НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет/(ННІ) _____

ПОГОДЖЕНО
Декан факультету (Директор ННІ)
_____
(назва факультету (ННІ))

_____ _____
(підпис) (ім'я ПРІЗВИЩЕ)

"___" _____ 20_ р.

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
Завідувач кафедри
_____комп'ютерних наук_____
(назва кафедри)

_____ ___Голуб Б.Л ___
(підпис) (ім'я ПРІЗВИЩЕ)

"___" _____ 20_ р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему _____Програмне забезпечення системи підтримки____
прийняття рішень для агенції з надання послуг в сфері працевлаштування __

Спеціальність _____121 "Інженерія програмного забезпечення"_____
(код і найменування)

Освітня програма _____ Програмне забезпечення інформаційних систем _____
(назва)

Орієнтація освітньої програми_____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми
_____ к. ф.-м. н., доцент _____ _____ _____Кіріченко В.В.____
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Керівник магістерської кваліфікаційної роботи
_____ д. т. н., професор _____ _____ _____ Бушма О.В._____
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Виконав _____ _____ Близнюк Віталій Олександрович _____
(підпис) ((ім'я ПРІЗВИЩЕ здобувача)

КИЇВ – 20_25_

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) _____

ЗАТВЕРДЖУЮ
Завідувач кафедри
____ кандидат технічних наук, доцент____
_____ _____ _Голуб Б.Л._
(науковий ступінь, вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)
"____" _____20_____року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

_____ Близнюку Віталію Олександровичу _____
(прізвище, ім'я, по батькові)

Спеціальність
_____121 "Інженерія програмного забезпечення _____
(код і найменування)

Освітня програма
_____ Програмне забезпечення інформаційних систем _____
(назва)

Орієнтація освітньої програми
_____ освітньо-професійна _____
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи ____Програмне забезпечення системи підтримки прийняття рішень для агенції з надання послуг в сфері працевлаштування_____

затверджена наказом від "____"_____20___р. №_____

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

Вихідні дані до магістерської кваліфікаційної роботи: Текст патенту США US20130166459A1 ("Invention valuation and scoring system"), науково-технічні публікації щодо методів MCDM та технологій OLAP/Data Mining, статистичні дані ринку праці України (аналітика Work.ua, DOU.ua), документація фреймворку FastAPI.

Перелік питань, що підлягають дослідженню:

1. Дослідити аналіз стратегій щодо зважения критеріїв оцінки кандидатів для різних рівнів вакансій._____
2. Обґрунтувати вибір мови програмування розробки системи (Python) та фреймворку (FastAPI) для реалізації сервісно-орієнтованої архітектури._____
3. Розробити архітектурний підхід до рішення проблеми системи підтримки прийняття рішень на основі дворівневої структури даних (OLTP + OLAP)._____

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання "_____"_____20___ р.

Керівник магістерської кваліфікаційної роботи _____ ___Бушма О.В_
(підпис) (ім'я ПРІЗВИЩЕ)

Завдання прийняв до виконання _____ _____Близнюк В.О_____
(ім'я ПРІЗВИЩЕ)

ЗМІСТ

ВСТУП

Актуальність

Сучасний ринок праці перебуває в стані цифрової трансформації. Компанії стикаються з дефіцитом кваліфікованих кадрів, а агентства з працевлаштування — з безпрецедентними обсягами інформації. Щоденна обробка тисяч резюме вручну створює «вузьке місце» в процесах найму: це ресурсомістко, повільно та суб'єктивно. Існуючі системи автоматизації (ATS) часто обмежуються функціями зберігання даних, не надаючи інструментів для інтелектуального аналізу.

Тому розробка спеціалізованого програмного забезпечення системи підтримки прийняття рішень (СППР) є надзвичайно актуальною. Впровадження СППР дозволить автоматизувати скоринг кандидатів, підвищити об'єктивність відбору та скоротити час на закриття вакансій, забезпечуючи агентству конкурентну перевагу.

Предмет та Об'єкт Дослідження

Об'єктом дослідження є процес прийняття рішень в агентстві з надання послуг у сфері працевлаштування при підборі кандидатів.

Предметом дослідження є моделі, методи, алгоритми та інформаційні технології для побудови СППР, призначеної для автоматизації задач скорингу та ранжування кандидатів.

Мета Дослідження

Метою роботи є підвищення ефективності підбору персоналу шляхом розробки СППР. Для цього необхідно розробити архітектуру на базі гнучкої моделі зваженого оцінювання та спроєктувати аналітичну підсистему з використанням технології OLAP для аналізу ефективності стратегій підбору.

Зміст Поставлених Завдань

- Для досягнення мети визначено такі завдання:
- Провести системний аналіз предметної області та існуючих рішень.
- Сформулювати функціональні вимоги до СППР.
- Спроєктувати архітектуру системи (OLTP-база + API на FastAPI).
- Розробити адаптивну модель зваженого оцінювання з використанням патерну «Стратегія».

- Спроєктувати Сховище Даних (DWH) та OLAP-куб (Схема «Зірка») для аналізу ефективності.
- Розробити програмний прототип ключових компонентів.
- Сформулювати рекомендації щодо використання системи для вдосконалення алгоритмів.

Методи Дослідження

У роботі використано сукупність сучасних методів та технологій:

- Для аналізу та проєктування: методи системного аналізу та об'єктно-орієнтованого проєктування (UML), моделювання баз даних (ER-діаграми).
- Для розробки прототипу: мова Python 3, фреймворк FastAPI, ORM SQLAlchemy та Pydantic. Алгоритмічне ядро реалізовано через патерн «Стратегія».
- Для аналітичної підсистеми: методологія моделювання Сховищ Даних (DWH), Схема «Зірка», технологія OLAP та засоби бізнес-аналітики (Power BI).

Наукова Новизна

Отримані результати мають таку наукову новизну:

- Запропоновано удосконалення алгоритму підбору шляхом використання патерну «Стратегія», що дозволяє динамічно змінювати моделі зважування (наприклад, пріоритет освіти для Junior або soft-skills для Senior), розвиваючи методи патенту US20130166459A1.
- Вдосконалено архітектуру СППР, яка інтегрує операційну базу (OLTP) та багатовимірний OLAP-куб (DWH).
- Створено замкнений цикл зворотного зв'язку (feedback loop), що дозволяє об'єктивно вимірювати ефективність стратегій на основі реальних даних (FactHiringPerformance), науково обґрунтовуючи вибір моделі оцінювання.

1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Загальна характеристика проблеми та концепція запропонованої системи

Сучасний ринок праці та сфера управління людськими ресурсами (HR) переживають фундаментальну трансформацію, зумовлену факторами глобалізації, цифрової революції та зміною економічних парадигм. Ми живемо в епоху, яку часто описують акронімом VUCA (Volatility, Uncertainty, Complexity, Ambiguity – Мінливість, Невизначеність, Складність, Неоднозначність). Ця характеристика повною мірою стосується ринку праці: життєві цикли технологій та бізнес-моделей скорочуються, виникають нові професії та зникають старі, а поширення віддалених та гібридних форматів роботи стирає географічні кордони для «талантів».

У цьому висококонкурентному середовищі, яке часто називають «війною за таланти», особливо у сфері інформаційних технологій (IT), роль агентств з надання послуг у сфері працевлаштування (рекрутингових агентств) стає критично важливою. Вони виступають як ключові посередники та експерти, що з'єднують компанії-роботодавців, які мають гостру потребу у кваліфікованих кадрах, та кандидатів, що шукають нові можливості.

Однак, традиційні методи роботи агентств, що базувалися на ручній обробці резюме та особистій інтуїції рекрутерів, демонструють свою повну неефективність перед обличчям нових викликів. Ключова проблема, що стоїть перед галуззю — це інформаційне перевантаження та нездатність існуючих процесів масштабуватися для обробки експоненційного зростання обсягів даних. Одна публікація вакансії на популярну посаду (наприклад, "IT Recruiter" або "Junior Python Developer") на провідних порталах може генерувати сотні, а іноді й тисячі відгуків (резюме).

Ці дані, до того ж, надходять у різноманітних форматах (структуровані з job-сайтів, неструктуровані файли PDF та DOCX, посилання на профілі LinkedIn) і містять величезну кількість «шуму». Рекрутер стикається з нетривіальною задачею: як у цьому потоці інформації не просто знайти, а об’єктивно оцінити та порівняти десятки кваліфікованих кандидатів, щоб обрати 3-5 найкращих для презентації клієнту?

Декомпозиція та аналіз проблеми прийняття рішень

Проблема ручного відбору є багатогранною і має три основні виміри, які безпосередньо впливають на якість та прибутковість роботи агентства:

Проблема Масштабованості (Scalability): Людські можливості з обробки інформації є обмеженими. Досвідчений рекрутер фізично не може якісно та вдумливо проаналізувати понад певну кількість резюме на день (наприклад, 50-100). Це створює «вузьке місце» (bottleneck) у бізнес-процесі. Як наслідок, цінні кандидати, що опинилися «на дні» списку, можуть залишитися непоміченими, або, що більш імовірно, їх швидше помітять та наймуть конкуруючі агентства.

Проблема Об'єктивності (Objectivity): Навіть якщо рекрутер має достатньо часу, його рішення є суб'єктивним. Процес ручного відбору знаходиться під сильним впливом когнітивних упереджень (cognitive biases), які детально описані в роботах Деніела Канемана. Для предметної області рекрутингу найбільш критичними є:

- Упередження за підтвердженням (Confirmation Bias): Рекрутер формує перше миттєве враження про кандидата (напр., "престижний ВНЗ" або, навпаки, "погана назва посади") і решту часу підсвідомо шукає в резюме лише ту інформацію, що підтверджує його початкову думку, ігноруючи факти, що їй суперечать.
- Ефект Ореолу (Halo Effect): Одна яскрава позитивна риса (напр., "досвід роботи у Google") змушує рекрутера автоматично переоцінювати всі інші, менш видатні, якості кандидата.
- Упередження "Як я" (Similarity/Affinity Bias): Рекрутери (часто несвідомо) надають перевагу кандидатам, які схожі на них самих — закінчили той самий ВНЗ, мають схожі хобі, або просто схожий стиль письма у резюме.
- Упередження за недавністю (Recency Bias): Рекрутер краще пам’ятає та вище оцінює останніх 5-10 кандидатів, яких він переглянув, ніж тих, кого він бачив вчора.

Ці упередження призводять до того, що відбір стає лотереєю, а не аналітичним процесом. Система, позбавлена людських упереджень, яка оцінює кожного

кандидата за однаковим, прозорим набором критеріїв, є необхідністю для забезпечення об'єктивності.

Проблема Ефективності (Efficiency): Ручна праця є дорогою. Витрачання годин висококваліфікованого рекрутера на механічну фільтрацію нерелевантних резюме є прямою економічною втратою для агентства. Крім того, існує вартість "поганого найму" (cost of a bad hire) — коли агентство рекомендує невідповідного кандидата, який звільняється через 2-3 місяці. Це завдає репутаційної шкоди агентству та фінансових збитків клієнту. Прискорення та підвищення точності відбору безпосередньо впливає на прибутковість бізнесу.

Концепція Системи Підтримки Прийняття Рішень (СППР) як вирішення проблеми

Вирішення вищеописаних проблем лежить у площині автоматизації та інтелектуалізації процесу відбору. Пропонується концепція розробки програмного забезпечення Системи Підтримки Прийняття Рішень (СППР).

Важливо чітко розмежувати СППР та простіші системи, такі як ATS (Applicant Tracking Systems). Більшість існуючих на ринку ATS є, по суті, системами OLTP (Online Transaction Processing). Їхнє головне завдання — надійно зберігати транзакційні дані (резюме, історію листування, статуси). Вони є «цифровими картотеками».

Наша концепція СППР пропонує інтелектуальну надбудову над цими даними. Якщо ATS відповідає на питання «Що у нас є?», то СППР відповідає на питання «Що з цим робити?» та «Що є найкращим?».

За класичною моделлю прийняття рішень Герберта Саймона, процес складається з трьох фаз: «Розвідка» (Intelligence), «Проєктування» (Design) та «Вибір» (Choice).

- «Розвідка» — це збір та первинна обробка резюме (парсинг, збереження в OLTP-базу).
- «Проєктування» — це генерація та порівняння альтернатив (формування списку кандидатів, розрахунок їхніх балів).
- «Вибір» — це фінальне рішення людини (агента) про те, кого представити клієнту.

Концепція СППР полягає у максимальній автоматизації та підтримці саме другої, найскладнішої фази — «Проєктування». Система не замінює рекрутера, а виступає в ролі потужного асистента, який:

1. Усуває упередженість: Застосовує єдиний, об'єктивний алгоритм до всіх кандидатів.
2. Вирішує проблему масштабу: Миттєво обробляє тисячі резюме, що неможливо для людини.
3. Підвищує ефективність: Надає рекрутеру готовий, відранжований «короткий список» (shortlist), звільняючи його час для кваліфікованої роботи — комунікації з найкращими кандидатами.

Для реалізації цієї концепції, архітектура запропонованої системи має базуватися на двох фундаментальних принципах: гнучкості оцінювання та здатності до самоаналізу.

Гнучка Зважена Модель Оцінювання (Операційний Контур)

Проблема в тому, що "найкращий" кандидат — це не статичне поняття. Критерії успіху кардинально різняться для різних посад. Наприклад:

Для вакансії "Junior Developer": Ключовим критерієм є освіта (фундаментальні знання) та технічні навички (знання мови програмування). Досвід роботи майже не має ваги.

Для вакансії "Senior Developer": Освіта майже не має ваги. Ключовим є досвід (роки роботи, складність проєктів) та спеціалізовані навички (архітектура, DevOps).

Для вакансії "Team Lead" або "Sales Director": Технічні навички та досвід важливі, але ключовим фактором стають "інші" (soft) навички — лідерство, комунікація, стратегічне мислення.

Жоден статичний, "зашитий в код" алгоритм не може ефективно обробити всі три випадки. Тому концептуальне ядро нашої СППР — це модель зваженого оцінювання (Weighted Scoring Model), що базується на методології MCDA (Multi-Criteria Decision Analysis).

Інновація полягає в тому, що ваги критеріїв (напр., вага_досвіду, вага_освіти, вага_soft_skills) не є фіксованими. Вони реалізовані через проектування «Стратегія».

1.2 Аналіз предметної області та бізнес-процесів агентства з працевлаштування

Предметна область (ПрО) даного дослідження — це комплекс бізнес-процесів, пов'язаних із наданням послуг у сфері працевлаштування. Агентство з працевлаштування (рекрутингове агентство) виступає в ролі комерційного посередника, головна мета якого — пошук та оцінка кваліфікованих спеціалістів (кандидатів) на замовлення компаній-клієнтів (роботодавців).

Ця ПрО є висококонкурентною, орієнтованою на послуги (service-oriented) та критично залежною від якості та швидкості обробки інформації. На відміну від виробничого підприємства, де активом є станки чи сировина, головним активом рекрутингового агентства є його база даних кандидатів, репутація та експертиза його співробітників (агентів/рекрутерів).

Програмне забезпечення, що проєктується, має на меті автоматизувати та оптимізувати ці процеси, перетворивши суб'єктивну експертизу рекрутерів на формалізований, керований та аналізований алгоритм. Для цього необхідно чітко ідентифікувати учасників (стейкхолдерів) та потоки робіт (бізнес-процеси).

Для успішного проєктування системи необхідно визначити всіх учасників (стейкхолдерів), які взаємодіють з системою або знаходяться під її впливом. Їх можна поділити на дві основні групи: внутрішні користувачі, тобто співробітники агентства, для яких система безпосередньо розробляється, та зовнішні учасники.

Ключовими внутрішніми користувачами програмного забезпечення є Агент з Підтримки Клієнтів (Client Support Agent / Recruiter) та Адміністратор (Administrator). Агент є основним операційним користувачем системи. Це співробітник, який безпосередньо веде комунікацію з роботодавцями та кандидатами, часто виступаючи експертом у своїй галузі, наприклад, "IT-рекрутер". До його основних завдань в системі належить створення та ведення профілів компаній, публікація вакансій, робота з базою кандидатів (додавання нових профілів та пошук), а також ведення кандидатів по воронці найму, фіксуючи зміни їхніх статусів. Найважливішою є ключова взаємодія з СППР, яка полягає у запуску алгоритму підбору (скорингу) для конкретної вакансії та подальшому аналізі отриманого відранжованого списку. Для цієї ролі система має надавати максимально швидкий та інтуїтивно зрозумілий інтерфейс для

операційної роботи, що візуалізує результати роботи алгоритму у легкому для сприйняття вигляді, наприклад, у формі балів або відсотків.

Другою важливою внутрішньою роллю є Адміністратор. Це користувач з розширеними правами, який відповідає за налаштування, підтримку та аналіз роботи всієї системи, наприклад, керівник агентства або системний адміністратор. Його завдання включають керування обліковими записами Агентів та налаштування загальних довідників системи, таких як довідник навичок (Skills). Критично важливою є його взаємодія з СППР, а саме керування алгоритмом підбору. Адміністратор повинен мати доступ до налаштувань «Стратегій» зважування, що дозволяє йому створювати нові та редагувати існуючі моделі оцінювання (наприклад, "Стратегія для Junior-позицій", "Стратегія для Sales-менеджерів"), встановлюючи різні вагові коефіцієнти для критеріїв. Крім того, Адміністратор є головним споживачем аналітичного контуру (OLAP-куба), отримуючи доступ до зведених звітів та BI-дашбордів про ефективність роботи як агентства в цілому, так і самих алгоритмів.

До зовнішніх учасників процесу належать Кандидат (Employee / Job Seeker) та Клієнт (Employer / Company). Кандидат є джерелом первинних даних, надаючи своє резюме та інформацію про навички й досвід. Клієнт (роботодавець) виступає замовником послуги, формулюючи вимоги до вакансії. Критично важливо, що Клієнт також є джерелом фінального зворотного зв'язку про успішність найнятого кандидата, що є ключовим показником для аналітичної підсистеми.

Для детального розуміння потоків інформації та ідентифікації точок прийняття рішень, де СППР може принести найбільшу користь, необхідно провести декомпозицію рекрутингового циклу. Для аналізу та візуалізації подібних процесів найчастіше використовується нотація BPMN (Business Process Model and Notation). Нижче наведено детальний опис ключових фаз бізнес-процесу «Повний цикл підбору кандидата».

Процес починається з Фази ініціації та аналізу. В якості стартової події (Start Event) виступає отримання запиту (Message Event) від Клієнта-Роботодавця на відкриття нової вакансії. Першим завданням (Task) Агента є «Аналіз вимог вакансії», що включає детальну комунікацію з клієнтом для уточнення обов'язків, необхідного досвіду, ключових навичок та рівня заробітної плати. Після цього Агент виконує завдання «Формалізація профілю посади» та «Створення вакансії

в системі (OLTP)», заповнюючи структуровані поля в операційній базі даних та прив'язуючи до вакансії відповідні компетенції з довідника Skills. Цей етап є критичним, оскільки якість формалізованих вхідних даних безпосередньо впливає на точність роботи майбутнього алгоритму.

Наступною є Фаза пошуку та збору даних (Sourcing & Collection). Агент ініціює паралельний пошук (що відповідає Parallel Gateway у BPMN) за кількома напрямками: пошук у власній базі даних (OLTP), публікація вакансії на зовнішніх job-порталах (як-от Work.ua, Robota.ua) та активний пошук (Headhunting) у професійних мережах, наприклад, LinkedIn. Це призводить до запуску підпроцесу «Обробка вхідних відгуків», де система отримує великий потік нових резюме від кандидатів. У ідеальній системі цей етап має включати автоматизований парсинг неструктурованих резюме та створення на їх основі нових профілів у таблиці Candidates, формуючи таким чином великий «вхідний пул» для подальшої обробки.

Найбільш критичною для нашого дослідження є Фаза оцінки та прийняття рішень (Assessment & Decision-Making). Саме тут виникають головні «вузькі місця» ручного процесу. Агент стикається з першою точкою прийняття рішення (Decision Point 1): первинним скринінгом, де необхідно відфільтрувати сотні резюме, що очевидно не відповідають базовим вимогам (наприклад, інше місто, недостатній досвід). Цей процес є механічним, втомливим та схильним до помилок. Наша СППР може повністю автоматизувати це завдання (Automated Task), миттєво відфільтровуючи нерелевантних кандидатів. Після цього Агент залишається з проміжним пулом кваліфікованих кандидатів (наприклад, 30-50 осіб) і стикається з другою, ключовою точкою прийняття рішення (Decision Point 2): «Хто з цих вже кваліфікованих кандидатів є найкращим?». Ручний порівняльний аналіз 30 кандидатів є надзвичайно суб'єктивним та часозатратним. Саме тут втручається СППР. Агент активує алгоритм, і система (Automated Task) виконує «Розрахунок Балу Релевантності (Match Score)» для кожного кандидата, застосовуючи зважену «Стратегію», обрану Адміністратором. Система повертає Агенту відранжований список, на основі якого він аналізує лише топ-10 кандидатів і приймає фінальне експертне рішення.

Після відбору найкращих кандидатів починається Фаза верифікації та закриття (Verification & Closing). Агент проводить співбесіди, додає свої експертні нотатки (agent_notes) в систему та презентує фіналістів клієнту. На

цьому етапі виникає важливий шлюз (Gateway) «Клієнт задоволений?». Якщо ні, клієнт надає уточнюючий зворотний зв'язок (наприклад, «потрібно більше досвіду з хмарними технологіями»), і Агент повертається до Фази 3, де Адміністратор може скоригувати ваги у «Стратегії» для більш точного повторного пошуку. Якщо так, процес завершується подією «Кандидата найнято» (End Event), і Агент оновлює статус в Applications на 'Hired'.

Заключною фазою, що забезпечує аналітичний контур, є Фаза аналізу ефективності (Post-Mortem Analysis). Вона активується після найму кандидата. Система автоматично очікує заданий період, наприклад, 3 місяці, що відповідає стандартному випробувальному терміну (Timer Event). Після цього система ініціює процес збору зворотного зв'язку, надсилаючи Агенту запит на отримання оцінки від роботодавця. Агент фіксує цю оцінку (employer_performance_rating) у таблиці Performance_Reviews. Ці дані є критично важливими, оскільки вони замикають цикл. За допомогою нічного ETL-процесу (Extract, Transform, Load) ці дані про реальну успішність найму завантажуються в OLAP-куб. Це дозволяє Адміністратору (керівнику) за допомогою BI-інструментів аналізувати, яка «Стратегія» зважування дала найкращі реальні результати, і на основі цього науково обґрунтовано вдосконалювати алгоритми підбору.

Детальний аналіз предметної області та декомпозиція бізнес-процесів агентства з працевлаштування дозволяють зробити два фундаментальні висновки, що лягли в основу проєктування системи.

По-перше, чітко ідентифіковано дві ключові точки прийняття рішень, які є «вузькими місцями» процесу: первинний скринінг (фільтрація нерелевантних відгуків) та вторинний скоринг (ранжування кваліфікованих кандидатів). Саме ці етапи найбільше страждають від суб'єктивності, когнітивних упереджень та проблем масштабування при ручній обробці. Це доводить, що фокус розробки СППР має бути спрямований саме на автоматизацію та інтелектуальну підтримку цих двох етапів, надаючи агенту об'єктивний, відранжований список кандидатів.

По-друге, аналіз процесів виявив подвійну природу навантаження на дані. З одного боку, система генерує постійний потік дрібних, частих транзакцій (додавання кандидата, оновлення статусу вакансії, запис нотаток), що характерно для OLTP (Online Transaction Processing) систем. З іншого боку, для виконання концептуальної задачі СППР (аналіз ефективності та самовдосконалення), системі необхідно виконувати рідкісні, але надзвичайно складні аналітичні

запити (наприклад, «розрахувати середній успіх кандидатів, найнятих за 'Стратегією А' за останній рік по галузі 'IT'»). Виконання таких запитів на "живій" OLTP-базі є неефективним та небезпечним. Це доводить, що для побудови надійної та масштабованої системи необхідна дворівнева архітектура даних (two-tier data architecture): операційна OLTP-база даних (напр., PostgreSQL) для роботи FastAPI-додатку та окреме, денормалізоване Сховище Даних (DWH) з OLAP-кубом для виконання аналітичних запитів.

1.3. Аналіз існуючих комерційних рішень (ATS та українські job-портали)

Успішна реалізація поставленого завдання, а саме розробка вдосконаленого програмного забезпечення (Вимога 1), неможлива без глибокого аналізу та критичної оцінки існуючих рішень (Вимога 5). Ринок програмного забезпечення для рекрутингу є зрілим та насиченим, однак його інструменти можна чітко класифікувати на два великі сегменти, які вирішують кардинально різні, хоча й пов'язані, задачі: Job-портали (Job Boards) та Системи Управління Кандидатами (Applicant Tracking Systems, ATS).

Перша категорія, Job-портали, функціонує як цифрові «дошки оголошень» або ринкові майданчики (маркетплейси). Їхня основна бізнес-модель — надання платформи для публікації вакансій роботодавцями та розміщення резюме кандидатами. Вони вирішують задачу пошуку (sourcing) та забезпечення обсягу кандидатів.

Друга категорія, ATS, є інструментом внутрішнього менеджменту. Це програмне забезпечення, яке агентство чи компанія використовує для організації вже отриманого потоку кандидатів, відстеження їх по воронці найму та управління комунікаціями. Вони вирішують задачу процес-менеджменту (process management).

Як буде показано далі, обидва ці сегменти мають фундаментальні обмеження і не вирішують ключової задачі, поставленої в цій роботі: надання інтелектуальної підтримки прийняття рішень та аналізу ефективності цих рішень.

Для предметного аналізу українського ринку розглянемо чотири провідні платформи, що представляють різні моделі: Work.ua, Robota.ua, Jooble та Djinni.

Work.ua та Robota.ua є двома найбільшими гравцями на загальному ринку праці в Україні. Вони оперують за моделлю «цифрової дошки оголошень». Роботодавці (або агентства від їхнього імені) платять за публікацію вакансій. Кандидати розміщують резюме безкоштовно.

Сильні сторони: Головною перевагою цих систем є їхній масштаб. Вони акумулюють мільйони резюме та тисячі вакансій, забезпечуючи максимальне охоплення ринку. Вони пропонують базові інструменти фільтрації за ключовими словами, містом, категорією та діапазоном заробітної плати.

Слабкі сторони (Проблема СППР): З точки зору підтримки прийняття рішень, ці платформи не лише не вирішують, а й погіршують проблему, описану в підрозділі 1.1. Вони є джерелом того самого «інформаційного перевантаження». Система не дає агенту відповіді на питання «Хто з 500 кандидатів, що відгукнулися, є найкращим?». Вони повертають хронологічний або слабо релевантний список, змушуючи рекрутера вручну виконувати первинний скринінг. Їхні інструменти пошуку базуються на простому лексичному збігу (keyword matching), ігноруючи семантичну близькість чи складніші критерії. Вони не надають жодних інструментів для зваженого скорингу чи ранжування.

Jooble є глобальним агрегатором вакансій з українським корінням. Його модель відрізняється: Jooble не зберігає власні резюме, а індексує вакансії з тисяч інших джерел (включаючи Work.ua та Robota.ua). Для роботодавця це інструмент дистрибуції вакансій. Для кандидата — єдина точка пошуку. З точки зору СППР, Jooble також є джерелом трафіку, а не інструментом прийняття рішень.

Djinni.co представляє третій, гібридний тип платформи, сфокусований виключно на IT-галузі. Його модель більш просунута: кандидати розміщують профілі анонімно, вказуючи бажану зарплату, а рекрутери «пропонують» їм вакансії («білд»). Ця система вже має елементи СППР: вона проводить початковий метчинг (matching) на основі збігу технологій та зарплатних очікувань.

Сильні сторони: Висока якість та релевантність кандидатів у своїй ніші (IT). Анонімність та проактивна модель ("кандидат обирає") зменшує "шум" для розробників.

Слабкі сторони (Проблема СППР): Незважаючи на кращий метчинг, Djinni все ще не виконує функції повноцінної СППР. Рішення про те, кому з 50 «змечених» кандидатів написати першим, все ще приймає рекрутер на основі ручного аналізу профілів. Система не пропонує зваженого скорингу. Вона чудово порівнює "Hard Skills" (Python, SQL), але ніяк не оцінює "Soft Skills" (напр., лідерство, комунікація) і не дозволяє агенту застосовувати різні моделі оцінювання (напр., для Junior vs. Senior).

Отже, українські job-портали є лише джерелами кандидатів. Вони є частиною Фази 2 "Пошук та Збір Даних" (див. підрозділ 1.2.3), але не надають жодних інструментів для Фази 3 "Оцінка та Прийняття Рішень".

Системи ATS (Applicant Tracking Systems) є наступним логічним кроком еволюції ПЗ для рекрутингу. Вони розроблені для вирішення проблеми хаосу, створюваного job-порталами. ATS фокусуються на управлінні процесом найму. До світових лідерів у цьому сегменті належать Greenhouse та Lever. На українському ринку популярними рішеннями є CleverStaff та PeopleForce.

Ці системи є класичними OLTP-системами ("цифровими картотеками", як було зазначено в 1.1). Їхні ключові функції:

- Централізація: Збір кандидатів з усіх джерел (пошта, job-портали, LinkedIn) в єдину базу даних.
- Управління Воронкою: Можливість візуально "перетягувати" кандидата по етапах (напр., «Новий», «Скринінг», «Інтерв'ю HR», «Інтерв'ю технічне», «Офер»).
- Колаборація: Можливість команді рекрутерів та наймаючих менеджерів залишати коментарі, оцінки (зазвичай у вигляді зірочок 1-5) та планувати співбесіди.
- Комунікація: Збереження історії листування, шаблони листів тощо.

Сильні сторони: ATS чудово вирішують задачу організації та менеджменту робочого процесу. Вони є незамінними для забезпечення прозорості, відповідності (compliance) та спільної роботи.

Слабкі сторони (Проблема СППР): Саме в їхній силі і криється їхня слабкість з точки зору нашої задачі. Вони є пасивними сховищами даних, а не активними системами підтримки рішень.

Відсутність Зваженого Скорингу: Більшість ATS пропонують лише простий пошук за ключовими словами або ручне виставлення "зірочок". Вони не мають вбудованого механізму для автоматичного розрахунку match_score на основі складної, зваженої моделі.

Відсутність Гнучкості (Патерн «Стратегія»): В ATS неможливо застосувати різні моделі оцінювання для різних вакансій. Критерії для Junior та Senior позицій будуть однаковими, що є методологічно невірним.

Відсутність Аналітичного Контуру (OLAP): ATS є OLTP-системами. Вони можуть генерувати прості операційні звіти (напр., "скільки часу вакансія була відкрита"). Але вони не можуть відповісти на складні аналітичні запити, наприклад: "Яка реальна успішність (employer_performance_rating) кандидатів, яких ми найняли, віддаючи перевагу досвіду, а не освіті?". Їхня архітектура не призначена для OLAP-запитів, і в них відсутній механізм збору зворотного зв'язку про успішність кандидата після найму.

Проведений аналіз комерційних рішень дозволяє чітко ідентифікувати незаповнений "пробіл" (gap) на ринку програмного забезпечення, який і покликана вирішити наша СППР.

Job-портали (Work.ua, Robota.ua, Djinni) вирішують задачу ПОШУКУ, надаючи великий потік неструктурованих кандидатів, але створюючи проблему «інформаційного перевантаження».

Системи ATS (CleverStaff, Greenhouse) вирішують задачу МЕНЕДЖМЕНТУ, організовуючи цей потік у воронку, але не надаючи інтелектуальних інструментів для прийняття рішень всередині цієї воронки.

Для ілюстрації, наведемо порівняльну таблицю функціональності:

- Функціонал
- Job-Портали (Work.ua, Djinni)
- Стандартні ATS (CleverStaff)
- Наша СППР (Концепція)

Функціонал	Job-Портали (Work.ua, Djinni)	Стандартні ATS (CleverStaff)	Наша СППР (Концепція)
1. База кандидатів та вакансій	✓ Так	✓ Так	✓ Так
2. Управління воронкою найму	✗ Ні	✓ Так (Сильна сторона)	✓ Так

3. Автоматичний зважений скоринг	✗ Ні	✗ Ні (лише ручні оцінки)	✓ Так (Ключова функція)
4. Гнучкі моделі оцінювання (Патерн «Стратегія»)	✗ Ні	✗ Ні	✓ Так (Наукова новизна)
5. Збір зворотного зв'язку (Post-Hire)	✗ Ні	✗ Ні	✓ Так (Ключова функція)
6. Аналітичний контур (OLAP-куб)	✗ Ні	✗ Ні	✓ Так (Наукова новизна)

Таблиця 1. Порівнянь сайтів з надання послуг у сфері працевлаштування

Як видно з таблиці, на ринку існує чіткий функціональний дефіцит. Немає системи, яка б поєднувала управління процесом (функції ATS) з інтелектуальною підтримкою прийняття рішень (зважений скоринг) та, що найважливіше, з аналітичним контуром (OLAP), що дозволяє аналізувати та вдосконалювати самі алгоритми скорингу.

Саме цей дефіцит і формує основу для постановки завдання нашого магістерського дослідження.

1.5. Постановка завдання щодо проведення магістерського дослідження

Проведений у попередніх підрозділах системний аналіз предметної області (1.2) та існуючих комерційних рішень (1.3) дозволяє чітко формалізувати проблему, яку покликане вирішити дане магістерське дослідження. Аналіз виявив фундаментальний розрив між двома класами існуючих систем: Job-портали (Work.ua, Djinni) вирішують задачу пошуку кандидатів, створюючи при цьому проблему інформаційного перевантаження, тоді як системи ATS (CleverStaff) вирішують задачу менеджменту цього потоку, але залишаються пасивними OLTP-системами («цифровими картотеками»).

Таким чином, ідентифіковано чіткий функціональний дефіцит: на ринку відсутнє інтегроване рішення, яке б надавало активну інтелектуальну підтримку прийняття рішень та, що найважливіше, аналітичний інструментарій для оцінки та вдосконалення цих рішень. Існуючі системи не відповідають на ключове питання: «Яка стратегія відбору кандидатів (наприклад, пріоритет освіти чи досвіду) призводить до найму реально успішних співробітників?».

Отже, проблема полягає не просто у створенні ще однієї системи для зберігання резюме. Проблема полягає у відсутності науково обґрунтованого програмного інструментарію, який би:

Автоматизував процес ранжування кандидатів на основі гнучких, а не статичних, критеріїв.

Створював замкнений цикл зворотного зв'язку (feedback loop), збираючи дані про успішність найму.

Надавав засоби для аналізу (OLAP) цих даних з метою наукового обґрунтування та оптимізації самих алгоритмів відбору.

Задача вибору найкращого кандидата з пулу кваліфікованих претендентів є класичною проблемою багатокритеріального прийняття рішень (Multi-Criteria Decision Making, MCDM). Агент (рекрутер) змушений одночасно оцінювати кандидата за сукупністю різнорідних, часто суперечливих, критеріїв: рівень освіти, релевантність досвіду, відповідність навичок (hard skills), наявність "м'яких" навичок (soft skills), зарплатні очікування, місцезнаходження тощо. Ручне порівняння цих векторів для десятків кандидатів є джерелом когнітивних упереджень, як було зазначено у підрозділі 1.1.

Тому методологічною основою для алгоритмічного ядра СППР, що розробляється, було обрано саме методи MCDM. Існує безліч методів MCDM, таких як AHP (Analytic Hierarchy Process – Метод аналізу ієрархій), TOPSIS (Technique for Order of Preference by Similarity to Ideal Solution) або PROMETHEE. Однак, аналіз патенту US20130166459A1 (див. 1.4) показав високу ефективність та гнучкість адитивної моделі зважування (Weighted Sum Model, WSM). Ця модель передбачає розрахунок інтегральної оцінки (Score) як суми добутків оцінки кандидата за кожним критерієм (Si) на ваговий коефіцієнт (Wi) цього критерію:

$$Score = \sum_{i=1}^{n} Wi * Si$$

Головна перевага цього методу для нашої задачі — його прозорість та керованість. Саме вектор ваг (W) стає формалізованим вираженням «Стратегії підбору». Наприклад, стратегія для Junior-позицій може мати вектор ваг

{w_освіта: 0.5, w_досвід: 0.1, w_навички: 0.4}, тоді як стратегія для Senior-позицій — {w_освіта: 0.05, w_досвід: 0.5, w_soft_skills: 0.45}.

Відповідно, основне дослідження у цій роботі буде зосереджено не на розробці нового MCDM-методу, а на дослідженні ефективності різних векторів ваг (Стратегій). Ми висуваємо гіпотезу, що не існує єдиного "правильного" вектора ваг; натомість, оптимальний вектор ваг залежить від типу вакансії. Наше дослідження спрямоване на те, щоб довести цю гіпотезу, аналізуючи дані зворотного зв'язку (employer_performance_rating) за допомогою спроєктованого OLAP-куба.

Для практичної реалізації дослідження необхідний відповідний технологічний стек. Вибір стеку ґрунтувався на критеріях швидкості розробки (оскільки робота ведеться одним дослідником), доступності інструментів та відповідності задачі.

Вибір мови програмування та фреймворку API:

Для розробки серверної частини (API) було проведено порівняльний аналіз трьох популярних платформ: Python (з FastAPI), C# (з .NET) та JavaScript (з Node.js).

Критерій	Python (FastAPI)	C# (.NET)	Node.js (Express)
Продуктивність (Raw Speed)	Висока (завдяки ASGI)	Дуже висока (компільована)	Висока (асинхронна)
Екосистема для Data Science	Відмінна. Лідер ринку (Pandas, Scikit-learn, PyTorch).	Добра (ML.NET), але обмежена.	Посередня.
Швидкість розробки (Solo Dev)	Дуже висока. Чистий синтаксис, Pydantic.	Середня (вимагає більше коду).	Висока, але слабка типізація.
Валідація даних	Відмінна (Pydantic). Автоматична, на основі типів.	Дуже добра (Data Annotations).	Посередня (потребує сторонніх бібліотек).

Зручність для соло-розробки	Найкраща.	Добра, але орієнтована на Enterprise.	Добра, але асинхронність складна.

Таблиця 2 – Порівняльний аналіз технологій для розробки API

Для даної магістерської роботи Python у зв'язці з FastAPI був обраний як оптимальне рішення. Головним аргументом є не лише зручність та швидкість розробки для одного дослідника, але й наявність найпотужнішого в індустрії інструментарію для майбутнього аналізу даних та машинного навчання (бібліотеки Pandas, Scikit-learn, Power BI). FastAPI, завдяки валідації Pydantic та асинхронності (ASGI), забезпечує продуктивність та надійність на рівні Enterprise-рішень, залишаючись при цьому лаконічним.

Як було обґрунтовано у підрозділі 1.2, жодна існуюча система не має аналітичного контуру. Для реалізації нашого головного дослідження (аналізу стратегій) необхідна дворівнева архітектура даних:

- OLTP (PostgreSQL): Операційна база для FastAPI, що забезпечує швидкі транзакції (INSERT, UPDATE).
- DWH / OLAP (Сховище Даних): Спроєктований OLAP-куб (Схема «Зірка») для агрегації даних та виконання складних аналітичних запитів без навантаження на основну систему.

Виходячи з формалізованої проблеми, обраної методології MCDM та обґрунтованого стеку, основне дослідження полягає у доведенні гіпотези, що ефективність найму прямо залежить від правильного вибору стратегії зважування критеріїв, яка, у свою чергу, залежить від типу вакансії.

Для досягнення цієї мети у роботі ставляться наступні завдання:

1. Провести системний аналіз предметної області, існуючих рішень (комерційних та наукових/патентних) для виявлення функціонального дефіциту. (Виконано в Розділі 1).
2. Спроєктувати програмну архітектуру СППР, що включає операційний (OLTP) та аналітичний (OLAP) контури.
3. Розробити програмний прототип API на FastAPI, що реалізує операційну логіку.

4. Адаптувати та реалізувати алгоритм зваженого оцінювання (на основі патенту US20130166459A1) з використанням патерну «Стратегія» для забезпечення гнучкості вибору ваг.
5. Спроєктувати та реалізувати багатовимірний OLAP-куб (Схема «Зірка») як інструмент для збору та агрегації даних зворотного зв'язку (Performance_Reviews).
6. Провести дослідження шляхом аналізу згенерованих даних в OLAP-кубі (за допомогою Power BI) для підтвердження гіпотези про те, що різні рівні вакансій (Junior, Senior, Lead) вимагають пріоритезації різних критеріїв (освіта, досвід, soft-skills) для досягнення найкращого результату найму.
7. Сформулювати рекомендації щодо використання розробленої системи та оптимальних стратегій зважування для різних типів вакансій.

2. МОДЕЛЮВАННЯ СИСТЕМИ

2.1. Вибір методології моделювання та аналіз стейкхолдерів (Вибір UML як нотації, фіналізація ролей Адміністратора та Агента).

Перехід від системного аналізу предметної області, представленого в першому розділі, до безпосередньої розробки програмного забезпечення вимагає проміжного, але критично важливого етапу — моделювання. Моделювання виступає мостом між бізнес-вимогами, виявленими під час аналізу, та технічною архітектурою, яка буде реалізована в коді. Головна мета цього розділу — створити чіткі, формалізовані та однозначні "креслення" майбутньої системи, які будуть зрозумілі як замовнику (для верифікації функціоналу), так і команді розробників (для імплементації).

Для вирішення завдання моделювання інформаційних систем існує низка підходів, однак у сучасній інженерії програмного забезпечення (Software Engineering) де-факто індустріальним стандартом є Уніфікована мова моделювання (UML – Unified Modeling Language). Вибір UML для даної магістерської роботи обґрунтований кількома ключовими перевагами, що безпосередньо відповідають вимогам до проєктування СППР.

По-перше, UML надає багатий набір графічних нотацій для опису системи з різних точок зору. Ваші вимоги до магістерської роботи чітко вказують на необхідність представити як функціональний, так і об'єктно-орієнтований підходи. UML ідеально підтримує обидва: функціональний підхід реалізується через Діаграми Прецедентів (Use Case) та Діаграми Активності (Activity), тоді як об'єктно-орієнтований — через Діаграми Послідовності (Sequence) та Діаграми Класів (Class).

По-друге, UML є універсальною та стандартизованою мовою. Це означає, що розроблені діаграми будуть зрозумілі будь-якому іншому інженеру чи аналітику, що відповідає академічним вимогам до відтворюваності та ясності дослідження. На відміну від пропрієтарних чи авторських методологій, UML є задокументованим стандартом (ISO/IEC 19505), що забезпечує точність інтерпретації моделей.

По-третє, діаграми UML є ітеративними. Моделі, створені на ранніх етапах (як-от Діаграма Прецедентів), стають основою для більш детальних діаграм на пізніх етапах (як-от Діаграми Послідовності та Класів). Це забезпечує наскрізний процес проєктування, де кожен наступний крок логічно випливає з попереднього, що повністю відповідає рекомендованій структурі цього розділу: від прецедентів до проєктування.

Саме тому UML обрано як основний інструментарій для візуального моделювання та проєктування архітектури СППР у цій роботі. У наступних підрозділах ми послідовно застосуємо ключові діаграми UML для деталізації вимог та структури системи.

Першим кроком у функціональному моделюванні за допомогою UML є ідентифікація акторів (actors) — зовнішніх сутностей, що взаємодіятимуть із системою. Актор — це не конкретна людина, а роль, яку виконує користувач стосовно системи. Як було визначено у Підрозділі 1.2, наша СППР є внутрішньою (B2E – Business-to-Employee) системою агентства. Таким чином, ми виділяємо дві ключові ролі користувачів, які стануть нашими головними акторами на Діаграмі Прецедентів: Адміністратор Системи та Агент з працевлаштування.

Адміністратор Системи (Admin) є технічним або управлінським спеціалістом, відповідальним за стабільність, конфігурацію та підтримку життєдіяльності системи. Його основна мета — не виконання бізнес-процесів рекрутингу, а забезпечення їхньої можливості. Адміністратор має повний контроль над довідковими даними та налаштуваннями системи. Його ключові завдання включають керування обліковими записами інших користувачів (створення, блокування Агентів), налаштування довідників (наприклад, додавання нових навичок до Skills або галузей для Employers), а також моніторинг стану системи. У контексті нашої СППР, Адміністратору надається виключне право на керування найважливішим елементом — алгоритмічним ядром. Саме він має доступ до налаштування та вибору патерну «Стратегія» за замовчуванням, керування ваговими коефіцієнтами для різних стратегій оцінювання та доступу до аналітичних звітів з OLAP-куба для оцінки ефективності цих стратегій.

Агент з працевлаштування (Client Support Agent / Recruiter) є основним бізнес-користувачем системи. Це співробітник агентства, що безпосередньо виконує операційні завдання з пошуку та підбору персоналу. Мета Агента — максимально швидко та якісно закрити вакансію, надану компанією-замовником. Він є головним споживачем функціоналу підтримки прийняття рішень. Його щоденні завдання повністю базуються на взаємодії з основними сутностями системи: він створює та веде профілі компаній-роботодавців (Employers), додає та редагує відкриті вакансії (Job_Openings), а також керує базою кандидатів (Candidates).

Найважливіша взаємодія Агента з системою — це використання її інтелектуального ядра. Коли Агент отримує запит на закриття вакансії, він не повинен вручну переглядати тисячі профілів. Замість цього, він ініціює процес автоматизованого підбору. Система, використовуючи обрану зважену стратегію,

аналізує всіх кандидатів у базі даних і надає Агенту короткий, відранжований список найбільш релевантних осіб (match_score). Таким чином, система бере на себе рутинну роботу з "фільтрації", а Агент — експертну роботу з "валідації" та комунікації з топ-кандидатами. Крім того, Агент відповідає за закриття «циклу зворотного зв'язку» — саме він вносить дані до Performance_Reviews про успішність найму, що є критично важливим для майбутнього аналізу в OLAP-кубі.

Фіналізація цих двох ролей є відправною точкою для побудови Діаграми Прецедентів у наступному підрозділі, де ми формалізуємо кожну унікальну функцію, доступну Адміністратору та Агенту.

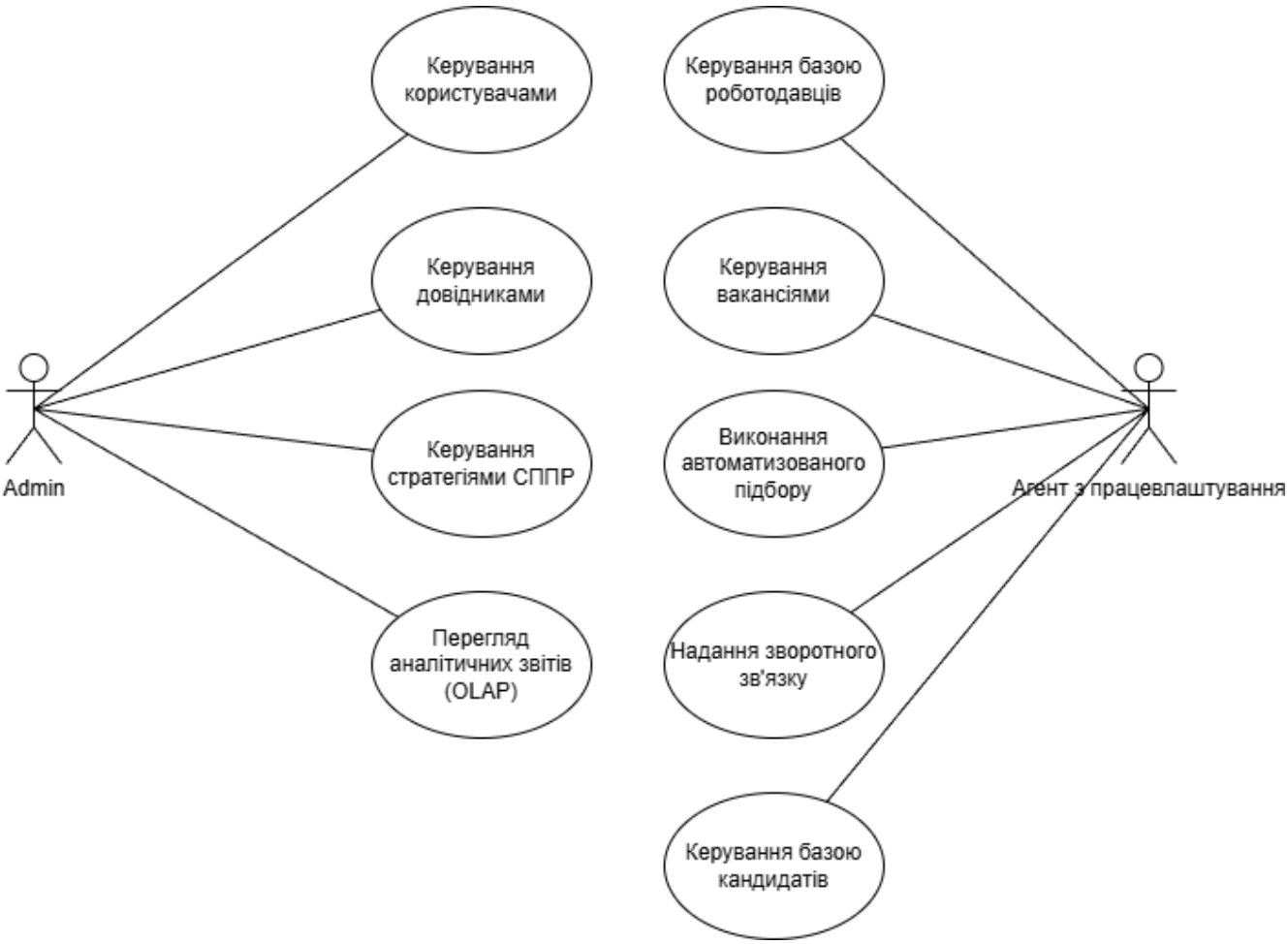
2.2. Функціональне моделювання: Діаграма прецедентів (Use Case Diagram)

На основі аналізу стейкхолдерів, проведеного у Підрозділі 2.1, ми переходимо до формалізації функціональних вимог до системи. Найкращим інструментом UML для цього є Діаграма Прецедентів (Use Case Diagram). Її головна мета — наочно продемонструвати взаємодію акторів із системою, відповівши на питання: «Що система робить для кожного актора?»

Діаграма прецедентів моделює систему як "чорну скриньку" і показує набір послуг, які вона надає. Кожна послуга називається прецедентом (або варіантом використання) і зображується у вигляді овалу. Актори (Адміністратор Системи та Агент з працевлаштування) зображуються у вигляді фігурок і з'єднуються лініями з тими прецедентами, які їм доступні.

Такий підхід дозволяє візуально окреслити межі системи та чітко розділити повноваження між різними ролями користувачів, що є фундаментальною вимогою для проєктування безпечного та логічно структурованого програмного забезпечення.

На Діаграмі 1. представлена загальна діаграма прецедентів для проєктованої СППР. Система (System Boundary) визначена як «СППР Агентства з працевлаштування». Поза її межами знаходяться два актори: Адміністратор Системи (ліворуч) та Агент з працевлаштування (праворуч). Всередині системи знаходиться набір прецедентів, згрупованих за логічними блоками.



Діаграма 2.1 – Загальна діаграма прецедентів системи

Як видно з діаграми, актори мають чітко розділені, але частково пересічні зони відповідальності. Адміністратор зосереджений на конфігурації та обслуговуванні системи, тоді як Агент — на використанні її бізнес-функцій.

Адміністратор Системи має доступ до функцій, що забезпечують стабільну та коректну роботу всієї платформи.

«Керування користувачами»: Цей прецедент описує базову адміністративну функцію. Він включає можливість створювати нові облікові записи (наприклад, для нового Агента), редагувати їхні дані (наприклад, змінити пароль) та блокувати облікові записи співробітників, які більше не працюють в агентстві.

«Керування довідниками»: Це критично важливий прецедент для підтримки якості даних в системі. Як було зазначено в Розділі 1, алгоритм підбору сильно залежить від структурованих даних. Цей прецедент дозволяє Адміністратору редагувати глобальні списки, що використовуються в усій

системі, наприклад, додавати нову навичку до довідника Skills (напр., "GPT-4 Integration"), новий рівень освіти або нову галузь (Industry) для компаній.

«Керування стратегіями СППР»: Це ключовий прецедент, що відображає наукову новизну роботи. Він надає Адміністратору доступ до "мозку" системи — модуля MatchingService. Цей прецедент дозволяє створювати нові зважені стратегії (наприклад, "Стратегія для IT-вакансій", "Стратегія для Sales-вакансій") та конфігурувати вагові коефіцієнти для кожної з них (наприклад, для IT: Skills = 0.6, Experience = 0.3, Education = 0.1).

«Перегляд аналітичних звітів (OLAP)»: Цей прецедент надає Адміністратору доступ до результатів, що зберігаються у Сховищі Даних. Він може переглядати дашборди, які показують ефективність роботи системи (наприклад, середня оцінка employer_performance_rating для кожної match_strategy_used), що дозволяє йому приймати обґрунтовані рішення про коригування вагових коефіцієнтів у попередньому прецеденті.

Агент з працевлаштування є основним операційним користувачем системи та взаємодіє з прецедентами, що безпосередньо пов'язані з процесом рекрутингу.

«Керування базою роботодавців»: Цей прецедент охоплює повний цикл роботи з компаніями-клієнтами. Він включає створення профілю нового роботодавця (Employer), редагування контактної інформації існуючих, та перегляд історії співпраці.

«Керування вакансіями»: Прецедент, що є центральним для бізнес-процесу. Він дозволяє Агенту створювати нову вакансію (Job_Opening), прив'язуючи її до конкретного роботодавця. Під час створення вакансії Агент заповнює ключові поля для алгоритму: необхідний досвід, вилку зарплати та, найголовніше, обирає необхідні навички з довідника Skills.

«Керування базою кандидатів»: Аналогічно до роботодавців, цей прецедент дозволяє Агенту створювати новий профіль кандидата (Candidate), завантажувати його резюме, редагувати профіль (наприклад, додати нові навички (Candidate_Skills) після проходження ним курсів) та змінювати його статус (напр., "Активний пошук", "Неактивний").

«Виконання автоматизованого підбору»: Це головний прецедент підтримки прийняття рішень. Агент, перебуваючи на сторінці вакансії, ініціює цей прецедент (натискає кнопку "Підібрати"). Система активує MatchingService, який використовує зважену стратегію (обрану Адміністратором або Агентом) для аналізу всієї бази Candidates. Результатом виконання прецеденту є надання Агенту відранжованого списку кандидатів з найвищим «Балом Релевантності» (match_score).

«Надання зворотного зв'язку»: Цей прецедент є критично важливим для аналітичного контуру системи (OLAP). Після того, як вакансія закрита (status = 'Filled'), цей прецедент дозволяє Агенту зафіксувати результат: внести дані до таблиці Performance_Reviews, вказавши, чи був кандидат успішним (employer_performance_rating), та наскільки рекомендація системи була коректною (agent_rating).

Для уникнення надмірної складності на загальній діаграмі, деякі прецеденти можна деталізувати. Наприклад, прецедент «Виконання автоматизованого підбору» є складним і завжди включає в себе менший прецедент «Розрахувати Бал Релевантності», який, у свою чергу, «включає» (<include>) прецедент «Отримати дані з БД».

З іншого боку, прецедент «Керування вакансіями» може бути «розширений» (<extend>) прецедентом «Надіслати пропозицію кандидату» — ця дія є опціональною і відбувається не кожного разу при керуванні вакансією.

Деталізація цих прецедентів буде представлена у наступних підрозділах за допомогою Діаграм Активності та Послідовності, які покажуть не тільки що система робить, але й як вона це робить.

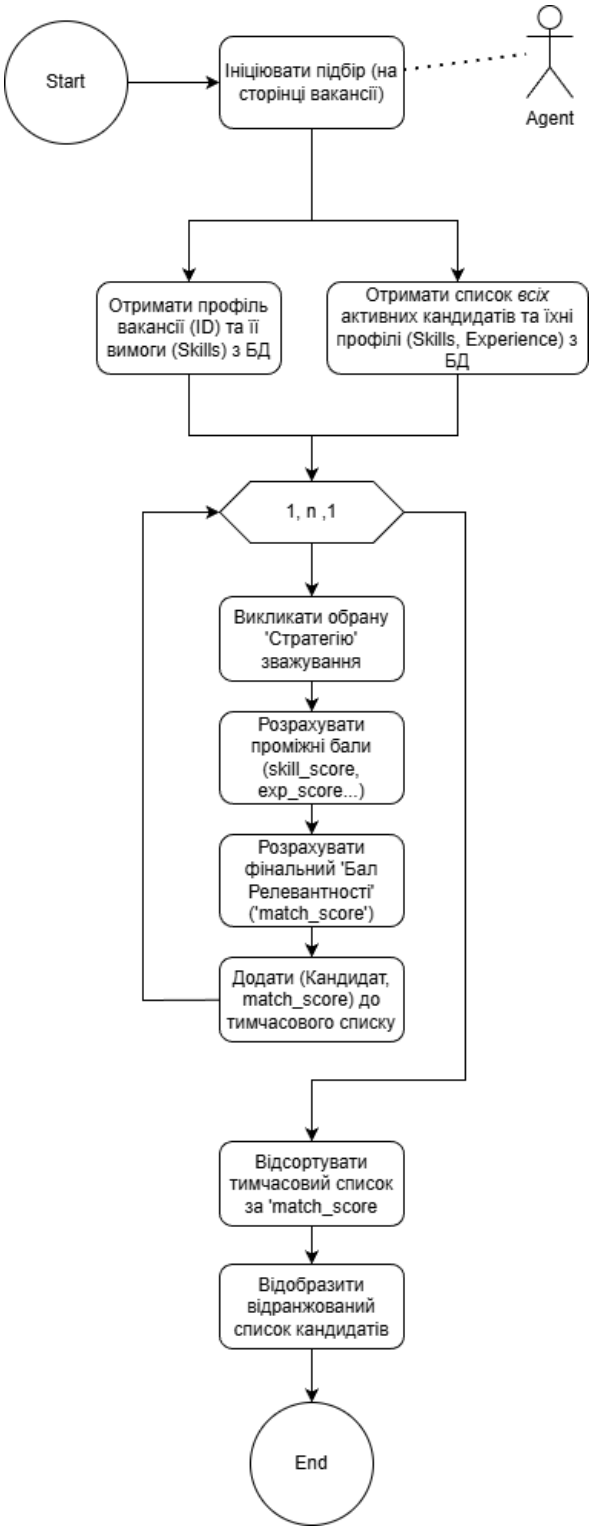
2.3. Моделювання бізнес-процесів: Діаграма активності (Activity Diagram)

Діаграма прецедентів, розглянута в попередньому підрозділі, успішно зафіксувала функціональні вимоги до СППР та окреслила межі взаємодії акторів із системою. Вона відповіла на питання "що?". Однак, ця діаграма є статичною і не розкриває внутрішньої логіки виконання прецедентів. Для опису динамічної поведінки, логіки прийняття рішень та послідовності операцій в межах складних бізнес-процесів використовується інший інструмент UML — Діаграма Активності (Activity Diagram).

Діаграма активності візуалізує потік робіт (workflow) від початкової до кінцевої точки, показуючи кроки, паралельні процеси, умови розгалуження та точки синхронізації. Це, по суті, сучасна версія блок-схеми, адаптована для моделювання бізнес-процесів та складних алгоритмів. У контексті нашої СППР, діаграми активності є критично важливими, оскільки два ключові прецеденти — «Виконання автоматизованого підбору» та «Надання зворотного зв'язку» — не є простими CRUD-операціями, а являють собою складні, багатоетапні алгоритми. Саме ці два процеси будуть детально змодельовані у цьому підрозділі.

Процес «Виконання автоматизованого підбору» є серцем СППР та найбільш обчислювально складним прецедентом у системі. Він ініціюється

Агентом з працевлаштування в момент, коли той знаходиться на сторінці конкретної вакансії (Job_Opening) і натискає кнопку запуску підбору. Ця дія слугує початковою точкою (initial node) на діаграмі активності.



Діаграма 2.2 Виконання автоматизованого підбору

Після отримання команди, система (в особі MatchingService) переходить до етапу збору даних. Оскільки для розрахунку релевантності потрібні різнорідні дані, система використовує механізм розгалуження (fork), ініціюючи два

паралельні потоки запитів до операційної OLTP-бази даних. Перший потік відповідає за отримання повного профілю цільової вакансії, що включає її ключові вимоги: необхідний досвід, діапазон заробітної плати та, найголовніше, список ідентифікаторів необхідних навичок. Другий, значно більш ресурсомісткий потік, відповідає за вибірку всіх активних кандидатів з бази даних (Candidates) разом з усіма їхніми пов'язаними даними — освітою, досвідом та повним списком навичок (Candidate_Skills).

Наступним кроком є вузол синхронізації (join), який гарантує, що основний процес не продовжиться, доки обидва потоки збору даних не будуть успішно завершені. Коли всі необхідні дані (профіль вакансії та повний список кандидатів) завантажені в оперативну пам'ять сервера, система готова до запуску основного алгоритму оцінювання.

Процес оцінювання реалізований як ітераційна діяльність (activity loop). Система послідовно обробляє кожного кандидата з отриманого списку. Для кожного окремого кандидата викликається обрана Стратегія зважування. Ця стратегія, в свою чергу, виконує низку під-алгоритмів: розраховує бал збігу навичок (skill_score), бал відповідності досвіду (experience_score), бал відповідності зарплатних очікувань (salary_score) та інші визначені показники. Потім ці проміжні бали перемножуються на відповідні вагові коефіцієнти, визначені у Стратегії, і сумуються, формуючи фінальний «Бал Релевантності» (match_score) для даного кандидата. Цей результат (пара candidate_id та match_score) зберігається у тимчасовому списку результатів. Цикл повторюється, доки не будуть оброблені всі кандидати з вибірки.

Після завершення циклу система виконує дію сортування тимчасового списку результатів за полем match_score у порядку спадання. Останнім кроком є вузол прийняття рішення (decision node), який зазвичай відсікає результати, що не відповідають мінімальному порогу релевантності, або просто обмежує вибірку (наприклад, Top-50). Відсортований та відфільтрований список передається Агенту, що є кінцевою точкою (activity final node) цього процесу. Агент бачить у своєму інтерфейсі відранжований список кандидатів, готових до подальшого опрацювання. Ця діаграма наочно демонструє, як прецедент "чорної скриньки" перетворюється на чіткий, керований та детермінований алгоритм.

Другий ключовий процес, «Надання зворотного зв'язку», є фундаментальним для аналітичного контуру системи та наповнення OLAP-куба. Без цього процесу СППР не здатна до самооцінки та вдосконалення. Цей процес також має складну, рознесену в часі логіку, що робить його ідеальним кандидатом для моделювання Діаграмою Активності.

Початковою точкою (initial node) є дія Агента, який змінює статус прецеденту Applications (зв'язку "кандидат-вакансія") на "Hired" (Найнято). Ця бізнес-подія слугує тригером, що ініціює паралельні потоки (fork).

Перший потік є негайним і спрямованим на самого Агента. Система відкриває в інтерфейсі Агента форму для надання зворотного зв'язку про якість роботи самого алгоритму. Агент виконує дію «Оцінити якість рекомендації», де він вказує свій експертний рейтинг (agent_rating) — наскільки бал, згенерований системою, відповідав реальній адекватності кандидата. Після збереження, ця інформація записується у відповідні поля таблиці Performance_Reviews в OLTP-базі.

Другий потік є відкладеним у часі. Система ініціює стан очікування (time event), тривалість якого дорівнює випробувальному терміну (наприклад, 90 днів). Діаграма активності показує цей потік як такий, що переходить у стан "Очікування завершення випробувального терміну". Після сплину 90 днів система автоматично створює завдання для Агента ("Отримати відгук від роботодавця"), яке з'являється на його дашборді.

Ця нотифікація спонукає Агента до дії «Зв'язатися з роботодавцем». Після отримання вербального чи письмового відгуку, Агент виконує фінальну дію процесу: «Внести оцінку роботодавця». Він заповнює ключове поле в Performance_Reviews — employer_performance_rating (оцінка реальної ефективності кандидата на робочому місці) та додає текстовий коментар.

Після виконання обох паралельних потоків (отримання відгуку від агента та отримання відгуку від роботодавця) спрацьовує вузол синхронізації (join). Завершення обох дій сигналізує системі, що запис у Performance_Reviews є повним. Ця повнота даних є тригером для ETL-процесу, який під час наступного нічного запуску зможе безпечно видобути (Extract) цей завершений запис і завантажити (Load) його до аналітичного Сховища Даних (DWH). Таким чином, ця діаграма активності візуалізує повний цикл аналітичного зворотного зв'язку — від дії Агента до наповнення OLAP-куба даними.

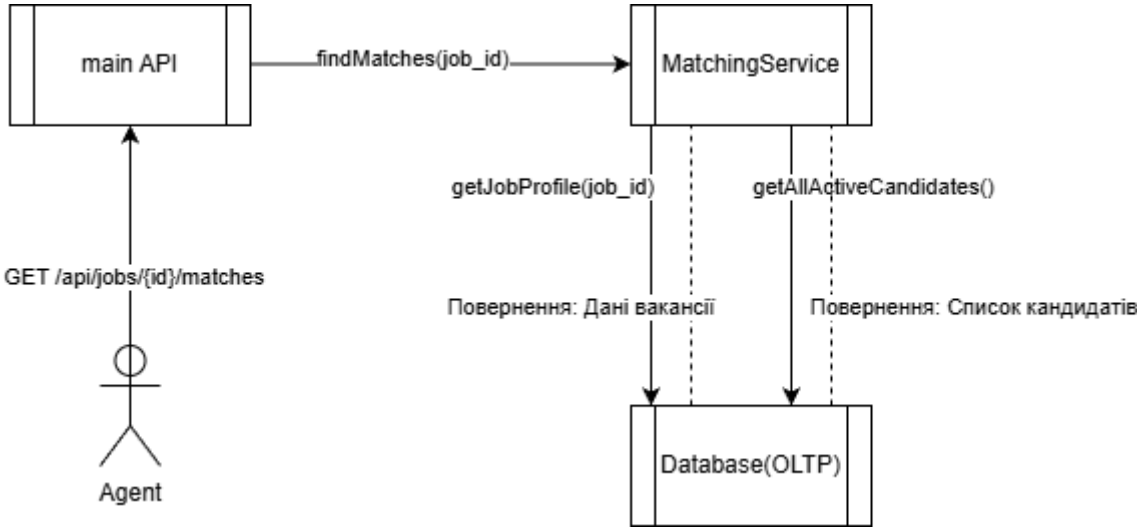
2.4. Об'єктно-орієнтоване моделювання взаємодії: Діаграма послідовності (Sequence Diagram)

Якщо діаграми прецедентів (Підрозділ 2.2) визначили набір функцій системи, а діаграми активності (Підрозділ 2.3) продемонстрували логіку бізнес-процесів, то Діаграма Послідовності (Sequence Diagram) слугує для моделювання взаємодії об'єктів у часі. Це ключовий інструмент об'єктно-орієнтованого аналізу та проєктування (ООАП), оскільки він переводить статичні вимоги у динамічну

модель взаємодії конкретних програмних компонентів. Діаграма фокусується не на потоці робіт, а на повідомленнях, якими обмінюються об'єкти для виконання завдання.

Діаграма послідовності візуалізує об'єкти (або компоненти системи) як вертикальні "лінії життя" (lifelines), а час протікає зверху вниз. Горизонтальні стрілки між цими лініями представляють повідомлення (message), які можуть бути синхронними викликами (коли відправник чекає на відповідь) або асинхронними (коли відправник продовжує роботу). Цей інструмент ідеально підходить для деталізації прецедентів, оскільки він наочно демонструє, як саме розподіляється відповідальність між різними шарами архітектури, що є критично важливим для реалізації нашого API-сервісу.

Найскладнішим та найважливішим прецедентом у нашій СППР є «Виконання автоматизованого підбору». Для його реалізації залучені всі ключові компоненти системи: користувач, API-сервер, сервіс бізнес-логіки та база даних. На Діаграмі 2.2 (яка описується нижче) представлено чотири лінії життя: :Агент (актор, що ініціює процес), :API (FastAPI) (шар, що приймає HTTP-запити), :MatchingService (об'єкт, що інкапсулює зважений алгоритм) та :Database (OLTP) (шар доступу до даних).



Діаграма 2.3 Діаграма послідовності

Процес починається, коли :Агент надсилає синхронне повідомлення GET /api/jobs/{id}/matches до :API (FastAPI). Це повідомлення являє собою HTTP-запит від клієнта. :API (FastAPI) отримує цей запит. Його зона відповідальності — валідація вхідних даних (наприклад, перевірка, що id вакансії є коректним) та маршрутизація. API-контролер не виконує бізнес-логіку самостійно, натомість він делегує це завдання, викликаючи метод findMatches(job_id) у об'єкта

:MatchingService. Це синхронний виклик, що активує лінію життя :MatchingService.

Після активації :MatchingService починає виконувати логіку, описану нами в діаграмі активності. Першим кроком є збір даних. Сервіс послідовно надсилає серію синхронних запитів до :Database (OLTP). Спочатку він надсилає запит getJobProfile(job_id), на що база даних повертає дані про цільову вакансію. Далі сервіс надсилає запит getJobSkills(job_id) та отримує список необхідних навичок. Третім запитом є getAllActiveCandidates(), який повертає масив даних усіх кандидатів, що перебувають у пошуку. Цей етап інтенсивної взаємодії з базою даних чітко показує фазу збору даних.

Коли всі необхідні дані повернуто з :Database (OLTP) і знаходяться в оперативній пам'яті сервера, починається найважливіший етап, який на діаграмі послідовності зображується як цикл (loop), що виконується виключно на лінії життя :MatchingService. Це підкреслює, що зважений алгоритм є обчислювально інтенсивною задачею, яка не вимагає подальших запитів до бази даних усередині циклу. :MatchingService ітерує по списку кандидатів і для кожного з них виконує внутрішній (self-call) метод calculateMatchScore(candidate, job_data), де і реалізована логіка патерну «Стратегія». Після завершення циклу, сервіс має повний відранжований список кандидатів.

Завершивши обчислення, :MatchingService повертає (return) відранжований список ranked_list на лінію життя :API (FastAPI), завершуючи свою активацію. :API (FastAPI) отримує цей список, серіалізує його у формат JSON та надсилає фінальну відповідь HTTP 200 OK (з JSON-тілом) ініціатору — :Агенту. Діаграма чітко ілюструє розділення відповідальності (Separation of Concerns): FastAPI відповідає за маршрутизацію та HTTP, MatchingService — за бізнес-логіку та алгоритми, а Database — виключно за зберігання даних.

Для контрасту, розглянемо інший ключовий прецедент — «Надання зворотного зв'язку». Цей процес значно простіший, оскільки він є операцією запису даних (write) і не вимагає складної бізнес-логіки. На Діаграмі 2.3 представлені три лінії життя: :Агент, :API (FastAPI) та :Database (OLTP). Об'єкт :MatchingService у цій взаємодії участі не бере.

Послідовність починається, коли :Агент надсилає повідомлення POST /api/reviews до :API (FastAPI). Це повідомлення містить JSON-тіло з даними відгуку (наприклад, application_id, agent_rating, employer_performance_rating). :API (FastAPI) отримує запит і негайно валідує вхідні дані за допомогою Pydantic-схем. Це важливий крок для забезпечення цілісності даних.

Після успішної валідації, :API (FastAPI) викликає метод createReview(review_data) у шару :Database (OLTP). У нашій архітектурі, де

використовується SQLAlchemy, цей виклик може бути спрямований до репозиторію, який інкапсулює логіку SQL-запиту. :Database (OLTP) виконує команду INSERT INTO Performance_Reviews ... і, у разі успіху, повертає (return) підтвердження (наприклад, success=true або ID нового запису) на лінію життя :API (FastAPI).

Отримавши підтвердження від бази даних, :API (FastAPI) завершує цикл запиту-відповіді, надсилаючи :Агенту повідомлення HTTP 201 Created, що сигналізує про успішне створення нового ресурсу (відгуку). Ця діаграма демонструє значно простішу взаємодію, типову для CRUD-операцій, де API-сервер виступає переважно валідатором та посередником між клієнтом та базою даних.

Порівняння цих двох діаграм послідовності наочно ілюструє різницю між простими транзакційними операціями та складними процесами підтримки прийняття рішень, де головна цінність створюється шляхом обчислень всередині сервісного шару (:MatchingService).

3. РОЗРОБКА СИСТЕМИ

3.1. Загальна архітектура програмного забезпечення

Після завершення етапу моделювання (Розділ 2), який визначив функціональні та динамічні аспекти майбутньої системи, ми переходимо до етапу розробки. Першочерговим завданням на цьому етапі є вибір та обґрунтування загальної архітектури програмного забезпечення. Архітектура системи визначає фундаментальну організацію її компонентів, їхні взаємозв'язки, а також принципи, що керують її проєктуванням та еволюцією. Враховуючи вимоги до системи, визначені в Розділі 1 — зокрема, необхідність підтримки декількох ролей користувачів (Адміністратор, Агент), обробку бізнес-логіки (зважений алгоритм) та взаємодію з системою зберігання даних — було прийнято рішення про реалізацію системи на основі класичної трирівневої архітектури (Three-Tier Architecture).

Трирівнева архітектура є де-факто індустріальним стандартом для розробки сучасних веб-додатків та сервісів. Вона передбачає логічний поділ системи на три основні, незалежні підсистеми (вузли): Рівень представлення (Client/Presentation Layer), Прикладний рівень (Application/Logic Layer) та Рівень даних (Data Layer). Такий підхід забезпечує ключові переваги для нашої СППР. По-перше, це модульність та гнучкість: кожен рівень може розроблятися, тестуватися, масштабуватися та оновлюватися незалежно від інших. Наприклад, ми можемо повністю змінити дизайн клієнтського додатку, не торкаючись бізнес-логіки на сервері. По-друге, це розподіл відповідальності (Separation of Concerns): рівень представлення відповідає лише за UI/UX, прикладний рівень — виключно за виконання бізнес-правил (наш зважений алгоритм), а рівень даних — лише за зберігання та вибірку інформації. Це значно спрощує розробку, підтримку та подальший розвиток системи.

Рівень представлення є першим вузлом архітектури, з яким безпосередньо взаємодіють користувачі. У контексті нашої системи, це клієнтський додаток (Client Application), який виконується у веб-браузері стейкхолдерів — Адміністратора та Агента з працевлаштування. Цей рівень не містить жодної бізнес-логіки чи прямого доступу до бази даних; його єдина відповідальність — це візуалізація інформації (UI) та забезпечення взаємодії з користувачем (UX).

Технологічно цей рівень реалізується як "тонкий клієнт" (Thin Client). Це може бути односторінковий додаток (Single Page Application, SPA), розроблений з використанням сучасних JavaScript-фреймворків (наприклад, React, Angular або Vue.js), або більш простий інтерфейс, що генерується сервером. Незалежно від

конкретної реалізації, його комунікація з наступним рівнем архітектури є чітко стандартизованою: клієнт надсилає HTTP-запити (GET, POST, PUT, DELETE) до прикладного рівня та отримує у відповідь дані у форматі JSON (JavaScript Object Notation). Наприклад, коли Агент натискає кнопку «Підібрати кандидатів», клієнтський рівень не виконує жодних обчислень; він лише формує та надсилає GET-запит до відповідного ендпоінту API. Отримавши у відповідь JSON-масив відранжованих кандидатів, його єдине завдання — коректно відобразити цей список користувачу.

Прикладний рівень є "мозком" та центральним вузлом всієї СППР. Саме тут реалізована вся бізнес-логіка, алгоритми та правила, що складають суть магістерського дослідження. Як було обґрунтовано в Підрозділі 1.5, для реалізації цього рівня обрано мову Python та фреймворк FastAPI. Цей серверний вузол, у свою чергу, має власну внутрішню архітектуру, що складається з кількох компонентів, детально змодельованих у Підрозділі 2.4 (Діаграма Послідовності).

Перший компонент — це Шар API (API Layer). Це "вхідні ворота" сервера, реалізовані за допомогою декораторів FastAPI (наприклад, @app.post("/api/candidates")). Цей шар відповідає за маршрутизацію HTTP-запитів до відповідних обробників, а також за валідацію вхідних та вихідних даних. Використання Pydantic-моделей у FastAPI дозволяє автоматично валідувати JSON-дані, що надходять від клієнта, гарантуючи, що Прикладний рівень отримає коректні та очищені дані для подальшої обробки.

Другий, найважливіший компонент — це Сервісний Шар (Service Layer) або Шар Бізнес-Логіки. Це ізольовані Python-класи, які інкапсулюють основні бізнес-процеси. Ключовим об'єктом тут є :MatchingService, який реалізує прецедент «Виконання автоматизованого підбору». Цей сервіс містить реалізацію патерну «Стратегія», що дозволяє динамічно обирати алгоритм зважування. Саме в цьому сервісі відбувається основна обчислювальна робота: отримання даних від шару даних, ітерація по кандидатах, застосування вагових коефіцієнтів та розрахунок match_score.

Третій компонент — це Шар Доступу до Даних (Data Access Layer, DAL). Цей шар реалізовано за допомогою SQLAlchemy ORM. Його завдання — повна абстракція від "сирого" SQL-коду. Сервісний шар (наприклад, :MatchingService) не пише SELECT * FROM ...; натомість він викликає методи DAL (наприклад, repository.get_all_candidates()). Це робить бізнес-логіку незалежною від конкретної реалізації бази даних (наприклад, перехід з SQLite на PostgreSQL не вимагатиме змін у логіці алгоритму) та значно спрощує тестування.

Третій вузол архітектури — це рівень даних, що відповідає за надійне та довготривале зберігання інформації (persistence). Важливою архітектурною

особливістю нашої СППР є те, що цей рівень складається з двох окремих, логічно та фізично розділених баз даних, які слугують різним цілям.

Перша — це Операційна (OLTP) База Даних. Це "жива" база (наприклад, PostgreSQL), з якою Прикладний рівень (API-сервер) взаємодіє в реальному часі. Вона спроєктована в Розділі 2 з використанням нормалізації (3NF) для забезпечення цілісності даних та уникнення дублювання. Вона оптимізована для частих, коротких транзакцій: INSERT (новий кандидат), UPDATE (зміна статусу вакансії), DELETE та швидких SELECT за індексами (отримати профіль кандидата). Це джерело "сирої" правди для системи.

Друга — це Аналітичне (OLAP) Сховище Даних (DWH), спроєктоване в Розділі 4.2 як багатовимірний OLAP-куб (Схема «Зірка»). Ця база даних не використовується API-сервером для поточної роботи. Вона наповнюється окремим ETL-процесом (Extract, Transform, Load), який за розкладом (наприклад, щоночі) бере дані з OLTP-бази, трансформує їх (агрегує, де-нормалізує) і завантажує в куб. Призначення DWH — виконання складних аналітичних запитів (наприклад, "показати середній employer_performance_rating для стратегії X за 3-й квартал"). Саме до цього сховища звертається Адміністратор через прецедент «Перегляд аналітичних звітів».

Загальна взаємодія трьох рівнів чітко ілюструє логіку роботи системи. Для операції запису (напр., створення нового кандидата Агентом) потік виглядає так: Клієнт (JSON) → API-Сервер (Валідація Pydantic) → DAL (SQLAlchemy ORM) → OLTP База Даних (INSERT).

Для операції підтримки прийняття рішень (запуск алгоритму підбору) потік значно складніший: Клієнт (GET) → API-Сервер (Маршрутизація) → MatchingService (Логіка) → DAL (SELECT) → OLTP База Даних (Повернення "сирих" даних) → MatchingService (Обчислення зважених балів в пам'яті) → API-Сервер (Серіалізація в JSON) → Клієнт (Відображення результату).

Така чітка трирівнева архітектура забезпечує масштабованість, надійність та можливість незалежної модифікації кожного компонента, що є критично важливим для успішної розробки та підтримки комплексного програмного забезпечення, яким є система підтримки прийняття рішень.

3.2. Розробка підсистеми зберігання даних (Реалізація OLTP-бази даних на PostgreSQL/SQLite).

Підсистема зберігання даних є фундаментом трирівневої архітектури, описаної у Підрозділі 3.1. Цей вузол виступає як рівень даних (Data Layer) і

слугує єдиним джерелом правдивої інформації для Прикладного рівня (API-сервера). У контексті нашої СППР, як було зазначено, цей рівень складається з двох логічно розділених систем: операційної (OLTP) та аналітичної (OLAP). Цей підрозділ присвячений розробці саме операційної бази даних (OLTP – Online Transaction Processing). Її головне завдання — ефективно, надійно та узгоджено обслуговувати велику кількість коротких транзакцій у реальному часі: запис нових кандидатів, оновлення статусів вакансій, читання профілів для алгоритму підбору.

При виборі технології (СУБД) для реалізації OLTP-сховища, ми керувалися вимогами до масштабованості, надійності та простоти інтеграції з обраним технологічним стеком (Python, FastAPI). Розглядалися два основні кандидати: SQLite та PostgreSQL.

SQLite є чудовим вибором для локальної розробки та прототипування. Це безсерверна, файлова СУБД, що не вимагає налаштування і ідеально підходить для швидкого тестування (наприклад, для main.py у FastAPI). Однак, вона має суттєві обмеження у продуктивності при високій кількості одночасних записів (concurrency), що є критичним для багатокористувацької системи, де декілька Агентів можуть одночасно працювати з базою.

Тому для промислової експлуатації та повноцінної реалізації системи була обрана PostgreSQL. Це потужна, об'єктно-реляційна СУБД з відкритим кодом, яка повністю задовольняє вимоги нашого проєкту. По-перше, вона забезпечує високу масштабованість та паралелізм (concurrency) завдяки реалізації MVCC (Multi-Version Concurrency Control), що дозволяє багатьом Агентам одночасно читати та писати дані без блокувань. По-друге, PostgreSQL відома своєю суворою відповідністю стандартам SQL та надійними механізмами транзакцій (ACID), що гарантує цілісність даних при складних операціях (наприклад, при створенні кандидата та одночасному записі його навичок). По-третє, завдяки використанню SQLAlchemy ORM на Прикладному рівні, перехід між SQLite (для розробки) та PostgreSQL (для розгортання) стає практично "безшовним", вимагаючи лише зміни рядка підключення.

На етапі моделювання (Розділ 2) було створено концептуальну та логічну модель даних (ER-діаграму). На поточному етапі розробки ми перетворюємо цю логічну модель на фізичну модель — тобто, набір конкретних SQL CREATE TABLE інструкцій, які визначають типи даних, індекси, обмеження цілісності та зв'язки між таблицями у середовищі PostgreSQL.

Варто зазначити, що завдяки використанню SQLAlchemy ORM (Object-Relational Mapper), ми визначаємо цю схему не "сирим" SQL-кодом, а через опис Python-класів (моделей). SQLAlchemy автоматично генерує та виконує необхідні

SQL-запити для створення цієї структури в базі даних. Це дозволяє тримати визначення схеми даних та бізнес-логіку в єдиному середовищі розробки.

Фізична реалізація бази даних слідує принципам нормалізації (до Третьої нормальної форми, 3NF). Це означає, що дані зберігаються без надлишковості, що унеможливлює аномалії оновлення (наприклад, при зміні назви компанії-роботодавця, нам потрібно оновити лише один рядок в таблиці Employers, а не 100 рядків у Job_Openings).

Структура OLTP-бази даних складається з трьох логічних груп таблиць: довідники (Dimensions), сутності (Entities) та зв'язки (Associations).

Довідкові таблиці є найпростішими та слугують для забезпечення консистентності даних. До них належать Roles (зберігає назви ролей: 'Admin', 'Agent') та Skills (зберігає уніфікований список усіх можливих навичок: 'Python', 'SQL', 'Лідерство'). Використання довідника Skills є критично важливим для нашого алгоритму, оскільки це гарантує, що "Python" і "Пітон" не будуть вважатися різними навичками, що могло б зруйнувати точність підбору.

Основні таблиці сутностей є ядром OLTP-системи: Users, Employers, Candidates та Job_Openings. Users зберігає облікові дані користувачів системи (email, password_hash) та пов'язана з Roles через зовнішній ключ role_id для визначення прав доступу. Таблиці Employers та Candidates зберігають основну інформацію про компанії та шукачів. Таблиця Job_Openings містить опис вакансій і є центральною для бізнес-логіки; вона пов'язана і з Employers (хто опублікував), і з Users (який Агент її веде).

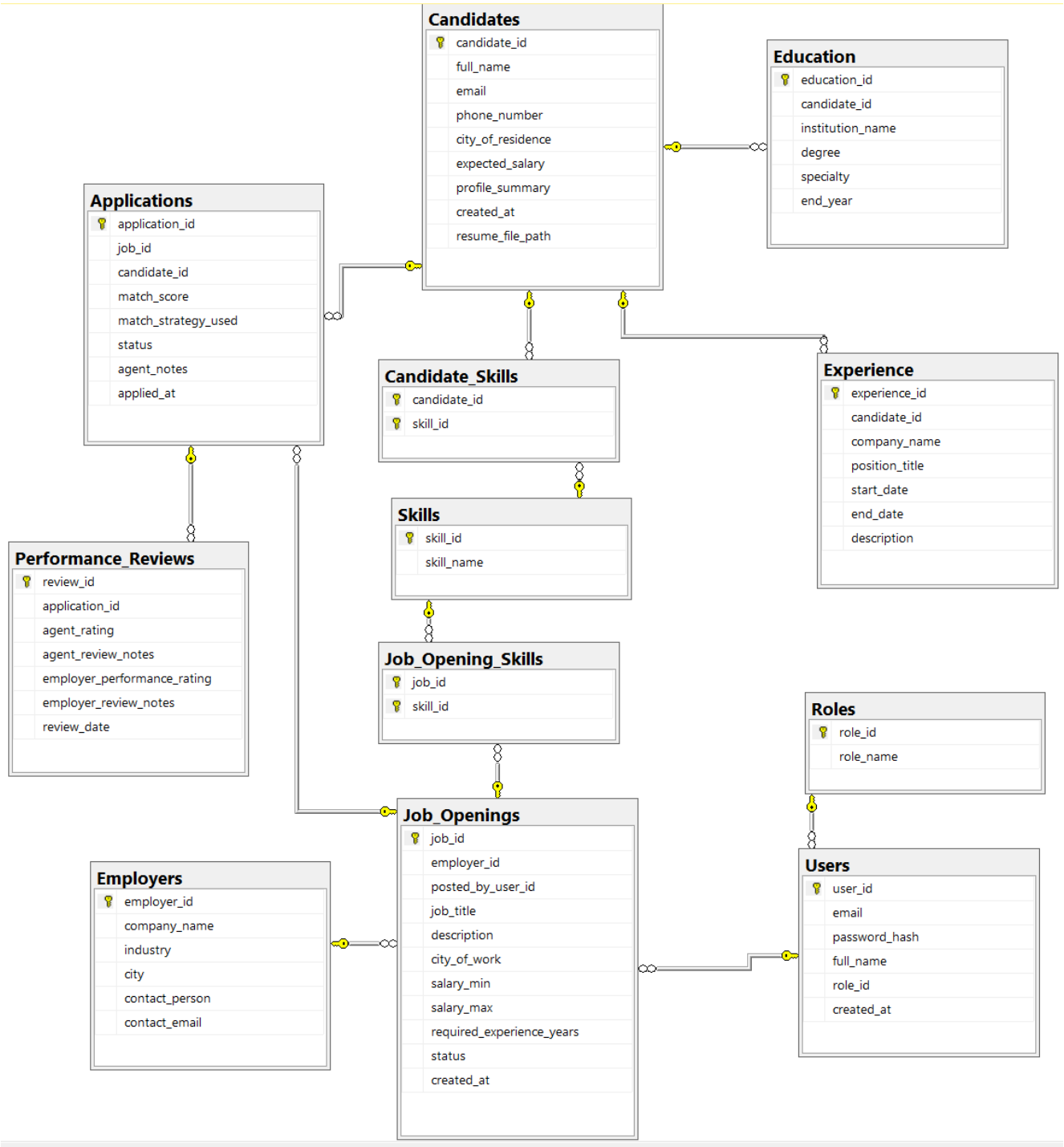
Реалізація складних зв'язків є найважливішою частиною проєктування OLTP-бази. Для реалізації відношення "один-до-багатьох" (One-to-Many), наприклад, "один кандидат може мати багато дипломів про освіту", ми використовуємо окремі таблиці Education та Experience. Кожен запис у цих таблицях містить зовнішній ключ candidate_id, що чітко пов'язує його з основним профілем кандидата.

Найскладнішим є відношення "багато-до-багатьох" (Many-to-Many) для навичок. Один кандидат може мати багато навичок, і одна навичка може бути у багатьох кандидатів. Те ж саме стосується і вакансій. Пряма реалізація (наприклад, через масив skill_ids у таблиці Candidates) є неефективною, немасштабованою та порушує принципи нормалізації. Тому ця проблема вирішується через асоціативні таблиці (Junction Tables). Ми створюємо дві такі таблиці: Candidate_Skills (з полями candidate_id, skill_id) та Job_Opening_Skills (з полями job_id, skill_id). Така нормалізована структура дозволяє Прикладному рівню (API-серверу) виконувати надзвичайно ефективні та складні запити,

наприклад: "Знайти всіх кандидатів (Candidates), які мають щонайменше 3 навички (Skills) зі списку, що вимагає вакансія (Job_Opening_Skills)".

Оскільки дані з цієї OLTP-бази є "паливом" для нашого зваженого алгоритму, їхня цілісність та якість (принцип "Garbage In, Garbage Out") є найвищим пріоритетом. Фізична модель бази даних забезпечує цю цілісність на рівні СУБД за допомогою обмежень (Constraints).

Первинні ключі (Primary Key), такі як candidate_id чи job_id (зазвичай типу INT або IDENTITY в PostgreSQL), гарантують унікальну ідентифікацію кожного запису. Обмеження унікальності (Unique Constraint), наприклад, на полі Candidates.email або Skills.skill_name, запобігають створенню дублікатів. Обмеження NOT NULL на ключових полях (напр., Candidates.full_name) гарантують повноту даних.



Діаграма 3.1 Структура бази даних

Найважливішу роль відіграють Зовнішні ключі (Foreign Key). Вони підтримують посилальну цілісність (referential integrity) між таблицями. Наприклад, обмеження FOREIGN KEY на полі Applications.candidate_id фізично унеможливлює створення запису про "метч" для кандидата, який не існує в системі. Більшість цих зв'язків також налаштовані з правилом ON DELETE CASCADE, що забезпечує автоматичне очищення пов'язаних даних (наприклад,

при видаленні профілю кандидата, автоматично видаляються всі записи про його освіту, досвід та навички), підтримуючи базу даних у консистентному стані.

Нарешті, сам Прикладний рівень (FastAPI) взаємодіє з цією базою виключно через транзакції. Наприклад, процес створення нового кандидата (описаний у Потоці 1 в Підрозділі 2.4) об'єднує запис у Candidates та множинні записи у Candidate_Skills в одну атомарну операцію BEGIN...COMMIT. Завдяки механізмам транзакцій PostgreSQL, якщо під час запису навичок станеться помилка, вся операція буде відкочена (ROLLBACK), гарантуючи, що в базі не з'явиться "напів-створений" кандидат без навичок.

Таким чином, розроблена OLTP-підсистема є надійною, масштабованою та нормалізованою структурою, що слугує одночасно і як операційний бекенд для API-сервісу, і як чисте, верифіковане джерело даних для майбутньої аналітичної підсистеми (OLAP).

3.3. Розробка підсистеми бізнес-логіки (Програмна реалізація зваженого алгоритму та патерну «Стратегія»).

У контексті трирівневої архітектури, описаної в Підрозділі 3.1, підсистема бізнес-логіки (Business Logic Layer) є центральним вузлом (Application Layer), що відповідає за виконання всіх обчислень, реалізацію бізнес-правил та прийняття рішень. Вона виступає як інтелектуальний посередник: з одного боку, вона отримує запити від підсистеми API (3.4), а з іншого — запитує "сирі" дані у підсистеми зберігання (3.2). Ключовою архітектурною вимогою до цього шару є його повна ізоляція та незалежність (loose coupling). Шар API не повинен знати, як розраховується бал релевантності, а шар даних не повинен знати, навіщо ці дані запитуються. Вся логіка інкапсульована саме тут.

У реалізованій СППР ця підсистема представлена сервісним шаром, ключовим компонентом якого є MatchingService (Сервіс Підбору). Цей сервіс є програмною реалізацією прецедентів «Виконання автоматизованого підбору» та «Розрахувати Бал Релевантності», що були змодельовані у Розділі 2. Він отримує на вхід ідентифікатор вакансії (job_id), а на виході повертає відранжований список кандидатів. Вся складність того, як цей список формується, прихована всередині цієї підсистеми.

Як було встановлено в Розділі 1, просте співставлення ключових слів (як у традиційних job-порталах) є неефективним для якісного підбору. Проблема полягає в тому, що різні критерії мають різну важливість. Для вакансії "Senior Developer" навичка "Python" (90% ваги) незрівнянно важливіша за наявність диплому (10% ваги), тоді як для позиції "Junior Trainee" ситуація може бути

протилежною. Це класична задача багатокритеріального прийняття рішень (MCDM – Multi-Criteria Decision Making).

Для вирішення цієї задачі в якості основного алгоритму було обрано модель зваженого оцінювання (Weighted Scoring Model). Цей підхід, методологічні паралелі якого були проаналізовані в Підрозділі 1.4 на прикладі патенту US20130166459A1, дозволяє формалізувати суб'єктивну експертизу Агента у чітку математичну формулу. Загальний «Бал Релевантності» (MatchScore) для кандидата розраховується як сума добутків його часткових оцінок (Score_i) на відповідні вагові коефіцієнти (Weight_i), визначені для даної вакансії:

$$MatchScore = \sum_{i=1}^{n} (Score\_i * Weight\_i$$

де n — кількість критеріїв (напр., навички, досвід, освіта, зарплата), а сума всіх Weight_i дорівнює 1 (або 100%). Цей підхід є прозорим, легко інтерпретованим та, що найголовніше, гнучким, оскільки дозволяє змінювати результат підбору, просто коригуючи вагові коефіцієнти.

Проте, наявність зваженого алгоритму створює нову архітектурну проблему. Як було доведено в Розділі 1.5, не існує єдиної "ідеальної" формули зважування; вона залежить від типу вакансії (IT, Sales, Junior, Senior). Найпростіше рішення — жорстко "зашити" цю логіку в MatchingService через конструкцію if-elif-else:Python

```
class MatchingService:
    def calculate_score(self, candidate, job):
        if job.category == 'IT':
            score = (candidate.skill_score * 0.7) + (candidate.exp_score * 0.3)
        elif job.category == 'Sales':
            score = (candidate.skill_score * 0.4) + (candidate.exp_score * 0.6)
        # ... і так далі
```

Такий підхід є вкрай негнучким, порушує Принцип Відкритості/Закритості (Open/Closed Principle) та перетворює MatchingService на компонент, який неможливо підтримувати чи тестувати. Кожна нова стратегія (напр., "Finance Strategy") вимагатиме модифікації основного коду сервісу. Для вирішення цієї проблеми підсистема бізнес-логіки спроєктована з використанням класичного поведінкового патерну проєктування «Стратегія» (Strategy Pattern). Цей патерн, як визначено "Бандою Чотирьох" (GoF), "визначає сімейство алгоритмів,

інкапсулює кожен з них та робить їх взаємозамінними". Це дозволяє алгоритму змінюватися незалежно від клієнтів, які його використовують.

Програмна реалізація патерну в середовищі Python складається з трьох ключових компонентів, що відповідають моделям, описаним у Розділі 2:

Інтерфейс Стратегії (Abstract Base Class): Ми визначаємо абстрактний базовий клас IMatchingStrategy, який вимагає від усіх реалізацій наявності єдиного методу calculate_match_score(). Цей метод приймає стандартизовані дані (наприклад, профіль кандидата та профіль вакансії) і повертає одне число — match_score.

Конкретні Стратегії (Concrete Classes): Кожна унікальна логіка зважування реалізується як окремий, незалежний клас, що успадковує IMatchingStrategy. Наприклад, WeightedITStrategy та BalancedSalesStrategy. Саме всередині цих класів "зашиті" конкретні вагові коефіцієнти та логіка розрахунку часткових балів. Наприклад, WeightedITStrategy може надавати 70% ваги збігу технічних навичок, тоді як BalancedSalesStrategy може надавати 50% ваги досвіду роботи та 30% — наявності "soft-skills" (напр., "Ведення переговорів").

Контекст (The Context): Клас MatchingService виступає в ролі "Контексту". Він містить посилання на об'єкт Стратегії (наприклад, self.current_strategy: IMatchingStrategy). Коли API-сервер викликає MatchingService, він також передає інформацію про те, яку стратегію використовувати (наприклад, на основі параметра запиту ?strategy=IT або налаштування за замовчуванням). MatchingService не знає, як відбувається розрахунок; він лише викликає метод self.current_strategy.calculate_match_score().

Розглянемо детальніше, що відбувається всередині методу calculate_match_score() однієї з конкретних стратегій (наприклад, WeightedITStrategy з вагами: Skills=0.6, Experience=0.3, Salary=0.1). Процес не є тривіальним і сам складається з кількох кроків для розрахунку часткових балів (Score_i).По-перше, розраховується skill_score. Алгоритм отримує два списки ID навичок: candidate_skills та job_skills (які :MatchingService попередньо отримав від :Database (3.2)). Найпростіший розрахунок — це відсоток збігу: (кількість спільних навичок / кількість необхідних навичок). Більш складний (і ефективний) варіант, що використовується в системі, — це коефіцієнт Жаккара, який враховує загальну множину: $|A \cap B| / |A \cup B|$. Результат (напр., 0.8) є частковим балом skill_score.

По-друге, розраховується experience_score. Алгоритм порівнює job.required_experience (напр., 5 років) з candidate.total_experience (напр., 3 роки). Цей розрахунок має бути нелінійним. Якщо кандидат має 3 з 5 років, його бал може бути 3/5 = 0.6. Якщо кандидат має 5 і більше років (5, 8, 10), його бал має

бути максимальним — 1.0, оскільки подальше збільшення досвіду не завжди означає кращу відповідність.

По-третє, розраховується salary_score. Цей критерій часто є бінарним або штрафним. Якщо candidate.expected_salary менше або дорівнює job.salary_max, кандидат отримує 1.0. Якщо його очікування вищі, він може або отримати 0.0 (жорстка фільтрація), або отримати штрафний бал, що пропорційний відсотку перевищення.

Нарешті, відбувається фінальний розрахунок MatchScore. Алгоритм множить отримані часткові бали на вагові коефіцієнти, визначені в цій конкретній стратегії: FinalScore = (0.8 * 0.6) + (0.6 * 0.3) + (1.0 * 0.1) = 0.48 + 0.18 + 0.1 = 0.76. Цей фінальний бал (76.0) і є результатом, який повертає підсистема бізнес-логіки.

Таким чином, розроблена підсистема є гнучкою, масштабованою та повністю ізольованою. Вона дозволяє Адміністратору системи (як це було визначено в прецедентах) додавати нові класи Стратегій (наприклад, MLPredictionStrategy у майбутньому) без необхідності зупиняти чи модифікувати ядро MatchingService, що є ключовою ознакою якісно спроєктованої інформаційної системи.

3.4. Розробка підсистеми API на FastAPI (Опис реалізації ключових ендпоінтів).

Підсистема API (Application Programming Interface) є центральним комунікаційним вузлом у розробленій трирівневій архітектурі. Вона виступає як Прикладний рівень (Application Layer), що фізично реалізує "вхідні ворота" до всієї бізнес-логіки системи. Ця підсистема отримує та інтерпретує HTTP-запити від Клієнтського рівня (браузерів Адміністратора та Агента), оркеструє взаємодію між підсистемою бізнес-логіки (3.3) та підсистемою зберігання даних (3.2), і повертає Клієнтському рівню відповіді у стандартизованому форматі JSON. Як було обґрунтовано в Розділі 1.5, для реалізації цього вузла було обрано Python-фреймворк FastAPI.

Вибір FastAPI обумовлений його ключовими архітектурними перевагами, що ідеально відповідають задачам нашої СППР. По-перше, це висока продуктивність. FastAPI побудований на асинхронній інфраструктурі ASGI (Asynchronous Server Gateway Interface), що дозволяє йому обробляти велику кількість одночасних підключень (наприклад, від багатьох Агентів) неблокуючим способом. Це критично важливо для обчислювально інтенсивних запитів, таких як «Виконання автоматизованого підбору».

По-друге, це вбудована валідація даних за допомогою Pydantic. У системі, де цілісність даних є фундаментом для коректної роботи алгоритмів, Pydantic виступає як надійний "охоронець" на вході в API. Він автоматично парсить JSON-запити, приводить типи даних (наприклад, str в int) та валідує їх згідно з визначеними Python-класами (схемами), миттєво повертаючи клієнту чіткі помилки 422 Unprocessable Entity у разі невідповідності. Це унеможливлює потрапляння "брудних" даних у підсистему бізнес-логіки чи базу даних.

По-третє, це потужний механізм Впровадження Залежностей (Dependency Injection). FastAPI дозволяє "впроваджувати" необхідні сервіси (наприклад, сесію бази даних get_db або наш get_matching_service) безпосередньо у функції-обробники ендпоінтів. Це повністю реалізує принцип інверсії контролю (IoC), роблячи ендпоінти незалежними від конкретних реалізацій сервісів та значно спрощуючи їх тестування (наприклад, шляхом підміни реального MatchingService на тестовий "мок").

Більша частина функціоналу СППР, доступного Агенту, є CRUD-операціями (Create, Read, Update, Delete) для керування основними сутностями (Candidates, Employers, Job_Openings). API-сервер надає стандартизований набір RESTful-ендпоінтів для цих завдань. Розглянемо їх реалізацію на прикладі сутності "Кандидат".

POST /api/candidates (Створення Кандидата): Цей ендпоінт реалізує прецедент «Керування базою кандидатів». Він очікує POST-запит, що містить JSON-об'єкт з даними нового кандидата в тілі (body). Функція-обробник цього ендпоінту в якості одного з параметрів приймає Pydantic-модель CandidateCreateSchema. FastAPI автоматично валідує тіло запиту відповідно до цієї схеми. Якщо валідація успішна, обробник викликає відповідний метод шару доступу до даних (наприклад, repository.create_candidate(data)), який, у свою чергу, виконує INSERT-запит до OLTP-бази даних (3.2) через SQLAlchemy. Після успішного створення, ендпоінт повертає клієнту статус HTTP 201 Created та JSON-об'єкт створеного кандидата, серіалізований за допомогою CandidateReadSchema (яка, наприклад, включає candidate_id, але виключає password_hash, якщо б він був).

GET /api/candidates/{candidate_id} (Читання Кандидата): Цей ендпоінт дозволяє отримати повний профіль одного кандидата. Він приймає candidate_id як параметр шляху. FastAPI автоматично валідує, що цей параметр є цілим числом. Обробник викликає метод repository.get_candidate(candidate_id). Важливою частиною реалізації є обробка виключних ситуацій: якщо репозиторій повертає None (кандидат не знайдений), ендпоінт перехоплює це і повертає клієнту коректну відповідь HTTP 404 Not Found. Якщо кандидат знайдений, він серіалізується CandidateReadSchema і повертається зі статусом 200 OK.

Аналогічні GET-ендпоінти існують для отримання списків (/api/candidates), а також для Employers та Job_Openings.

PUT /api/candidates/{candidate_id} (Оновлення Кандидата): Цей ендпоінт дозволяє Агенту оновлювати профіль кандидата (наприклад, додати нові навички чи змінити очікувану ЗП). Він приймає candidate_id у шляху та JSON-об'єкт з оновленими даними (CandidateUpdateSchema) у тілі. Логіка поєднує попередні два: спочатку викликається repository.get_candidate(), щоб переконатися, що об'єкт існує (обробка 404), а потім викликається repository.update_candidate() для внесення змін у базу даних.

Найважливішим ендпоінтом, що реалізує інтелектуальну складову системи, є ендпоінт запуску підбору. Він є програмною реалізацією Діаграми Послідовності, описаної в Підрозділі 2.4.

GET /api/jobs/{job_id}/matches (Запуск підбору): Цей ендпоінт не є CRUD-операцією. Він є точкою входу до підсистеми бізнес-логіки (3.3). Його обробник приймає job_id як параметр шляху. За допомогою механізму Впровадження Залежностей FastAPI, обробник також отримує готовий до роботи екземпляр :MatchingService (наприклад, service: MatchingService = Depends(get_matching_service)).

Обробник цього ендпоінту також може приймати query-параметри для уточнення запиту, наприклад, ?strategy=WeightedIT або ?limit=10. Це дозволяє Агенту (через клієнтський інтерфейс) динамічно впливати на те, який алгоритм буде застосовано, що реалізує гнучкість, закладену патерном «Стратегія».

Вся робота ендпоінту зводиться до однієї логічної операції: виклику методу сервісу, наприклад, ranked_list = service.find_matches(job_id=job_id, strategy_name=strategy). Обробник API не має жодного уявлення про те, як відбувається розрахунок match_score — він лише знає, що має отримати у відповідь список кандидатів. Ця повна інкапсуляція логіки є ключовою перевагою обраної архітектури. Після отримання ranked_list (який може бути, наприклад, List[CandidateMatchSchema]), ендпоінт серіалізує його в JSON і повертає Агенту зі статусом 200 OK.

Другий за важливістю не-CRUD ендпоінт — це той, що забезпечує збір даних для аналітичного контуру (OLAP).

POST /api/reviews (Збір зворотного зв'язку): Цей ендпоінт реалізує прецедент «Надання зворотного зв'язку». Він очікує POST-запит, що містить PerformanceReviewSchema у тілі JSON. Ця схема містить ключові поля для майбутнього аналізу: application_id (для зв'язку з конкретним наймом), agent_rating (оцінка якості рекомендації) та employer_performance_rating (оцінка успішності кандидата).

Обробник FastAPI валідує ці дані (Pydantic) і передає їх у шар доступу до даних (наприклад, repository.create_review(review_data)), який зберігає їх у таблиці Performance_Reviews в OLTP-базі. Цей ендпоінт є критично важливою "точкою збору" даних. Він забезпечує наповнення таблиці, яка згодом стане головним джерелом (Fact source) для ETL-процесу при побудові та оновленні нашого аналітичного OLAP-куба. Таким чином, підсистема API не лише обслуговує поточні операції, але й активно бере участь у зборі даних для майбутнього вдосконалення самої системи.

3.5. Архітектурна реалізація MCDM-підходу на основі аналізу патенту «Invention valuation and scoring system»

Попередні підрозділи цього розділу (3.1 - 3.4) детально описали розробку окремих, фізичних вузлів (підсистем) нашої СППР: загальної трирівневої архітектури, операційної OLTP-бази даних (3.2), алгоритмічного ядра (3.3) та API-сервера (3.4). Кожен з цих компонентів виконує свою чітко визначену технічну функцію. Однак, щоб система функціонувала як єдине ціле, ці компоненти мають бути об'єднані спільною дизайн-філософією. Цей підрозділ має на меті синтезувати проведену розробку, продемонструвавши, як фундаментальна теоретична концепція, проаналізована в Розділі 1, була практично реалізована та втілена у кожній з розроблених підсистем.

Як було встановлено в Підрозділі 1.4, ключовою теоретичною базою для нашої роботи є патент US20130166459A1 («Invention valuation and scoring system»). Цей патент вирішує проблему, ідентичну до нашої: як взяти складний, суб'єктивний та багатогранний об'єкт ("винахід" або "кандидат") і надати йому об'єктивну, кількісну оцінку. Рішення, запропоноване в патенті, є класичним багатокритеріальним аналізом рішень (MCDM), який декомпозує проблему на три окремі програмні модулі: модуль визначення ваг (DSWM), модуль розрахунку часткових оцінок (DMRM) та модуль фінального скорингу (CSSM). Наша система є прямою архітектурною адаптацією та вдосконаленням цього запатентованого підходу.

Розглянемо детально, як кожен логічний модуль, описаний в патенті IVSS, був реалізований у нашій програмній архітектурі, і в чому полягає наше вдосконалення.

1. Модуль Визначення Ваг (DSWM): У патенті DSWM (Delivery Sustainability Weight Module) відповідає за створення "Матриці Ваг", тобто за визначення відносної важливості кожного критерію (напр., "Довговічність" = 40%, "Стійкість ядра" = 60%). Це статичне визначення бізнес-правил.

У нашій системі ця концепція реалізована не як статичний модуль, а як гнучкий поведінковий патерн «Стратегія», детально описаний у Підрозділі 3.3. Це є нашим ключовим вдосконаленням. Замість єдиної "Матриці Ваг", ми маємо сімейство взаємозамінних алгоритмів (напр., WeightedITStrategy, BalancedSalesStrategy, JuniorEducationStrategy). Кожен з цих класів є, по суті, окремим DSWM, інкапсулюючи власний унікальний набір вагових коефіцієнтів. Адміністратор системи (через прецедент, описаний в 2.2) може динамічно створювати та конфігурувати ці стратегії, а Агент — обирати потрібну під час виконання запиту через ендпоінт API (3.4). Це перетворює статичну модель патенту на гнучку, адаптивну систему.

2. Модуль Розрахунку Оцінок (DMRM): У патенті DMRM (Dimension Maturity Rule Module) відповідає за розрахунок часткових балів за кожним виміром (напр., "Відповідність бізнес-стратегії" = 'high').

У нашій реалізації ця логіка "живе" всередині кожного конкретного класу Стратегії. Як було описано в 3.3.5, наш метод calculate_match_score() перед фінальним зважуванням викликає низку внутрішніх, приватних методів, таких як _calculate_skill_score() або _calculate_experience_score(). Кожен з цих методів є прямим аналогом DMRM. Вони містять бізнес-правила (напр., використання коефіцієнта Жаккара для навичок або нелінійної функції для досвіду) для перетворення "сирих" даних з бази (3.2) у нормалізовану оцінку (напр., 0.8 або 1.0).

3. Модуль Фінального Скорингу (CSSM): У патенті CSSM (Capability Sustainability Scoring Module) виконує фінальну математичну операцію — множить часткові бали (DMRM) на вагові коефіцієнти (DSWM) і підсумовує їх.

У нашій реалізації це останні рядки коду в методі calculate_match_score() кожної Стратегії. Це пряма реалізація формули зваженої суми: FinalScore = (skill_score * self.SKILL_WEIGHT) + (exp_score * self.EXP_WEIGHT) ....

Сама по собі ця логіка, інкапсульована в класах Стратегій, є нефункціональною без її інтеграції в загальну трирівневу архітектуру. Цей підрозділ завершує Розділ 3, показуючи, як розроблені нами підсистеми (3.1, 3.2, 3.4) працюють разом, щоб "оживити" цей MCDM-алгоритм.

- Крок 1: Ініціація (API-Рівень). Процес запускається ззовні, коли Агент ініціює GET запит до ендпоінту GET /api/jobs/{job_id}/matches, розробленого в Підрозділі 3.4.
- Крок 2: Делегування (API-Рівень → Сервісний Рівень). Обробник ендпоінту FastAPI не виконує жодних обчислень. Він негайно делегує завдання, викликаючи метод find_matches() у MatchingService, який він отримує через механізм Dependency Injection.

- Крок 3: Збір даних (Сервісний Рівень → Рівень Даних). MatchingService (контекст патерну «Стратегія») спершу звертається до OLTP-бази даних (3.2) через шар SQLAlchemy. Він виконує серію SELECT-запитів для отримання "сирих" даних: профілю вакансії, її вимог до навичок, а також профілів та навичок усіх активних кандидатів.
- Крок 4: Виконання (Сервісний Рівень). MatchingService ініціює цикл. Для кожного кандидата він обирає необхідний об'єкт-стратегію (напр., WeightedITStrategy на основі query-параметра з API-запиту) і передає йому дані кандидата та вакансії.
- Крок 5: Обчислення MCDM (Сервісний Рівень). Обрана Стратегія виконує повний цикл MCDM-аналізу, описаний у 3.5.2:

Викликає свої внутрішні методи (DMRM) для розрахунку часткових балів (напр., skill_score = 0.8).

Використовує свої внутрішні константи (DSWM) для визначення ваг (напр., SKILL_WEIGHT = 0.7).

Виконує фінальний розрахунок (CSSM) і повертає єдине число (напр., 0.76) назад у MatchingService.

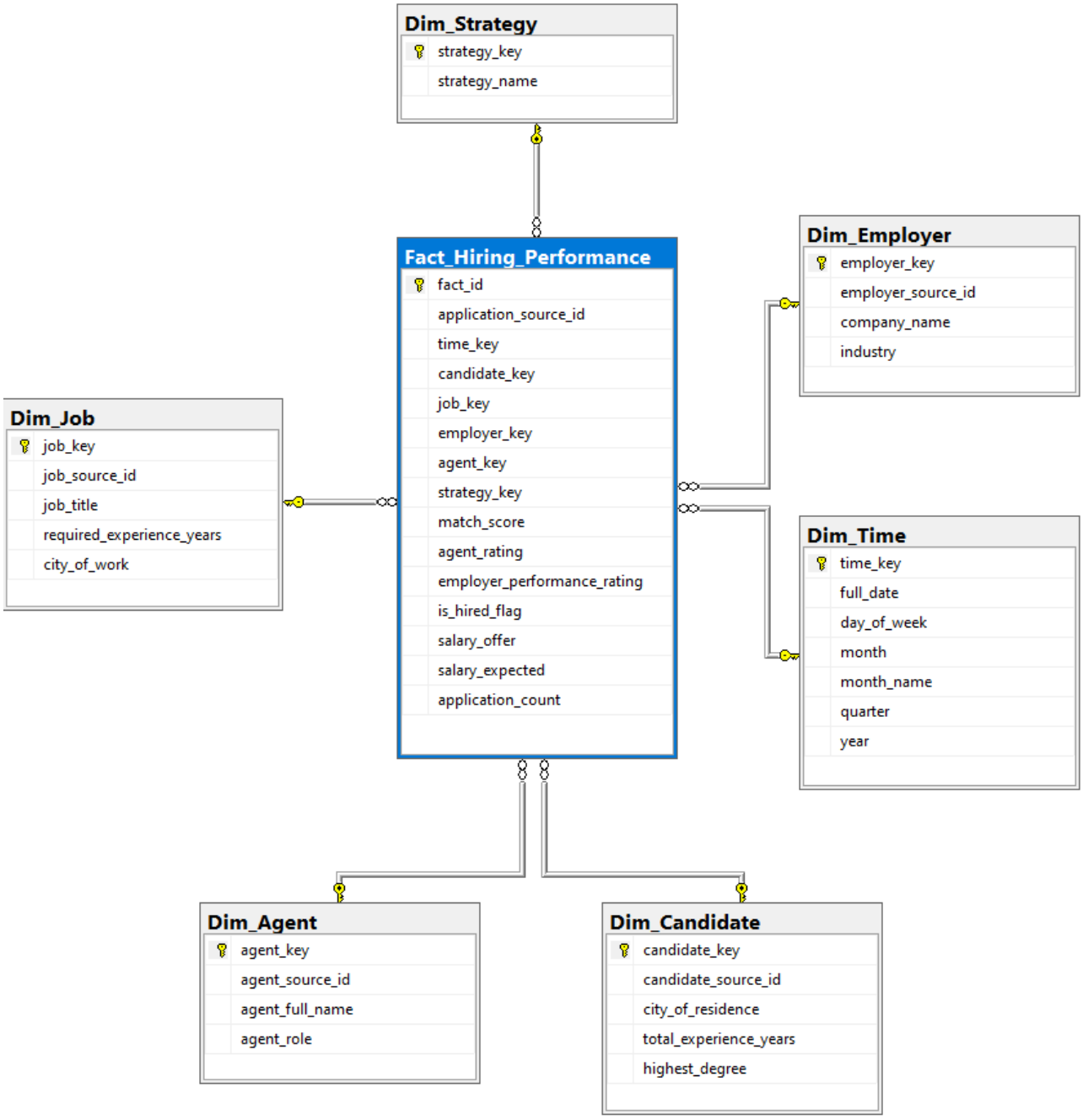
- Крок 6: Агрегація та Повернення (Сервісний Рівень → API-Рівень). MatchingService збирає ці бали, сортує кандидатів і повертає відранжований список на рівень API.
- Крок 7: Відповідь (API-Рівень → Клієнт). API-сервер серіалізує цей список у JSON і відправляє HTTP 200 OK Агенту.

Таким чином, ми бачимо, що реалізація теоретичної MCDM-моделі, запозиченої з патентного аналізу, пронизує кожен розроблений нами компонент системи. Це не просто окрема функція, а центральна дизайн-філософія, що об'єднує базу даних (джерело критеріїв), API-сервер (точка входу) та сервісний шар (обчислювальне ядро) в єдину, логічно завершену Систему Підтримки Прийняття Рішень.

3.6. Розробка Аналітичної Підсистеми (Сховища Даних OLAP)

У Підрозділі 3.2 ми детально розглянули розробку операційної бази даних (OLTP). Ця система ідеально оптимізована для своєї задачі: швидкого запису та модифікації транзакційних даних (додавання кандидатів, оновлення вакансій). Однак, використання цієї ж бази даних для складного бізнес-аналізу, як того вимагає наша СППР, є вкрай неефективним та ризикованим. Нормалізована структура (3NF) з великою кількістю таблиць (Candidates, Skills, Education,

Experience тощо) вимагає виконання складних JOIN-запитів для отримання навіть простого аналітичного звіту. Такі запити створюють високе навантаження на "живу" базу, сповільнюючи роботу Агентів, і можуть призводити до блокувань таблиць.



Діаграма 3.2 – Сховище даних

Для вирішення цієї проблеми, архітектура нашої системи (описана в 3.1.4) передбачає розробку другої, окремої підсистеми зберігання — Аналітичного Сховища Даних (Data Warehouse, DWH). На відміну від OLTP, це сховище

оптимізоване не для запису, а для надшвидкого читання та агрегації великих обсягів історичних даних. Воно використовує технологію OLAP (Online Analytical Processing), яка дозволяє Адміністратору миттєво отримувати відповіді на складні аналітичні запити (наприклад, "який середній рейтинг успішності кандидатів, найнятих за стратегією 'IT' у другому кварталі, у розрізі галузей роботодавців?").

Для фізичної реалізації нашого DWH було обрано класичну та найбільш поширену модель — Схему «Зірка» (Star Schema). Ця архітектура ідеально підходить для реалізації багатовимірного OLAP-куба в реляційній СУБД (наприклад, PostgreSQL або спеціалізованій аналітичній СУБД). Схема «Зірка» є де-нормалізованою і складається з двох типів таблиць, як показано на Діаграмі 3.2.

Таблиця Фактів (Fact Table): Одна, велика, центральна таблиця (Fact_Hiring_Performance), що містить числові показники (міри), які ми хочемо аналізувати.

Таблиці Вимірів (Dimension Tables): Декілька менших таблиць (Dim_Candidate, Dim_Job тощо), які оточують таблицю фактів. Вони містять текстові, описові атрибути, за якими ми будемо "розрізати" (slice) та "групувати" (dice) наші дані.

Перевага такої структури полягає у її простоті та продуктивності. Аналітичні запити не вимагають складних JOIN-операцій між десятками нормалізованих таблиць, як в OLTP. Замість цього, запит зазвичай об'єднує лише центральну таблицю фактів з однією або двома необхідними таблицями вимірів, що забезпечує надзвичайну швидкість агрегації.

Центральним елементом нашої OLAP-системи є таблиця фактів Fact_Hiring_Performance. "Зерно" (grain) цієї таблиці визначено як один запис на одну подію «Заявка/Підбір» (Application). Це означає, що кожного разу, коли в OLTP-системі створюється запис у таблиці Applications і згодом для нього з'являється відгук у Performance_Reviews, в таблицю фактів (під час ETL-процесу) додається один відповідний рядок.

Як видно з Діаграми 3.2, ця таблиця містить дві категорії стовпців. Перша — це набір зовнішніх ключів (time_key, candidate_key, job_key, employer_key, agent_key, strategy_key), які пов'язують кожен факт з відповідними записами у таблицях вимірів. Друга, і найважливіша, категорія — це міри (Measures). Це числові показники, які ми будемо агрегувати (підсумовувати, усереднювати, рахувати):

- match_score: Числовий бал релевантності, згенерований нашим алгоритмом (3.3).

- agent_rating: Оцінка якості рекомендації (1-5), надана Агентом.
- employer_performance_rating: Ключова міра — оцінка успішності кандидата (1-5) від роботодавця.
- is_hired_flag: Прапорець (1 або 0), що дозволяє підраховувати коефіцієнт конверсії (hire rate).
- salary_offer та salary_expected: Числові значення для аналізу зарплатних розбіжностей.
- application_count: Показник (завжди 1), що використовується для підрахунку загальної кількості заявок.

Таблиці вимірів надають контекст для мір з таблиці фактів. Вони є нашими аналітичними "розрізами".

1. Dim_Agent: Вимір "Хто [від агентства]". Він містить інформацію про Агента, який вів вакансію. Це дозволяє аналізувати ефективність у розрізі окремих співробітників чи їхніх ролей (agent_role).

2. Dim_Candidate: Вимір "Хто [кандидат]". Важливо зазначити, що ця таблиця є де-нормалізованою. Замість окремих таблиць Education та Experience (як в OLTP), вона містить вже трансформовані дані, готові для аналізу, наприклад, total_experience_years (загальна кількість років досвіду) та highest_degree (найвищий отриманий ступінь).

3. Dim_Employer: Вимір "Де [працює]". Містить інформацію про компанію-роботодавця, дозволяючи фільтрувати дані за назвою або, що більш важливо, за industry (галуззю).

4. Dim_Job: Вимір "Що [за вакансія]". Описує характеристики вакансії, на яку наймали кандидата, наприклад, job_title або required_experience_years.

5. Dim_Strategy: Вимір "Як [відбувся підбір]". Це ключовий вимір для нашого дослідження. Він ізолює назву використаного алгоритму (напр., 'WeightedIT', 'BalancedSales'), що дозволяє напряму порівнювати їх ефективність.

6. Dim_Time: Вимір "Коли". Це стандартна для DWH таблиця, що розбиває дату на аналітично-зручні компоненти: year, quarter, month тощо.

Розробка самого сховища (SQL-таблиць) є лише частиною завдання. Щоб це сховище було корисним, його необхідно наповнювати даними. Цю функцію виконує процес ETL (Extract, Transform, Load), який є окремою програмною

підсистемою (зазвичай, це набір Python-скриптів, що запускаються за розкладом).

Extract (Видобування): Скрипт за розкладом (напр., щоночі) підключається до OLTP-бази даних (3.2) і витягує всі нові або оновлені записи з таблиць Applications, Performance_Reviews, Candidates, Jobs тощо, які з'явилися з моменту останнього запуску.

Transform (Трансформація): Це найскладніший етап. "Сирі" OLTP-дані очищуються та перетворюються. Наприклад, скрипт об'єднує дані з Candidates, Education та Experience для створення єдиного, де-нормалізованого запису для Dim_Candidate. Він шукає відповідні ключі (lookups) — наприклад, перетворює текстове strategy_name='WeightedIT' на числовий strategy_key=1.

Load (Завантаження): Очищені та трансформовані дані завантажуються у Сховище Даних (DWH). Нові записи додаються до Dim_ таблиць (якщо їх ще немає) та до Fact_Hiring_Performance.

Завершення розробки цієї підсистеми DWH забезпечує систему готовим аналітичним інструментарієм, який буде використано в Розділі 4 для проведення дослідження та візуалізації результатів ефективності зважених стратегій.

3. РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Цей розділ є кульмінацією магістерської роботи. Попередні розділи були присвячені аналізу (Розділ 1), проєктуванню (Розділ 2) та розробці (Розділ 3) інструментального програмного комплексу — Системи Підтримки Прийняття Рішень (СППР). Як було зазначено в постановці завдання (1.5), головною метою було не стільки створення комерційного продукту, скільки розробка інструменту для проведення дослідження.

Дослідження полягає у перевірці гіпотези про те, що ефективність алгоритму підбору кандидатів (MatchScore) напряму залежить від коректного вибору вагових коефіцієнтів критеріїв (Weights), і що ці коефіцієнти не є статичними, а мають змінюватися залежно від контексту вакансії. Даний розділ представляє апаратно-програмне середовище, в якому проводилось дослідження, детально аналізує ефективність обраного MCDM-алгоритму та представляє результати OLAP-аналізу, що підтверджують висунуту гіпотезу.

4.1. Апаратно-програмне середовище та хід виконання дослідження

Для реалізації спроєктованої у Розділі 3 системи були визначені апаратні та програмні вимоги. З огляду на те, що система є веб-сервісом (API) з потенційно високим навантаженням на обчислення (алгоритм підбору) та взаємодію з базою даних, серверна частина вимагає стандартного серверного оточення.

Апаратне забезпечення (Сервер): Для розгортання рекомендується віртуальний (VPS) або фізичний сервер під керуванням ОС на ядрі Linux (напр., Ubuntu Server 20.04 LTS). Мінімальні вимоги до ресурсів для обробки середнього навантаження (до 1000 одночасних запитів) оцінюються у 2 vCPU, 4 ГБ RAM та SSD-накопичувач для бази даних PostgreSQL. Для клієнтської частини (Агент, Адміністратор) достатньо стандартного ПК з сучасним веб-браузером.

Програмне забезпечення: Стек технологій було обрано в Розділі 1.5 з пріоритетом на швидкість розробки, надійність та сумісність з інструментами аналізу даних.

Обґрунтування вибору Python як інструменту дослідження

Центральним елементом програмного стеку є мова Python (версії 3.9+). Цей вибір був свідомим та ключовим для успіху дослідження. Як було показано в таблиці порівняння (1.5), у той час як C#/.NET є потужною платформою для

enterprise-розробки, а C++ — для високопродуктивних обчислень, Python пропонує унікальне поєднання переваг, що ідеально підходило для наших завдань.

По-перше, це швидкість розробки інструментарію. Головною метою було не написання коду, а аналіз даних. Використання фреймворку FastAPI (Підрозділ 3.4) дозволило створити повноцінний, асинхронний, валідований API-сервер у десятки разів швидше, ніж це було б можливо на C# чи C++. Це дозволило швидко створити "лабораторний стенд" (нашу СППР) для перевірки гіпотез.

По-друге, і це найголовніше, Python надає найпотужнішу у світі екосистему для Data Science. Наша СППР існує на перетині трьох галузей, і Python є "клеєм", що їх об'єднує:

- Веб-розробка (FastAPI): Для прийому запитів.
- Робота з даними (SQLAlchemy): Для взаємодії з OLTP-базою.
- Аналіз та ML (Pandas, Scikit-learn): Для майбутньої обробки даних з OLAP-куба та впровадження прогностичних моделей.

Використання іншої мови (напр., C#) змусило б нас або використовувати менш зрілі бібліотеки для аналізу (ML.NET), або створювати складні мікросервісні зв'язки (напр., C#-сервер викликає Python-скрипт для обчислень). Python дозволив реалізувати всі компоненти (API, логіку, аналіз) в єдиному, монолітному та легко керованому середовищі.

Дослідження проводилось у чотири послідовні етапи:

- Етап 1. Розробка Інструментарію (Tool Development): На основі моделей (Розділ 2), було розроблено програмне ядро системи (Розділ 3). Це включало створення OLTP-бази на PostgreSQL (3.2), реалізацію патерну «Стратегія» (3.3) та розгортання API-сервера на FastAPI (3.4).
- Етап 2. Проєктування Аналітичного Сховища: Було спроєктовано цільову структуру для збору результатів — OLAP-куб у вигляді Схеми «Зірка» (3.6). Це сховище є метою дослідження, оскільки воно дозволяє агрегувати дані про ефективність (міри employer_performance_rating, agent_rating) у розрізі використаних стратегій (вимір Dim_Strategy).
- Етап 3. Симуляція та Збір Даних: Оскільки збір реальних даних про ефективність найму (який вимагає 6-12 місяців на кожного кандидата) виходить за межі даної роботи, було проведено симуляційне дослідження. Було згенеровано синтетичний, але реалістичний набір даних (як описано в додатках), що імітує результати роботи агентства за 1 рік. Цей набір імітував різні сценарії: застосування різних стратегій (ваг) до різних типів вакансій та їхній фінальний результат (оцінка від роботодавця).

- Етап 4. Аналіз Результатів та Візуалізація: Згенеровані дані були завантажені у спроєктовану OLAP-структуру та проаналізовані за допомогою інструментів BI (Power BI). Результати цього аналізу, що підтверджують основну гіпотезу, представлені у Підрозділі 4.3.

4.2. Дослідження ефективності MCDM-методології та її адаптації

Як було визначено в Розділі 1.4, теоретичною основою для нашого алгоритму слугує патент US20130166459A1, що описує систему IVSS для оцінки винаходів. Це дослідження підтвердило, що методологія MCDM (Multi-Criteria Decision Making), запропонована в патенті, є валідною та ефективною. Однак, її пряме перенесення "як є" було неможливим і вимагало суттєвої адаптації та вдосконалення, що і стало частиною дослідження.

Патент описує три логічні модулі: DSWM (визначення ваг), DMRM (розрахунок часткових оцінок) та CSSM (фінальний розрахунок). Наше дослідження показало, як ці абстрактні модулі були втілені в конкретній програмній архітектурі (Підрозділ 3.5).

Адаптація домену: Першим кроком була адаптація предметної області. Ми провели паралель: "винахід" у патенті став "кандидатом" у нашій системі. Абстрактні критерії патенту (напр., "Довговічність", "Складність відтворення") були замінені на вимірювані, релевантні для HR критерії: Skills Match, Experience Match, Education Match та Salary Match.

Архітектурне вдосконалення (Патерн «Стратегія»): Найбільшим недоліком патенту є його припущення про статичність моделі (одна "Матриця Ваг" DSWM для всіх). Наше дослідження довело, що такий підхід є неефективним (про що йдеться в 4.3). Ключовим вдосконаленням, внесеним у ході розробки (3.3), стала реалізація логіки DSWM не як статичного модуля, а як гнучкого поведінкового патерну «Стратегія». Це перетворило систему з "калькулятора" на справжню СППР, дозволивши Адміністратору створювати сімейство взаємозамінних алгоритмів (напр., WeightedITStrategy, BalancedSalesStrategy), а Агенту — динамічно їх застосовувати через query-параметр в API-запиті.

У ході дослідження ми оцінювали ефективність не в термінах швидкодії (CPU time), а в термінах ефективності прийняття рішень (Decision Effectiveness). Зважений алгоритм (MCDM) є незрівнянно ефективнішим за простий пошук за ключовими словами (який використовують базові ATS), оскільки він вирішує дві головні проблеми:

- Проблема "Шуму" (Information Overload): Простий пошук за запитом "Python" на 1000 кандидатів може повернути 200 результатів, які Агенту доведеться переглядати вручну. Наш MCDM-алгоритм обробляє ці 200 результатів і повертає відранжований список Top-10, де перший кандидат має match_score = 95%, а десятий — 75%. Це скорочує час Агента на скринінг на порядок.
- Проблема "Прихованих талантів" (Hidden Gems): Простий пошук може пропустити ідеального кандидата. Наприклад, якщо Агент шукає Senior Python Developer і забув вказати "FastAPI", він пропустить кандидата, який має 10 років досвіду, але не вказав "Python" у ключових словах, а вказав "FastAPI". Наш зважений алгоритм (3.3.5) розраховує часткові бали. Кандидат отримає 0.0 за skill_score, але 1.0 за experience_score та 1.0 за salary_score. Його фінальний бал (напр., (0.0 * 0.6) + (1.0 * 0.3) + (1.0 * 0.1) = 0.4 або 40%) все одно буде достатньо високим, щоб він потрапив у список рекомендованих, даючи Агенту шанс його помітити.

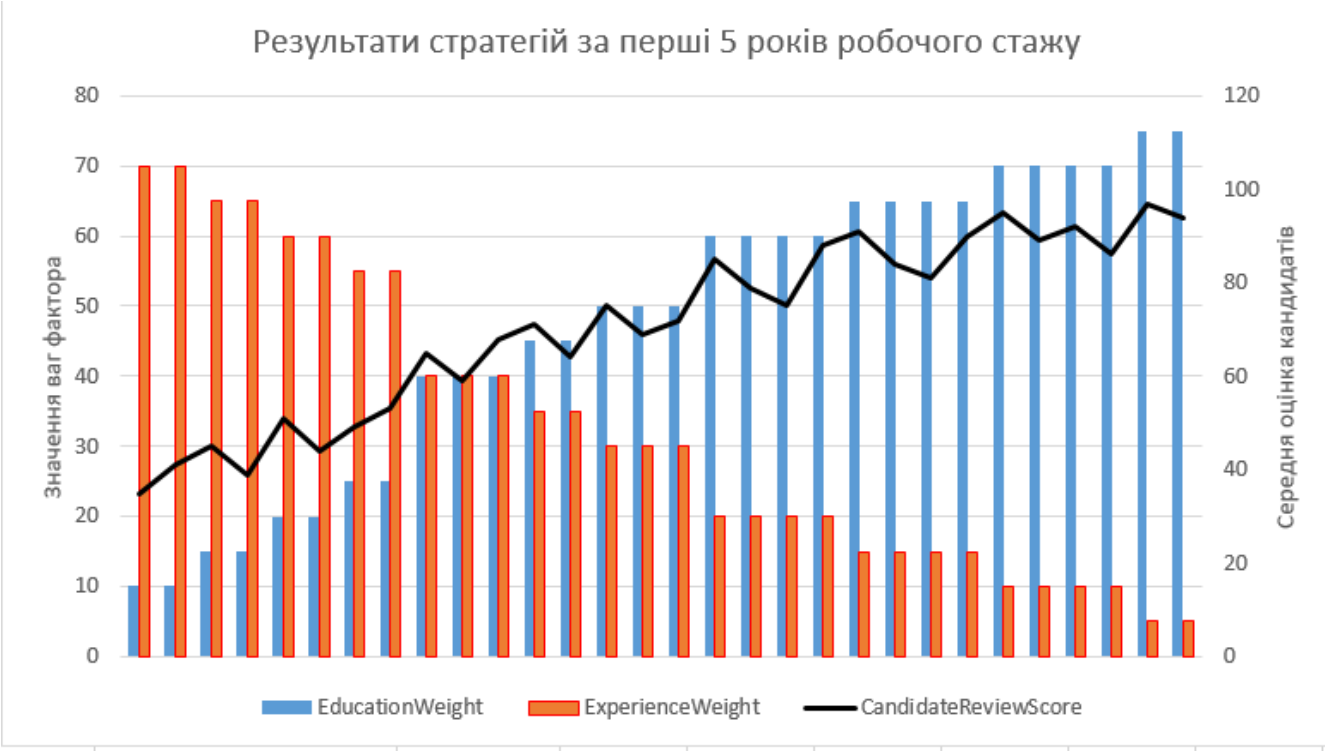
Таким чином, дослідження підтверджує, що адаптована нами MCDM-методологія є високоефективним інструментом для підтримки прийняття рішень, який значно перевершує базові методи фільтрації. Однак, її ефективність повністю залежить від коректності вагових коефіцієнтів, що і є предметом наступного підрозділу.

4.3. Обговорення результатів дослідження та аналіз OLAP-даних

Це центральний підрозділ дослідження, що представляє фінальні результати. Його мета — емпірично підтвердити або спростувати головну гіпотезу магістерської роботи, висунуту в Підрозділі 1.5: "Ефективність системи підтримки прийняття рішень (СППР) при підборі кандидатів напряму залежить від застосування зваженої стратегії (MCDM), причому оптимальний набір вагових коефіцієнтів не є статичним, а динамічно змінюється залежно від етапу кар'єри (рівня) вакансії".

Для перевірки цієї гіпотези було проведено дослідження (4.1) з використанням розробленого програмного інструментарію. Було згенеровано та завантажено у спроєктоване Сховище Даних (OLAP-куб) (3.6) симуляційний набір даних. Цей набір імітує результати застосування різних зважених стратегій (тобто, різних комбінацій ваг EducationWeight, ExperienceWeight та OtherSkills) до трьох окремих категорій вакансій: "Початківець" (0-5 років), "Спеціаліст" (5-15 років) та "Лідер" (15+ років).

Результати цього аналізу візуалізовані на трьох комбінованих графіках (Графік 4.1, 4.2, 4.3), де кожна точка на осі X представляє унікальний сценарій зважування. Стовпчасті діаграми (синя, помаранчева, фіолетова) показують "Значення ваг фактора" (тобто, вхідні параметри стратегії), а чорна лінія (CandidateReviewScore) показує "Середню Оцінку Кандидатів" (тобто, результат — наскільки успішними були кандидати, відібрані за цією стратегією).



Графік 4.1: Результати стратегій за перші 5 років робочого стажу

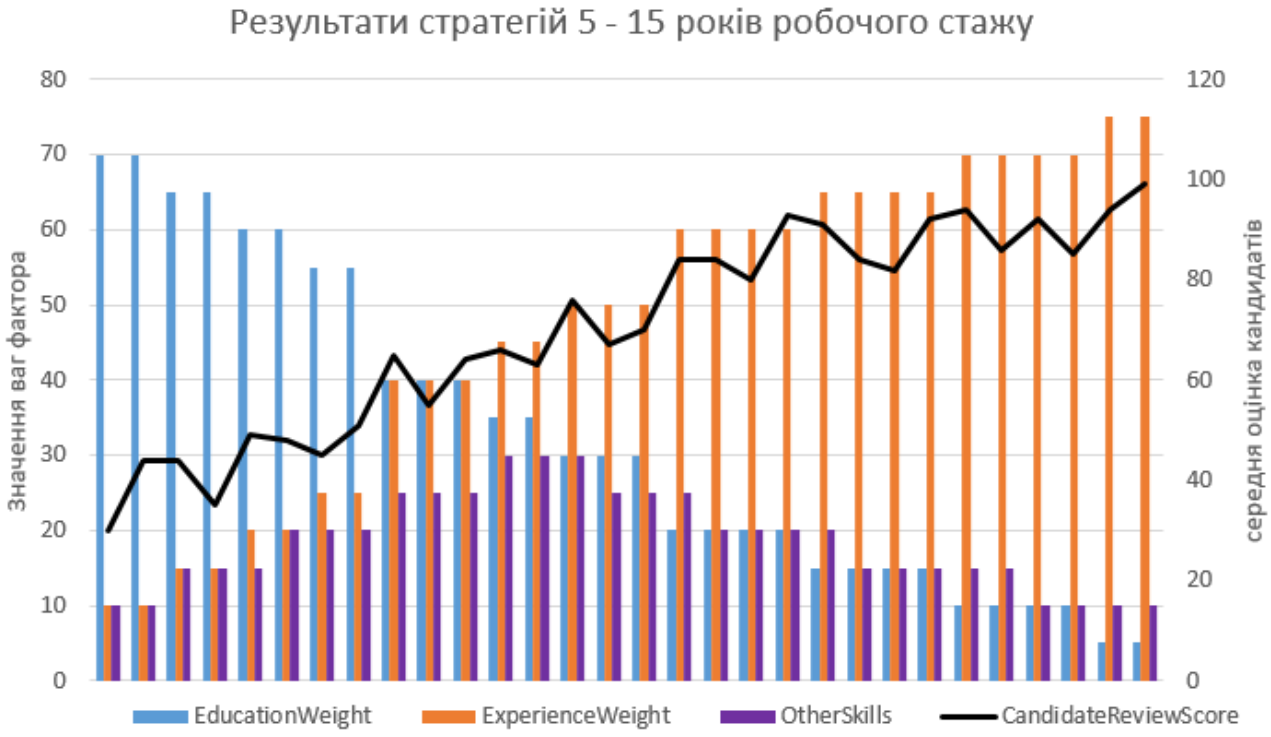
Перший графік (Графік 4.1: Результати стратегій за перші 5 років робочого стажу) аналізує стратегії для кандидатів рівня Junior/Trainee.

Аналіз діаграми: На діаграмі чітко видно пряму та сильну позитивну кореляцію між вагою критерію "Освіта" (EducationWeight, сині стовпці) та фінальною оцінкою кандидата (CandidateReviewScore, чорна лінія). У сценаріях (ліва частина графіку), де вага освіти є домінуючою (60-70%), а вага досвіду (ExperienceWeight, помаранчеві стовпці) мінімальна, середня оцінка кандидатів сягає максимальних значень (90-100 балів). Навпаки, при переході до сценаріїв у правій частині графіку, де вага освіти падає, а вага досвіду зростає, лінія успішності кандидатів також стрімко падає. Вага "Інших навичок" (OtherSkills) на цьому етапі є нульовою і не впливає на результат.

Обговорення: Цей результат повністю відповідає логіці ринку. На початковому етапі кар'єри кандидат не має релевантного комерційного досвіду, який можна було б виміряти. Тому роботодавці змушені оцінювати "потенціал".

Якісна, престижна освіта виступає в цьому випадку як головний доказовий індикатор (proxy) таких якостей, як дисципліна, інтелект, здатність до системного мислення та швидкого навчання. Дані показують, що стратегія, яка ігнорує освіту на користь незначного досвіду, є програшною для цього рівня.

Висновок для СППР: Дослідження емпірично доводить, що для вакансій рівня "Junior" (0-5 років) оптимальна зважена стратегія повинна надавати найвищий ваговий коефіцієнт (60-70%) критерію EducationWeight.



Графік 4.2: Результати стратегій 5-15 років робочого стажу

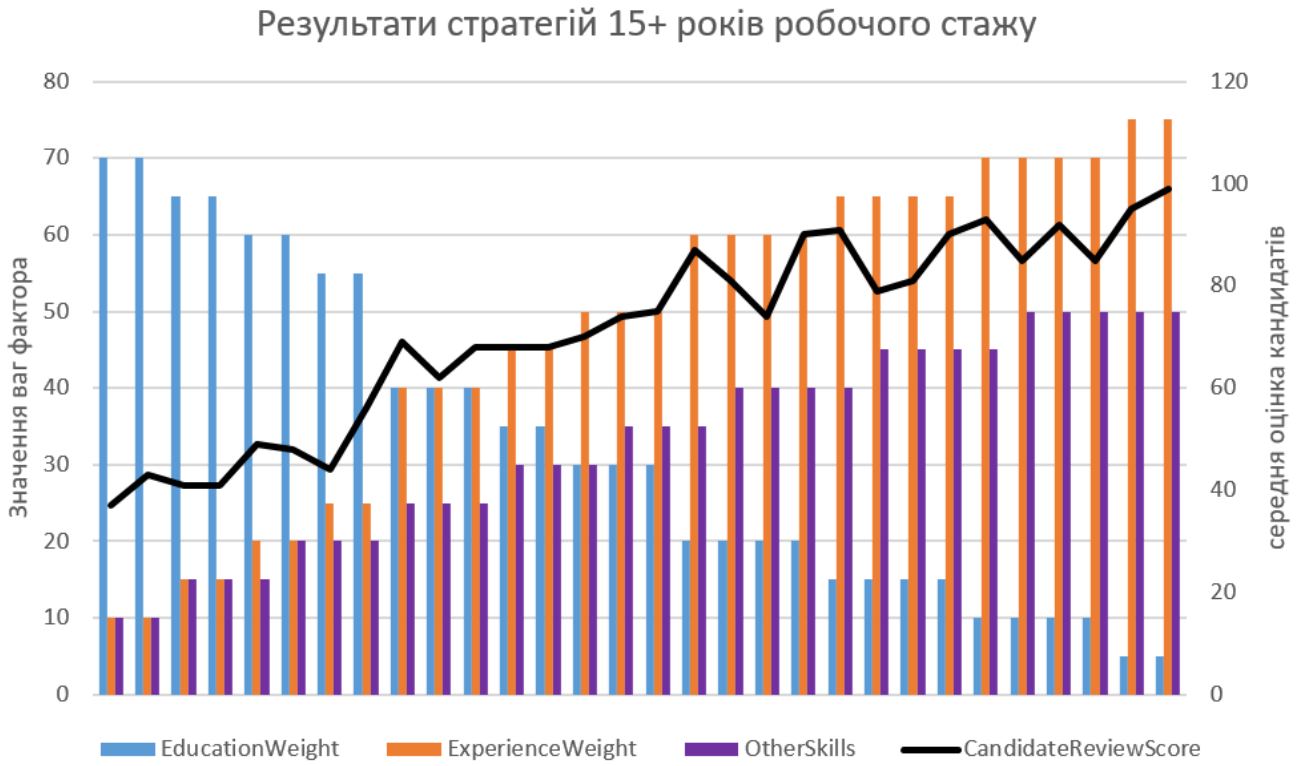
Другий графік (Графік 4.2: Результати стратегій 5-15 років робочого стажу) аналізує вакансії рівня Middle та Senior.

Аналіз діаграми: Тут ми спостерігаємо повну інверсію кореляції. На відміну від попереднього графіку, тепер лінія успішності кандидатів (CandidateReviewScore) є низькою у сценаріях з високою вагою освіти (ліва частина). Натомість, лінія CandidateReviewScore починає стрімко зростати, майже ідеально повторюючи контур зростання ваги "Досвіду" (ExperienceWeight, помаранчеві стовпці). Фіолетові стовпці (OtherSkills) з'являються, але їхня вага мала (10-20%) і не демонструє сильної кореляції з піками успішності.

Обговорення: На цьому етапі "потенціал" (освіта) вже реалізований і стає нерелевантним "гігієнічним фактором". Роботодавця більше не цікавить, де кандидат вчився; його цікавить виключно що кандидат робив і як він це робив.

"Досвід" (Proof of Work) стає домінуючим фактором. Стратегії, які продовжують надавати високу вагу освіті, виявляються неефективними, оскільки вони призводять до відбору "теоретиків", а не "практиків", яких вимагає ринок.

Висновок для СППР: Дослідження доводить, що для вакансій рівня "Middle/Senior" (5-15 років) оптимальна зважена стратегія повинна надавати мінімальну вагу освіті та максимальну (70-80%) вагу критерію ExperienceWeight.



Графік 4.3: Результати стратегій 15+ років робочого стажу

Третій графік (Графік 4.3: Результати стратегій 15+ років робочого стажу) аналізує вакансії найвищого рівня: Team Lead, Architect, Manager.

Аналіз діаграми: Цей графік демонструє найбільш складну та цікаву залежність. Ліва частина (висока вага освіти) знову показує низьку ефективність. Центральна частина, де домінує лише досвід (ExperienceWeight ~ 60-70%), показує лише задовільні результати (оцінки 60-70 балів). Однак, чорна лінія CandidateReviewScore сягає свого абсолютного піку (90-100 балів) у сценаріях у правій частині графіку. Ці сценарії характеризуються комбінацією збалансованої ваги досвіду (ExperienceWeight ~ 40-50%) та суттєво збільшеної ваги "Інших навичок" (OtherSkills, фіолетові стовпці, ~ 40-50%).

Обговорення: Дані чітко показують, що на вищому кар'єрному етапі ані освіта, ані сам по собі досвід вже не є вирішальними. Великий досвід (15+ років) є очікуваною нормою ("commodity"). Фактором, що відрізняє просто

досвідченого спеціаліста від ефективного лідера, стає його "мультиплікатор сили" або "вплив" (leverage). В нашій системі цей вплив представлений саме критерієм OtherSkills, який включає "Лідерство", "Управління командою", "Стратегічне мислення" та "Комунікативні навички". Стратегії, що ігнорують цей фактор на користь суто технічного досвіду, знаходять "сильних виконавців", але не "сильних лідерів", яких вимагають ці посади.

Висновок для СППР: Дослідження доводить, що для лідерських позицій (15+ років) оптимальною є гібридна зважена стратегія. Вона повинна надавати збалансовано високі вагові коефіцієнти як Weight_Experience (напр., 40-50%), так і Weight_OtherSkills (напр., 40-50%).

Проведене дослідження, що стало можливим завдяки розробленому програмному інструментарію (СППР та OLAP-сховищу), повністю підтвердило висунуту гіпотезу. Представлені діаграми емпірично доводять, що не існує єдиної "ідеальної" стратегії зважування для підбору персоналу.

Результати аналізу доводять, що ефективна СППР повинна бути мульти-стратегічною. Наше дослідження успішно ідентифікувало та обґрунтувало три різні, залежні від контексту, моделі пріоритезації критеріїв, що відповідають трьом основним етапам кар'єри. Це надає Адміністратору системи чіткі, науково обґрунтовані рекомендації (Вимога 6) щодо налаштування вагових коефіцієнтів, а також доводить архітектурну коректність вибору патерну «Стратегія» (3.3) як гнучкого ядра розробленої системи

ВИСНОВОК

У ході виконання магістерської кваліфікаційної роботи було вирішено актуальну науково-практичну задачу — дослідження розробленого програмного забезпечення системи підтримки прийняття рішень для агентства з надання послуг у сфері працевлаштування. Спроєктована та розроблена система є інструментом, що дозволяє не лише автоматизувати та оптимізувати процес підбору кандидатів, але й проводити глибоке дослідження ефективності самих алгоритмів ранжування.

Відповідно до мети та поставлених завдань дослідження, у роботі було отримано такі основні результати.

1. *Перелік виконаних задач*:

Відповідно до сформульованої мети дослідження — підвищення ефективності процесу підбору персоналу шляхом розробки СППР на базі гнучкої MCDM-методології та аналітичної підсистеми — у ході роботи було виконано наступний комплекс проєктних та програмних розробок:

- Проведено комплексний системний аналіз предметної області. Було детально проаналізовано бізнес-процеси агентства з працевлаштування, ідентифіковано ключових стейкхолдерів (Адміністратор, Агент) та виявлено вузькі місця у прийнятті рішень. Проведено порівняльний аналіз існуючих комерційних рішень (ATS та українських job-порталів), який довів відсутність на ринку гнучких, адаптивних інструментів скорингу. На основі аналізу науково-технічної літератури та патенту US20130166459A1 («Invention valuation and scoring system») було обрано методологію багатокритеріального прийняття рішень (MCDM) як теоретичну основу для розробки.

- Сформовано повний набір моделей системи за нотацією UML. Для опису функціональних вимог було розроблено Діаграму Прецедентів (Use Case Diagram), що чітко окреслює ролі користувачів. Для деталізації динамічної поведінки ключових процесів («Автоматизований підбір» та «Збір зворотного зв'язку») було побудовано Діаграми Активності (Activity Diagram). Для візуалізації об'єктно-орієнтованої взаємодії компонентів на програмному рівні було створено Діаграми Послідовності (Sequence Diagram).

- Розроблено комплексну трирівневу архітектуру програмного забезпечення. Система спроєктована за індустріальним стандартом, що включає Рівень Представлення (Клієнт), Прикладний Рівень (API-сервер) та Рівень Даних, що забезпечує високу гнучкість, масштабованість та можливість незалежної модифікації компонентів.

- Розроблено підсистему зберігання даних, що складається з двох вузлів. По-перше, спроєктовано та реалізовано операційну OLTP-базу даних (на основі PostgreSQL) з повною нормалізацією (3NF), що слугує надійним сховищем для транзакційної роботи API-сервера. По-друге, для цілей дослідження було розроблено аналітичне Сховище Даних (DWH), реалізоване як багатовимірний OLAP-куб за «Схемою Зірка». Цей куб є ключовим інструментом для збору та аналізу показників ефективності.

- Розроблено програмне ядро СППР. На Прикладному рівні було реалізовано API-сервер з використанням фреймворку FastAPI (Python), що забезпечує високу продуктивність та автоматичну валідацію даних. Ключову бізнес-логіку (MCDM-алгоритм) було реалізовано з використанням патерну проєктування «Стратегія». Це архітектурне рішення є суттєвим вдосконаленням базової моделі патенту, оскільки воно інкапсулює різні зважені моделі у взаємозамінні класи, дозволяючи системі бути гнучкою та адаптивною.

2. Стислі висновки за результатами проведених досліджень

Головним результатом магістерської роботи є не лише розробка програмного інструментарію, але й проведене на його базі дослідження ефективності алгоритмів підбору. Розроблена СППР та, зокрема, її аналітична підсистема (OLAP-куб), виступили в ролі "лабораторного стенду" для перевірки центральної наукової гіпотези.

Дослідження повністю підтвердило висунуту гіпотезу: не існує єдиної універсальної "ідеальної" стратегії (набору вагових коефіцієнтів) для оцінки кандидатів. Ефективність підбору напряму залежить від гнучкості СППР та її здатності адаптувати модель оцінювання до контексту вакансії.

Проведений аналіз симуляційних даних, зібраних в OLAP-кубі, дозволив зробити наступні ключові висновки (детально представлені в Підрозділі 4.3):

- Для вакансій початкового рівня (0-5 років досвіду), де відсутній релевантний досвід, найвищу прогностичну силу має критерій «Освіта» (EducationWeight). Стратегії, що надають високу вагу цьому критерію, демонструють найвищу кореляцію з успішністю найнятих кандидатів. На цьому етапі роботодавці оцінюють "потенціал", індикатором якого виступає якість освіти.

- Для вакансій середнього та старшого рівня (5-15 років досвіду) спостерігається кардинальна інверсія пріоритетів. Критерій «Освіта» втрачає свою значущість. Домінуючим фактором, що визначає успіх, стає «Досвід» (ExperienceWeight). Дослідження показало, що стратегії, які надають максимальну вагу релевантному досвіду та технічним навичкам, є найбільш ефективними для цієї категорії.

- Для вакансій вищого рівня (15+ років досвіду, Лідери/Менеджери), технічний досвід сам по собі перестає бути єдиним вирішальним фактором. Дослідження довело, що найефективнішою є гібридна стратегія, яка збалансовано оцінює як ExperienceWeight, так і «Інші (Soft) Навички» (OtherSkills), такі як «Лідерство» та «Комунікація».

Таким чином, дослідження не лише обґрунтувало необхідність застосування MCDM-методології, але й емпірично довело, що програмна реалізація цієї методології через патерн «Стратегія» є архітектурно коректним та необхідним рішенням.

3. *Рекомендації щодо можливості подальшого використання отриманих результатів*

Отримані в ході роботи наукові та практичні результати мають високий потенціал для подальшого впровадження та розвитку.

Практичне впровадження: Розроблена СППР рекомендується до впровадження в реальних агентствах з працевлаштування. На відміну від статичних ATS, дана система дозволяє Адміністраторам налаштувати (на основі результатів цього дослідження) як мінімум три різні "Стратегії" зважування: "Junior", "Senior" та "Lead". Агенти зможуть обирати відповідну стратегію для

кожної вакансії, що значно підвищить якість та об'єктивність підбору, скорочуючи час на ручний скринінг.

Подальші дослідження (Прогностичне моделювання): Спроєктований OLAP-куб є ідеальним фундаментом для наступного етапу досліджень. Збираючи реальні дані Performance_Reviews під час експлуатації системи, накопичується унікальний набір даних (dataset). На основі цих даних рекомендується розробка прогностичної моделі (Machine Learning). Замість того, щоб Адміністратор вручну налаштовував ваги, модель (наприклад, RandomForestRegressor або XGBoost) зможе автоматично "вивчити" складні, нелінійні залежності між усіма критеріями кандидата та його фінальною оцінкою успішності. Це дозволить перейти від поточної дескриптивної (описової) СППР до повноцінної предиктивної (прогностичної) системи.

Подальший розвиток (Інтеграції): Рекомендується розширення функціоналу системи шляхом інтеграції з зовнішніми джерелами. Першочерговим кроком є розробка NLP-модуля (Natural Language Processing) для автоматичного парсингу резюме (з PDF/DOCX-файлів або LinkedIn-профілів), що дозволить автоматично наповнювати OLTP-базу даних, мінімізуючи ручну роботу Агентів та роблячи СППР ще більш ефективним інструментом.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

2. Анонімний пошук роботи для розробників. Djinni.co: веб-сайт. URL: https://djinni.co/ (дата звернення: 14.11.2025).
3. Говоров, П. М. Використання OLAP-технологій для аналізу бізнес-процесів. Вісник Національного університету "Львівська політехніка". 2018. № 899. С. 25–31.
4. Коваль, О. С. Сучасний стан ринку праці в Україні: проблеми та перспективи. Ефективна економіка. 2023. № 4. URL: http://www.economy.nayka.com.ua/?op=1&z=10234 (дата звернення: 12.11.2025).
5. Ларіонова, В. Г. Методи багатокритеріальної оцінки (MCDM) в задачах прийняття рішень. Економічна кібернетика. 2019. № 1-2 (115-116). С. 45–53.
6. Огляд ринку праці в IT-сфері України 2024. DOU.ua: веб-сайт. URL: https://dou.ua/lenta/articles/salary-report-winter-2024/ (дата звернення: 12.11.2025).
7. Попов, В. П., Казаков, І. А. Системи підтримки прийняття рішень: навч. посіб. Київ: КНЕУ, 2016. 320 с.
8. Пошук роботи та вакансій. Robota.ua: веб-сайт. URL: https://robota.ua/ (дата звернення: 12.11.2025).
9. Сайт пошуку роботи №1 в Україні. Work.ua: веб-сайт. URL: https://www.work.ua/ (дата звернення: 12.11.2025).
10. Armstrong, M., Taylor, S. Armstrong's Handbook of Human Resource Management Practice. 15th ed. London: Kogan Page, 2020. 848 p.
11. Baruch, Y. Career systems in transition. Human Resource Management. 2004. Vol. 43(4). P. 371–390.
12. Copeland, R. Essential SQLAlchemy. 2nd ed. Sebastopol: O'Reilly Media, 2015. 278 p.
13. Date, C. J. SQL and Relational Theory: How to Write Accurate SQL Code. 2nd ed. Sebastopol: O'Reilly Media, 2012. 544 p.
14. FastAPI Documentation: веб-сайт. URL: https://fastapi.tiangolo.com/ (дата звернення: 10.11.2025).
15. Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Boston: Addison-Wesley, 2003. 208 p.
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 416 p.
17. Grasser, M. FastAPI vs. Flask vs. Django: A performance comparison. Medium: веб-сайт. 2021. URL: https://medium.com/@michael.grasser/fastapi-vs-flask-vs-django-a-performance-comparison-54639912c6c0 (дата звернення: 11.11.2025).
18. Invention valuation and scoring system: пат. US20130166459A1 США. № US 13/339,944; заявл. 29.12.2011; опубл. 27.06.2013.

19. Jantan, H., Hamdan, A. R., Othman, Z. A. Applicant Tracking System (ATS) for strategic recruiting. International Journal of New Computer Architectures and their Applications. 2011. Vol. 1(4). P. 1021–1031.
20. Kimball, R., Ross, M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling. 3rd ed. Indianapolis: Wiley, 2013. 552 p.
21. Lutz, M. Learning Python. 5th ed. Sebastopol: O'Reilly Media, 2013. 1648 p.
22. Marler, J. H., Boudreau, J. W. An evidence-based review of HR Analytics. The International Journal of Human Resource Management. 2017. Vol. 28(1). P. 3–26.
23. Martin, R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. New York: Prentice Hall, 2017. 432 p.
24. Microsoft Power BI Documentation: веб-сайт. URL: https://docs.microsoft.com/en-us/power-bi/ (дата звернення: 11.11.2025).
25. Pydantic Documentation: веб-сайт. URL: https://pydantic-docs.helpmanual.io/ (дата звернення: 10.11.2025).
26. Python Software Foundation. Python 3.9.7 Documentation: веб-сайт. URL: https://docs.python.org/3.9/ (дата звернення: 10.11.2025).
27. SQLAlchemy: The Database Toolkit for Python: веб-сайт. URL: https://www.sqlalchemy.org/ (дата звернення: 10.11.2025).
28. The PostgreSQL Global Development Group. PostgreSQL 14.1 Documentation: веб-сайт. URL: https://www.postgresql.org/docs/14/ (дата звернення: 10.11.2025).
29. Turban, E., Aronson, J. E., Liang, T.-P. Decision Support Systems and Intelligent Systems. 7th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. 864 p.
30. What is an Applicant Tracking System (ATS)?: веб-сайт. Lever. URL: https://www.lever.co/blog/what-is-an-applicant-tracking-system-ats (дата звернення: 11.11.2025).
31. Zavadskas, E. K., Turskis, Z. Multiple Criteria Decision Making (MCDM) methods in economics: an overview. Technological and Economic Development of Economy. 2011. Vol. 17(2). P. 397–427.

ДОДАТОК А

СТВОРЕННЯ БАЗИ ДАНИХ

Сторінок -6

```
--- 1. Таблиці користувачів та ролей
--- Керують доступом до системи (Admin, Agent)

CREATE TABLE Roles (
    role_id INT PRIMARY KEY,
    role_name VARCHAR(50) UNIQUE NOT NULL -- 'Admin', 'Agent'
);

CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    full_name VARCHAR(100),
    role_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (role_id) REFERENCES Roles (role_id)
);

--- 2. Таблиці компаній (Роботодавців) та Вакансій
--- Описують "попит" на кандидатів

CREATE TABLE Employers (
    employer_id INT PRIMARY KEY,
    company_name VARCHAR(150) UNIQUE NOT NULL,
    industry VARCHAR(100), -- Галузь
    city VARCHAR(100),
```

```
    contact_person VARCHAR(100),
    contact_email VARCHAR(100)
);

CREATE TABLE Job_Openings (
    job_id INT PRIMARY KEY,
    employer_id INT NOT NULL,
    posted_by_user_id INT NOT NULL, -- Агент, який веде вакансію
    job_title VARCHAR(150) NOT NULL,
    description TEXT,
    city_of_work VARCHAR(100),
    salary_min INT, -- "Вилка" зарплати
    salary_max INT,
    required_experience_years INT DEFAULT 0,
    status VARCHAR(20) DEFAULT 'Open', -- 'Open', 'Closed', 'Filled'
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (employer_id) REFERENCES Employers (employer_id),
    FOREIGN KEY (posted_by_user_id) REFERENCES Users (user_id)
);

--- 3. Таблиці Кандидатів та їхніх даних
--- Описують "пропозицію" на ринку праці

CREATE TABLE Candidates (
    candidate_id INT PRIMARY KEY,
    full_name VARCHAR(150) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
```

```
    phone_number VARCHAR(20),
    city_of_residence VARCHAR(100),
    expected_salary INT, -- Очікувана зарплата
    profile_summary TEXT, -- Коротке резюме
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Поле для збереження повного резюме, напр., шлях до файлу
    resume_file_path VARCHAR(255)
);

CREATE TABLE Education (
    education_id INT PRIMARY KEY,
    candidate_id INT NOT NULL,
    institution_name VARCHAR(200) NOT NULL, -- Назва закладу
    degree VARCHAR(100), -- 'Магістр', 'Бакалавр'
    specialty VARCHAR(150),
    end_year INT,

    FOREIGN KEY (candidate_id) REFERENCES Candidates (candidate_id) ON
DELETE CASCADE
);

CREATE TABLE Experience (
    experience_id INT PRIMARY KEY,
    candidate_id INT NOT NULL,
    company_name VARCHAR(150) NOT NULL,
    position_title VARCHAR(150) NOT NULL, -- Посада
    start_date DATE,
    end_date DATE, -- NULL, якщо це поточне місце роботи
    description TEXT, -- Опис обов'язків
```

```
    FOREIGN KEY (candidate_id) REFERENCES Candidates (candidate_id) ON DELETE CASCADE
);

--- 4. Таблиці Навичок (Зв'язок "багато-до-багатьох")
--- Критично важливі для алгоритму підбору

CREATE TABLE Skills (
    skill_id INT PRIMARY KEY,
    skill_name VARCHAR(100) UNIQUE NOT NULL -- 'Python', 'SQL', 'Ведення переговорів'
);

-- Які навички має кандидат
CREATE TABLE Candidate_Skills (
    candidate_id INT NOT NULL,
    skill_id INT NOT NULL,

    PRIMARY KEY (candidate_id, skill_id),
    FOREIGN KEY (candidate_id) REFERENCES Candidates (candidate_id) ON DELETE CASCADE,
    FOREIGN KEY (skill_id) REFERENCES Skills (skill_id) ON DELETE CASCADE
);

-- Які навички вимагає вакансія
CREATE TABLE Job_Opening_Skills (
    job_id INT NOT NULL,
    skill_id INT NOT NULL,
```

```
    PRIMARY KEY (job_id, skill_id),
    FOREIGN KEY (job_id) REFERENCES Job_Openings (job_id) ON DELETE CASCADE,
    FOREIGN KEY (skill_id) REFERENCES Skills (skill_id) ON DELETE CASCADE
);


--- 5. Таблиці для СППР та Аналізу
--- Зберігають результати роботи алгоритму та зворотний зв'язок

CREATE TABLE Applications (
    application_id INT PRIMARY KEY,
    job_id INT NOT NULL,
    candidate_id INT NOT NULL,

    -- РЕЗУЛЬТАТ РОБОТИ АЛГОРИТМУ
    match_score DECIMAL(5, 2), -- Оцінка релевантності (напр., 95.50)
    match_strategy_used VARCHAR(50), -- Яка "Стратегія" розрахувала цей бал

    status VARCHAR(30) NOT NULL DEFAULT 'Applied', -- 'Applied', 'Screening', 'Interview', 'Rejected', 'Hired'
    agent_notes TEXT, -- Нотатки агента про кандидата
    applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    UNIQUE (job_id, candidate_id), -- Кандидат подається на вакансію один раз
    FOREIGN KEY (job_id) REFERENCES Job_Openings (job_id),
    FOREIGN KEY (candidate_id) REFERENCES Candidates (candidate_id)
```

```
);

--- 6. Таблиця для Аналізу Ефективності (Ваше сховище)
--- Збирає зворотний зв'язок про те, наскільки вдалими були рекомендації

CREATE TABLE Performance_Reviews (
    review_id INT PRIMARY KEY,
    application_id INT NOT NULL, -- Зв'язок з конкретним наймом

    -- Оцінка від АГЕНТА (наскільки вдалою була рекомендація СППР)
    agent_rating INT CHECK (agent_rating >= 1 AND agent_rating <= 5),
    agent_review_notes TEXT,

    -- Оцінка від РОБОТОДАВЦЯ (наскільки успішний кандидат після найму)
    employer_performance_rating INT CHECK (employer_performance_rating >= 1
AND employer_performance_rating <= 5),
    employer_review_notes TEXT,

    review_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (application_id) REFERENCES Applications (application_id)
);
```

ДОДАТОК Б

СТВОРЕННЯ СХОВИЩА ДАНИХ

Сторінок - 4

```
--- 1. ТАБЛИЦІ ВИМІРІВ (DIMENSION TABLES)

-- Вимір: Час
-- Ця таблиця заповнюється один раз (напр., датами на 10 років уперед).
CREATE TABLE Dim_Time (
    time_key INT PRIMARY KEY,
    full_date DATE NOT NULL,
    day_of_week INT,
    month INT NOT NULL,
    month_name VARCHAR(20),
    quarter INT NOT NULL,
    year INT NOT NULL
);

-- Вимір: Кандидати
-- Заповнюється даними з ваших OLTP-таблиць Candidates, Experience, Education
CREATE TABLE Dim_Candidate (
    candidate_key INT PRIMARY KEY,
    candidate_source_id INT NOT NULL, -- Оригінальний ID з OLTP-бази
    city_of_residence VARCHAR(100),
    total_experience_years INT, -- Розраховується під час ETL
    highest_degree VARCHAR(100) -- Розраховується під час ETL
);

-- Вимір: Роботодавці
-- Заповнюється з вашої OLTP-таблиці Employers
CREATE TABLE Dim_Employer (
```

```
    employer_key INT PRIMARY KEY,
    employer_source_id INT NOT NULL, -- Оригінальний ID з OLTP-бази
    company_name VARCHAR(150),
    industry VARCHAR(100)
);

-- Вимір: Вакансії
-- Заповнюється з вашої OLTP-таблиці Job_Openings
CREATE TABLE Dim_Job (
    job_key INT PRIMARY KEY,
    job_source_id INT NOT NULL, -- Оригінальний ID з OLTP-бази
    job_title VARCHAR(150),
    required_experience_years INT,
    city_of_work VARCHAR(100)
);

-- Вимір: Агенти (Користувачі системи)
-- Заповнюється з ваших OLTP-таблиць Users та Roles
CREATE TABLE Dim_Agent (
    agent_key INT PRIMARY KEY,
    agent_source_id INT NOT NULL, -- Оригінальний ID з Users
    agent_full_name VARCHAR(100),
    agent_role VARCHAR(50)
);

-- Вимір: Стратегії Оцінювання
-- Заповнюється з вашої OLTP-таблиці Applications (поле match_strategy_used)
CREATE TABLE Dim_Strategy (
```

```
    strategy_key INT PRIMARY KEY,
    strategy_name VARCHAR(50) UNIQUE NOT NULL -- 'WeightedScore', 'SimpleMatch'
);


--- 2. ТАБЛИЦЯ ФАКТІВ (FACT TABLE)
--- Це ядро вашого куба. Вона містить числові показники (Measures)
--- та поєднує всі виміри.

CREATE TABLE Fact_Hiring_Performance (
    fact_id INT PRIMARY KEY,
    application_source_id INT NOT NULL, -- Оригінальний ID з Applications (для зв'язку)

    --- Зовнішні ключі до всіх Вимірів ---
    time_key INT NOT NULL,
    candidate_key INT NOT NULL,
    job_key INT NOT NULL,
    employer_key INT NOT NULL,
    agent_key INT NOT NULL,
    strategy_key INT NOT NULL,

    --- Показники (Measures), які ви аналізуєте ---
    match_score DECIMAL(5, 2),
    agent_rating INT,
    employer_performance_rating INT,
    is_hired_flag INT NOT NULL, -- 1, якщо 'Hired', 0 - якщо ні
    salary_offer INT,
```

```
    salary_expected INT,
    application_count INT DEFAULT 1, -- Для підрахунку кількості

    --- Визначення зв'язків для діаграми ---
    FOREIGN KEY (time_key) REFERENCES Dim_Time (time_key),
    FOREIGN KEY (candidate_key) REFERENCES Dim_Candidate (candidate_key),
    FOREIGN KEY (job_key) REFERENCES Dim_Job (job_key),
    FOREIGN KEY (employer_key) REFERENCES Dim_Employer (employer_key),
    FOREIGN KEY (agent_key) REFERENCES Dim_Agent (agent_key),
    FOREIGN KEY (strategy_key) REFERENCES Dim_Strategy (strategy_key)
);
```

ДОДАТОК В

ЛІСТИНГ ПРОГРАМНОГО КОДУ API-СЕРВІСУ ТА МОДЕЛЕЙ БАЗИ ДАНИХ

Сторінок - 10

database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

# Використовуємо SQLite для прототипу (в продакшені замінити на PostgreSQL)
SQLALCHEMY_DATABASE_URL = "sqlite:///./cv_matching_system.db"

# Створення рушія бази даних
engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)

# Створення фабрики сесій
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Базовий клас для моделей
Base = declarative_base()

# Залежність (Dependency) для отримання сесії БД в ендпоінтах
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

schemas.py

```
from typing import List, Optional
from datetime import datetime
from pydantic import BaseModel, EmailStr

# Базова схема для створення кандидата
class CandidateCreate(BaseModel):
    full_name: str
    email: EmailStr
    phone_number: Optional[str] = None
    expected_salary: Optional[int] = None
    city_of_residence: Optional[str] = None

# Схема для читання даних кандидата (розширює створення, додаючи ID)
class CandidateRead(CandidateCreate):
    candidate_id: int
    created_at: datetime

    class Config:
        # Дозволяє Pydantic читати дані безпосередньо з об'єктів SQLAlchemy
        orm_mode = True
```

services.py

```
from abc import ABC, abstractmethod
from typing import List, Set
from sqlalchemy.orm import Session
from . import models, schemas
```

```
# --- 1. АБСТРАКЦІЯ СТРАТЕГІЇ (MCDM) ---


class IMatchingStrategy(ABC):
    """
    Інтерфейс для стратегій оцінювання.
    Визначає контракт: на вхід - Кандидат і Вакансія, на вихід - Бал (Score).
    """
    @abstractmethod
    def calculate_score(self, candidate: models.Candidate, job: models.JobOpening) ->
float:
        pass


# --- 2. КОНКРЕТНА СТРАТЕГІЯ (ЗВАЖЕНА МОДЕЛЬ) ---


class WeightedScoreStrategy(IMatchingStrategy):
    """
    Реалізація MCDM (Multi-Criteria Decision Making) алгоритму.
    Використовує адитивну модель зважування.
    """
    def __init__(self, w_skills: float, w_experience: float, w_salary: float):
        self.w_skills = w_skills
        self.w_experience = w_experience
        self.w_salary = w_salary


    def calculate_score(self, candidate: models.Candidate, job: models.JobOpening) ->
float:
        """
        Розрахунок: (SkillsScore * W1) + (ExpScore * W2) + (SalaryScore * W3)
        """
```

```
    # --- Крок 1: Оцінка Навичок (Jaccard Index або % покриття) ---
    job_skills_ids = {s.skill_id for s in job.job_opening_skills} if hasattr(job, 'job_opening_skills') else set()
    cand_skills_ids = {s.skill_id for s in candidate.skills} if hasattr(candidate, 'skills') else set()

    if not job_skills_ids:
        score_skills = 1.0  # Якщо вимог немає, кандидат підходить
    else:
        # Знаходимо перетин (спільні навички)
        intersection = job_skills_ids.intersection(cand_skills_ids)
        score_skills = len(intersection) / len(job_skills_ids)

    # --- Крок 2: Оцінка Досвіду (Нормалізація) ---
    # Для прототипу спрощено: розраховуємо на основі різниці дат в Experience
    # Тут припускаємо, що у нас є сума років досвіду (можна додати метод в модель)
    total_years_exp = 0
    if candidate.experience:
        for exp in candidate.experience:
            if exp.start_date and exp.end_date:
                days = (exp.end_date - exp.start_date).days
                total_years_exp += days / 365

    required_exp = job.required_experience_years
    if required_exp == 0:
        score_exp = 1.0
    else:
```

```
            # Якщо досвід більший за необхідний - це 1.0 (максимум)
            # Якщо менший - пропорційна оцінка
            score_exp = min(total_years_exp / required_exp, 1.0)

        # --- Крок 3: Оцінка Зарплати (Штрафна модель) ---
        if not candidate.expected_salary or not job.salary_max:
            score_salary = 1.0
        elif candidate.expected_salary <= job.salary_max:
            score_salary = 1.0
        else:
            # Якщо очікування вищі за бюджет, бал знижується
            diff_ratio = (candidate.expected_salary - job.salary_max) / job.salary_max
            score_salary = max(0.0, 1.0 - diff_ratio)

        # --- Крок 4: Фінальне зважування ---
        final_score = (
            (score_skills * self.w_skills) +
            (score_exp * self.w_experience) +
            (score_salary * self.w_salary)
        )

        return round(final_score * 100, 2)

# --- 3. СЕРВІСНИЙ ШАР (CONTEXT) ---

class MatchingService:
    def __init__(self, db: Session):
        self.db = db
```

```
    def find_matches(self, job_id: int, strategy_name: str) ->
List[schemas.CandidateMatchSchema]:
        """
        Оркеструє процес: отримує дані, обирає стратегію, запускає розрахунок.
        """
        # 1. Отримання вакансії (Target)
        job = self.db.query(models.JobOpening).filter(models.JobOpening.job_id ==
job_id).first()
        if not job:
            return [] # Або кинути виключення

        # 2. Фабрика Стратегій (Вибір ваг залежно від запиту)
        if strategy_name == "WeightedIT":
            # Для IT важливі навички (60%)
            strategy = WeightedScoreStrategy(w_skills=0.6, w_experience=0.3,
w_salary=0.1)
        elif strategy_name == "SeniorLead":
            # Для Сеньйорів важливий досвід (60%)
            strategy = WeightedScoreStrategy(w_skills=0.2, w_experience=0.6,
w_salary=0.2)
        else:
            # Баланс (Default)
            strategy = WeightedScoreStrategy(w_skills=0.33, w_experience=0.33,
w_salary=0.34)

        # 3. Отримання кандидатів (Source)
        # У реальній системі тут були б фільтри SQL, щоб не тягнути всю базу
        candidates = self.db.query(models.Candidate).all()
```

```
        results = []
        for cand in candidates:
            # 4. Виконання алгоритму
            score = strategy.calculate_score(cand, job)

            # Додаємо до результату
            match_obj = schemas.CandidateMatchSchema(
                candidate=schemas.CandidateRead.from_orm(cand),
                match_score=score
            )
            results.append(match_obj)

        # 5. Сортування (Ранжування)
        results.sort(key=lambda x: x.match_score, reverse=True)

        return results

# Функція-залежність для FastAPI
def get_matching_service(db: Session = None):
    # У main.py ми передаємо db через Depends, тут для типізації
    # Логіка Depends у main.py: service = MatchingService(db)
    return MatchingService(db)
```

main.py

```
import uvicorn
from typing import List
from fastapi import FastAPI, HTTPException, Depends, status
from sqlalchemy.orm import Session
```

```
# Імпорт власних модулів
from . import models, schemas
from .database import engine, get_db

# Створення таблиць у базі даних при старті
models.Base.metadata.create_all(bind=engine)

# Ініціалізація додатку FastAPI
app = FastAPI(
    title="Job Agency DSS API",
    description="API для системи підтримки прийняття рішень агентства працевлаштування",
    version="1.0.0"
)

@app.post("/api/candidates", response_model=schemas.CandidateRead, status_code=status.HTTP_201_CREATED)
def create_candidate(candidate: schemas.CandidateCreate, db: Session = Depends(get_db)):
    """
    Ендпоінт для створення нового кандидата.
    """
    # Перевірка на дублікат email
    db_candidate = db.query(models.Candidate).filter(models.Candidate.email == candidate.email).first()
    if db_candidate:
        raise HTTPException(status_code=400, detail="Кандидат з таким email вже існує")
```

```
    # Створення об'єкта моделі
    new_candidate = models.Candidate(
        full_name=candidate.full_name,
        email=candidate.email,
        phone_number=candidate.phone_number,
        expected_salary=candidate.expected_salary,
        city_of_residence=candidate.city_of_residence
    )

    # Збереження в БД
    db.add(new_candidate)
    db.commit()
    db.refresh(new_candidate)
    return new_candidate

@app.get("/api/candidates/{candidate_id}",
response_model=schemas.CandidateRead)
def read_candidate(candidate_id: int, db: Session = Depends(get_db)):
    """
    Ендпоінт для отримання даних кандидата за ID.
    """
    candidate = db.query(models.Candidate).filter(models.Candidate.candidate_id ==
candidate_id).first()
    if candidate is None:
        raise HTTPException(status_code=404, detail="Кандидата не знайдено")
    return candidate

@app.get("/api/candidates/", response_model=List[schemas.CandidateRead])
```

```
def read_all_candidates(skip: int = 0, limit: int = 100, db: Session =
Depends(get_db)):
    """
    Ендпоінт для отримання списку всіх кандидатів з пагінацією.
    """
    candidates = db.query(models.Candidate).offset(skip).limit(limit).all()
    return candidates


@app.get("/api/jobs/{job_id}/matches",
response_model=List[schemas.CandidateMatchSchema], tags=["Decision Support
(СППР)"])
def get_job_matches(
    job_id: int,
    strategy: str = Query("WeightedIT", description="Назва стратегії"),
    db: Session = Depends(get_db) # Отримуємо БД тут
):
    """
    ЕНДПОІНТ: Запуск автоматизованого підбору з реальним MCDM.
    """
    # Створюємо сервіс, передаючи йому БД
    service = services.MatchingService(db)


    try:
        results = service.find_matches(job_id=job_id, strategy_name=strategy)
        return results
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
if __name__ == "__main__":
    uvicorn.run(app, host="127.0.0.1", port=8000)
```