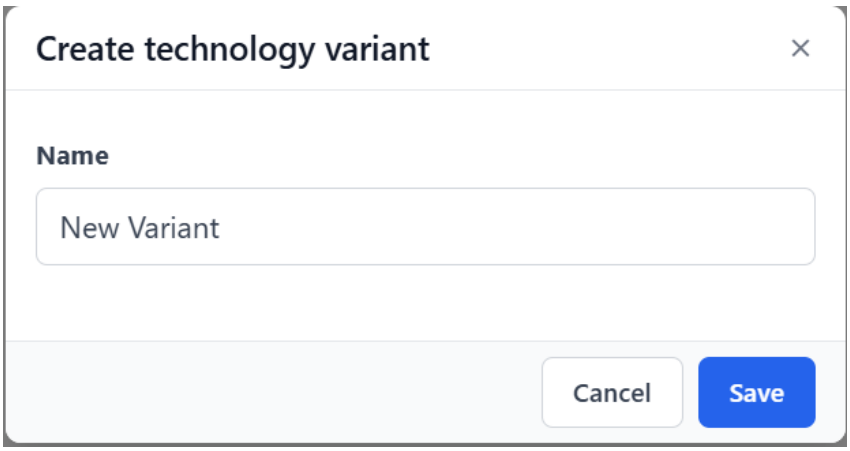

Рисунок 3.8 – Модальне вікно для додавання варіанту технічного рішення

Як було зазначено вище рядки це критерії за якими порівнюються варіанти. Змінювати ці варіанти в даному меню забороняється, але дозволяється видалити критерій для оцінки даної варіанти – для цього необхідно натиснути на іконку мінуса зліва від назви критерія. Для того щоб навпаки додати критерій необхідно натиснути на плюс внизу таблиці і обрати критерій у розкривному списку у модальному вікні (рис. 3.9), яке відкрилось.

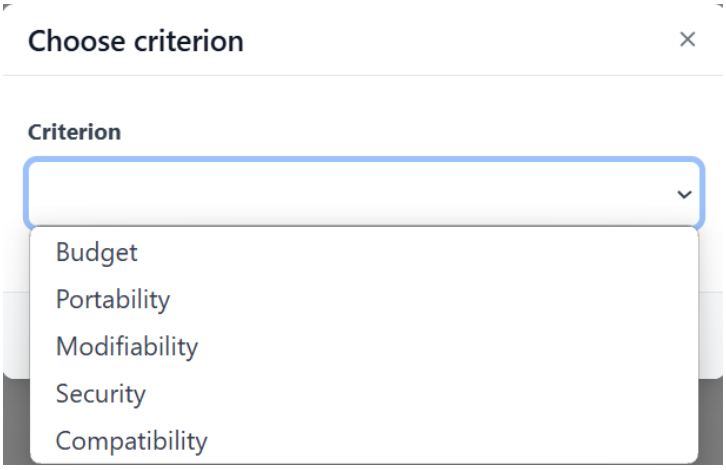

Рисунок 3.9 – Модальне вікно для вибору критерію

Також в правому нижньому куті таблиці можна помітити кнопку збереження: при додаванні критерія або варіанти усі дані зберігаються одразу, але самі коефіцієнти будуть збережені в базу даних тільки при натисканні на кнопку збереження.

3.1.3 Розробка інтерфейсу для перегляду альтернатив

Програмний застосунок – є набором якихось варіантів технічних рішень. Враховуючи, що різні технічні рішення мають особисті критерії оцінювання і різні коефіцієнти в залежності від критерію, то наявність загальної таблиці відображення всіх можливих альтернатив стає необхідністю, ніж простим бонусом.

На рисунку 3.10 відображається перелік усіх можливих альтернатив і їх ваги за конкретними критеріями. В останній колонці відображається сума коефіцієнтів альтернативи, яка є фінальним значенням до накладання регуляторів. Тобто дане значення показує, наскільки альтернативи стабілізовані при базових коефіцієнтах. Дане відображення допоможе домогтись найбільш рівномірних коефіцієнтів при базових вхідних даних. Тож при наданні рекомендацій користувачеві стосовно архітектури програмного забезпечення можна гарантувати вибір конкретної альтернативи в залежності від вимог користувача, а не від базових даних асистента.

Alternatives	Performance	Compatibility	Reliability	Security	Modifiability	Portability	Time Behavior	Resource Utilization	Capacity	Budget	Sum
Monolith X REST X Local Server X SQL	0.237	0.087	0.145	0.120	0.250	0.063	0.287	0.233	0.175	0.175	0.41
Monolith X REST X Local Server X NoSQL	0.237	0.087	0.145	0.120	0.200	0.063	0.238	0.282	0.225	0.175	0.40
Monolith X REST X Cloud X SQL	0.237	0.087	0.295	0.220	0.250	0.063	0.287	0.233	0.175	0.075	0.44
Monolith X REST X Cloud X NoSQL	0.237	0.087	0.295	0.220	0.200	0.063	0.238	0.282	0.225	0.075	0.43
Monolith X GraphQL X Local Server X SQL	0.250	0.087	0.145	0.120	0.250	0.063	0.300	0.253	0.175	0.175	0.42
Monolith X GraphQL X Local Server X NoSQL	0.250	0.087	0.145	0.120	0.200	0.063	0.250	0.302	0.225	0.175	0.41
Monolith X GraphQL X Cloud X SQL	0.250	0.087	0.295	0.220	0.250	0.063	0.300	0.253	0.175	0.075	0.45
Monolith X GraphQL X Cloud X NoSQL	0.250	0.087	0.295	0.220	0.200	0.063	0.250	0.302	0.225	0.075	0.44
Monolith X gRPC X Local Server X SQL	0.250	0.087	0.158	0.120	0.250	0.063	0.300	0.233	0.175	0.175	0.42
Monolith X gRPC X Local Server X NoSQL	0.250	0.087	0.158	0.120	0.200	0.063	0.250	0.282	0.225	0.175	0.42
Monolith X gRPC X Cloud X SQL	0.250	0.087	0.307	0.220	0.250	0.063	0.300	0.233	0.175	0.075	0.45
Monolith X gRPC X Cloud X NoSQL	0.250	0.087	0.307	0.220	0.200	0.063	0.250	0.282	0.225	0.075	0.45
Monolith X Message Bus X Local Server X SQL	0.212	0.087	0.182	0.120	0.250	0.063	0.262	0.233	0.225	0.175	0.42
Monolith X Message Bus X Local Server X NoSQL	0.212	0.087	0.182	0.120	0.200	0.063	0.212	0.282	0.275	0.175	0.42
Monolith X Message Bus X Cloud X SQL	0.212	0.087	0.333	0.220	0.250	0.063	0.262	0.233	0.225	0.075	0.45
Monolith X Message Bus X Cloud X NoSQL	0.212	0.087	0.333	0.220	0.200	0.063	0.212	0.282	0.275	0.075	0.45
SOA X REST X Local Server X SQL	0.250	0.087	0.155	0.130	0.225	0.075	0.267	0.208	0.200	0.175	0.41
SOA X REST X Local Server X NoSQL	0.250	0.087	0.155	0.130	0.175	0.075	0.217	0.258	0.250	0.175	0.41
SOA X REST X Cloud X SQL	0.250	0.087	0.305	0.230	0.225	0.075	0.267	0.208	0.200	0.075	0.44
SOA X REST X Cloud X NoSQL	0.250	0.087	0.305	0.230	0.175	0.075	0.217	0.258	0.250	0.075	0.44
SOA X GraphQL X Local Server X SQL	0.263	0.087	0.155	0.130	0.225	0.075	0.280	0.228	0.200	0.175	0.42
SOA X GraphQL X Local Server X NoSQL	0.263	0.087	0.155	0.130	0.175	0.075	0.230	0.277	0.250	0.175	0.42
SOA X GraphQL X Cloud X SQL	0.263	0.087	0.305	0.230	0.225	0.075	0.280	0.228	0.200	0.075	0.45
SOA X GraphQL X Cloud X NoSQL	0.263	0.087	0.305	0.230	0.175	0.075	0.230	0.277	0.250	0.075	0.45
SOA X gRPC X Local Server X SQL	0.263	0.087	0.167	0.130	0.225	0.075	0.280	0.208	0.200	0.175	0.42
SOA X gRPC X Local Server X NoSQL	0.263	0.087	0.167	0.130	0.175	0.075	0.230	0.258	0.250	0.175	0.42
SOA X gRPC X Cloud X SQL	0.263	0.087	0.318	0.230	0.225	0.075	0.280	0.208	0.200	0.075	0.45
SOA X gRPC X Cloud X NoSQL	0.263	0.087	0.318	0.230	0.175	0.075	0.230	0.258	0.250	0.075	0.45
SOA X Message Bus X Local Server X SQL	0.225	0.087	0.193	0.130	0.225	0.075	0.242	0.208	0.250	0.175	0.42

Рисунок 3.10 – Перелік усіх можливих альтернатив і їх ваги за конкретними критеріями

Даний інтерфейс існує тільки для перегляду, тож додати якісь альтернативи, або навпаки видалити якісь дані є неможливим.

3.1.4 Розробка інтерфейсу для керування регуляторами

Дане меню дозволяє керувати регуляторами, які підвищують або понижують коефіцієнт якоїсь альтернативи. Тільки перейшовши адміністратор бачить лише перелік усіх регуляторів (рис. 3.11). Здебільшого тут мають існувати регулятори, які показують як технологічні рішення співіснують, або які уточнюючі питання можна поставити користувачеві. Регулятори можна як редагувати, так і видаляти.

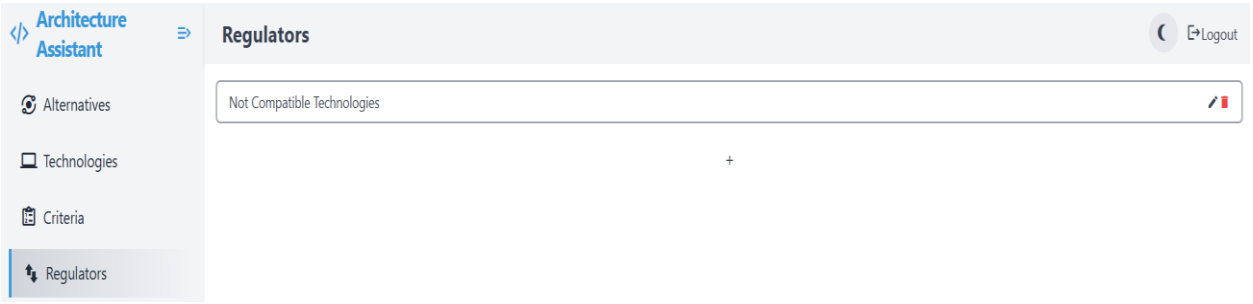

Рисунок 3.11 – Інтерфейс для керування регуляторами

Щоб додати новий регулятор необхідно натиснути на іконку плюса під усіма регуляторами. Після натискання замість іконки плюса з’явиться покрокове меню заповнення інформації про регулятор. На першому кроці (рис. 3.12) необхідно ввести назву регулятора (її можна зазначити на будь-якому кроці після) і обрати тип регулятора: автоматичний – активується без участі користувача, питання – ставить питання користувачеві із булевим варіантом відповіді, або оціночним від 1 до 9.

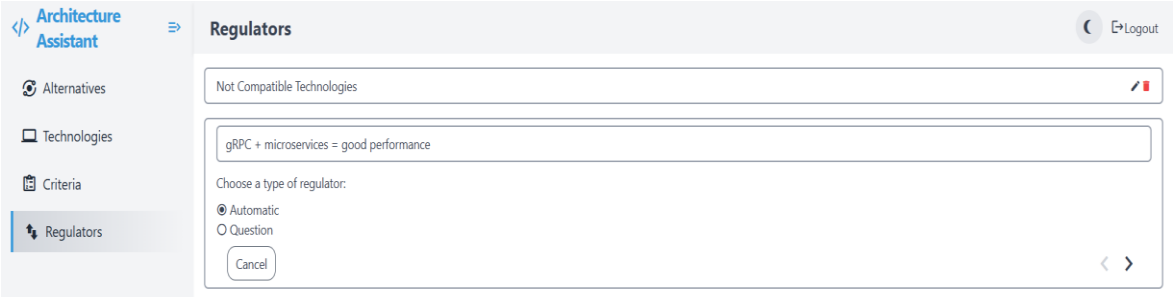

Рисунок 3.12 – Перший крок додавання регулятора: обрання типу регулятора

В залежності від типу регулятора кроки додавання можуть відрізнятись. При додаванні автоматичного регулятора адміністратор має пройти через наступні кроки: другий крок (рис. 3.13) – необхідно визначити альтернативи, до яких буде застосовано регулятор: все, що має зв’язку gRPC та мікросервіси. Після цього, на кроці 3 (рис. 3.14), вимагається обрати критерії, за якими буде застосований регулятор. Дані кроки дозволяють визначити, які альтернативи і за якими критеріями необхідно регулювати.

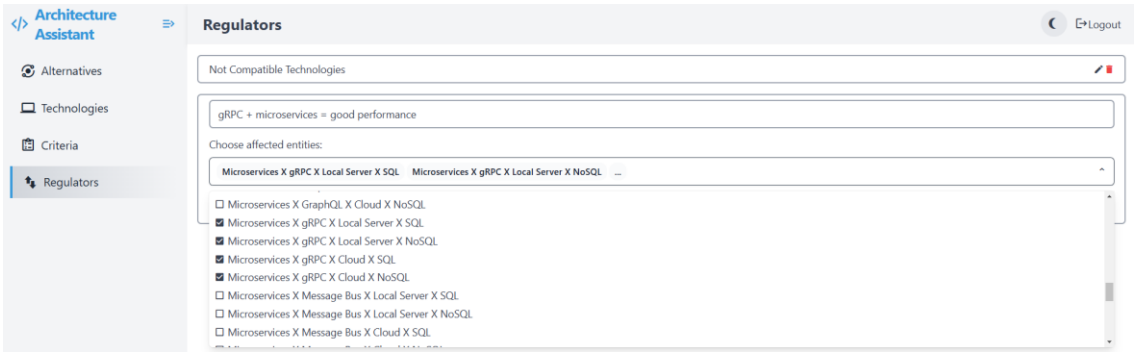

Рисунок 3.13 – Другий крок додавання регулятора (автоматичний регулятор): вибір альтернатив

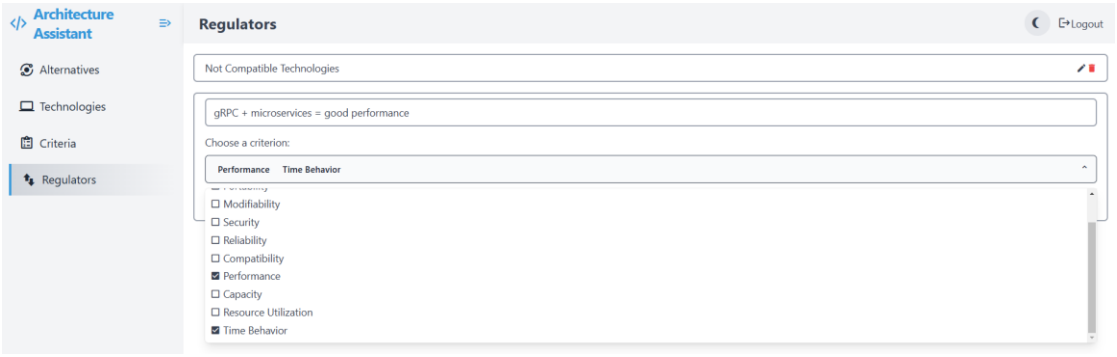

Рисунок 3.14 – Третій крок додавання регулятора (автоматичний регулятор): вибір критеріїв

Заключним кроком додавання автоматичного регулятора (рис. 3.15) є введення коефіцієнту в межах від 0 до 2, де 0 буде означати невраховування альтернатив за даними критеріями, 1 – залишить значення без змін (але від такого регулятора немає сенсу), усі інші збільшать або зменшать відповідно значення альтернатив за певними критеріями (менше 1 – зменшить, більше 1 – збільшить відповідно).

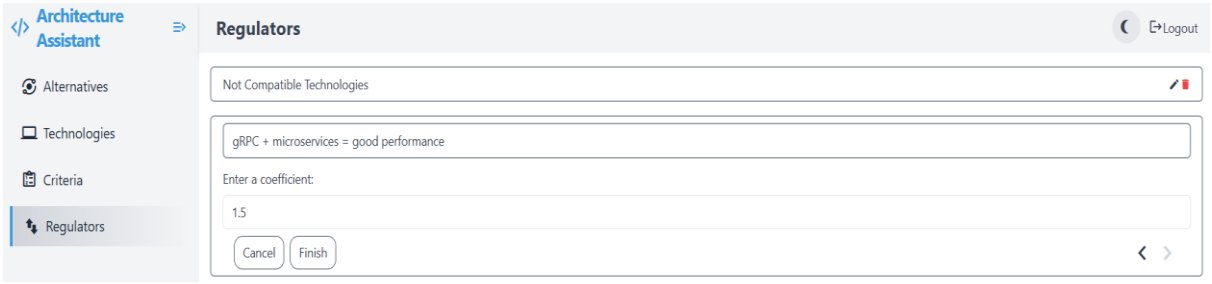

Рисунок 3.15 – Четвертий крок додавання регулятора (автоматичний регулятор): введення коефіцієнту

Але існують ситуації, коли необхідно відредагувати значення альтернатив за певними критеріями в залежності від вимог та наявних ресурсів користувача. Для цього і існує другий тип регуляторів «питання». При обранні його на першому кроці додавання регулятора, адміністраторові відобразиться другий крок (рис. 3.16), на якому застосунок попросить ввести текст питання.

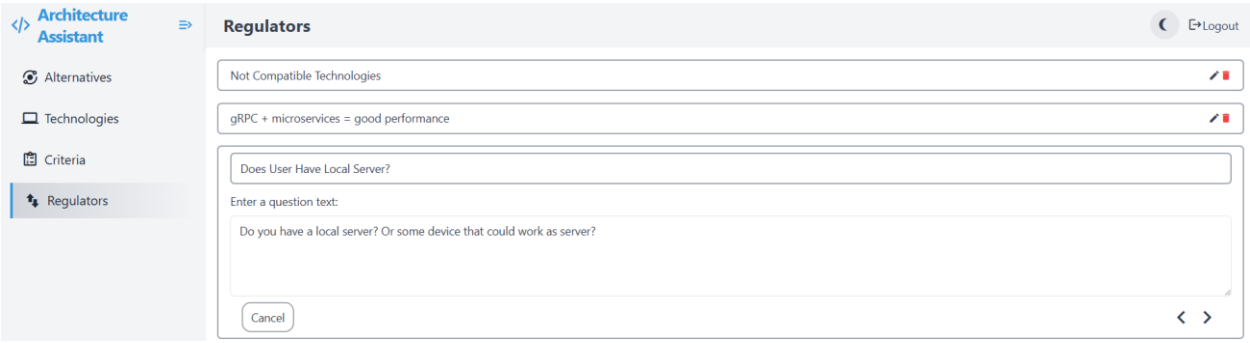

Рисунок 3.16 – Другий крок додавання регулятора (регулятор-питання): введення питання

Третій і четвертий кроки не відрізняються від того, що питалось для автоматичного регулятора: обрати альтернативи (рис. 3.17) і критерії (3.18), за якими буде проведене регулювання.

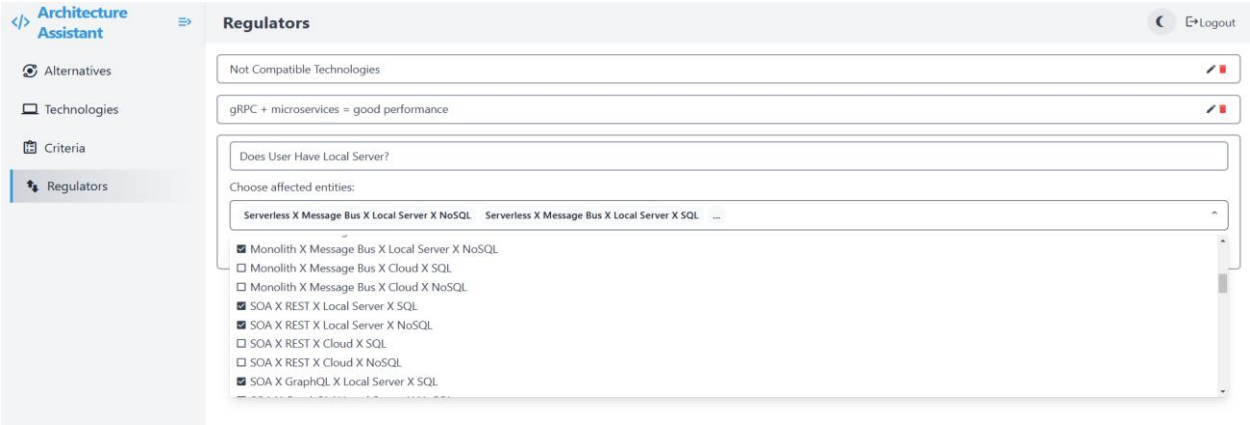

Рисунок 3.17 – Третій крок додавання регулятора (регулятор-питання): вибір альтернатив

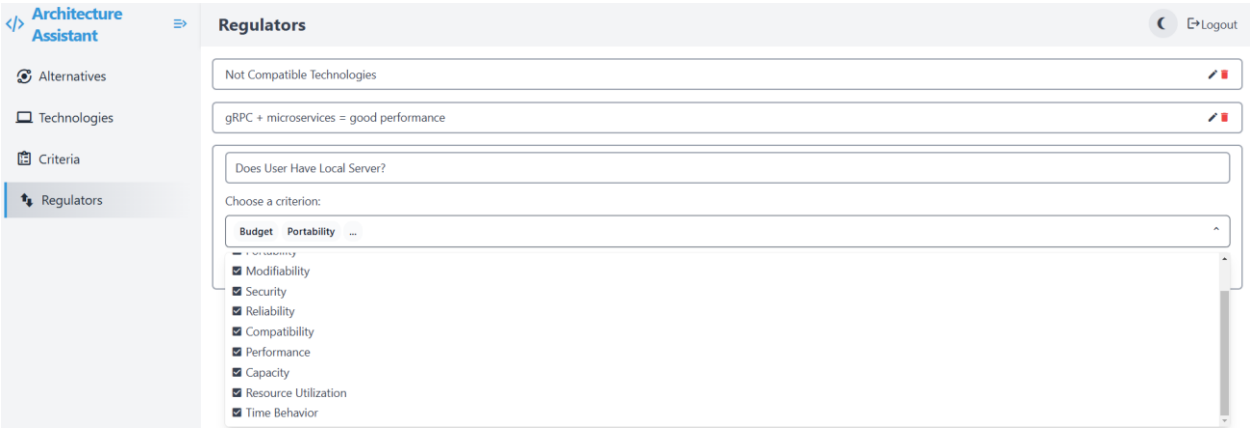

Рисунок 3.18 – Четвертий крок додавання регулятора (регулятор-питання): вибір альтернатив

В заключному п’ятому кроці (рис 3.19) додавання регулятора-питання адміністраторові необхідно обрати тип надання відповіді: оціночний або булевий.

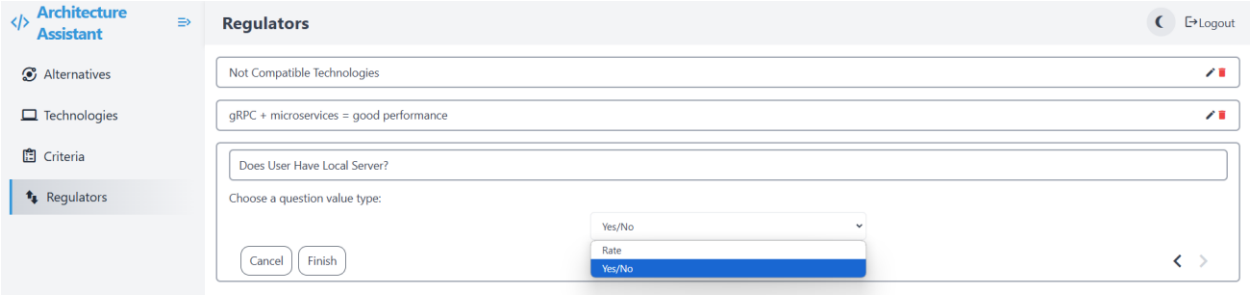

Рисунок 3.19 – П’ятий крок додавання регулятора (регулятор-питання): вибір критеріїв

3.2 Розробка користувацького інтерфейсу

Застосунок був розроблений для єдиної мети користувача: отримати рекомендації до того, яке архітектурне рішення прийняти. Саме через це користувач має мінімальні можливості: створити чат і відповідати в ньому на питання необхідні для визначення найкращої альтернативи.

3.2.1 Розробка меню керування чатами

На рисунку 3.20 показане головне меню користувача. На цьому меню відображаються усі чати, які були створенні користувачем. Кожен чат ідентифікується іменем для зручної навігації між ними. Існує можливість видалити чат натисканням кнопки з іконкою сміттєвого баку. Для кожного чату відображається скорочений текст останнього повідомлення і дата надсилання цього повідомлення.

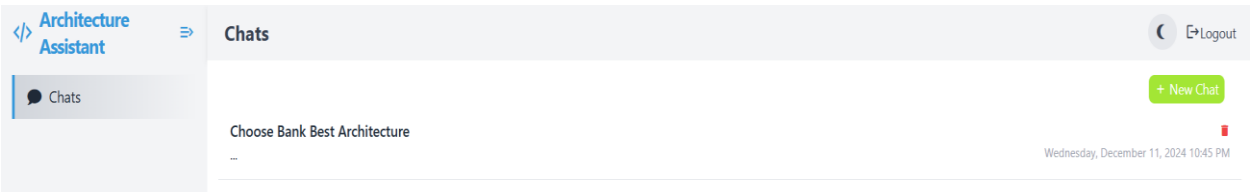

Рисунок 3.20 – Меню керування чатами

3.2.2 Розробка інтерфейсу створення чату

Чат представляє собою механізм збирання усієї необхідної інформації про вимоги користувача. При натисканні на кнопку «+ New Chat», тобто при створенні нового чату відкривається модальне вікно для заповнення необхідного контексту (рис. 3.21).

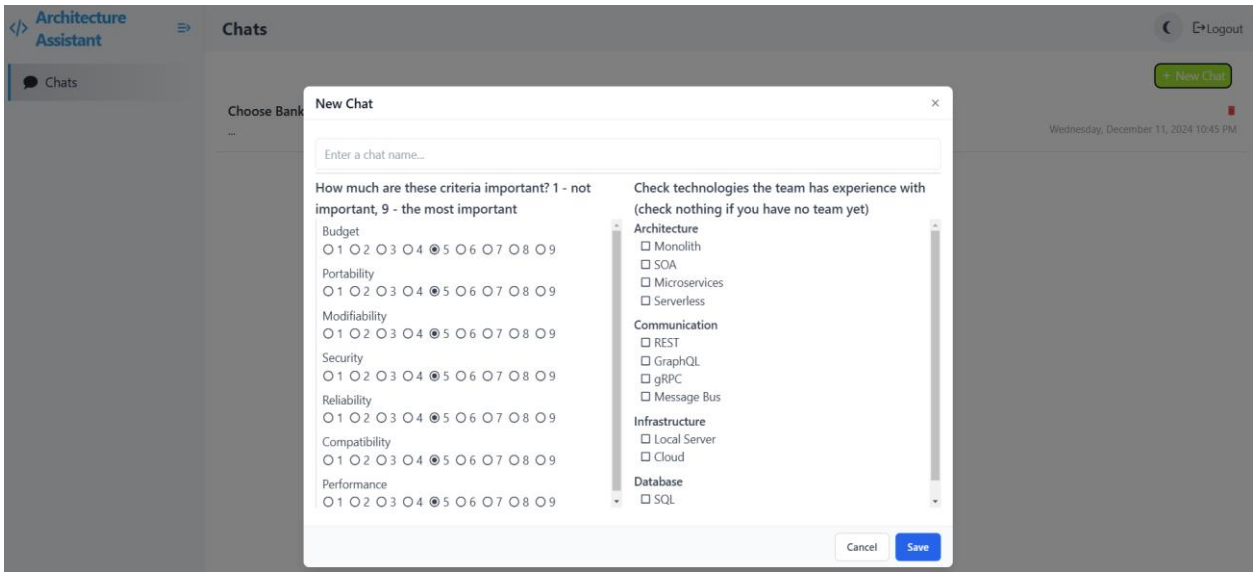

Рисунок 3.21 – Модальне вікно створення чату

В даному вікні на лівій панелі застосунок просить користувача оцінити критерії оцінювання. Дані значення визначають важливість критерію, деякі технічні рішення є кращими за одним критерієм і гіршими за іншим. На правій панелі від користувача вимагається зазначити із якими технологіями ознайомлена команда, що дасть можливість надати рекомендації відповідно до можливостей команди. Окрім цього над обома панелями існує поле для вводу назви чату, яке може бути змінене в будь-який момент часу.

Після створення чату користувачеві відкривається інтерфейс для подальшого ознайомлення з вимогами користувача (рис. 3.22). Одразу із створенням чату, зберігаються значення автоматичних регуляторів, які йдуть у зазначеному адміністратором порядку. Усі автоматичні регулятори відображаються у форматі «Applied: {Назва регулятора}». Біля кожного

повідомлення показана дата останнього його оновлення. Контекст чату (рис. 3.23), який користувач зазначив при його створенні, є можливість переглянути. Відредагувати цей контекст – неможливо, він необхідний для перегляду для розуміння того, які вимоги користувач ставив для визначення найкращого архітектурного рішення.

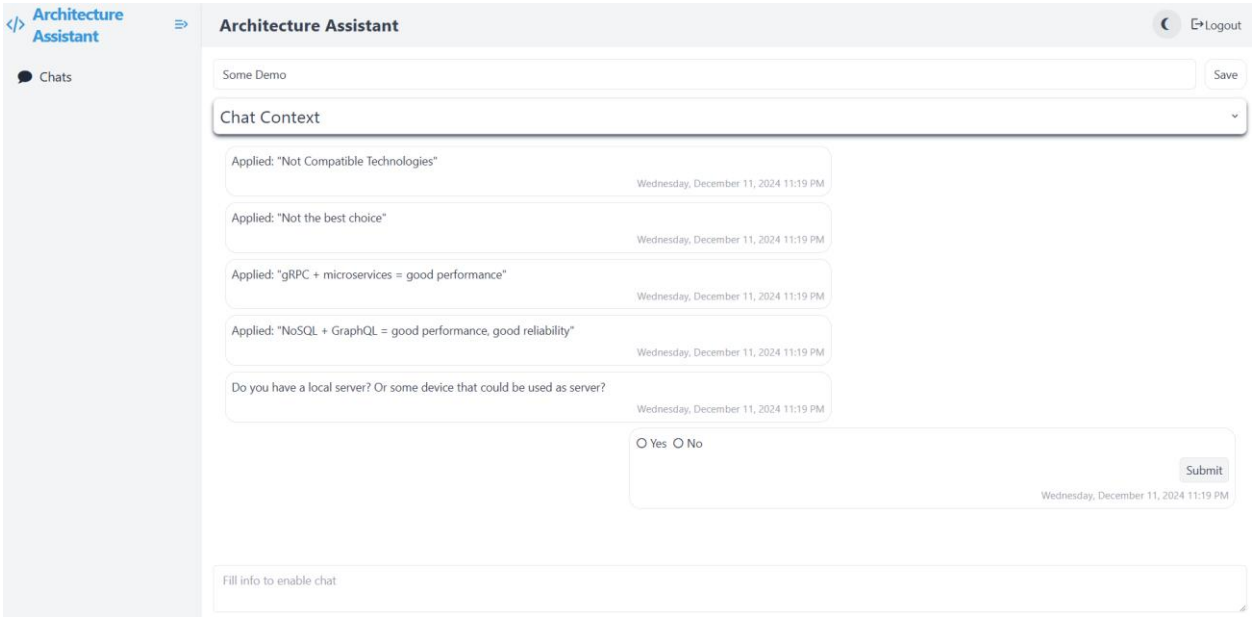

Рисунок 3.22 – Інтерфейс чату

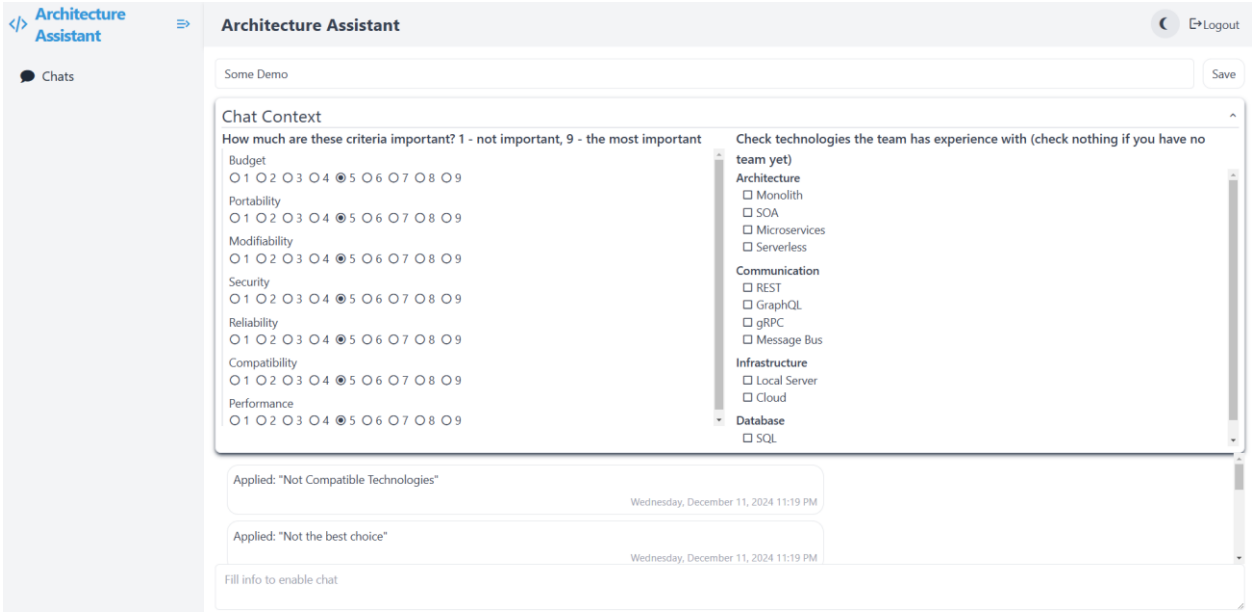

Рисунок 3.23 – Інтерфейс чату із розкритими контекстом

Повідомлення користувача, які необхідні для збереження значень регуляторів-запитань, мають кнопку збереження коефіцієнту. Після збереження коефіцієнту дані регулятора відображаються, але забороняється їх змінювати.

Після надання відповідей на всі базові запитання, які були передбаченні регуляторами створеними адміністратором, користувач отримує повідомлення про те, яке найкраще архітектурне рішення вибрав базовий алгоритм застосунку (рис. 3.24).

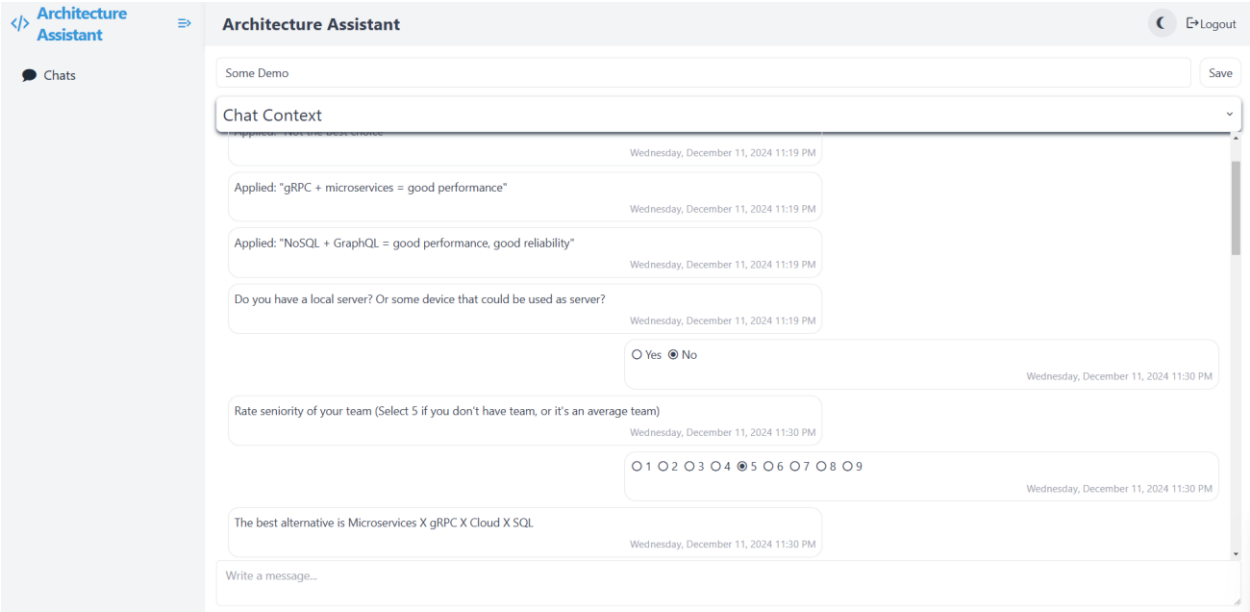

Рисунок 3.24 – Завершений чат

Окрім цього після завершення базового алгоритму визначення найкращого архітектурного рішення програмний застосунок виконає запит до Chat GPT, щоб отримати рекомендації стосовно імплементації (рис. 3.25).

Програмний застосунок надає можливість продовжити обговорення виданих результатів LLM або алгоритму (рис. 3.26), а також ставити уточнюючи питання. Усі ці питання будуть спрямовані до Chat GPT із зазначенням історії переписки і контексту, що допоможе отримати більш якісні рекомендації в майбутньому.

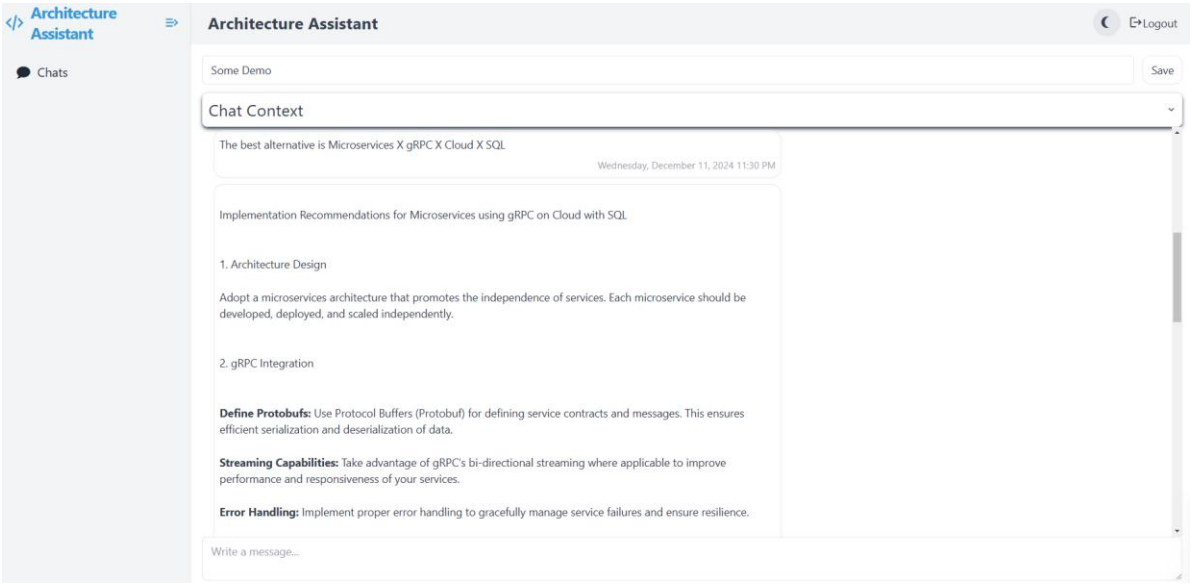

Рисунок 3.25 – Рекомендації щодо імплементації надані LLM

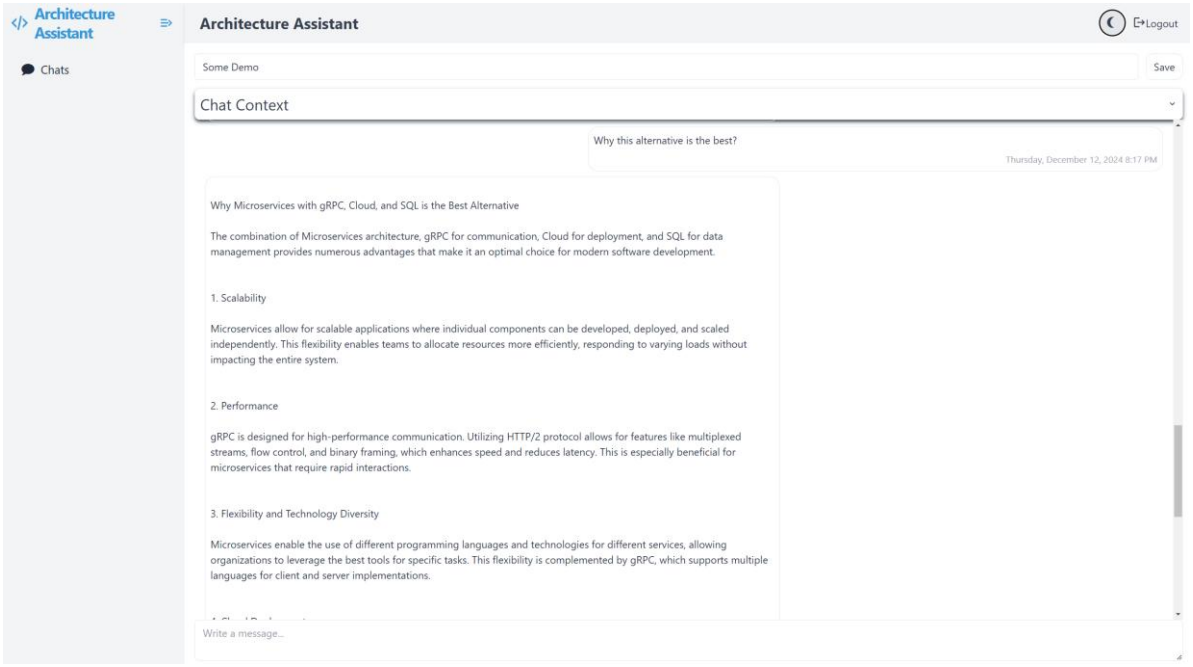

Рисунок 3.26 – Продовжений діалог із Chat GPT в межах контексту чату

3.3 Тестування алгоритму визначення найкращого архітектурного рішення

Алгоритм буде протестований на завданні аналізу вимог до застосунку, призначеного для оцінки знань студентів з мови SQL. Основним функціоналом цього застосунку буде проведення тестування студентів з

використанням SQL-запитів, а також забезпечення автоматизованої перевірки результатів. Крім цього, застосунок матиме додаткову функцію розпізнавання облич студентів для підтвердження їхньої особи, що підвищить безпеку та уникне можливості шахрайства під час складання тестів.

3.3.1 Введення тестових даних

Як було зазначено вище, початковим кроком для користувача (рис. 3.27) – є оцінка важливості критеріїв відносно один одного. І визначення технологічних рішень із якими ознайомлена команда.

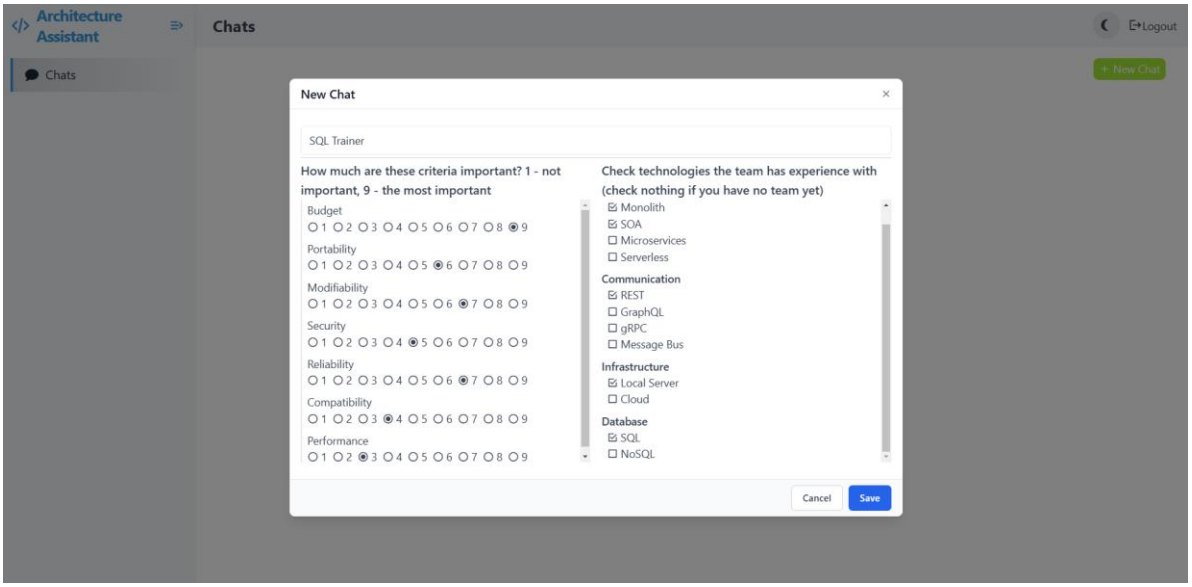

Рисунок 3.27 – Введені початкові тестові дані

Для даного програмного застосунку бюджет є найкритичнішою складовою, через те, що розроблявся одним студентом із обмеженими фінансовими ресурсами. Через обмеженість в фінансових ресурсах програмний застосунок має бути легким в перенесені, тому що при його розвитку може стати необхідним масштабувати застосунок. Даний проєкт має бути легким в модифікації для додавання різних баз даних, розширення функціоналу для оцінки знань студентів також із різних мов програмування.

Безпека є критичною складовою, але не на початкових стадіях розвитку ПЗ. Надійність системи сформує позивну репутацію навколо застосунку, що є дуже важливим для стартапу. Сумісність може бути тісно пов’язано із переносністю, але це не є критичним через те, що навіть при перенесені застосунку в іншу інфраструктуру, можна забезпечити схоже середовище. Продуктивність системи є важливою ознакою ПЗ, але не є одним із найважливіших саме для цього застосунку.

Студент, який розробляє систему для оцінки знань студента, мав досвід роботи із монолітними архітектурними рішеннями і сервісно-орієнтованими системами. Студент працював тільки із застосунками, які мали спілкування за допомогою REST. Усі системи були на локальних серверах або на віртуальних машинах. Окрім цього студент мав досвід роботи тільки із реляційними базами даних.

3.3.2 Регулювання ваг відповідями на запитання-регулятори

Наступним кроком користувач має відповісти на усі базові запитання алгоритму (рис. 3.28).

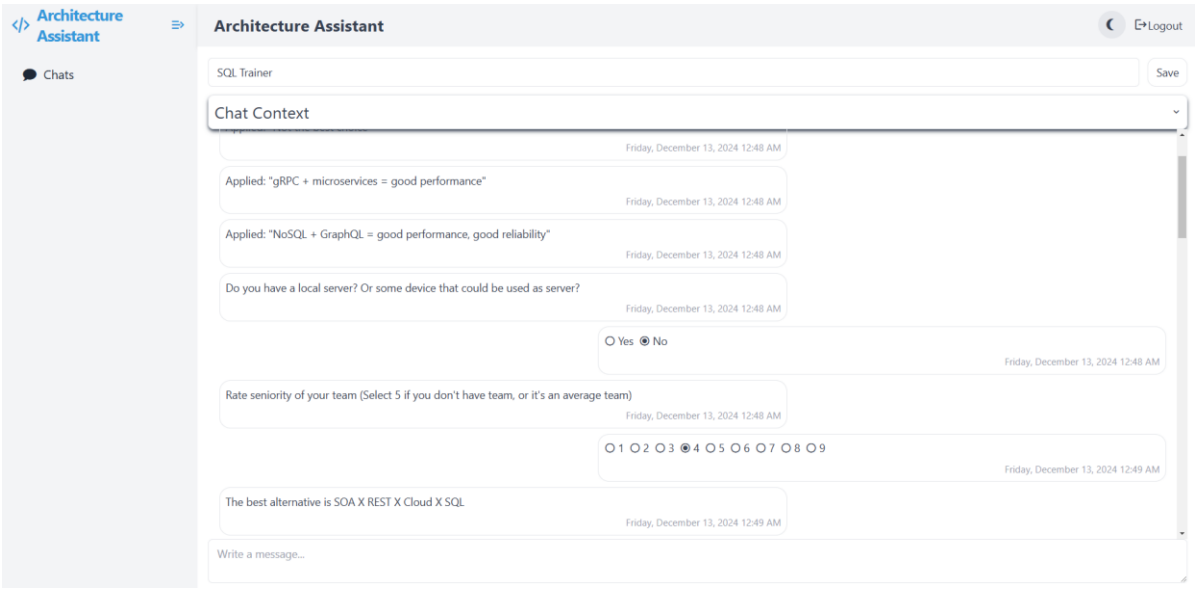

Рисунок 3.28 – Відповіді на питання-регулятори

Студент не має жодного пристрою, який міг би працювати як сервер. Також студент має недостатній рівень експертизи в питаннях архітектури та розробки. Для демонстративних цілей алгоритм має лише декілька запитань-регуляторів. Базовий алгоритм визначив найкращою альтернативою для вхідних даних – поєднання технологій SOA та REST, які мають працювати на хмарному провайдері і зберігати дані в реляційній базі даних.

Після визначення найкращої альтернативи базових алгоритмом застосунок звертається до Chat GPT, щоб надати рекомендації щодо реалізації застосунку із застосуванням даних технологічних рішень. (рис. 3.29).

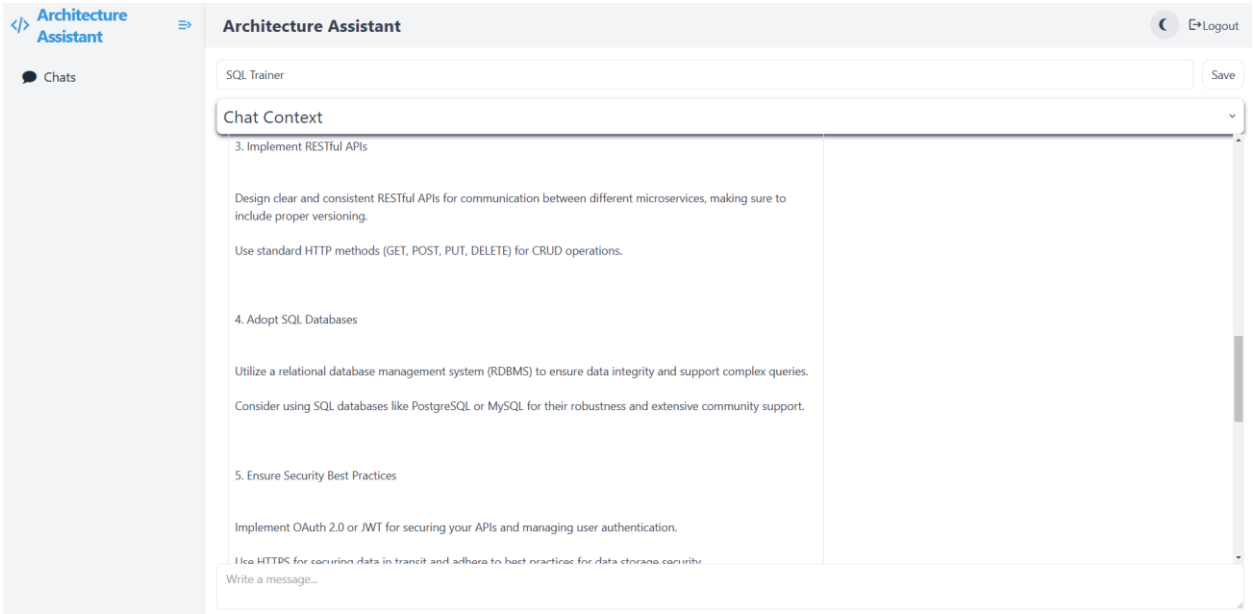

Рисунок 3.29 – Надані Chat GPT рекомендації

3.3.3 Аналізування результатів базового алгоритму

Запропонована альтернатива є обґрунтованою через те, що сам виконавець є знайомим із даними технологіями.

Вибір сервісної архітектури замість моноліту є суперечливим, тому що розробка моноліту може зайняти менше часу, і через це вимагати менше ресурсів. Гарною практикою є написання спочатку моноліту розподіленого на

декілька модулів, який із часом і за потреби можна буде перевести на сервісну архітектуру. Важливим в такому підході є розподіл моноліту на модулі, які в майбутньому і стануть окремими сервісами, залежності між такими модулями мають бути мінімальними для забезпечення більш легкої міграції. Можна сказати, що алгоритм надав правильну рекомендацію стосовно архітектурного рішення, але ця рекомендація не є найкращою.

Однією із проблем даної версії алгоритму і застосунку в цілому – це не врахування можливості поєднання різних варіантів різних технологічних рішень. Наприклад, в даній ситуації алгоритм рекомендує використовувати REST для комунікації між сервісами, ця рекомендація є хорошою, враховуючи, що виконавець знайомий тільки із REST. Але для даного застосунку необхідно поєднати декілька різних способів комунікації між сервісами і між сервером і клієнтом. Основним методом комунікації має бути і є REST, що може бути за потреби замінений на gRPC чи GraphQL, якщо команда краще знайома із цими технологіями. Але при збереженні результатів тестування дуже важливим є надійність системи, тобто вебсервіс має обов’язково обробити запит. Для такої цілі використання черг стає необхідністю, а не опцією для вибору. Окрім цього під час проходження тесту система має завжди ідентифікувати студента, для цього необхідно кожну мить перевіряти чи студент є в камері та чи інша людина не проходить тест за студента. Ця мета може бути досягнена або встановленням системи ідентифікації обличчя на пристрій студента, або налаштування комунікації в режимі реального часу, наприклад, використовуючи технологію вебсокетів. Алгоритм надав правильну рекомендацію, але не повну через його обмеження.

Далі застосунок відрекомендував розмістити застосунок у хмарного провайдера. Дане рішення може бути суперечливим, тому що студент визначив бюджет як найкритичнішим критерієм, а хмарний провайдер може бути не дешевим. Окрім цього студент ніколи не мав досвіду роботи із хмарними рішеннями. Але з іншої сторони студент зазначив, що в нього немає

жодного пристрою, який міг би виступити у ролі серверу, що має автоматично казати про те, що локальний сервер не може бути використаний.

Останньою частиною альтернативи є використання реляційних баз даних. Таке рішення є правильним і з точки зору того, що студент не ознайомлений із NoSQL базами даних і загалом через те, що для даного застосунку більше підходять структуровані дані.

Враховуючи усе вищесказане, можна зробити висновок, що рекомендація, надана програмним застосунком, є правильною в загальних рисах, оскільки вона базується на чітких даних і враховує ключові вимоги проєкту. Проте вона не є ідеальною через низку факторів, які залишають простір для вдосконалення. Рекомендація демонструє базовий рівень адаптації до поставлених завдань, але не враховує багатогранності реальних умов і потреб, що є критично важливим для розробки складних програмних систем.

Зокрема, програма надала рішення, які відповідають поточним знанням та досвіду виконавця, але не запропонувала альтернатив чи комбінованих підходів, які могли б значно покращити ефективність або гнучкість системи. Також рекомендація не враховує довгострокову перспективу, де початково обране рішення може стати обмеженням у майбутньому через зміну вимог чи збільшення навантаження на систему.

Існують аспекти, які потребують вдосконалення. Наприклад, застосунок міг би враховувати можливість комбінування різних технологій для вирішення складних завдань, таких як забезпечення високої надійності чи адаптація до змін у реальному часі. Крім того, рекомендація могла б бути доповнена деталізованими підходами до оптимізації використаних ресурсів, врахуванням обмежень бюджету чи знань виконавця.

Таким чином, хоча рекомендація відповідає базовим вимогам, її потенціал не реалізовано повністю. Це підкреслює необхідність доопрацювання алгоритму, включаючи розширення його функціоналу для більш глибокого аналізу та врахування складних сценаріїв, що зробить результати більш релевантними та практичними.

ВИСНОВКИ

У рамках кваліфікаційної роботи було досліджено питання обрання архітектури вебзастосунку на основі аналізу вимог прикладної задачі. Робота надає глибокий аналіз сучасного стану сектору інформаційних технологій, наявних архітектурних рішень, сучасних технологій і можливостей їх поєднання. У результаті дослідження були виокремлені критерії оцінювання якості програмного забезпечення, а також сформовані групи технологічних рішень і їх варіантів. Розроблено алгоритм для визначення найкращого архітектурного рішення на основі вимог зазначених користувачем. Розроблено і протестоване технічне рішення для автоматичного виконання алгоритму.

Експериментальні дослідження показали, що розроблений алгоритм справляється із своєю задачею. Він надає обґрунтовані рекомендації, за допомогою яких можливо розробити застосунок високої якості.

Проте результати даного алгоритму не є ідеальними, тому що не враховують багато різних аспектів, наприклад, можливість поступового масштабування застосунку та поєднання технологічних рішень і використання конкретних варіантів у вузькоспеціалізованих випадках.

Наукова новизна роботи полягає у розробці алгоритму щодо рекомендації архітектурного рішення для вебзастосунку з урахуванням вимог прикладної задачі та використанням LLM для надання рекомендацій щодо застосування запропонованих технологічних рішень. Розроблений на основі алгоритму сервіс може покращити загальну ситуацію із якістю коду програмного забезпечення.

Таким чином, проведене дослідження показало проблему недостатнього приділення уваги вибору архітектурних рішень, та наслідки такої халатності. Було запропоновано алгоритм, який допомагає обрати архітектурне рішення і надає рекомендації щодо його імплементації, навіть користувачам, які не мають великого досвіду в розробці застосунків, або навіть не є розробниками або архітекторами програмного забезпечення. Алгоритм був перевірений на

тестових вимогах до вебзастосунку і показав свій потенціал в цьому напрямку. Покращення даного алгоритму і розробка схожих рішень допоможе будь-кому спроєктувати вебзастосунок відносно вимог користувача, що в майбутньому може призвести до покращення середньої якості коду тим, що розробник стартапу зможе швидко розробити дизайн застосунку.

Результати дослідження апробовано у вигляді тез доповідей під час X Міжнародної науково-технічної конференції «Computerintegrated technologies of automation of technological processes» [27] та XV Міжнародної науково-технічної конференції «Free and open source software» [6].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Yakovleva, O., & Nikolaieva, K. (2020). Research Of Descriptor Based Image Normalization And Comparative Analysis Of SURF, SIFT, BRISK, ORB, KAZE, AKAZE Descriptors. *Advanced Information Systems, 4*(4), 89-101. doi:10.20998/2522-9052.2020.4.13.

2 Gorokhovatskyi V., Tvoroshenko I., and Yakovleva O. (2024) Transforming image descriptions as a set of descriptors to construct classification features, Indonesian Journal of Electrical Engineering and Computer Science, vol. 33, no. 1, pp. 113-125. DOI: 10.11591/ijeecs.v33.i1.pp113-125.

3 Gorokhovatskyi , O., & Yakovleva , O. (2024). Medoids as a Packing Of ORB Image Descriptors. Advanced Information Systems, 8(2), pp. 5–11.

4 Gorokhovatskyi V., Tvoroshenko I., Yakovleva O., Hudáková M., and Gorokhovatskyi O. (2024) Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set, IEEE Access, vol. 12, pp. 73376-73385. DOI: 10.1109/ACCESS.2024.3404371.

5 Ковтуненко, А. Р., Яковлева, О. В., Любченко, В. А., & Янголенко, О. В. (2020). Дослідження сумісного використання математичної морфології та згорткових нейронних мереж для вирішення задачі розпізнавання цінників. *Вісник Національного технічного університету ХПІ (3). 24-31.*

6 Naumenko V., Shelest V., Yakovleva O. (2024). Combination of .Net technology and Angular framework to develop application for testing SQL language knowledge. Proceedings of the XVth International Scientific and Practical Conference «Free And Open Source Software», Ukraine, Kharkiv, February 13-14, 2024. pp.63-66.

7 Yakovleva, O., Kovtunenko, A., Liubchenko, V., Honcharenko, V., & Kobylin, O. (2023). Face Detection for Video Surveillance-based Security System. *CEUR Workshop Proceedings* Vol. 3403. pp. 69-86. ISSN 1613-0073.

8 Yakovleva O., Nebeský L, Liakhov P. (2023). Research methods of texture image analysis to solve the texture search problem. Proceedings of the IV

International Scientific and Practical Conference «The world of modern technologies and inventions». Vienna, Austria. 2023. pp. 252-261.

9 Daga, A., de Cesare, S., & Lycett, M. (2006). Separation of concerns: Techniques, issues and implications. *Journal of Intelligent Systems*, 15(1-4), 153–176. https://doi.org/10.1515/JISYS.2006.15.1-4.153.

10 Nierstrasz, O., & Achermann, F. (2000). Separation of concerns through unification of concepts. *Proceedings of the ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*.

11 Aksit, M., Tekinerdogan, B., & Bergmans, L. (2001). The six concerns for separation of concerns. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*.

12 Wagner, S., & Deissenboeck, F. (2017). Abstractness, specificity, and complexity in software design. *IEEE Software*, 34(5), 54–60.

13 Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR 95-98.

14 Android UI Architecture Migration to MVVM. URL: https://www.dashlane.com/blog/android-ui-architecture-mvvm (дата звернення: 14.12.2024).

15 Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

16 MVC. URL: https://developer.mozilla.org/en-US/docs/Glossary/MVC (дата звернення 14.12.2024).

17 REST API Basics - 4 Things you Need to Know. URL: https://mannhowie.com/rest-api (дата звернення 14.12.2024).

18 Box, D. (2000). Simple object access protocol (SOAP). URL: *http://www.w3.org/TR/SOAP/* (дата звернення 03.10.2024).

19 SOAP-based Web services architecture. URL: https://www.researchgate.net/figure/SOAP-based-Web-services-architecture_fig3_220144364 (дата звернення 14.12.2024).

20 gRPC – система віддаленого виклику процедур з відкритим кодом. URL: https://grpc.io/docs/what-is-grpc/introduction/ (дата звернення 03.10.2024).

21 GraphQL – мова запитів і маніпуляції даними для API з відкритим кодом. URL: https://graphql.org/learn/ (дата звернення: 03.10.2024).

22 GraphQL Protocol. URL: https://www.wallarm.com/what/what-is-graphql-definition-with-example (дата звернення 14.12.2024).

23 gRPC vs REST - A Brief Comparison. URL: https://refine.dev/blog/grpc-vs-rest/ (дата звернення 14.12.2024).

24 International Electrotechnical Commission. (2001). *Software Engineering-Product Quality* (Vol. 9126). ISO/IEC.

25 International Organization for Standardization. (2011). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models (Vol. 25010). ISO/IEC.

26 Froala – вебредактор тексту. URL: https://froala.com/ (дата звернення 05.10.2024).

27 Kubernetes – відкрита система автоматичного розгортання, масштабування та управління застосунками у контейнерах. URL: https://kubernetes.io/ (дата звернення 20.10.2024).

28 Blazor – це безкоштовна вебплатформа з відкритим вихідним кодом, яка дозволяє розробникам створювати вебінтерфейс користувача на основі компонентів за допомогою C# та HTML. URL: https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor (дата звернення 22.10.2024).

29 Protocol Buffers – формат серіалізації даних, запропонований корпорацією Google, як альтернатива XML. URL: https://protobuf.dev/overview/ (дата звернення 23.10.2024).

30 RabbitMQ – платформа, що реалізує систему обміну повідомленнями між компонентами програмної системи на основі стандарту AMQP. URL: https://www.rabbitmq.com/docs (дата звернення 24.10.2024).

31 Apache Kafka – це розподілене сховище подій і платформа для їх багатопотокового оброблення. URL: https://kafka.apache.org/documentation/ (дата звернення 24.10.2024).

32 Azure Bus Service – це повністю керований корпоративний брокер повідомлень із чергами повідомлень і темами публікацій та підписки. URL: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview (дата звернення 24.10.2024).

33 Docker – платформа розроблена для допомоги розробникам будувати, поширювати і запускати застосунки-контейнери. URL: https://www.docker.com/get-started/ (дата звернення 10.12.2024).

34 Shelest V., Yakovleva O. (2024) Research on selecting web application architecture based on the analysis of applied requirements. Proceedings of the X International Scientific and Practical Conference «Computerintegrated technologies of automation of technological processes». Hamburg, Germany, pp. 46-54.