НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG

Виконав : студент __4__ курсу, групи _ІМ-21_
(шифр групи)

Гундарев Ростислав Денисович
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент, аспірант Туганських О. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ст. викл., д.ф.к.і., Міщенко Л. Д.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доцент кафедри СПСКС, к.т.н., доцент Тарасенко-Клятченко О. В
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Гундарева Ростислава Денисовича

1. Тема проєкту *Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG*
керівник проєкту Туганських Олександр Антонович, асистент,,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від від 03 червня 2026 року No2055-с

2. Термін здачі студентом закінченого проєкту *30 травня 2026.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються):
Аналіз існуючих підходів до реалізації автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG. Розробка програмної системи з автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG. Прикладні аспекти застосування програмної системи з автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG.

5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень) структура системи, діаграма класів, алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Міщенко Л. Д.		

7. Дата видачі завдання «17» січня 2026 р.

Календарний план

№ п/р	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	17.01.2026	
2.	*Вивчення та аналіз завдання*	30.01.2026	
3.	*Розробка архітектури та загальної структури системи*	19.02.2026	
4.	*Розробка структур окремих підсистем*	10.03.2026	
5.	*Програмна реалізація системи*	20.05.2026	
6.	*Оформлення пояснювальної записки*	25.05.2026	
7.	*Захист програмного продукту*	31.05.2026	
8.	*Передзахист*	10.06.2026	
9.	*Захист*	18.06.2026	

Студент-дипломник _____ Ростислав ГУНДАРЕВ
(підпис)

Керівник проєкту _____ Олександр ТУГАНСЬКИХ
(підпис)

АНОТАЦІЯ

У дипломному проєкті досліджено існуючі підходи до автоматизації обробки електронної пошти та методології побудови систем штучного інтелекту, зокрема проведено порівняльний аналіз комерційних рішень і методів класифікації тексту. За результатами аналізу розроблено спеціалізовану систему автоматизованої обробки листів абітурієнтів на основі архітектури RAG, яка реалізує семантичну класифікацію вхідних листів за допомогою моделі all-MiniLM-L6-v2 та векторної бази даних ChromaDB, розпізнавання типів вкладених документів мультимодальною моделлю CLIP ViT-B/32, верифікацію кваліфікованого електронного підпису через портал id.gov.ua, а також автоматичну генерацію чернеток відповідей засобами великої мовної моделі.

Ключові слова: автоматизації обробки електронної пошти, штучний інтелект, RAG, семантична класифікація, all-MiniLM-L6-v2, векторна база даних, ChromaDB, CLIP ViT-B/32, електронний підпис, велика мовна модель.

ANNOTATION

This diploma project investigates existing approaches to the automation of email processing and methodologies for building artificial intelligence systems; in particular, it conducts a comparative analysis of commercial solution and text classification methods. Based on the results of the analysis, a specialised system for the automated processing of applicants' emails was developed using the RAG architecture, which implements semantic classification of incoming emails using the all-MiniLM-L6-v2 model and the ChromaDB vector database, recognition of attached document types using the CLIP ViT-B/32 multimodal model, verification of qualified electronic signatures via the id.gov.ua portal, and automatic generation of draft replies using a large language model.

Keywords: email processing automation, artificial intelligence, RAG, semantic classification, all-MiniLM-L6-v2, ChromaDB, vector database, CLIP ViT-B/32, qualified electronic signature, large language model.

ОПИС АЛЬБОМУ

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: *«Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG»*

справки	Форма	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система автоматизації	4		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система автоматизації	72		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система автоматизації	1		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Структура системи			
	А4	***ІАЛЦ.467200.005 Д2***	Система автоматизації	1		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Діаграма класів			

Зм	*Лист*	*№ докум.*	*Підп*	*Дата*	***ІАЛЦ.467200.001 ОА***		
Розроб		Гундарев Р.Д.			*Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG* Опис альбому	Літ.	Аркуш 1 / Аркушів 2
Перев		Туганських О.А					
Норм		Міщенко Л. Д.				***КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21***	
Затв		Волокита А. М.					

справки	Форма	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
	А4	***ІАЛЦ.467200.006 Д3***	Система автоматизації	1		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Система автоматизації	25		
			обробки листів абітурієнтів			
			з використанням штучного			
			інтелекту на основі			
			архітектури RAG			
			Текст програмного коду			

Зм	*Лист*	*№ докум.*	*Підп*	*Дата*	***ІАЛЦ.467200.001 ОА***		
Розроб		Гундарев Р.Д.			*Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG* Опис альбому	Літ.	Аркуш 2 / Аркушів 2
Перев		Туганських О.А					
Норм		Міщенко Л. Д.				***КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21***	
Затв		Волокита А. М.					

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: *«Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG»*

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є здійснення процесу обробки та аналізу електронних листів абітурієнтів, аналізу та менеджменту великих обсягів листів від абітурієнтів, підтримка автоматизації роботи з поштою шляхом використання протоколу IMAP для читання, інтерпретування та категоризації вхідних листів та можливість відправляти автоматичні відповіді, допомога оператору завдяки автоматизації операцій над документами, формування пакетів документів у необхідному форматі та аналіз правильності підписів, збір статистики щодо якості листів та вкладень, для подальшого аналізу результатів роботи.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки системи автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», яке було затверджено факультетом інформатики та обчислювальної техніки кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою бакалаврського проєкту є підвищення оперативності та точності опрацювання звернень вступників, а також зниження навантаження на операторів приймальної комісії шляхом автоматизації процесів семантичної класифікації листів та верифікації цифрових підписів у супутніх документах

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Простий і інтуїтивно зрозумілий інтерфейс веб-додатка для зручної взаємодії оператора з компонентами системи.
- Надання можливості користувачам отримувати доступ до пошти через протокол IMAP для автоматичного зчитування та аналізу листів.
- Забезпечення інтерактивної взаємодії оператора з інструментами автоматизації роботи з документами та аналізу підписів.
- Надання користувачам можливості збору статистики щодо звернень абітурієнтів для аналізу в межах Data Science.
- Надання вичерпної та зрозумілої документації, включаючи пояснювальну записку та опис алгоритмів архітектури RAG.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Visual Studio 2017 IDE версії або вище.
- Python 3.13.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i5 або аналог, AVX.
- Вільний дисковий простір не менше ніж 10 ГБ.
- RAM не менше ніж 8 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Терміни виконання
Затвердження теми проєкту	17.01.2026
Вивчення та аналіз завдання	30.01.2026
Розробка архітектури та загальної структури системи	19.02.2026
Розробка структур окремих підсистем	10.03.2026
Програмна реалізація системи	20.05.2026
Виправлення помилок	21.05.2026
Оформлення пояснювальної записки	25.05.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG*»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AI	(Artificial Intelligence) Штучний інтелект
API	(Application Programming Interface) Прикладний програмний інтерфейс
ASGI	(Asynchronous Server Gateway Interface) Асинхронний серверний шлюзовий інтерфейс
CLIP	(Contrastive Language-Image Pretraining) Контрастивне попереднє навчання мови-зображень
CRUD	(Create, Read, Update, Delete) Створити, прочитати, оновити, видалити
FAISS	(Facebook AI Similarity Search) Пошук подібності від Facebook AI
IMAP	(Internet Message Access Protocol) Протокол доступу до інтернет-повідомлень
LLM	(Large Language Model) Велика мовна модель
PDF	(Portable Document Format) Портативний формат документів
RAM	(Random Acces Memory) Оперативна пам’ять
RAG	(Retrieval-Augmented Generation) Генерація доповнена пошуком
SaaS	(Software as a Service) Програмне забезпечення як послуга
SMTP	(Simple Mail Transfer Protocol) Простий протокол пересилання пошти
SSE	(Server-Sent Events) Події, надіслані сервером
UID	(Unique Identifier) Унікальний ідентифікатор
ViT	(Vision Transformer) Візуальний трансформер
ГБ	Гігабайт
ГП	Графічний процесор
ЗВО	Заклад вищої освіти

КЕП	Кваліфікований електронний підпис
ОП	Оперативна пам’ять
ОС	Операційна система
ЦП	Центральний процесор

ВСТУП

Станом на зараз, та з кожною новою вступною кампанією у системи вищої освіти в Україні ми можемо спостерігати збільшення необхідних об’ємів інформації та взаємодії між вступниками та приймальними комісіями закладів вищої освіти. Головним каналом здійснення комунікації за часів воєнного стану стала електронна пошта, проте вона не тільки не полегшує взаємодію а і ускладнює її через необхідність дотримання норм вступу в заклади вищої освіти та верифікації вступників дистанційно, шляхом валідації документів на кваліфікований електронний підпис (КЕП) через окремі державні сервіси. Таким чином, збільшується час на ручну обробку великого переліку листів та абітурієнтів, зважаючи також на популярність Київського Політехнічного Інституту в Україні серед вступників. Листи абітурієнтів містять різноманітну інформацію: запитання щодо правил вступу, прикріплені документи, сканкопії паспортів та підписані заяви, що значно ускладнює їх систематизацію та своєчасне опрацювання.

Традиційні підходи до обробки вхідної пошти: ручне сортування, фільтри за ключовими словами або правила на основі регулярних виразів, є недостатньо ефективними та неробочими у випадку семантично різноманітних запитів, написаних у вільній формі. Постійне збільшення кількості вступників посилюють потребу у швидкому, надійному й автоматизованому інструменті для обробки та класифікації листів.

Актуальність теми зумовлена необхідністю автоматизації та прискорення операцій приймальних комісій закладів вищої оствіти (ЗВО) в умовах зростаючого навантаження та обмеженості ресурсів і часових рамок вступної кампанії. Використання та впровадження сучасних технологій штучного інтелекту, зокрема архітектури Retrieval-Augmented Generation (RAG), дозволяє суттєво полегшити роботу та час на обробку електронних листів, автоматизувати перевірку документів для прийняття рішень.

Мета дипломного проєкту: створення системи автоматизованої обробки електронних листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG, яка забезпечує класифікацію листів за тематичними категоріями, верифікацію вкладених документів та підтримку операторів приймальної комісії в процесі опрацювання вхідної кореспонденції.

Об'єкт дипломного проєкту: процес обробки та аналізу електронних листів абітурієнтів засобами протоколу Internet Message Access Protocol (IMAP) та інтелектуальних методів класифікації.

Предмет дипломного проєкту: методи побудови та оптимізації компонентів архітектури RAG для покращення якості автоматизованої класифікації та обробки листів абітурієнтів.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО РЕАЛІЗАЦІЇ АВТОМАТИЗАЦІЇ ОБРОБКИ ЛИСТІВ АБІТУРІЄНТІВ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ НА ОСНОВІ АРХІТЕКТУРИ RAG.

Вибір технологічного підходу до автоматизованої класифікації електронних листів абітурієнтів потребує критичного огляду наявних рішень у контексті конкретних умов вступної кампанії: довільної форми звернень, прикріплених сканкопій документів та необхідності інтеграції через протокол IMAP.

Оскільки основою розроблюваної системи є архітектура RAG, окремої уваги потребує дослідження методів побудови її ключових компонентів: моделей векторного ембедингу для україномовних текстів та механізмів налаштування порогів семантичної подібності, що на відміну від регулярних виразів чи навчання з учителем дозволяє класифікувати нові типи звернень без перенавчання моделі.

1.1. Аналіз існуючих технологічних рішень з реалізації автоматизації обробки листів абітурієнтів.

Існуючі поштові сервіси орієнтовані на корпоративний документообіг і вирішують категоризацію листів на рівні широких класів: «Основні», «Реклама», спам; не забезпечуючи семантичної класифікації за довільними тематичними правилами. Нижче проведено порівняльний аналіз п'яти комерційних рішень для визначення їх функціональних обмежень та обґрунтування доцільності розробки власної системи на основі RAG.

1.1.1. Gmail Smart Labels

Gmail Smart Labels — вбудована функція Google Gmail, що автоматично категоризує вхідні листи на фіксовані класи: «Основні», «Соціальні мережі», «Акції», «Сповіщення», «Форуми» засобами машинного навчання на стороні серверів Google без додаткового налаштування[1].

Головною перевагою є повна інтеграція з екосистемою Gmail та класифікація в реальному часі безкоштовно для всіх користувачів.

Проте фіксований набір категорій унеможливлює створення власних правил класифікації, алгоритм є «чорною скринькою» без семантичного пошуку, а обробка даних виключно на серверах Google робить рішення непридатним для задач, де критичними є конфіденційність та гнучкість налаштувань.

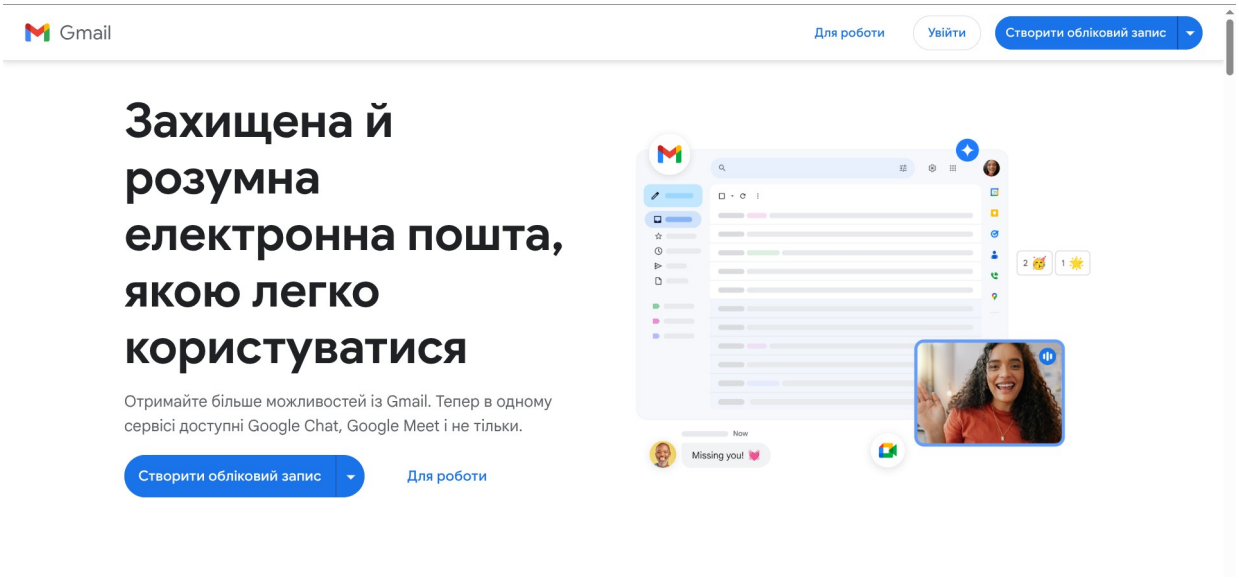


Рисунок 1.1 − Сайт сервісу Gmail від Google

1.1.2. Microsoft Outlook / Microsoft 365 Focused Inbox

Microsoft Outlook — один з найпоширеніших поштових клієнтів у корпоративному середовищі, функція «Focused Inbox» якого автоматично розділяє листи на «Важливі» та «Інші» засобами машинного навчання Microsoft, а в рамках Microsoft 365 доступні розширені правила фільтрації та автоматизація через Power Automate[2].

Перевагами є глибока інтеграція з корпоративною екосистемою, підтримка IMAP/Exchange та корпоративний рівень безпеки.

Однак класифікація зводиться до бінарного поділу без повноцінного семантичного пошуку, розширена автоматизація потребує платної підписки M365, а відсутність RAG-підходу унеможливлює інтелектуальну класифікацію за складними предметними запитами.

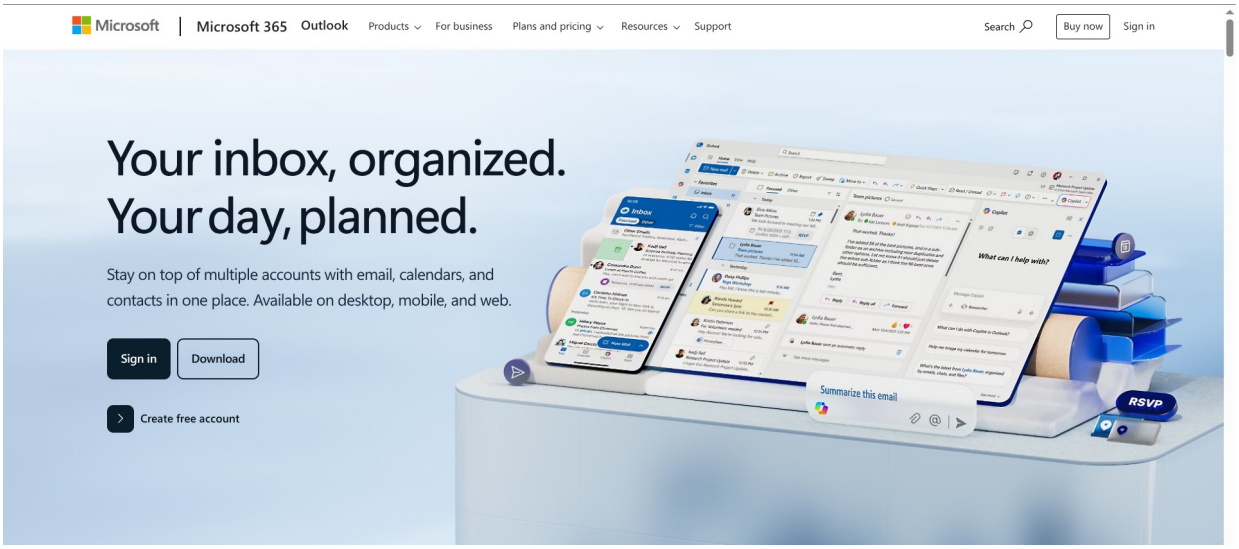


Рисунок 1.2 − Сайт сервісу Microsoft Outlook

1.1.3. SaneBox

SaneBox — хмарний Software as a Service (SaaS)-сервіс для управління поштою, що аналізує поведінку користувача та автоматично переміщує малозначущі листи до спеціальних тек (@SaneLater, @SaneBlackHole), працюючи через IMAP з будь-яким поштовим провайдером без встановлення програмного забезпечення[3].

Перевагою є універсальність та адаптивне навчання на основі дій користувача без ручного налаштування правил.

Проте класифікація базується виключно на поведінкових факторах без аналізу семантичного змісту, сервіс є платним від $7/міс, надає доступ до поштової скриньки третій стороні, що створює ризики конфіденційності, і не підходить для вирішення складних організаційних задач на кшталт автоматизованої класифікації офіційних заяв за їх змістом.

Рисунок 1.3 − Сайт сервісу SaneBox

1.1.4. SuperhumanAI

Superhuman — преміум поштовий клієнт для Gmail та Outlook з функцією «Superhuman AI», що автоматично формує короткі резюме листів, пропонує авторозгортання відповідей та визначає пріоритетність вхідних листів засобами великих мовних моделей[4].

Перевагами є використання потужних Large Language Model (LLM) для генерації резюме та відповідей, надшвидкий інтерфейс з підтримкою клавіатурних скорочень та інтеграція з Gmail і Outlook без втрати їх базової функціональності.

Проте висока вартість підписки $30/міс, відсутність підтримки власних правил класифікації на основі RAG, неможливість аналізувати вміст вкладених документів та закрита інфраструктура з обробкою даних на зовнішніх серверах роблять сервіс непридатним для специфічних задач вступної кампанії.

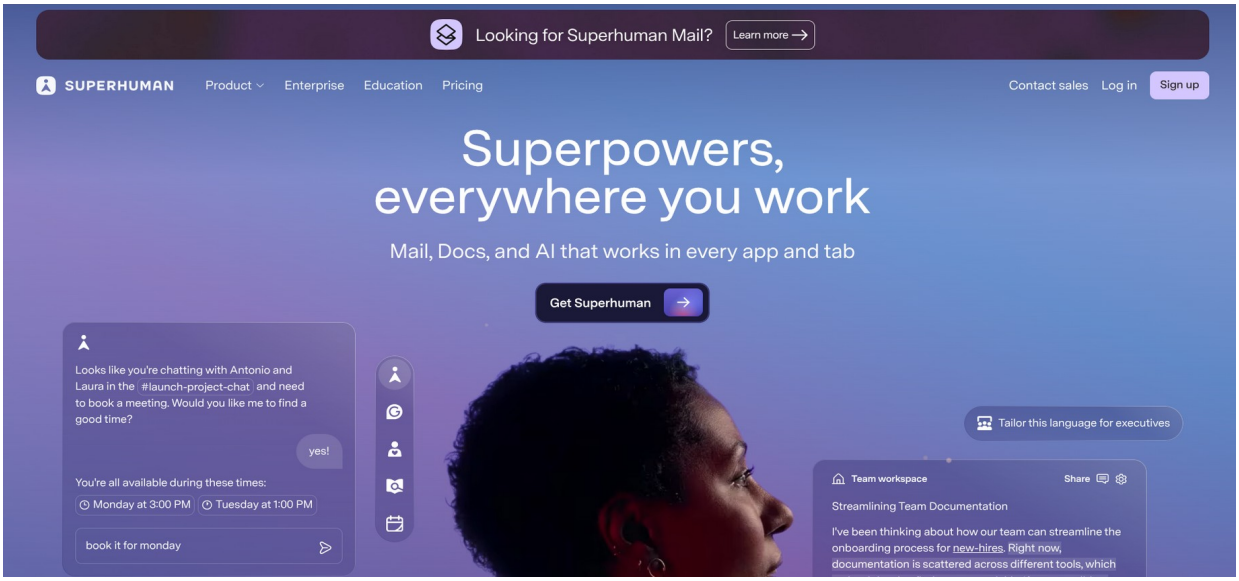


Рисунок 1.4 − Сайт сервісу SuperhumanAI

1.1.5. Spark by Readdle

Spark — поштовий клієнт від Readdle з вбудованим Artificial Inteligence (AI)-асистентом на базі OpenAI GPT, що пропонує інтелектуальне сортування листів, генерацію відповідей, резюмування листування та функцію «Smart Inbox» для автоматичного групування повідомлень за типом[5].

Перевагами є наявність безкоштовної версії з базовим AI-функціоналом, підтримка одночасної роботи з декількома поштовими акаунтами через Universal IMAP та інструменти командної роботи.

Проте робота AI-функцій залежить від зовнішнього OpenAI Application Programming Interface (API), відсутня підтримка RAG-архітектури та векторного пошуку, класифікаційні правила не дозволяють задавати довільні семантичні запити, а аналіз вмісту вкладень не підтримується.

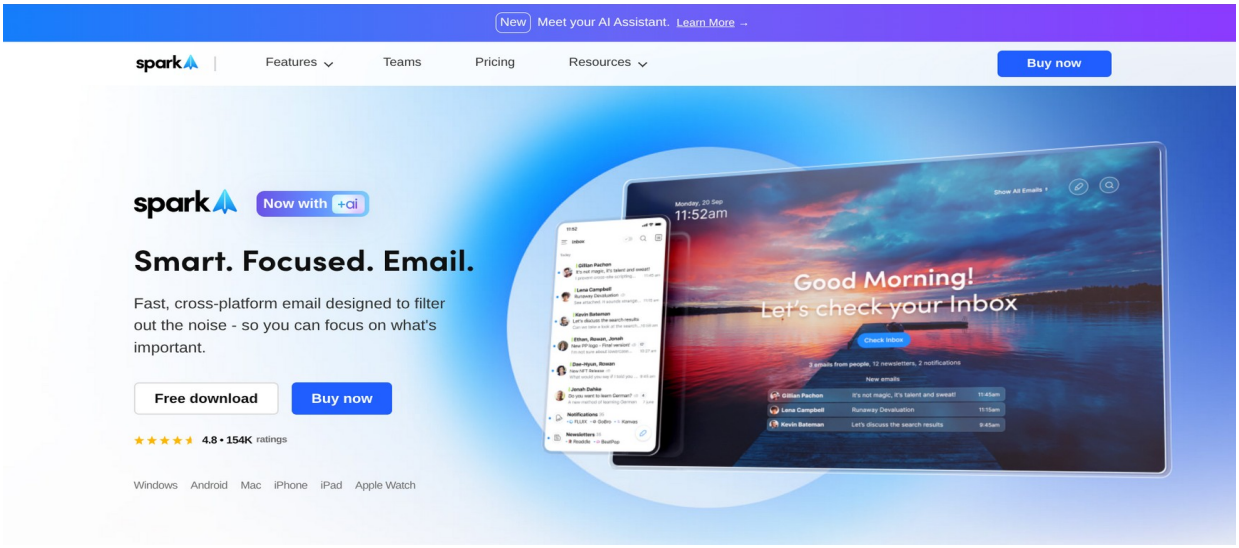


Рисунок 1.5 − Сайт сервісу Spark

На основі аналізу представлених технологічних рішень, можна дійти висновку, що сучасні поштові сервіси є переважно універсальними SaaS-продуктами із закритим кодом та фіксованою логікою, які не забезпечують необхідного рівня гнучкості та безпеки для специфічних задач.

Жодна з розглянутих систем не підтримує інноваційний RAG-підхід для класифікації через векторні бази даних, не має інструментів комп'ютерного зору для аналізу вкладених документів та не інтегрована з стандартами цифрового підпису, що робить їх непридатними для автоматизації обробки електронних листів під час вступної кампанії.

Таким чином, розробка власної системи, яка поєднує в собі семантичний пошук та валідацію документів вступників на підпис, дозволяє створити унікальний продукт який справляється з поставленою спеціалізованою задачою.

Таблиця 1.1. − Порівняльна таблиця існуючих технологічних рішень

Критерій	Gmail Smart Labels	MS Outlook	SaneBox	Superhuman AI	Spark AI
Інфраструктура та локалізація	Закритий SaaS-сервіс	Хмарна інфраструктура Microsoft	Хмарний SaaS-сервіс	Закрита зовнішня інфраструктура	Зовнішній сервіс OpenAI API
Гнучкість категорій	Фіксовані категорії	Ручні правила та теги	Адаптивне навчання на поведінці	Фіксована логіка LLM	Обмежене AI-групування
ШІ-генерація резюме/відповіді	-	+	-	+	-
Семантична класифікація	Фіксовані категорії	Бінарна	Лише поведінкова	LLM-пріоритетність	AI-групування
Тип інтеграції	Вбудована в Gmail	IMAP / Exchange	Universal IMAP	Gmail / Outlook	Universal IMAP
Конфіденційність	Дані на серверах Google	Дані в хмарі Microsoft	Доступ третіх сторін	Обробка на зовнішніх серверах	Залежність від OpenAI API
Вартість	Безкоштовно	Платна підписка M365	Від $7/міс	$30/міс	Безкоштовно / Платно

1.2. Аналіз існуючих методологій до створення та оптимізації штучного інтелекту на основі архітектури RAG

Архітектура RAG функціонує як ланцюжок взаємозалежних компонентів, де якість кінцевого результату визначається найслабшою ланкою, тому вибір кожного елементу потребує окремого обґрунтування в контексті конкретної задачі класифікації україномовних листів абітурієнтів в умовах локального розгортання без графічного процесора (ГП)[6].

Відправною точкою конвеєра є перетворення тексту теми листа на числовий вектор. Теми абітурієнтів мають характерну структуру: прізвище, форма фінансування, назва спеціальності; і саме цей структурний патерн визначає вибір моделі ембедингу.

Серед моделей сімейства sentence-transformers найменша all-MiniLM-L6-v2 розміром 22 МБ з часом інференсу ~5 мс на центральному процесорі (ЦП), попри лише часткову підтримку української мови ефективно обробляє такі теми завдяки тому, що власні назви та терміни спеціальностей транслітеруються або збігаються з англійськими відповідниками[7, 8].

Більші альтернативи, paraphrase-multilingual-MiniLM-L12-v2 (118 МБ) з повноцінною багатомовною підтримкою та LaBSE від Google (471 МБ) з найвищою якістю для довільних україномовних текстів забезпечують кращу точність для вільних формулювань, однак збільшені вимоги до пам'яті та часу завантаження не виправдані для структурованих коротких тем листів вступної кампанії[9].

Отримані вектори потрапляють до векторної бази даних, від можливостей якої залежить не лише швидкість пошуку, а й зручність інтеграції з рештою системи.

Facebook AI Similarity Search (FAISS) від Facebook демонструє найвищу продуктивність серед розглянутих рішень, проте зберігає індекс лише в оперативній пам'яті та потребує явної серіалізації при кожному завершенні роботи, а відсутність фільтрації за метаданими ускладнює ідентифікацію конкретних листів за IMAP Unique Identifier (UID)[10].

Qdrant вирішує обидві проблеми, але потребує окремого серверного процесу, що збільшує кількість залежностей системи.

ChromaDB займає оптимальну позицію: вбудована персистентність на диску, нативна інтеграція з sentence-transformers та фільтрація за метаданими дозволяють зберігати IMAP UID безпосередньо разом із вектором, забезпечуючи однозначний зв'язок між векторним сховищем та реляційною базою SQLite без додаткових таблиць відповідності[11].

Знайдені вектори потрібно порівняти з еталонними запитами категорій, і тут постає питання вибору міри подібності.

Косинусна відстань вимірює кут між векторами незалежно від їхньої довжини, евклідова — геометричну відстань між точками у просторі ембедингів[12].

$$d_{\cos}(\vec{u},\vec{v}) = 1 - \frac{\vec{u}\cdot\vec{v}}{norm\,\vec{u}\cdot norm\,\vec{v}} \qquad (1.1)$$

Де – $\vec{u}$ та $\vec{v}$ — вектори ембедингів двох текстів,

norm – евклідова норма вектора.

Для нормалізованих векторів sentence-transformers обидві метрики є математично еквівалентними, проте моделі цього сімейства навчаються саме з оптимізацією косинусної подібності, що робить її природним вибором[8]. Базове порогове значення 0.5 збалансовує точність та повноту для коротких структурованих текстів і може регулюватися оператором.

Паралельно з семантичною класифікацією система має розпізнавати типи вкладених документів: паспортів, військових квитків, заяв, без можливості навчати окрему модель на розміченій вибірці зображень.

Мультимодальна модель Contrastive Language-Image Pretraining (CLIP) вирішує цю задачу через zero-shot класифікацію: вона зіставляє векторне представлення зображення з текстовими підказками для кожного класу документа та обирає найближчий. З трьох конфігурацій — Vision Transformer (ViT)-B/32 (338 МБ), ViT-L/14 (890 МБ) та RN50 (224 МБ) — архітектура Vision Transformer у ViT-B/32 забезпечує ефективніший аналіз структурованих документів формату А4 порівняно з ResNet-50, тоді як збільшення моделі до ViT-L/14 майже втричі підвищує вимоги до пам'яті без суттєвого приросту якості для задачі розрізнення базових типів офіційних документів[13].

Завершальним етапом конвеєра є генерація персоналізованої чернетки відповіді на основі зібраних технічних міток.

Локальне розгортання через Ollama гарантує повну конфіденційність, однак квантизовані моделі потребують щонайменше 8 гігабайт (ГБ) оперативної пам’яті (ОП) та суттєво уповільнюють генерацію на CPU, що є критичним в умовах пікового навантаження вступної кампанії.

Хмарні сервіси Gemini та Groq позбавлені цих обмежень: безкоштовні ліміти обох платформ перевищують прогнозовані обсяги запитів кампанії, а якість генерації достатня для формування офіційних чернеток українською мовою.

Принциповим є те, що до зовнішнього API передаються виключно знеособлені технічні мітки: «паспорт: знайдено», «КЕП: валідний», без жодних персональних даних абітурієнта, що повністю знімає питання конфіденційності при використанні хмарного підходу.

1.3. Формалізація задачі та постановка часткових задач проєктування.

За результатами аналізу встановлено відсутність готових рішень для автоматизованої обробки листів абітурієнтів: наявні поштові платформи не підтримують семантичну класифікацію через векторні бази даних, аналіз вкладених документів засобами комп'ютерного зору та верифікацію КЕП за вітчизняними стандартами.

Предметом проєктування є система, що отримує листи через IMAP, класифікує їх за визначеними оператором категоріями, верифікує вкладені документи з перевіркою цифрового підпису та генерує чернетку відповіді на основі отриманих міток.

На підставі технічного завдання сформульовано вимоги до системи. Функціональні вимоги:

- підключення до поштової скриньки через IMAP та інкрементальне завантаження заголовків;
- отримання повного тексту та вкладень на вимогу оператора; векторне ембедування тем моделлю all-MiniLM-L6-v2 з індексацією у ChromaDB;
- семантична класифікація за правилами оператора з регульованим порогом косинусної подібності;
- Create, Read, Update, Delete (CRUD)-управління правилами через REST API та вебінтерфейс; класифікація Portable Document Format (PDF)-вкладень моделлю CLIP ViT-B/32;
- верифікація КЕП через портал id.gov.ua; генерація чернетки відповіді хмарним LLM API.

Нефункціональні вимоги:

- локальне розгортання всіх компонентів обробки персональних даних із передачею до зовнішніх API виключно знеособлених технічних міток;
- функціонування без GPU;
- модульна архітектура з розділеними сервісами бекенду та фронтенду через REST API.

На підставі сформульованих вимог визначено сім часткових задач проєктування:

1. реалізація IMAP-модуля з інкрементальною синхронізацією та lazy loading вкладень;
2. реалізація RAG-конвеєра класифікації з ембедуванням тем та пошуком найближчих сусідів за косинусною відстанню;
3. реалізація модуля верифікації документів CLIP-класифікація PDF-вкладень та перевірка .p7s-підписів через id.gov.ua засобами Playwright;
4. розробка FastAPI-бекенду з маршрутами для синхронізації, класифікації та CRUD правил;
5. реалізація RAG-оркестратора для генерації чернеток через Google Gemini або Groq з обов'язковою валідацією оператором перед відправкою;
6. розробка Streamlit-інтерфейсу оператора з переглядом листів, статусом верифікації та сторінкою класифікованих звернень.
7. проєктування специфікації REST API;

Визначені задачі формують повну специфікацію системи та є підставою для переходу до детального проєктування у наступному розділі.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі проаналізовано існуючі комерційні рішення з автоматизації обробки електронних листів та досліджено методології побудови ключових компонентів RAG-архітектури, що є підґрунтям для проєктних рішень наступного розділу.

Порівняльний аналіз п'яти комерційних продуктів — Gmail Smart Labels, Microsoft Outlook Focused Inbox, SaneBox, Superhuman AI та Spark показав, що жоден із них не підтримує семантичної класифікації через векторні бази даних, не аналізує вміст PDF-вкладень засобами комп'ютерного зору та не верифікує кваліфікований електронний підпис за вітчизняними стандартами. Усі розглянуті рішення залежать від закритої хмарної інфраструктури, що унеможливлює їх застосування в умовах вступної кампанії університету через вимоги до конфіденційності персональних даних абітурієнтів та необхідність локального розгортання.

За результатами аналізу методологій побудови RAG-архітектури визначено оптимальний набір компонентів системи: модель all-MiniLM-L6-v2 для векторного ембедингу тем листів завдяки мінімальним обчислювальним вимогам та сумісності зі структурованим форматом тем абітурієнтів; ChromaDB як векторне сховище з вбудованою персистентністю та підтримкою фільтрації за метаданими без окремого серверного процесу; косинусна відстань із порогом 0.5 як стандартна міра подібності для моделей сімейства sentence-transformers; CLIP ViT-B/32 для zero-shot класифікації вкладених документів без необхідності розміченої навчальної вибірки; хмарне API для генерації чернеток відповідей із передачею виключно знеособлених технічних міток, що виключає ризики для конфіденційності персональних даних абітурієнтів.

РОЗДІЛ 2
РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ З АВТОМАТИЗАЦІЇ ОБРОБКИ ЛИСТІВ АБІТУРІЄНТІВ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ НА ОСНОВІ АРХІТЕКТУРИ RAG

2.1. Розробка програмної компоненти системи з автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG.

Розробка програмної системи вимагала комплексного підходу, що поєднує методи обробки природної мови, комп'ютерного зору та автоматизованої веб-взаємодії.

Центральним принципом побудови системи є архітектура Retrieval-Augmented Generation (RAG), реалізована через поєднання методів семантичного пошуку та генеративного аналізу. Замість автономної генерації тексту, система використовує векторну базу даних для пошуку еталонних категорій та контексту, які разом із технічними мітками (лейблами) верифікації документів передаються до великої мовної моделі. Це дозволяє LLM виконувати роль інтелектуального судді: на основі отриманого контексту модель класифікує звернення та формує обґрунтовану чернетку відповіді. Такий підхід забезпечує детермінованість результатів, оскільки логіка прийняття рішень базується на чітко визначених критеріях верифікації, а генеративні можливості ШІ використовуються для фінальної агрегації даних та персоналізації комунікації.

2.1.1. Архітектура системи використання штучного інтелекту на основі архітектури RAG.

Архітектура RAG доповнює параметричні знання мовної моделі контекстом із зовнішніх баз даних за допомогою трьох компонентів: векторного індексу, механізму семантичного пошуку та генератора.

Оскільки обробка звернень абітурієнтів вимагає комплексного аналізу, у розробленій системі застосовано повноцінну архітектуру RAG. Велика мовна модель (LLM) виступає агрегатором: вона співставляє знайдені пошуком еталонні сценарії з результатами технічної верифікації (лейблами від моделей комп'ютерного зору та перевірки КЕП). Завдяки цьому система функціонує як інтелектуальний помічник, що класифікує заявки, обробляє нестандартні формулювання та аргументує рішення шляхом генерації персоналізованих чернеток відповідей.

З архітектурної точки зору систему реалізовано у вигляді двох незалежно розгорнутих Python-сервісів, що взаємодіють через HTTP REST API. Такий поділ забезпечує незалежне масштабування та оновлення кожного сервісу, чітке розмежування відповідальностей між бізнес-логікою та шаром представлення.

Серверна частина (backend) побудована на FastAPI та uvicorn. Вона координує роботу ШІ-компонентів: синхронізацію пошти (IMAP), управління базами даних (SQLite, ChromaDB), верифікацію документів (CLIP) та оркестрацію взаємодії з LLM для генерації відповідей. REST API має модульну структуру маршрутизаторів для управління листами та правилами класифікації. Архітектура реалізує принцип розділення відповідальностей (Separation of Concerns) через взаємодію маршрутів, сервісів та рівня доступу до даних.

Клієнтська частина (frontend) на базі Streamlit надає інтерфейс для управління обробкою звернень[21]. Оператор може динамічно налаштовувати параметри класифікації, ініціювати синхронізацію та переглядати результати верифікації документів у реальному часі. Особливу увагу приділено валідації згенерованих системою чернеток відповідей перед їх відправкою. Взаємодія сервісів через HTTP забезпечує гнучкість розгортання та незалежне масштабування компонентів.

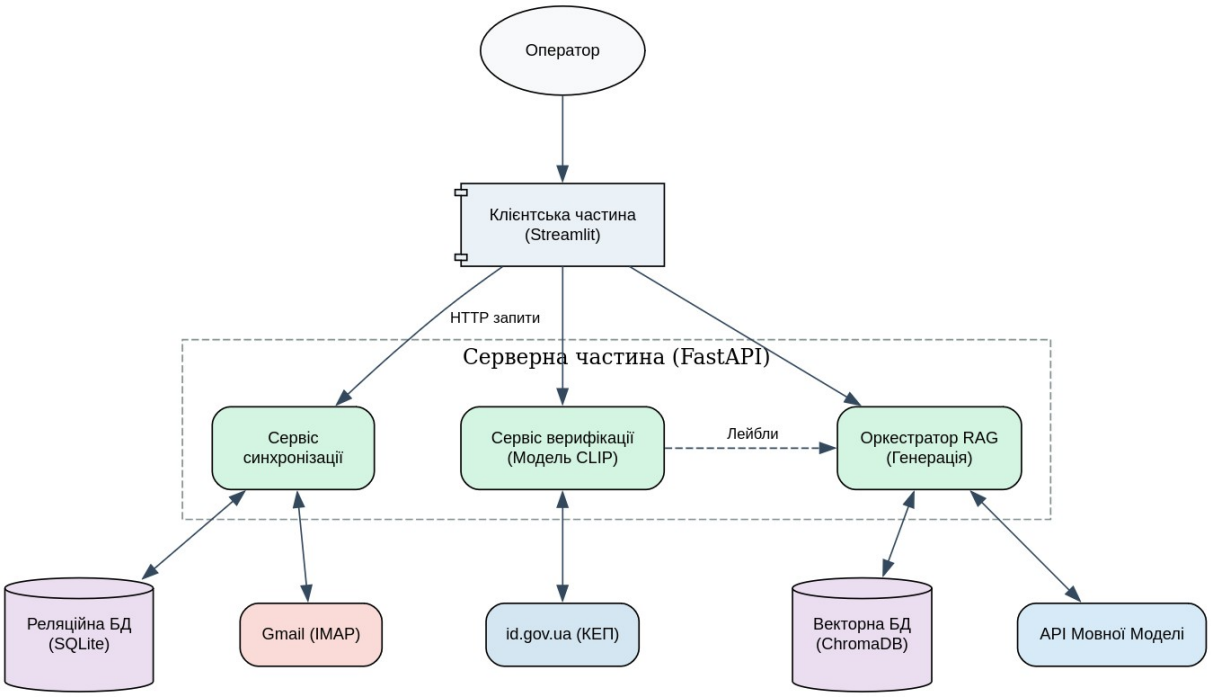


Рисунок 2.1 − Схема компонентної архітектури програмної системи

Конвеєр обробки даних у системі реалізовано як послідовний ланцюжок етапів, що трансформує вхідні листи абітурієнтів у структуровані результати за допомогою методів штучного інтелекту.

Загальний цикл обробки охоплює первинну синхронізацію пошти, семантичний пошук, мультимодальну верифікацію документів та фінальну генерацію чернеток відповідей. Такий підхід забезпечує комплексну автоматизацію рутинних завдань приймальної комісії без втрати точності.

Перший етап — синхронізація пошти. Оператор ініціює підключення до Gmail через IMAP. Система виконує вибіркову реплікацію: завантажує до 500 заголовків при першому запуску, а надалі лише нові листи (з UID > max). Метадані зберігаються в базі SQLite. Повний текст та вкладення завантажуються відкладено (lazy loading), лише за явним запитом оператора.

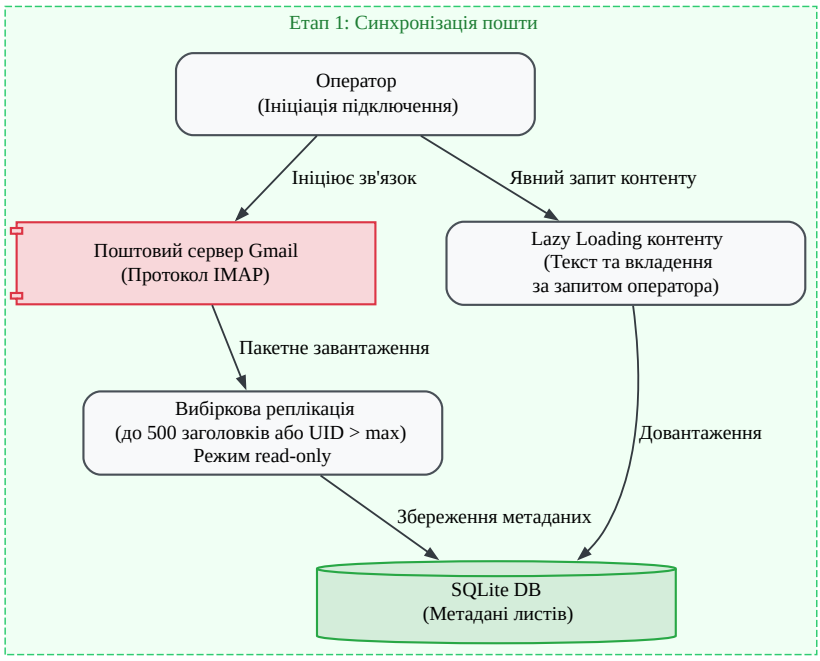


Рисунок 2.2 − Схема першого етапу синхронізації пошти

Другий етап — векторне індексування. Теми листів, що ще не мають векторного представлення у базі ChromaDB, передаються на обробку пакетами. Кожна тема перетворюється на 384-вимірний вектор за допомогою моделі all-MiniLM-L6-v2 архітектури Sentence Transformers. Модель оптимізована для вимірювання семантичної схожості речень і, незважаючи на орієнтацію на англійську мову, ефективно обробляє україномовні теми листів завдяки структурним патернам (прізвище, ім'я, тип вступу). Вектори зберігаються у постійній колекції разом із метаданими та UID листа як ідентифікатором.

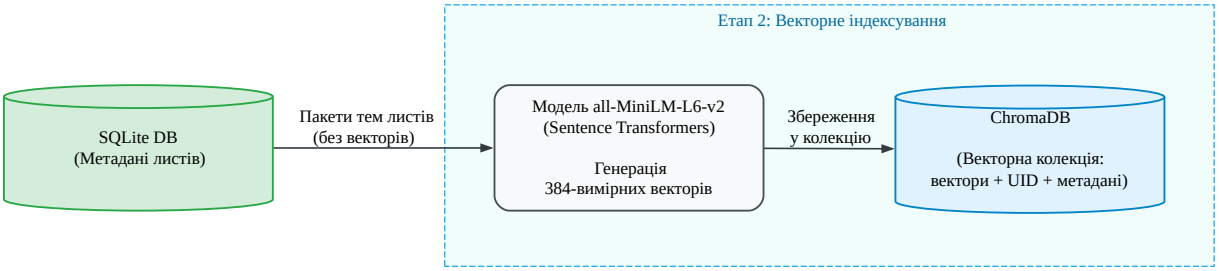


Рисунок 2.3 − Схема другого етапу векторного індексування

Третій етап — семантичний пошук та фільтрація (Retrieval). Система здійснює векторний пошук у ChromaDB за еталонним запитом. Знайдені листи оцінюються за косинусною відстанню: якщо показник < 0.5, лист маркується як релевантна заявка, інакше як загальний. Базовий поріг (0.5) можна змінювати в інтерфейсі. Результати разом із числовим значенням відстані зберігаються в базі для оцінки оператором.

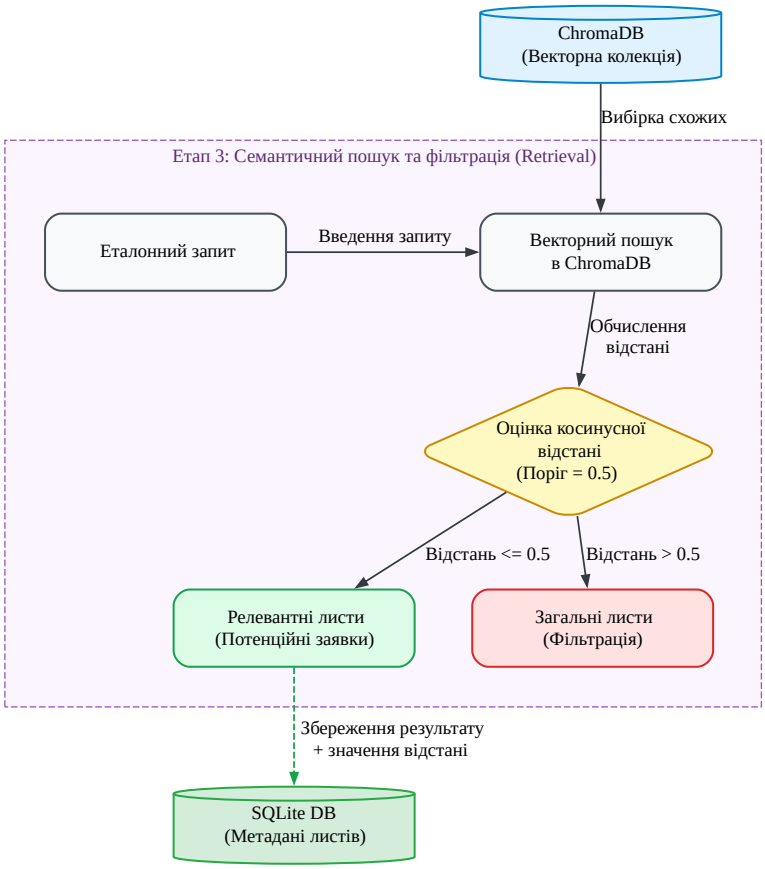


Рисунок 2.4 − Схема третього етапу семантичного пошуку та фільтрації

Четвертий етап — верифікація документів та екстракція міток. Для листів, що пройшли первинну фільтрацію, оператор ініціює процес перевірки вкладень. Цей процес є потоковим: результати передаються клієнту через механізм Server-Sent Events (SSE) у міру їх отримання. Верифікація охоплює класифікацію типів документів за допомогою локальної моделі комп'ютерного зору (CLIP) та перевірку автентичності файлів цифрового підпису (.p7s) через портал id.gov.ua. Результатом цього етапу є набір технічних міток (лейблів) для кожного вкладення.

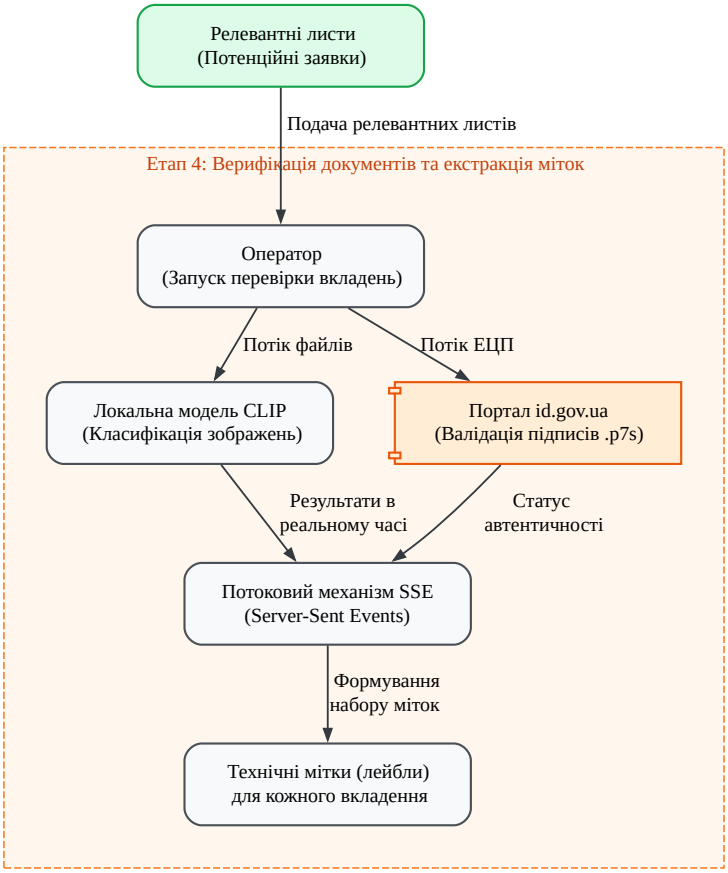


Рисунок 2.5 − Схема четвертого етапу верифікації документів та екстракції міток

П'ятий етап — інтелектуальний аналіз та генерація (Augmented Generation). На фінальному етапі система об'єднує текст релевантного листа з отриманими на попередньому кроці технічними мітками документів і передає цей контекст до великої мовної моделі (LLM). Модель виступає в ролі експертної системи: вона аналізує комплектність заявки та автоматично генерує чернетку зворотної відповіді абітурієнту (наприклад, підтвердження прийому документів або прохання дослати відсутній військовий квиток). Згенерований результат виводиться в інтерфейс для фінальної валідації оператором.

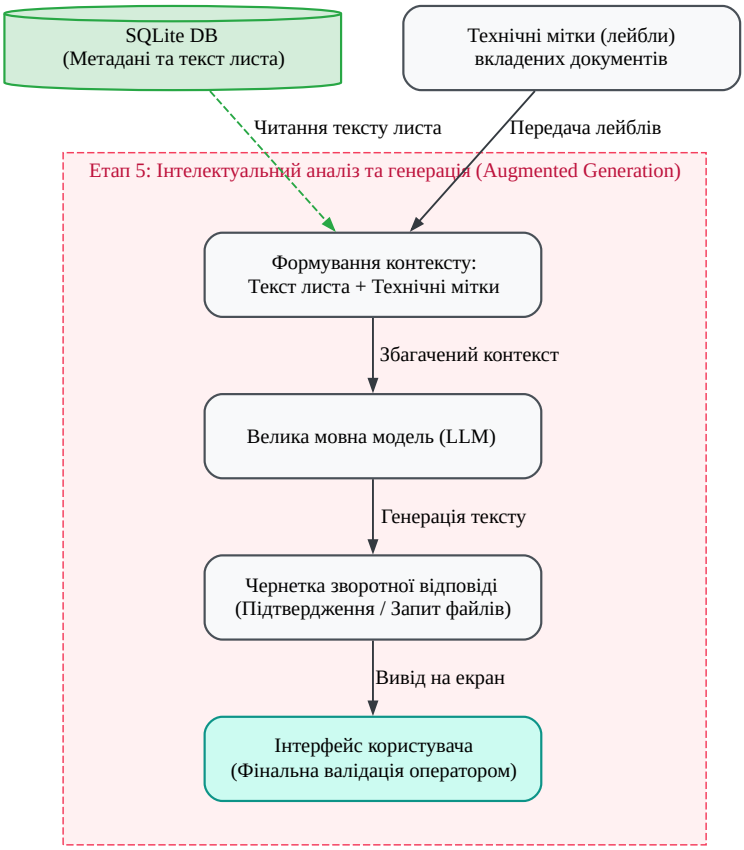


Рисунок 2.6 − Схема п’ятого етапу аналізу та генерації

Класифікація візуальних образів документів реалізована на основі мультимодальної архітектури CLIP (Contrastive Language-Image Pre-Training) моделі ViT-B-32, що пройшла попереднє навчання на наборі даних LAION-2B. Вибір даної моделі обумовлений її здатністю до Zero-Shot класифікації, що дозволяє ідентифікувати тип документа без необхідності донавчання на вузькоспеціалізованих вибірках. Для кожного цільового класу (паспорт, військовий квиток, довідка тощо) сформовано набір текстових дескрипторів, що описують візуальні характеристики об'єкта. Процес ідентифікації базується на обчисленні семантичної подібності між вектором зображення та набором промптів, де фінальний клас визначається як результат усередненої косинусної схожості[14].

$$\hat{c} = \text{argmax}_{c \in C} \frac{1}{|P_c|} \sum_{p \in P_c} \cos(\boldsymbol{v_I}, \ \boldsymbol{v_p}) \qquad (2.1)$$

Де – $\hat{c}$ передбачений клас документа,

C – множина всіх класів (паспорт, військовий квиток тощо),

P_c – набір текстових промптів для класу c,

v_I – вектор зображення (image embedding з CLIP),

v_p – вектор текстового промпту (text embedding з CLIP),

$\|\bullet\|$ – евклідова норма вектора.

Модуль верифікації цифрових підписів інтегрований з державним порталом id.gov.ua за допомогою інструментів автоматизації браузера Playwright[15, 16]. Це забезпечує автоматичну перевірку кваліфікованих електронних підписів (КЕП) на відповідність вимогам Закону України «Про електронні довірчі послуги» та стандарту ДСТУ ГОСТ 34.310. Алгоритм верифікації передбачає завантаження файлу відокремленого підпису (.p7s), ініціювання процедури перевірки та подальший парсинг результатів для отримання даних про підписувача[17; 18].

2.1.2. Інженерія вимог.

Для забезпечення всебічного опису системи автоматизації обробки листів абітурієнтів, вимоги до програмного продукту варто розглянути з двох стратегічних сторін: користувацької, технологічної. Такий підхід дозволяє чітко розмежувати функціональні обов'язки системи та технічні обмеження реалізації.

Користувацька група вимог визначає взаємодію представника приймальної комісії з інтерфейсом системи для виконання щоденних завдань:

- Управління доступом: система повинна забезпечувати автентифікацію через поштовий сервер Gmail за допомогою протоколу IMAP та паролів додатків. Інтерактивний контроль: оператор має володіти інструментами для ручного запуску процесів синхронізації, семантичної класифікації та верифікації документів .
- Гнучкість налаштувань: інтерфейс повинен містити регульований слайдер для встановлення порогу косинусної відстані, що дозволяє оператору самостійно визначати ступінь семантичної близькості для класифікації листів.
- Валідація результатів: система має відображати результати аналізу та автоматично згенеровані чернетки відповідей для їх фінального затвердження або редагування оператором перед відправкою.

Визначені користувацькі вимоги слугують основою для розробки сценаріїв використання системи, які описують функціональну логіку взаємодії оператора з компонентами RAG-архітектури. Основні функціональні вимоги до взаємодії оператора з компонентами RAG-архітектури формалізовано у вигляді компактної матриці користувацьких сценаріїв (таблиця 2.1).

Таблиця 2.1 – Матриця користувацьких сценаріїв системи

Назва сценарію	Вимоги користувача	Критерії роботи
Підключення до сервера	підключитися до Gmail через IMAP з паролем додатка для доступу до пошти.	Валідація облікових даних у реальному часі та збереження успішної сесії для наступних запусків.
Семантична класифікація	запускати класифікацію листів для автоматичного виявлення заявок на вступ.	Векторний пошук у ChromaDB за еталонним промптом із розрахунком косинусної відстані для кожної теми. Маркування листів як «релевантні» відбувається при відстані ≤ порогу , а результати аналізу зберігаються в SQLite.
Верифікація вкладень	ініціювати перевірку файлів для визначення їхнього типу та валідації КЕП.	Класифікація образів документів моделлю CLIP (паспорт, квиток тощо) та перевірка від'єднаних підписів .p7s через id.gov.ua. Реалізовано потокову передачу результатів на фронтенд через SSE та присвоєння технічних міток (лейблів) кожному файлу.
Редагування відповіді	переглянути, змінити та затвердити автоматично згенеровану чернетку відповіді.	Автоматична генерація чернетки відповіді оркестратором RAG на основі контексту й міток із відображенням в інтерактивному текстовому полі. Забезпечено можливість вільного редагування тексту оператором та повну заборону автономної відправки системою без його явного підтвердження.

Технічні вимоги визначають інженерні характеристики реалізації системи автоматизації обробки листів абітурієнтів: швидкодію, стійкість до відмов, захист даних та здатність до подальшого розвитку. Визначено три групи технічних вимог: продуктивність, зручність використання, а також супроводжуваність і переносимість.

Першою та базовою групою є вимоги до продуктивності, оскільки саме від швидкості реакції системи залежить практична цінність автоматизації в умовах пікового навантаження вступної кампанії. Векторна класифікація одного листа не повинна перевищувати 2 секунди за умови локального розгортання моделі, генерація чернетки відповіді засобами мовної моделі має завершуватися не довше ніж за 15 секунд з моменту передачі контексту, а синхронізація 500 листів у пакетному режимі повинна виконуватися не більше ніж за 3 хвилини за стабільного інтернет-з'єднання.

Система повинна також бути зручною для кінцевого користувача, оскільки складний інтерфейс нівелює переваги автоматизації, змушуючи оператора витрачати зайвий час на освоєння інструменту замість виконання фахових завдань. Інтерфейс системи повинен бути зрозумілим для співробітника приймальної комісії без спеціальної технічної підготовки, а всі ключові дії: синхронізація, класифікація, верифікація та генерація відповіді мають бути доступні не більше ніж у два кліки від головного екрана.

Нарешті, кодова база структурована за принципом розділення відповідальності, що дозволяє замінювати окремі компоненти без переписування суміжних модулів, а конфігураційні параметри виносяться у окремий файл. Система розгортається на корпоративному сервері університету під управлінням Linux без залежності від хмарних ресурсів і без виділеного графічного прискорювача.

2.2. Архітектурне проектування програмної компоненти.

Архітектурне проектування програмної компоненти системи автоматизації обробки листів абітурієнтів спрямоване на трансформацію визначених у попередньому розділі вимог у конкретні технічні рішення: структуру даних, специфікацію інтерфейсів взаємодії між модулями та детальну реалізацію кожного компонента конвеєру обробки. Загальний підхід ґрунтується на принципі модульності — система розбита на незалежні функціональні компоненти, кожен з яких відповідає за окремий етап обробки листа та може бути замінений або вдосконалений без впливу на решту системи. Взаємодія між компонентами здійснюється через чітко визначені інтерфейси REST API, що забезпечує слабку зв'язаність модулів та незалежність клієнтської частини від внутрішньої логіки серверних сервісів.

2.2.1. Розробка програмної компоненти системи з автоматизації обробки листів.

Серверна частина системи побудована на базі FastAPI — сучасного асинхронного веб-фреймворку для Python[20]. FastAPI забезпечує декларативне визначення маршрутів, автоматичну валідацію вхідних і вихідних даних на основі Pydantic-схем та генерацію OpenAPI-документації. Окремою перевагою є вбудована підтримка потокових відповідей типу Server-Sent Events, що використовується у системі для передачі прогресу верифікації документів у реальному часі. Для запуску застосунку використовується Asynchronous Server Gateway Interface (ASGI)-сервер Uvicorn, який реалізує асинхронну обробку запитів на основі asyncio, що дозволяє паралельно обслуговувати підключення до IMAP-сервера, векторної бази даних та браузерного автоматизатора без блокування.

База даних містить три таблиці.

Таблиця 2.2 — Структура таблиці emails

Поле	**Тип**	**Опис**
id	INTEGER PK	Автоінкрементний ідентифікатор
uid	TEXT UNIQUE	UID листа з IMAP-сервера
account_email	TEXT	Email-адреса облікового запису
from_nickname	TEXT	Відображуване ім'я відправника
from_email	TEXT	Email відправника
time	TEXT	Час отримання
subject	TEXT	Тема листа
text	TEXT	Повний текст/HTML тіло
attachments	TEXT	JSON-серіалізований список вкладень

Таблиця emails зберігає метадані кожного листа, отриманого з IMAP.

Таблиця 2.3 — Структура таблиці classification_rules

Поле	**Тип**	**Опис**
id	INTEGER PK	Автоінкрементний ідентифікатор
label	TEXT UNIQUE	Назва мітки
reference_query	TEXT	Еталонний запит для векторного пошуку
max_distance	FLOAT	Поріг косинусної відстані

Таблиця classification_rules зберігає правила класифікації, що визначаються користувачем.

Таблиця 2.4 — Структура таблиці email_labels

Поле	**Тип**	**Опис**
id	INTEGER PK	Автоінкрементний ідентифікатор
uid	TEXT FK	UID листа (посилання на emails.uid)
label	TEXT FK	Присвоєна мітка
distance	FLOAT	Косинусна відстань до еталонного запиту
synced_to_imap	INTEGER	Прапорець синхронізації з IMAP (0/1)

Таблиця email_labels фіксує результати класифікації. Пара (uid, label) має обмеження UNIQUE, що унеможливлює дублювання міток.

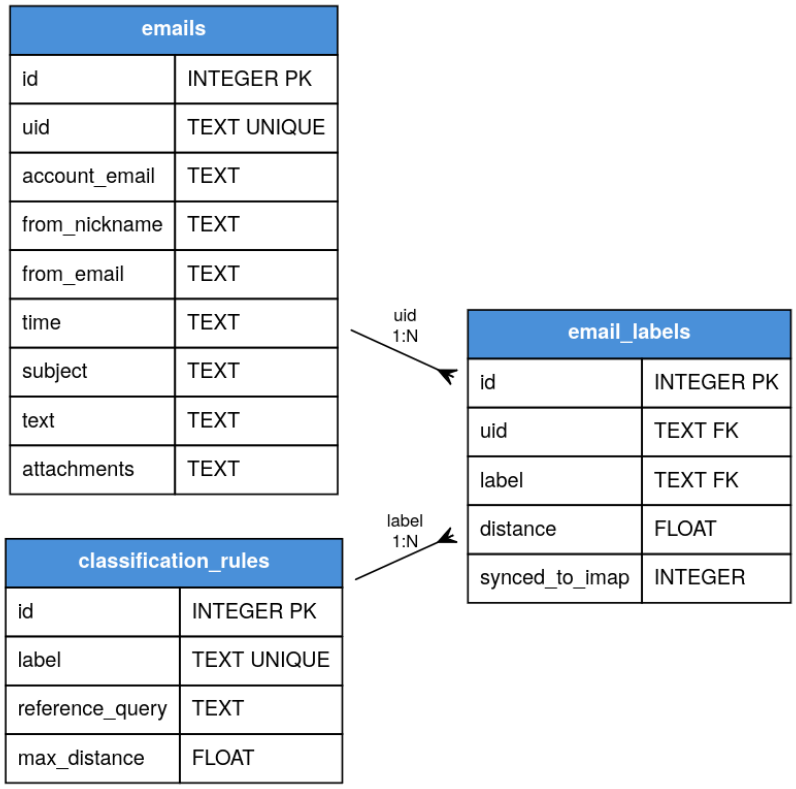


Рисунок 2.7 − ER-діаграма бази даних

ChromaDB зберігається на диску (data/chroma_db/) як persistent-клієнт. Система використовує єдину колекцію emails, де кожен документ це тема листа, векторизована моделлю all-MiniLM-L6-v2 (384-вимірні ембедінги, ONNX-формат).

Ідентифікатор документа формується як `{account_email}|{uid}`, що дозволяє фільтрувати результати запиту за конкретним обліковим записом через поле метаданих `account_email`. Метадані документа містять усі поля листа, що дозволяє отримати повну інформацію без звернення до SQLite.

Під час класифікації для кожного правила виконується запит `collection.query(query_texts=[reference_query])`, і листи, чия косинусна відстань не перевищує `max_distance`, отримують відповідну мітку.

SQLite та ChromaDB не мають зовнішніх ключів між собою, зв'язок відбувається через uid. SQLite є джерелом для метаданих та міток; ChromaDB виключно для семантичного пошуку[22]. Такий поділ дозволяє оновлювати ембедінги незалежно від структурованих даних.

Реалізацію модуля синхронізації пошти побудовано на основі бібліотеки imap_tools, яка забезпечує декларативний доступ до поштового сервера через протокол IMAP[19, 23]. Функція `sync_emails` реалізує інкрементальний режим завантаження: перед підключенням до сервера виконується запит до SQLite для визначення максимального UID вже збережених листів облікового запису. Якщо база містить попередньо завантажені листи, система формує IMAP-запит лише для нових повідомлень із діапазоном `uid > max_uid`, що унеможливлює повторне завантаження вже оброблених листів та суттєво зменшує трафік при повторних синхронізаціях.

За першого запуску, коли база порожня, завантажується 500 листів. Отримані метадані пакетно записуються до таблиці emails з використанням `INSERT OR IGNORE` для забезпечення ідемпотентності операції.

На рисунку 2.8 наведено фрагмент реалізації функції синхронізації.

```python
def sync_emails(email_address: str, imap_password: str, limit: int = 500) -> int:
    with get_connection() as conn:
        row = conn.execute(
            "SELECT MAX(CAST(uid AS INTEGER)) FROM emails WHERE account_email = ?",
            (email_address,),
        ).fetchone()
        max_uid = row[0] or 0

    with MailBox(GOOGLE_IMAP_SERVICE).login(
        email_address, imap_password, "INBOX"
    ) as mailbox:
        if max_uid > 0:
            messages = mailbox.fetch(
                AND(uid=f"{max_uid + 1}:*"),
                headers_only=True,
                bulk=100,
            )
        else:
            messages = mailbox.fetch(
                reverse=True, headers_only=True, limit=limit, bulk=100
            )

        empty_attachments = json.dumps([])
        rows_to_insert = []
        for msg in messages:
            nickname = msg.from_values.name if msg.from_values else ""
            email_addr = msg.from_values.email if msg.from_values else msg.from_
            subject = msg.subject or ""

            if not email_addr and not subject:
                continue

            rows_to_insert.append(
                (msg.uid, email_address, nickname, email_addr, msg.date_str, subject, empty_attachments)
            )

    if not rows_to_insert:
        return 0
```

Рисунок 2.8 − Приклад реалізації IMAP синхронізації

Модуль векторного індексування реалізовано через `chromadb.PersistentClient`, що забезпечує збереження колекції на диску без окремого серверного процесу.

Функція `add_emails_batch` виконує масове додавання векторів через upsert, зберігаючи тему листа як документ та повні метадані для подальшої фільтрації.

Функція `get_emails` реалізує семантичний пошук із фільтрацією за обліковим записом.

На рисунку 2.9 наведено реалізацію основних функцій модуля.

```python
def add_emails_batch(emails: list[dict]):
    if not emails:
        return
    account_email = emails[0].get("account_email", "")
    collection.upsert(
        documents=[e.get("subject", "") for e in emails],
        metadatas=[e for e in emails],
        ids=[_chroma_id(account_email, e["uid"]) for e in emails],
    )


def get_uids_for_account(account_email: str) -> set[str]:
    if not account_email:
        results = collection.get(include=[])
        return set(results.get("ids", []))
    try:
        results = collection.get(where={"account_email": account_email}, include=[])
        return {_strip_prefix(account_email, cid) for cid in results.get("ids", [])}
    except Exception:
        return set()
```

Рисунок 2.9 − Реалізація модуля векторного індексування та семантичного пошуку у ChromaDB

Семантичну класифікацію листів реалізовано у функції `classify_by_subject`, яка ітерує по правилах класифікації, визначених оператором у таблиці classification_rules.

Для кожного правила виконується векторний пошук у ChromaDB з еталонним запитом (`reference_query`) та отримуються UID листів разом із косинусними відстанями.

Алгоритм забезпечує однозначне присвоєння мітки: кожен UID отримує лише одну мітку правило якої зустрілося першим та відповідає порогу `max_distance`.

Листи, що не пройшли жодного правила, автоматично отримують мітку відхиленого з відстанню 1.0.

На рисунку 2.10 наведено реалізацію функції класифікації.

```python
def classify_by_subject(account_email: str) -> dict:
    rules = _get_rules()
    if not rules:
        return {"accepted": 0, "rejected": 0, "already_classified": 0}

    already = _already_classified_uids(account_email)
    matched: dict[str, tuple[str, float]] = {}
    skipped = 0

    for rule in rules:
        results = get_emails(
            query=rule["reference_query"], limit=5000, account_email=account_email
        )
        if not results or not results.get("distances"):
            continue
        for uid, distance in zip(results["ids"][0], results["distances"][0]):
            if uid in already:
                skipped += 1
                continue
            if uid in matched:
                continue
            if distance <= rule["max_distance"]:
                matched[uid] = (rule["label"], distance)

    all_uids = get_uids_for_account(account_email)
    rejected_uids = all_uids - matched.keys() - already

    accepted_rows = [(uid, label, dist) for uid, (label, dist) in matched.items()]
    rejected_rows = [(uid, SUBJECT_REJECTED, 1.0) for uid in rejected_uids]
    _insert_labels(accepted_rows + rejected_rows)

    return {
        "accepted": len(matched),
        "rejected": len(rejected_uids),
        "already_classified": skipped,
    }
```

Рисунок 2.10 − Реалізація модуля семантичної класифікації листів на основі векторного пошуку

Класифікацію вкладених документів реалізовано у функції `classify_attachment` на основі моделі CLIP ViT-B/32. Для кожного класу документів визначено набір текстових промптів, що описують його візуальні характеристики. Модель завантажується одноразово за першим зверненням, після чого обчислюється косинусна подібність між вектором ображення та векторами всіх промптів, клас із найвищою усередненою подібністю вважається результатом класифікації.

На рисунку 2.11 наведено реалізацію функції класифікації вкладень.

```python
def classify_attachment(file_path: str | Path) -> dict:

    file_path = Path(file_path)
    img = _load_image(file_path)
    if img is None:
        return {"label": "other", "confidence": 0.0, "scores": {}}

    _load_model()

    with torch.no_grad():
        img_tensor = _preprocess(img).unsqueeze(0).to(_device)
        image_feats = _model.encode_image(img_tensor)
        image_feats = image_feats / image_feats.norm(dim=-1, keepdim=True)
        sims = (image_feats @ _text_features.T).squeeze(0)


    scores: dict[str, float] = {}
    idx = 0
    for key in CLASS_KEYS:
        n = len(CLASS_PROMPTS[key])
        scores[key] = float(sims[idx : idx + n].mean().item())
        idx += n

    best_label = max(scores, key=scores.get)
    return {"label": best_label, "confidence": scores[best_label], "scores": scores}
```

Рисунок 2.11 − Реалізація модуля zero-shot класифікації документів моделлю CLIP ViT-B/32

Верифікацію КЕП реалізовано в асинхронній функції `verify_pdf_signature` засобами Playwright.

Браузерний екземпляр Chromium створюється одноразово та захищається від паралельних звернень через `asyncio.Lock`. Процес охоплює завантаження файлу .p7s до форми порталу id.gov.ua, ініціювання перевірки та зчитування результату з DOM-елементів #result і #signResults.

На рисунку 2.12 наведено реалізацію функції верифікації.

```python
async def verify_pdf_signature(file_path: str | Path, timeout_ms: int = 60000) -> dict:
    file_path = str(Path(file_path).resolve())

    async with _lock:
        ctx = await _get_context()
        page = await ctx.new_page()
        try:
            await page.goto(VERIFY_WIDGET_URL, wait_until="networkidle")
            await page.wait_for_selector("#chooseFilesInput", state="attached", timeout=30000)

            await page.locator("#chooseFilesInput").set_input_files(file_path)

            await page.wait_for_selector("#checkButton", state="visible", timeout=15000)
            await page.click("#checkButton")

            try:
                await page.wait_for_function(
                    """() => {
                        const result = document.querySelector('#result');
                        if (result && result.style.display !== 'none') return true;
                        const err = document.querySelector('#errorBlock');
                        if (err && err.style.display !== 'none') return true;
                        return false;
                    }""",
                    timeout=timeout_ms,
                )
            except PWTimeout:
                return {
                    "signed": False,
                    "valid": False,
                    "details": "Час очікування результату вичерпано",
                    "signer": None,
                    "content": None,
                }
```

Рисунок 2.12 − Реалізація модуля верифікації кваліфікованого електронного підпису через портал id.gov.ua

Усі маршрути зареєстровані під префіксом `/api/v1`. Таблиця 2.2 описує повний перелік ендпоїнтів.

Таблиця 2.5 — REST API серверної частини

Метод	Маршрут	Призначення
GET	`/health`	Перевірка доступності сервера
POST	`/api/v1/login_imap`	Підключення до Gmail IMAP, синхронізація заголовків листів у SQLite
GET	`/api/v1/emails`	Пагінований список листів
POST	`/api/v1/{uid}/details`	Завантаження повного тексту та вкладень конкретного листа
POST	`/api/v1/classify`	Векторне вбудовування тем і класифікація за схожістю
GET	`/api/v1/classification/summary`	Зведена статистика по мітках
GET	`/api/v1/classification/by_label`	Пагінований список листів за міткою
POST	`/api/v1/{uid}/classify/documents`	Верифікація вкладень листа: CLIP-класифікація, перевірка підписів
POST	`/api/v1/{uid}/classify/documents/stream`	Верифікація вкладень з потоковою передачею подій (SSE)
GET	`/api/v1/{uid}/attachments/{filename}`	Завантаження вкладення
GET	`/api/v1/{uid}/originals`	Список верифікованих оригіналів документів
GET	`/api/v1/{uid}/originals-meta`	Метадані підписантів верифікованих документів
GET	`/api/v1/{uid}/originals/{filename}`	Завантаження верифікованого оригіналу
GET	`/api/v1/statistics/timeline`	Хронологічний графік статистики

Ендпоїнт POST `/login_imap` приймає облікові дані `{email_address, imap_password}` та виконує дві послідовні операції: спочатку перевіряє коректність підключення до IMAP-сервера imap.gmail.com, після чого ініціює інкрементальну синхронізацію — завантажує з сервера лише ті листи, унікальний ідентифікатор UID яких є більшим за максимальний вже збережений у локальній базі даних.

Ендпоїнт POST `/classify` реалізує двофазну обробку вхідного потоку повідомлень. На першій фазі здійснюється обчислення ембедингів та вбудовування тем усіх ще не проіндексованих листів у векторну базу даних ChromaDB пакетами по 50 документів. На другій фазі виконується пошуковий запит до колекції за еталонним рядком із подальшою фільтрацією результатів за встановленою пороговою косинусною відстанню. Листи, показник близькості яких потрапляє в межі порогу, отримують мітку Accepted/Subject, тоді як решта маркується як Email Subject.

Ендпоїнт POST `/{uid}/classify/documents/stream` реалізований на основі технології Server-Sent Events (SSE) для організації асинхронного веб-зв'язку. Клієнтська частина отримує структуровані події в реальному часі в міру виконання кожного окремого кроку конвеєра верифікації: завантаження файлів-вкладень, мультимодальна CLIP-класифікація образів, пошук відокремленого цифрового підпису, передача об'єктів до порталу id.gov.ua та фінальне отримання результату валідації. Таке архітектурне рішення забезпечує високий рівень чутливості інтерфейсу (responsive UX) без тривалого блокуючого очікування з боку користувача.

Система використовує дві бази даних паралельно. SQLite зберігає структуровані метадані листів та результати класифікації у реляційних таблицях emails і email_labels. Векторна база даних ChromaDB зберігає обчислені ембединги тем листів у колекції emails, де кожен окремий документ має унікальний ідентифікатор — IMAP UID відповідного листа.

Модуль генерації чернеток відповідей є фінальним компонентом RAG-конвеєру системи та реалізує принцип Augmented Generation — збагачення контексту перед передачею до великої мовної моделі. На відміну від класифікаційних етапів, що використовують детерміновані алгоритми векторного пошуку, цей модуль застосовує генеративні можливості LLM виключно для формування відповіді абітурієнту на основі структурованих технічних даних.

Перед викликом LLM API оркестратор RAG збирає контекст із двох джерел. Першим є повний текст листа абітурієнта, завантажений із SQLite за UID листа. Другим є набір знеособлених технічних міток, згенерованих на етапі верифікації документів: тип кожного вкладення, визначений моделлю CLIP, та статус перевірки електронного підпису через портал id.gov.ua.

Персональні дані абітурієнта ім'я, номери документів, адреса, не передаються до зовнішнього API на жодному з етапів. До хмарного сервісу надходять виключно знеособлені технічні мітки та текст листа без вкладень.

Система підтримує два провайдери за вибором оператора: Google Gemini API та Groq API з моделями сімейства Llama 3 та Mixtral.

Оркестратор формує системний промпт, що визначає роль моделі як асистента приймальної комісії, та користувацький промпт із зібраним контекстом. Модель отримує інструкцію згенерувати чернетку відповіді українською мовою з урахуванням наявних та відсутніх документів. У разі відсутності певного документа або невалідного підпису модель формує відповідний запит до абітурієнта щодо надання необхідних матеріалів.

Таблиця 2.6 — REST API ендпоїнти модуля генерації чернеток

Метод	Маршрут	Призначення
POST	`/api/v1/{uid}/generate/draft`	Генерація чернетки відповіді для конкретного листа на основі тексту листа та технічних міток верифікованих документів
GET	`/api/v1/{uid}/draft`	Отримання збереженої чернетки відповіді для листа
PUT	`/api/v1/{uid}/draft`	Збереження відредагованої оператором чернетки відповіді
POST	`/api/v1/{uid}/draft/send`	Відправка затвердженої чернетки як відповіді на лист через IMAP/ Simple Mail Transfer Protocol (SMTP)

Ендпоїнт POST `/api/v1/{uid}/generate/draft` приймає необов'язковий параметр provider зі значеннями gemini або groq. За замовчуванням використовується Gemini API, відповідь містить текст чернетки та метадані генерації.

Ендпоїнт POST `/api/v1/{uid}/draft/send` вимагає наявності збереженої чернетки та прапорця confirmed: true, що унеможливлює автоматичну відправку без явного затвердження оператором.

Згенерована чернетка зберігається у таблиці email_drafts бази SQLite з прапорцем `is_sent`. Оператор редагує текст в інтерфейсі Streamlit і лише після підтвердження ініціює відправку через SMTP.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі розроблено архітектуру та програмну реалізацію системи автоматизованої обробки листів абітурієнтів на основі RAG.

Основою системи є архітектура RAG, за якою класифікація листів виконується через пошук за семантичною схожістю у векторній базі даних ChromaDB, а велика мовна модель залучається на фінальному етапі виключно для генерації персоналізованих чернеток відповідей на основі збагаченого контексту. Такий розподіл забезпечує детермінований і передбачуваний результат класифікації, тоді як генеративні можливості ШІ використовуються лише там, де важлива гнучкість формулювань, а не точність рішення.

Визначено конвеєр обробки даних із п'яти етапів: синхронізація листів через IMAP, векторне індексування тем у ChromaDB, семантична класифікація за еталонним запитом, потокова верифікація вкладених документів та генерація чернетки відповіді засобами хмарного LLM API. Кожен етап реалізований як окремий незалежний сервіс, що дозволяє запускати їх у довільному порядку та повторювати без впливу на стан системи.

Сформовано перелік вимог та описано основні сценарії використання системи оператором приймальної комісії. Для класифікації документів-вкладень застосовано мультимодальну модель CLIP, яка не потребує розміченої навчальної вибірки. Верифікація кваліфікованого електронного підпису реалізована через автоматизовану взаємодію з державним порталом id.gov.ua засобами Playwright.

Серверна частина реалізована на FastAPI з розподілом відповідальностей між маршрутами, сервісами та шаром даних. REST API охоплює повний цикл роботи з листами: від синхронізації до генерації та завантаження верифікованих документів. Модуль генерації чернеток реалізовано з підтримкою двох LLM-провайдерів: Google Gemini та Groq, з обов'язковою валідацією оператором перед відправкою через SMTP, що унеможливлює автономну відправку відповідей системою. Описано бібліотеки, що забезпечують роботу кожної підсистеми, та обґрунтовано їх вибір відповідно до вимог.

РОЗДІЛ 3
ПРИКЛАДНІ АСПЕКТИ ЗАСТОСУВАННЯ ПРОГРАМНОЇ СИСТЕМИ З АВТОМАТИЗАЦІЇ ОБРОБКИ ЛИСТІВ АБІТУРІЄНТІВ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ НА ОСНОВІ АРХІТЕКТУРИ RAG

Практична цінність будь-якої програмної системи визначається не лише коректністю її архітектурних рішень, а й здатністю ефективно функціонувати в умовах реального застосування. Система автоматизованої обробки листів абітурієнтів на основі архітектури RAG реалізує комплексний підхід до автоматизації рутинних операцій приймальної комісії, що охоплює семантичну класифікацію вхідної кореспонденції, мультимодальну верифікацію вкладених документів та генерацію персоналізованих чернеток відповідей засобами великої мовної моделі. Перевірка відповідності реалізованих функціональних можливостей задекларованим вимогам потребує всебічного розгляду системи з прикладної точки зору.

У підрозділі 3.1 наведено покрокову демонстрацію повного робочого циклу оператора приймальної комісії — від підключення до поштового сервера до отримання готової чернетки відповіді абітурієнту, — що дозволяє наочно оцінити ергономічність інтерфейсу та узгодженість роботи всіх компонентів конвеєру обробки даних. Підрозділ 3.2 містить результати тестування розробленої системи з аналізом відповідності досягнутих показників продуктивності та точності класифікації вимогам технічного завдання. У підрозділі 3.3 наведено програмну документацію на розроблене програмне забезпечення.

3.1. Демонстрація практичних можливостей розробленої системи.

Клієнтська частина системи реалізована у вигляді веб-застосунку на основі фреймворку Streamlit та розгортається локально. Інтерфейс орієнтований на оператора приймальної комісії без спеціальної технічної підготовки: навігація між розділами здійснюється через бічну панель. Доступ до системи захищено автентифікацією через IMAP — при вході оператор вводить адресу електронної пошти та пароль додатка Gmail, після чого облікові дані зберігаються у сесії для подальших запитів до серверної частини.

Після успішної автентифікації оператору доступні чотири розділи у групі «Сервіс»: Поштова скринька — перегляд вхідних листів із пагінацією та можливістю завантажити повний текст і вкладення конкретного листа; Абітурієнти — відображення результатів семантичної класифікації з індикаторами статусу верифікації документів; Правила класифікації — CRUD-управління правилами семантичного пошуку із налаштуванням порогу косинусної відстані; Статистика — зведені дані щодо опрацьованих звернень.

Розгляд практичних можливостей системи починається з опису процесу автентифікації та синхронізації поштової скриньки, оскільки саме ці операції є обов'язковою передумовою для роботи всіх подальших компонентів конвеєру обробки даних.

3.1.1. Автентифікація та синхронізація поштової скриньки.

Початком роботи з системою є процедура автентифікації. При відкритті веб-застосунку оператору відображається форма входу (рисунок 3.1).

Форма входу містить два поля: адреса електронної пошти Gmail та пароль додатка — спеціально згенерований App Password, що не розкриває основний пароль облікового запису. Після натискання кнопки «Увійти» фронтенд надсилає облікові дані до ендпоінту POST /api/v1/login_imap серверної частини.

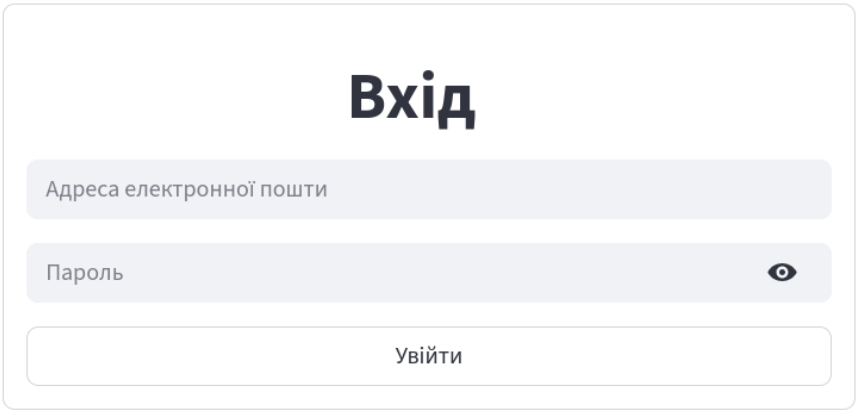


Рисунок 3.1 − Форма входу в систему

Бекенд виконує дві послідовні операції: спочатку встановлює тестове підключення до поштового сервера imap.gmail.com для перевірки коректності облікових даних, після чого ініціює інкрементальну синхронізацію, тобто завантажує заголовки лише тих листів, унікальний ідентифікатор UID яких перевищує максимальний збережений у локальній базі SQLite. За першого запуску завантажується до 500 заголовків; надалі синхронізуються виключно нові повідомлення. Метадані листів (відправник, тема, дата, UID) зберігаються у таблиці emails бази SQLite, тоді як повний текст та вкладення завантажуються відкладено, лише за явним запитом оператора.

Після успішної автентифікації оператор отримує доступ до розділу «Поштова скринька» (рисунок 3.2), де листи відображаються у вигляді пагінованого списку з показом відправника, теми та дати отримання. Для перегляду повного тексту конкретного листа оператор натискає кнопку «Переглянути», що ініціює окремий запит на завантаження тіла повідомлення та вкладень.

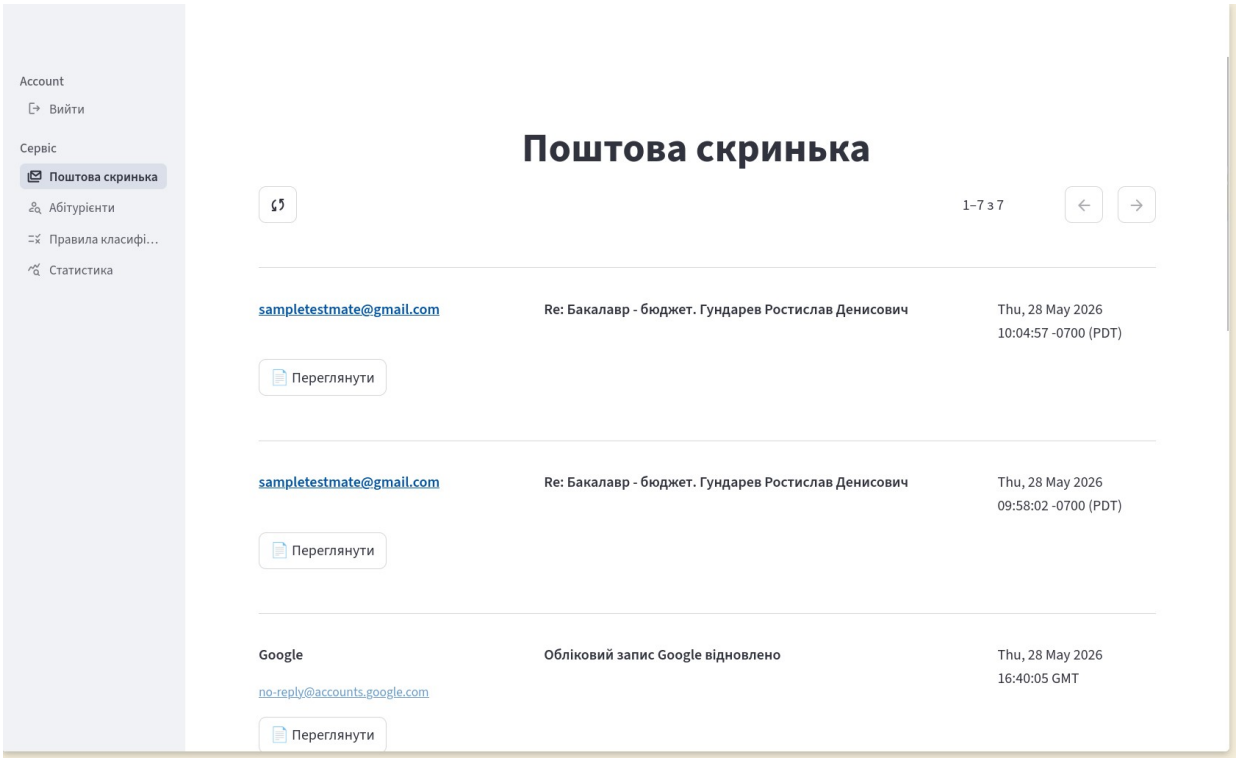


Рисунок 3.2 − Відображення розділу «Поштова скринька» системи

При натисканні кнопки «Переглянути» система виконує запит POST `/api/v1/{uid}/details`, що ініціює завантаження повного тексту листа та вкладень безпосередньо з IMAP-сервера. Результат розгортається безпосередньо під рядком листа у списку (рисунок 3.3): відображається тіло повідомлення у текстовому форматі та перелік вкладень у вигляді кнопок із назвами файлів. На наведеному прикладі лист від абітурієнта містить три файли відокремленого електронного підпису формату .p7s: Passport.p7s, Prypusne.p7s, Zayava.p7s та фотографію 3x4.pdf.

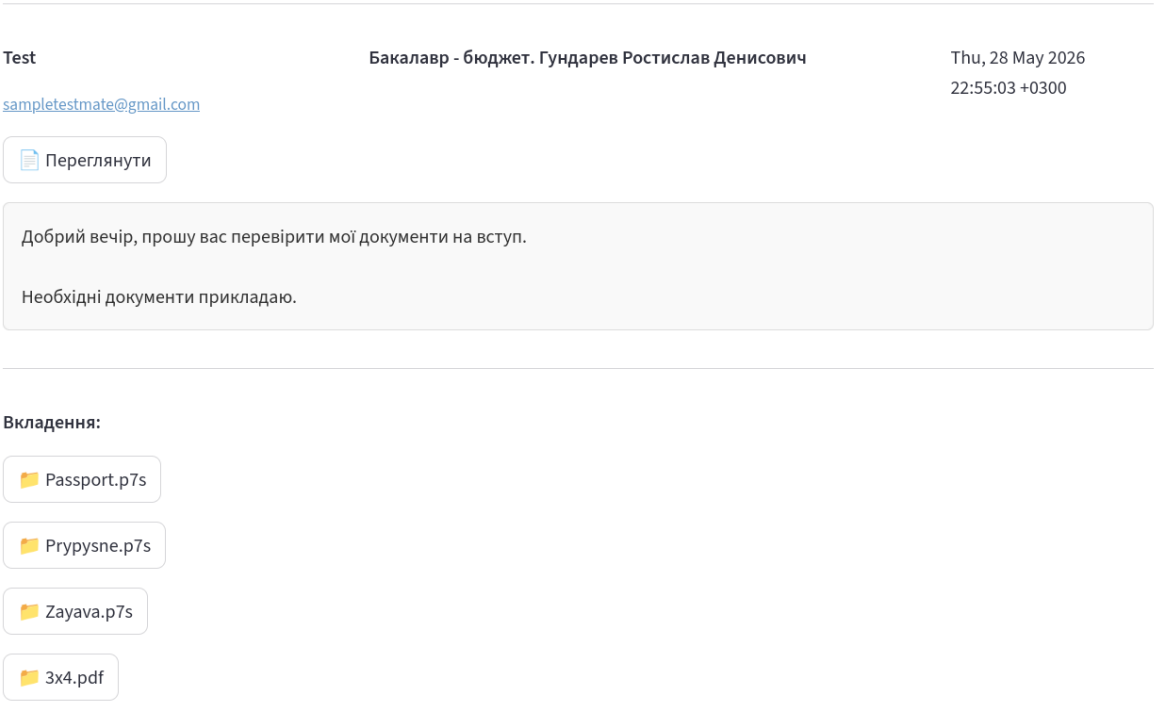


Рисунок 3.3 − Перегляд повного тексту та вкладень листа абітурієнта

Натискання на кнопку вкладення ініціює його завантаження через ендпоінт GET `/api/v1/{uid}/attachments/{filename}`. Завдяки принципу відкладеного завантаження (lazy loading) повний текст та вкладення запитуються лише за явною дією оператора, що суттєво скорочує час первинної синхронізації поштової скриньки.

Список листів відображається посторінково по 50 повідомлень, у правій частині панелі інструментів показано поточний діапазон та загальну кількість, наприклад, «1–7 з 7». Перехід між сторінками здійснюється кнопками навігації «←» та «→».

Для отримання нових листів, що надійшли після останньої синхронізації, оператор натискає кнопку «Оновити» (іконка ⟳), система повторно викликає ендпоінт POST `/api/v1/login_imap`, що запускає інкрементальну синхронізацію та завантажує лише листи з UID, більшим за максимальний збережений у базі.

3.1.2. Налаштування правил класифікації.

Перед запуском семантичної класифікації оператор має визначити правила, за якими система здійснюватиме розподіл листів. Налаштування виконується у розділі «Правила класифікації» (рисунок 3.4). За відсутності створених правил інтерфейс відображає інформаційне повідомлення «Правил немає. Додайте перше правило нижче.», а нижче розміщено форму додавання нового правила.

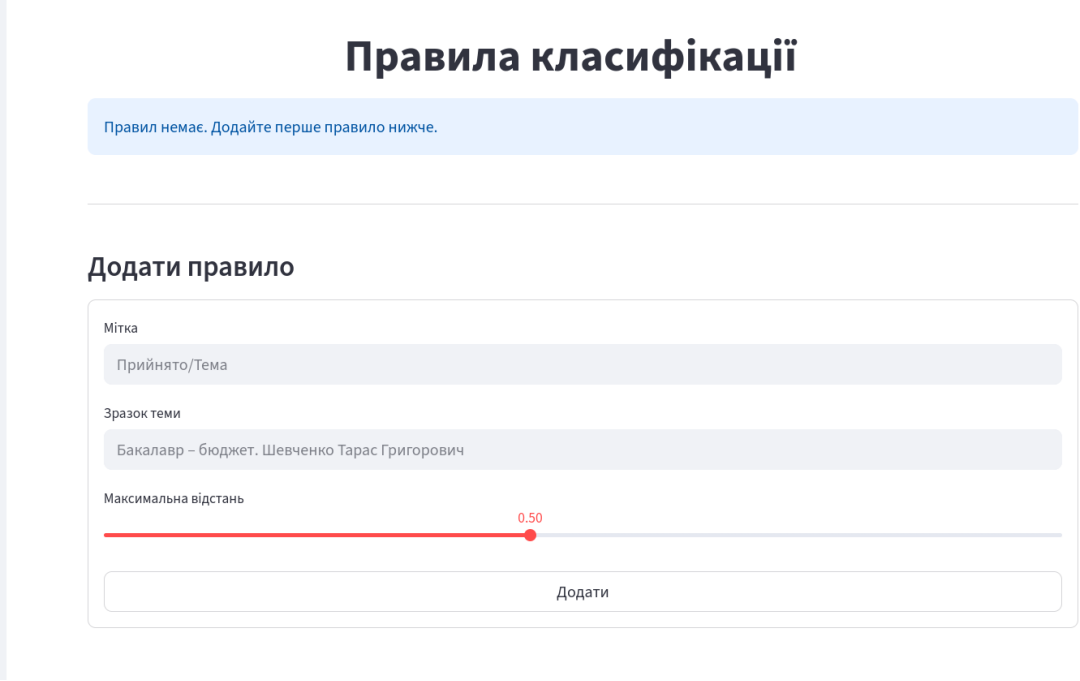


Рисунок 3.4 − Розділ «Правила класифікації» з формою додавання правила

Форма містить три параметри. Поле «Мітка» визначає назву категорії, яка присвоюватиметься листам, що відповідають правилу, наприклад, «Прийнято/Тема». Поле «Зразок теми» є еталонним запитом для векторного пошуку: система обчислює косинусну відстань між ембедингом теми кожного листа та ембедингом цього рядка, тому зразок має відтворювати типове формулювання теми листів цільової категорії.

Слайдер «Максимальна відстань» встановлює поріг косинусної відстані у діапазоні від 0.1 до 1.0 з кроком 0.05; листи з відстанню, що не перевищує встановленого порогу, отримують відповідну мітку. Значення за замовчуванням 0.50 є збалансованим між точністю та повнотою класифікації для моделей сімейства sentence-transformers.

Після заповнення форми оператор натискає «Додати», що надсилає запит POST `/api/v1/rules` із параметрами `{label, reference_query, max_distance}`.

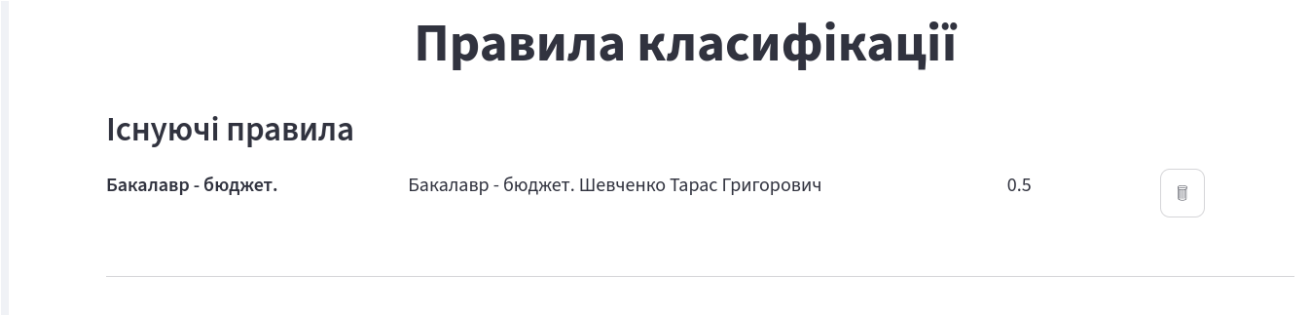


Рисунок 3.5 − Новостворене правило класифікації

Збережені правила відображаються у вигляді таблиці з колонками: мітка, зразок теми, порогове значення відстані та кнопка видалення. Видалення правила виконується натисканням іконки, що надсилає запит DELETE `/api/v1/rules/{id}` та негайно оновлює список. Система підтримує довільну кількість правил, що дозволяє оператору одночасно класифікувати листи за кількома тематичними категоріями в межах однієї кампанії.

3.1.3. Семантична класифікація листів.

Семантична класифікація запускається у розділі «Абітурієнти» натисканням кнопки «Класифікувати» (рисунок 3.6). Система надсилає запит POST `/api/v1/classify`, який виконує двофазну обробку: спочатку обчислює 384-вимірні вектори тем усіх ще не проіндексованих листів засобами моделі all-MiniLM-L6-v2 та записує їх до колекції ChromaDB пакетами по 50 документів; після чого для кожного збереженого правила класифікації виконує векторний пошук за еталонним запитом із фільтрацією результатів за пороговою косинусною відстанню. Листи, показник відстані яких не перевищує встановленого порогу, отримують мітку відповідного правила; решта маркується міткою «Неприйнято». Після завершення класифікації інтерфейс відображає сповіщення із кількістю оброблених, прийнятих та відхилених листів.

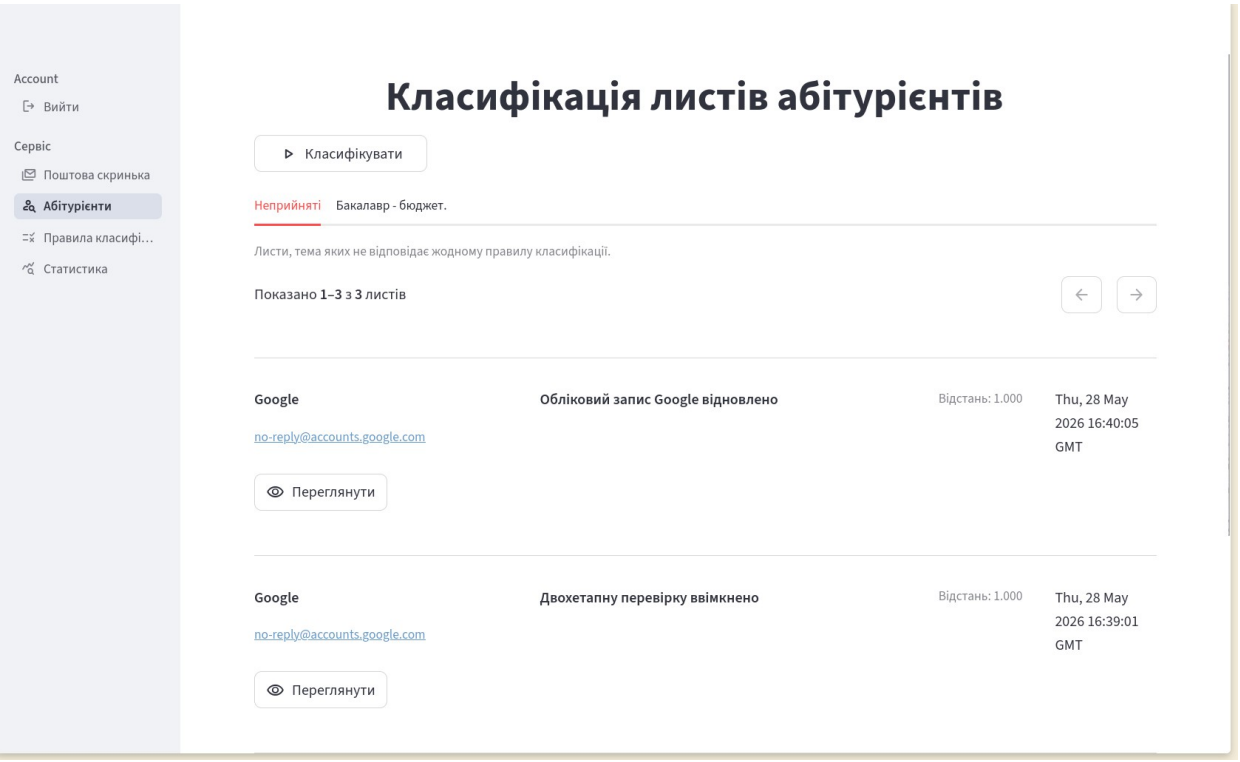


Рисунок 3.6 – Розділ «Класифікація листів абітурієнтів», вкладка «Неприйняті»

Вкладка, що відповідає конкретному правилу класифікації, відображає лише ті листи, косинусна відстань яких не перевищила встановленого порогу (рисунок 3.7).

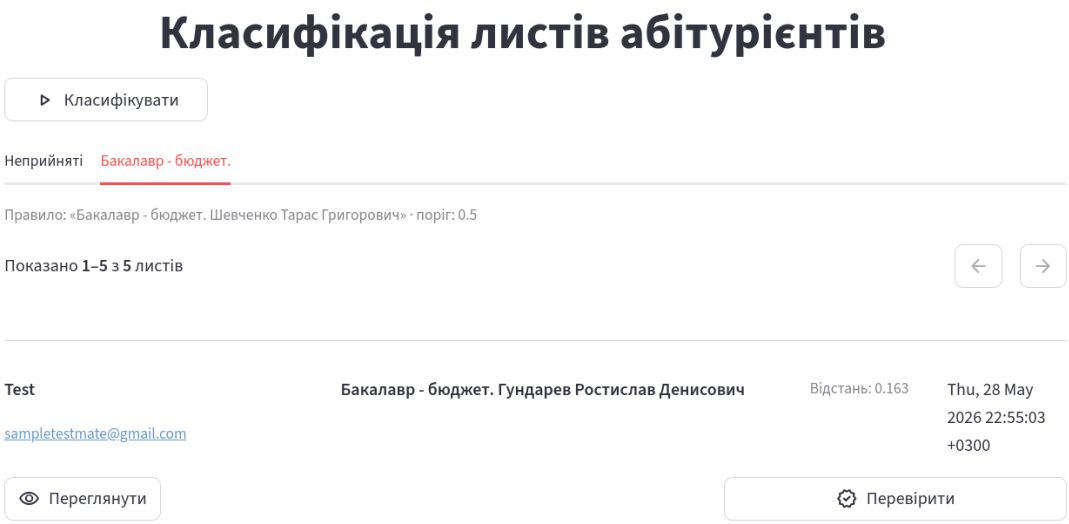


Рисунок 3.7 − Розділ «Класифікація листів абітурієнтів», вкладка «Неприйняті»

У підзаголовку вкладки виводиться еталонний запит правила та порогове значення у наведеному прикладі: «Бакалавр - бюджет. Шевченко Тарас Григорович» з порогом 0.5. Лист від абітурієнта з темою «Бакалавр - бюджет. Гундарев Ростислав Денисович» отримав відстань 0.163, що свідчить про високу семантичну близькість до еталонного запиту: структурний збіг формату теми (тип вступу, форма фінансування, прізвище) забезпечує стабільне розпізнавання попри відмінність конкретного прізвища.

На відміну від вкладки «Неприйняті», для листів у вкладках правил доступна кнопка «Перевірити», вона запускає конвеєр верифікації вкладених документів, що розглядається у підрозділі 3.1.4.

3.1.4. Верифікація вкладених документів.

Після того як лист класифіковано за темою та він потрапляє до відповідної вкладки, оператор може ініціювати перевірку вкладених документів, натиснувши кнопку «Перевірити». Процес верифікації виконується асинхронно на сервері та передається у браузер у режимі реального часу через механізм Server-Sent Events (SSE), кожен проміжний крок надходить окремою подією і відображається безпосередньо в інтерфейсі в міру виконання.

Верифікація охоплює два незалежні напрями: класифікацію документів за їх візуальним вмістом та перевірку цифрового підпису.

На першому етапі кожне вкладення, що є зображенням або PDF-документом, передається до моделі CLIP (ViT-B/32), яка порівнює векторне представлення сторінки з набором текстових підказок для кожного типу документа та визначає найбільш вірогідний клас. Як показано на рисунку 3.8, система визначила перший файл як «військово-обліковий документ (Резерв+)», а другий як «паспорт».

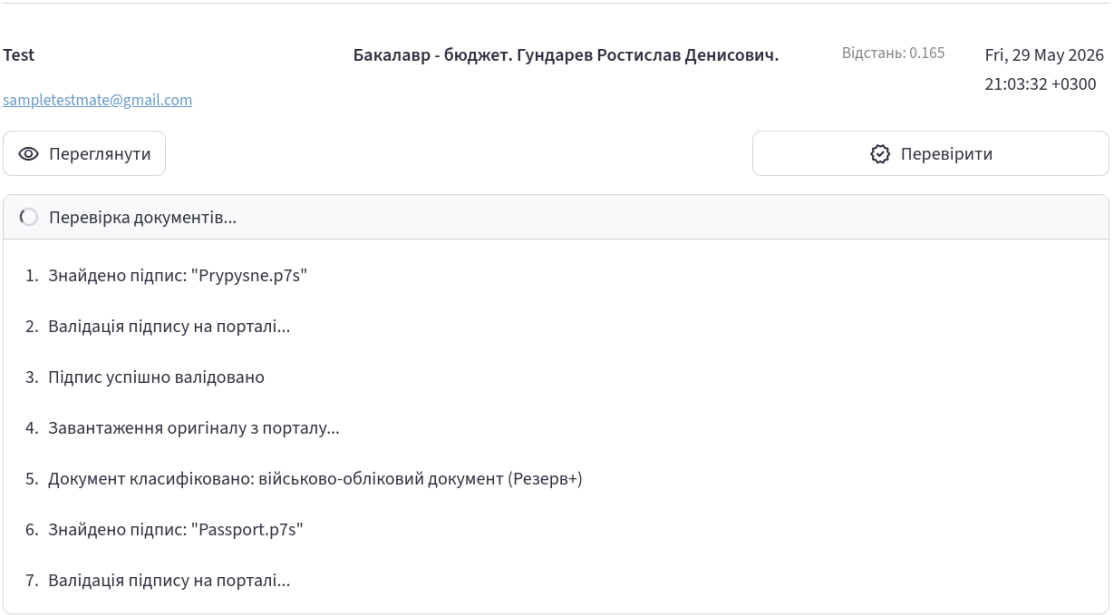


Рисунок 3.8 − потоковий вивід SSE під час верифікації

На другому етапі система перевіряє наявність файлу електронного підпису з розширенням .p7s, назва якого відповідає назві знайденого документа наприклад, passport.pdf.p7s. За наявності такого файлу він автоматично завантажується на портал id.gov.ua через браузерну автоматизацію Playwright, де відбувається верифікація кваліфікованого електронного підпису (КЕП).

Ім'я підписанта зчитується з результату перевірки та відображається поруч із документом.

Якщо файл підпису відсутній, документ отримує позначку «без підпису».За результатами верифікації кожному листу автоматично присвоюються мітки стану документів: «Passport OK» (документ знайдено й підпис верифіковано), «Passport Unsigned» (документ знайдено, підпис відсутній) або «Passport Missing» (документ не виявлено).

Аналогічна логіка застосовується до заяви та військово-облікового документа. Ці мітки відображаються як кольорові індикатори у рядку листа (рисунок 3.9) і дозволяють оператору миттєво оцінити комплектність пакету документів без необхідності відкривати лист.

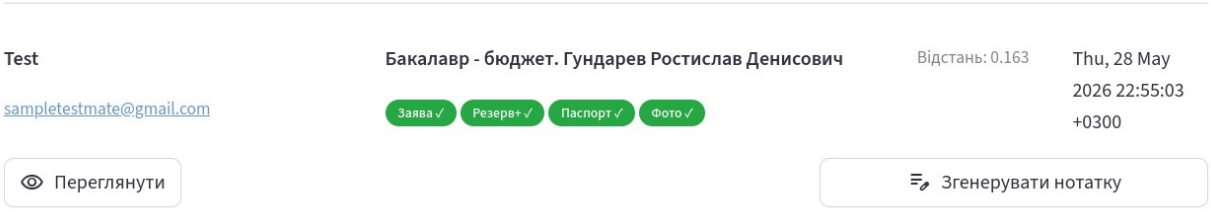


Рисунок 3.9 − Відображення міток стану документів у рядку листа після верифікації

3.1.5. Генерація нотатки та відправка відповіді.

Після завершення верифікації вкладених документів кнопка «Перевірити» у рядку листа замінюється кнопкою «Згенерувати нотатку».

Такий стан інтерфейсу сигналізує оператору про готовність системи до формування персоналізованої чернетки відповіді: всі технічні мітки документів уже присвоєно і можуть бути використані як контекст для великої мовної моделі.

При натисканні кнопки фронтенд надсилає запит POST `/api/v1/{uid}/draft` до серверної частини та відображає індикатор очікування.

Серверний обробник виконує дві послідовні операції над базою даних SQLite: спочатку зчитує метадані листа: тему та адресу відправника; після чого отримує всі технічні мітки, що були присвоєні листу на попередніх етапах класифікації та верифікації.

Принциповою особливістю реалізації є те, що до зовнішнього API великої мовної моделі передаються виключно знеособлені технічні мітки наприклад: «Passport OK», «Military Unsigned», «Application Missing»; без жодних персональних даних абітурієнта: імені, номерів документів чи адреси електронної пошти. Це забезпечує відповідність вимогам конфіденційності при роботі з хмарними сервісами.

Оркестратор RAG формує два повідомлення для моделі. Системний промпт визначає роль моделі як асистента приймальної комісії КПІ та інструктує генерувати коротку офіційну чернетку відповіді українською мовою виключно на основі наданих міток, не вигадуючи інформацію поза їхніми межами. Користувацьке повідомлення містить перелік міток у форматі списку.

Згенерована чернетка відображається у текстовому полі безпосередньо під рядком листа (рисунок 3.10). Поле є редагованим, що дозволяє оператору скоригувати формулювання перед відправкою. Якщо мітки вказують на відсутність або невалідний підпис певного документа, модель формує запит до абітурієнта щодо надання необхідних матеріалів; якщо всі документи в порядку підтверджує їх отримання.

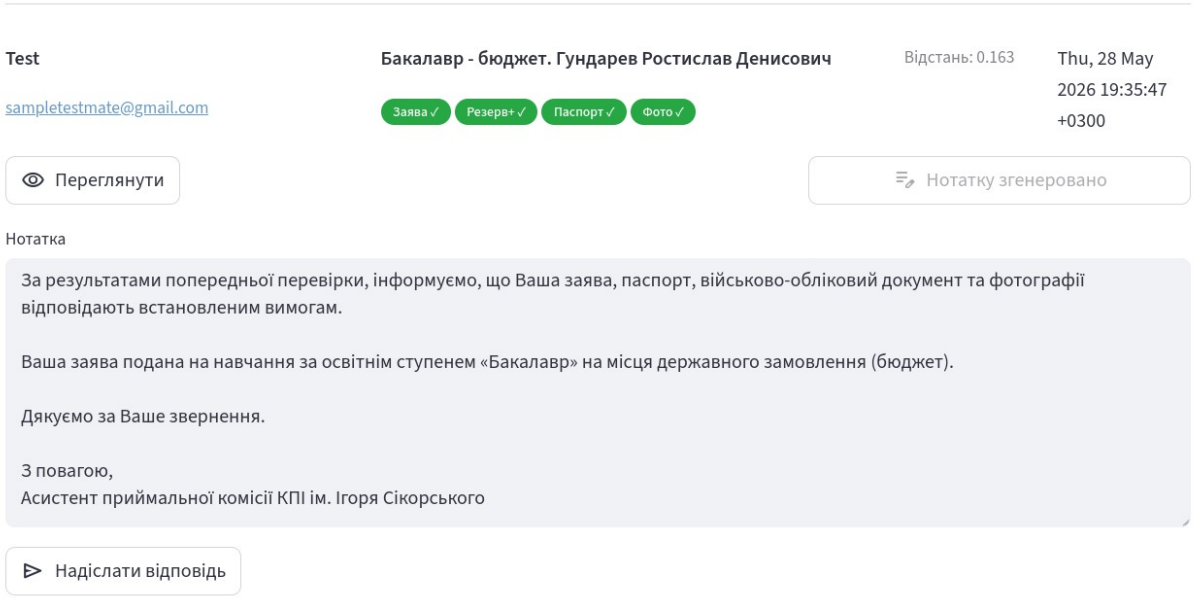


Рисунок 3.10 − Згенерована чернетка відповіді у редагованому текстовому полі

Після редагування оператор натискає кнопку «Надіслати відповідь». Фронтенд передає відредагований текст та облікові дані на ендпоінт POST `/api/v1/{uid}/reply`. Серверний обробник формує повідомлення формату MIME з темою Re: {тема оригінального листа} та відправляє його через протокол SMTP із шифруванням STARTTLS на сервер smtp.gmail.com порт 587, використовуючи той самий App Password, що й для IMAP-синхронізації. Після успішної відправки кнопка замінюється зеленим індикатором «Відповідь надіслано» (рисунок 3.11), що унеможливлює повторну відправку.

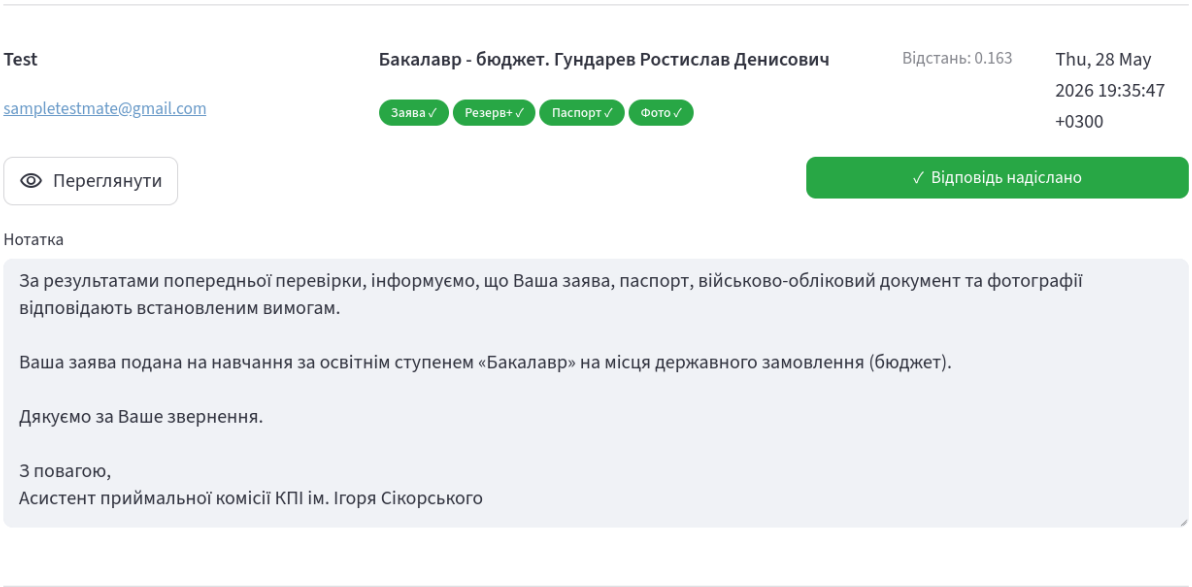


Рисунок 3.11 − Підтвердження успішної відправки відповіді абітурієнту

3.1.6. Перегляд статистики.

Розділ «Статистика» надає оператору зведений огляд результатів обробки вхідної кореспонденції у вигляді двох інтерактивних діаграм. Дані обох діаграм можна фільтрувати за конкретним правилом класифікації через випадний список «Категорія:» у верхній частині сторінки; за замовчуванням відображаються дані по всіх категоріях одночасно.

Перша діаграма «Надходження листів» (рисунок 3.12) відображає розподіл вхідних листів за датами у вигляді згрупованих стовпців з чотирма кольоровими категоріями: «Не перевірені», «Перевірено», «Прийняті за темою» та «Відхилені за темою». Набір категорій для відображення обирається через мультиселектор над діаграмою. Дані надходять з ендпоінту GET `/api/v1/statistics/timeline`, який групує листи за датою отримання та обчислює кількість по кожній категорії.

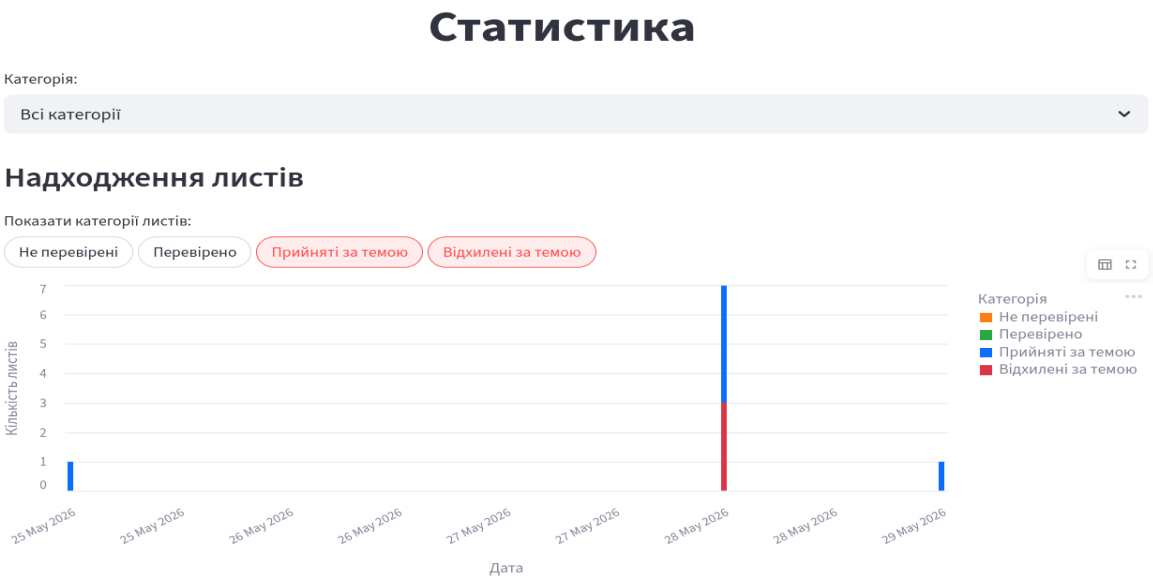


Рисунок 3.12 − Діаграма «Надходження листів»

Друга діаграма «Перевірка документів» (рисунок 3.13) відображає результати верифікації у вигляді стовпців за типами документів: «Паспорт», «Заява», «Резерв+» та «Фото», кожен з яких розфарбований за статусом: зелений — «ОК», синій — «Без підпису», червоний — «Відсутній». Дані надходять з ендпоінту GET `/api/v1/classification/summary`, що повертає кількість листів за кожною міткою верифікації з таблиці email_labels.

Рисунок 3.13 − Діаграма «Перевірка документів»

Обидві діаграми реалізовано на бібліотеці Altair з підтримкою адаптивної ширини контейнера. Такий підхід дозволяє оператору приймальної комісії швидко оцінити загальний стан кампанії, кількість необроблених звернень та типові проблеми з комплектністю документів без ручного аналізу кожного листа.

3.2. Тестування розробленої системи.

Тестування системи проводилося на трьох рівнях: юніт-тестування окремих функцій, інтеграційне тестування взаємодії компонентів та функціональне тестування повного робочого циклу оператора. Завершальним етапом є перевірка відповідності реалізованих можливостей вимогам технічного завдання.

Юніт-тестування охоплювало перевірку ізольованих функцій серверної частини. Функція `_parse_date()` перевірялась на коректне перетворення рядків формату RFC 2822 у рядок YYYY-MM-DD та коректне повернення None для відсутнього значення. Логіка порогової фільтрації у classifier_service.py тестувалась для граничних значень: лист з відстанню рівно 0.5 при порозі 0.5

класифікується як прийнятий, а при 0.501 — відхиляється. Логіка присвоєння міток у document_validation_service.py тестувалась для трьох сценаріїв: наявність .p7s-файлу → мітка OK; відсутність файлу підпису → Unsigned; відсутність документа серед вкладень → Missing.

Інтеграційне тестування перевіряло взаємодію між компонентами. Конвеєр POST `/api/v1/login_imap` → SQLite тестувався на реальних кредах: перевірялась наявність записів у таблиці emails, унікальність UID та інкрементальна логіка — повторний виклик не дублює збережені листи. Конвеєр POST `/api/v1/classify` → ChromaDB → email_labels тестувався на скриньці з 7 листами: після першого запуску з'явилось 7 векторів та відповідні записи з відстанями; повторний запуск не додавав дублікатів.

SSE-ендпоінт верифікації тестувався на листі з чотирма вкладеннями: перевірявся формат подій, наявність проміжних кроків та фінальної події "step": "done" з переліком міток. Модуль POST `/api/v1/{uid}/reply` тестувався зреальними SMTP-кредами: перевірялась доставка листа та коректність теми Re: {оригінальна тема}.

Функціональне тестування проводилося методом «чорного ящика» на тестовому листі з темою «Бакалавр - бюджет. Гундарев Ростислав Денисович» з вкладеннями Passport.p7s, Prypusne.p7s, Zayava.p7s та 3x4.pdf. Перевірялась відповідність поведінки інтерфейсу та серверних відповідей очікуваним результатам для десяти ключових сценаріїв — від автентифікації до відправки відповіді. Результати наведено у таблиці 3.1.

Таблиця 3.1 — Результати функціонального тестування

№	**Тестований сценарій**	**Очікуваний результат**	**Фактичний результат**	**Статус**
1	Автентифікація з валідними IMAP-кредами	Синхронізація листів	Завантажено 7 заголовків	Пройдено
2	Автентифікація з невалідним паролем	Повідомлення про помилку	Відображено «Помилка підключення»	Пройдено
3	Перегляд повного тексту листа	Завантаження тіла та вкладень	Відображено текст і 4 кнопки вкладень	Пройдено
4	Додавання правила класифікації	Правило збережено у таблиці	Правило з порогом 0.5 збережено	Пройдено
5	Семантична класифікація	Лист з відстанню < 0.5 у вкладці правила	Відстань 0.163, лист у вкладці правила	Пройдено
6	Верифікація документів (SSE)	Потоковий вивід, мітки після завершення	7 кроків у реальному часі, мітки присвоєно	Пройдено
7	Верифікація без файлу підпису	Мітка «Unsigned»	Присвоєно «Passport Unsigned»	Пройдено
8	Генерація нотатки	Чернетка українською на основі міток	Сформовано текст підтвердження документів	Пройдено
9	Відправка відповіді через SMTP	Лист доставлено	Лист отримано, зелений індикатор	Пройдено
10	Видалення правила класифікації	Правило видалено зі списку	Правило зникло після натискання	Пройдено

3.3. Програмна документація на програмне забезпечення.

Система розгортається у вигляді двох Docker-контейнерів, керованих через Docker Compose. Серверна частина (backend) побудована на Python 3.13 з фреймворком FastAPI та запускається на порту 8000; клієнтська частина (frontend) реалізована на Streamlit та запускається на порту 8501. Фронтенд залежить від бекенду через умову service_healthy, запуск клієнтської частини відкладається до успішної відповіді ендпоінту /health серверної частини.

Вимоги до апаратного та програмного забезпечення.

Для розгортання системи необхідно: центральний процесор з підтримкою AVX-інструкцій (Intel Core i5 або аналог), не менше 8 ГБ оперативної пам'яті (CLIP ViT-B/32 займає ~700 МБ, ChromaDB та MiniLM — ще ~500 МБ), не менше 10 ГБ вільного дискового простору для моделей та даних. Наявність GPU не вимагається — усі моделі працюють на CPU. Програмне забезпечення: Docker Engine 24.0 або вище, Docker Compose v2, доступ до мережі Інтернет для завантаження образів та звернення до Gemini API.

Налаштування середовища.

Перед запуском необхідно створити файл .env у кореневій директорії проєкту зі значенням змінної GEMINI_API_KEY — ключа доступу до Google Gemini API, який отримується безкоштовно через Google AI Studio. Решта змінних мають значення за замовчуванням, наведені у таблиці 3.2.

Таблиця 3.2 — Змінні середовища системи

Змінна	Значення за замовчуванням	Призначення
GEMINI_API_KEY	— (обов'язкова)	Ключ доступу до Google Gemini API
BACKEND_URL	http://backend:8000	Адреса серверної частини для фронтенду
DB_PATH	./data/emails.db	Шлях до файлу SQLite
CHROMA_DB_PATH	./data/chroma_db	Шлях до сховища ChromaDB
ATTACHMENTS_DIR	./data/attachments	Директорія для збереження вкладень

Розгортання системи.

Для запуску системи достатньо виконати одну команду у кореневій директорії проєкту:

```
docker compose up —build
```

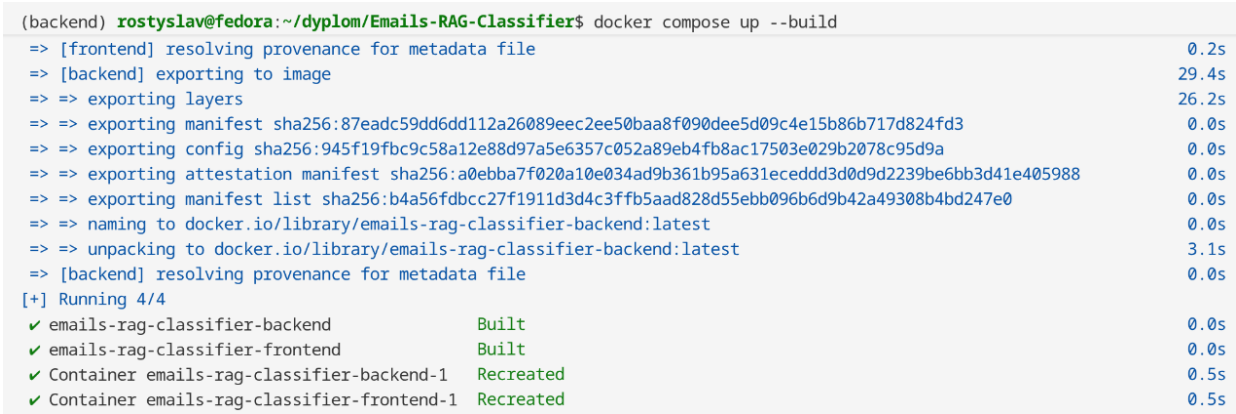


Рисунок 3.14 − Успішне збирання Docker-образів серверної та клієнтської частин

При першому запуску Docker автоматично встановлює залежності та завантажує моделі: all-MiniLM-L6-v2 (~22 МБ, кешується у backend/model_cache/huggingface) та CLIP ViT-B/32 (~338 МБ, кешується у backend/model_cache/chroma). Повторні запуски використовують кеш та стартують значно швидше. Після успішного запуску веб-інтерфейс доступний за адресою http://localhost:8501.

```
Attaching to backend-1, frontend-1
backend-1  | INFO:     Started server process [1]
backend-1  | INFO:     Waiting for application startup.
backend-1  | 2026-05-29 19:39:37,261 - root - INFO - Parsing model identifier. Schema: None, Identifier: ViT-B-32
backend-1  | 2026-05-29 19:39:37,261 - root - INFO - Loaded built-in ViT-B-32 model config.
backend-1  | 2026-05-29 19:39:37,532 - httpx - INFO - HTTP Request: HEAD https://huggingface.co/laion/CLIP-ViT-B-32-laion2B-s
34B-b79K/resolve/main/open_clip_model.safetensors "HTTP/1.1 302 Found"
backend-1  | Warning: You are sending unauthenticated requests to the HF Hub. Please set a HF_TOKEN to enable higher rate lim
its and faster downloads.
backend-1  | 2026-05-29 19:39:37,533 - huggingface_hub.utils._http - WARNING - Warning: You are sending unauthenticated reque
sts to the HF Hub. Please set a HF_TOKEN to enable higher rate limits and faster downloads.
backend-1  | 2026-05-29 19:39:37,533 - root - INFO - Instantiating model architecture: CLIP
backend-1  | 2026-05-29 19:39:38,420 - root - INFO - Loading full pretrained weights from: /root/.cache/huggingface/hub/model
s--laion--CLIP-ViT-B-32-laion2B-s34B-b79K/snapshots/1a25a446712ba5ee05982a381eed697ef9b435cf/open_clip_model.safetensors
backend-1  | 2026-05-29 19:39:39,012 - root - INFO - Final image preprocessing configuration set: {'size': (224, 224), 'mode'
: 'RGB', 'mean': (0.48145466, 0.4578275, 0.40821073), 'std': (0.26862954, 0.26130258, 0.27577711), 'interpolation': 'bicubic'
, 'resize_mode': 'shortest', 'fill_color': 0}
backend-1  | 2026-05-29 19:39:39,012 - root - INFO - Model ViT-B-32 creation process complete.
backend-1  | 2026-05-29 19:39:39,013 - root - INFO - Parsing tokenizer identifier. Schema: None, Identifier: ViT-B-32
backend-1  | 2026-05-29 19:39:39,013 - root - INFO - Attempting to load config from built-in: ViT-B-32
backend-1  | 2026-05-29 19:39:39,013 - root - INFO - Using default SimpleTokenizer.
backend-1  | INFO:     Application startup complete.
backend-1  | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
backend-1  | INFO:     127.0.0.1:38934 - "GET /health HTTP/1.1" 200 OK
```

Рисунок 3.15 − Завантаження моделі CLIP ViT-B/32 та запуск серверної частини

```
frontend-1  |
frontend-1  | Collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.
frontend-1  |
frontend-1  |
frontend-1  |   You can now view your Streamlit app in your browser.
frontend-1  |
frontend-1  |   Local URL: http://localhost:8501
frontend-1  |   Network URL: http://172.20.0.3:8501
frontend-1  |   External URL: http://95.158.43.50:8501
frontend-1  |
```

Рисунок 3.16 − Успішний запуск клієнтської частини, застосунок доступний за адресою http://localhost:8501

Для розгортання без Docker кожен сервіс запускається окремо за допомогою пакетного менеджера uv:

```
cd backend && uv sync && uv run uvicorn main:app --host 0.0.0.0 --port 8000
```

```
cd frontend && uv sync && uv run streamlit run streamlit_app.py
```

Ключові залежності серверної частини.

fastapi та uvicorn — фреймворк та ASGI-сервер;

chromadb — векторна база даних з вбудованою підтримкою sentence-transformers;

open-clip-torch та torch — мультимодальна модель CLIP[24];

playwright — браузерна автоматизація для верифікації КЕП;

imap-tools — бібліотека для роботи з IMAP;

google-genai — клієнт Google Gemini API;

pymupdf — конвертація PDF-сторінок у зображення для CLIP.

Дані та персистентність.

Усі дані зберігаються локально у директорії backend/data/: файл emails.db (SQLite), директорія chroma_db (ChromaDB), директорія attachments (вкладення листів). При розгортанні через Docker Compose ця директорія монтується як том, що забезпечує збереження даних між перезапусками контейнерів.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі виконано прикладну перевірку розробленої системи автоматизації обробки листів абітурієнтів: продемонстровано повний робочий цикл оператора приймальної комісії, проведено тестування всіх компонентів конвеєру обробки та складено програмну документацію.

Для підтвердження практичної цінності системи наведено покрокову демонстрацію всіх функціональних можливостей у контексті реального сценарію вступної кампанії.

Показано процес автентифікації та інкрементальної синхронізації поштової скриньки з підтримкою відкладеного завантаження вмісту листів, налаштування правил класифікації з регульованим порогом семантичної подібності, а також запуск RAG-конвеєру, який стабільно розпізнає листи-заявки за структурним форматом теми незалежно від конкретного прізвища вступника.

Продемонстровано верифікацію вкладених документів у потоковому режимі з мультимодальною класифікацією типів файлів та перевіркою кваліфікованого електронного підпису, результати якої відображаються як кольорові індикатори безпосередньо у списку листів.

Показано генерацію персоналізованої чернетки відповіді засобами LLM з дотриманням конфіденційності та обов'язковою валідацією оператором перед відправкою, а також зведений статистичний огляд кампанії у вигляді інтерактивних діаграм.

Тестування системи проводилося на трьох рівнях: юніт-тестування ізольованих функцій, інтеграційне тестування взаємодії між компонентами та функціональне тестування повного робочого циклу оператора методом «чорного ящика».

Усі десять ключових сценаріїв, від автентифікації до відправки відповіді, отримали результат «Пройдено», що підтверджує відповідність реалізованих можливостей задекларованим вимогам технічного завдання.

Програмна документація визначає мінімальні апаратні вимоги (CPU з підтримкою AVX-інструкцій, не менше 8 ГБ оперативної пам’яті (ОП), не менше 10 ГБ дискового простору), описує конфігурацію середовища через таблицю змінних оточення та процедуру розгортання системи єдиною командою docker compose up --build з автоматичним завантаженням моделей all-MiniLM-L6-v2 та CLIP ViT-B/32 при першому запуску.

Таким чином, у третьому розділі підтверджено практичну цінність розробленої системи: всі компоненти конвеєру функціонують у взаємодії, повний цикл від підключення поштового сервера до відправки верифікованої відповіді абітурієнту виконується без ручного втручання, а реалізована функціональність у повному обсязі відповідає вимогам технічного завдання.

ВИСНОВКИ

У дипломному проєкті розроблено спеціалізовану систему автоматизованої обробки листів абітурієнтів приймальної комісії закладу вищої освіти з використанням штучного інтелекту на основі архітектури RAG.

Аналіз існуючих комерційних рішень показав, що сучасні поштові сервіси орієнтовані на широкий корпоративний документообіг та не забезпечують необхідного рівня гнучкості для специфічних задач вступної кампанії: жодне з розглянутих рішень не підтримує семантичної класифікації за довільними тематичними правилами, мультимодального аналізу вкладених документів та верифікації кваліфікованого електронного підпису відповідно до вимог українського законодавства. Це підтвердило доцільність розробки власної системи.

На основі порівняльного аналізу методологій побудови компонентів RAG-архітектури обґрунтовано вибір технологічного стеку: модель all-MiniLM-L6-v2 для векторного представлення текстів, ChromaDB як векторна база даних з вбудованою персистентністю, мультимодальна модель CLIP ViT-B/32 для zero-shot класифікації типів документів без необхідності розміченої навчальної вибірки, та браузерна автоматизація Playwright для інтеграції з державним порталом id.gov.ua.

Розроблено п'ятиетапний конвеєр обробки даних, що охоплює синхронізацію листів через протокол IMAP, семантичне індексування та класифікацію вхідної кореспонденції, потокову верифікацію вкладених документів у реальному часі та генерацію персоналізованих чернеток відповідей засобами великої мовної моделі. Архітектура системи побудована на принципі модульності та розділення відповідальностей, що забезпечує незалежність компонентів та можливість їх вдосконалення без впливу на решту системи.

Захист персональних даних реалізовано на архітектурному рівні: до зовнішніх хмарних сервісів передаються виключно знеособлені технічні мітки без персональної інформації абітурієнта.

Практичну цінність розробленої системи підтверджено трирівневим тестуванням. Усі десять ключових функціональних сценаріїв, від автентифікації через IMAP до відправки відповіді через SMTP отримали результат «Пройдено». Повний цикл опрацювання звернення вступника виконується без ручного втручання оператора та відповідає задекларованим вимогам технічного завдання. Система розгортається на стандартному серверному обладнанні без графічного прискорювача єдиною командою, що забезпечує її практичну придатність для впровадження у реальних умовах вступної кампанії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gmail Smart Labels. Google LLC. URL: https://support.google.com/mail/answer/3094499 (дата звернення: 25.05.2026).

2. Focused Inbox for Outlook. Microsoft Corporation. URL: https://support.microsoft.com/en-us/office/focused-inbox-for-outlook (дата звернення: 25.05.2026).

3. SaneBox — email management service. SaneBox Inc. URL: https://www.sanebox.com (дата звернення: 25.05.2026).

4. Superhuman — the fastest email experience ever made. Superhuman Inc. URL: https://superhuman.com (дата звернення: 25.05.2026).

5. Spark — smart email app by Readdle. Readdle Inc. URL: https://sparkmailapp.com (дата звернення: 25.05.2026).

6. Lewis P., Perez E., Piktus A. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems. 2020. Vol. 33. P. 9459–9474. arXiv:2005.11401.

7. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of NAACL-HLT 2019. 2019. P. 4171–4186. arXiv:1810.04805.

8. Reimers N., Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of EMNLP 2019. 2019. P. 3982–3992. arXiv:1908.10084.

9. Wang W., Wei F., Dong L. et al. MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. Advances in NeurIPS 2020. Vol. 33. P. 5776–5788. arXiv:2002.10957.

10. Johnson J., Douze M., Jégou H. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data. 2021. Vol. 7. No. 3. P. 535–547. arXiv:1702.08734.

11. Chroma: the AI-native open-source embedding database. Chroma Core, 2024. URL: https://docs.trychroma.com (дата звернення: 25.05.2026).

12. Manning C. D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge : Cambridge University Press, 2008. 496 p.

13. Radford A., Kim J. W., Hallacy C. et al. Learning Transferable Visual Models From Natural Language Supervision. Proceedings of ICML 2021. 2021. P. 8748–8763. arXiv:2103.00020.

14. Schuhmann C., Beaumont R., Vencu R. et al. LAION-5B: An open large-scale dataset for training next generation image-text models. Advances in NeurIPS 2022. Vol. 35. arXiv:2210.08402.

15. Playwright for Python. Microsoft, 2024. URL: https://playwright.dev/python (дата звернення: 25.05.2026).

16. Портал перевірки кваліфікованих електронних підписів. Міністерство цифрової трансформації України. URL: https://id.gov.ua (дата звернення: 25.05.2026).

17. Про електронні довірчі послуги : Закон України від 05.10.2017 № 2155-VIII. URL: https://zakon.rada.gov.ua/laws/show/2155-19 (дата звернення: 25.05.2026).

18. ДСТУ ГОСТ 34.310:2004. Інформаційні технології. Криптографічний захист інформації. Процедури вироблення та перевірки електронного цифрового підпису на основі асиметричного криптографічного алгоритму. Київ : Держспоживстандарт України, 2005. 12 с.

19. RFC 9051: Internet Message Access Protocol (IMAP) — Version 4rev2 . IETF, 2021. URL: https://datatracker.ietf.org/doc/html/rfc9051 (дата звернення: 25.05.2026).

20. FastAPI: modern, fast web framework for building APIs with Python . S. Ramírez, 2024. URL: https://fastapi.tiangolo.com (дата звернення: 25.05.2026) .

21. Streamlit: a faster way to build and share data apps. Snowflake Inc., 2024. URL: https://docs.streamlit.io (дата звернення: 25.05.2026).

22. SQLite: self-contained, serverless, zero-configuration database engine. D. Richard Hipp. URL: https://www.sqlite.org/docs.html (дата звернення: 25.05.2026).

23. imap-tools: A high-level working with email by IMAP for Python. URL: https://github.com/ikvk/imap_tools (дата звернення: 25.05.2026).

24. open_clip: Open Source implementation of CLIP. LAION-AI, 2024. URL: https://github.com/mlfoundations/open_clip (дата звернення: 25.05.2026).

ДОДАТОК А

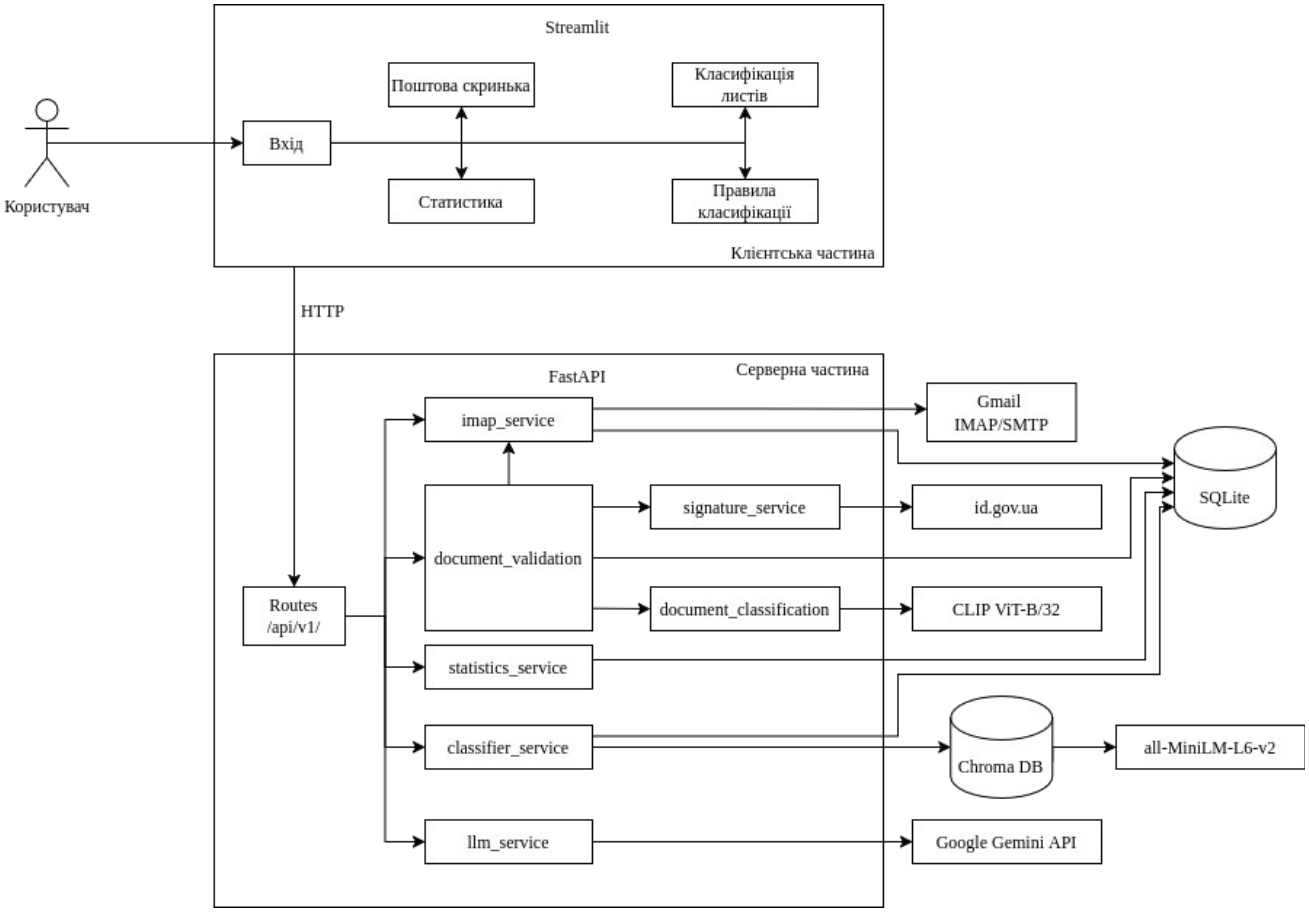
Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG

Структура системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.004 Д1	
		№ докум.	Підпис	Дата		
Розробив		Гундарев Р. Д.			**Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG** Структура системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Туганських О. А.				
Реценз.		Тарасенко-Клятченко О. В.				***КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21***
Н. Контр.		Міщенко Л. Д.				
Затвердив		Волокита А. М.				

ДОДАТОК Б

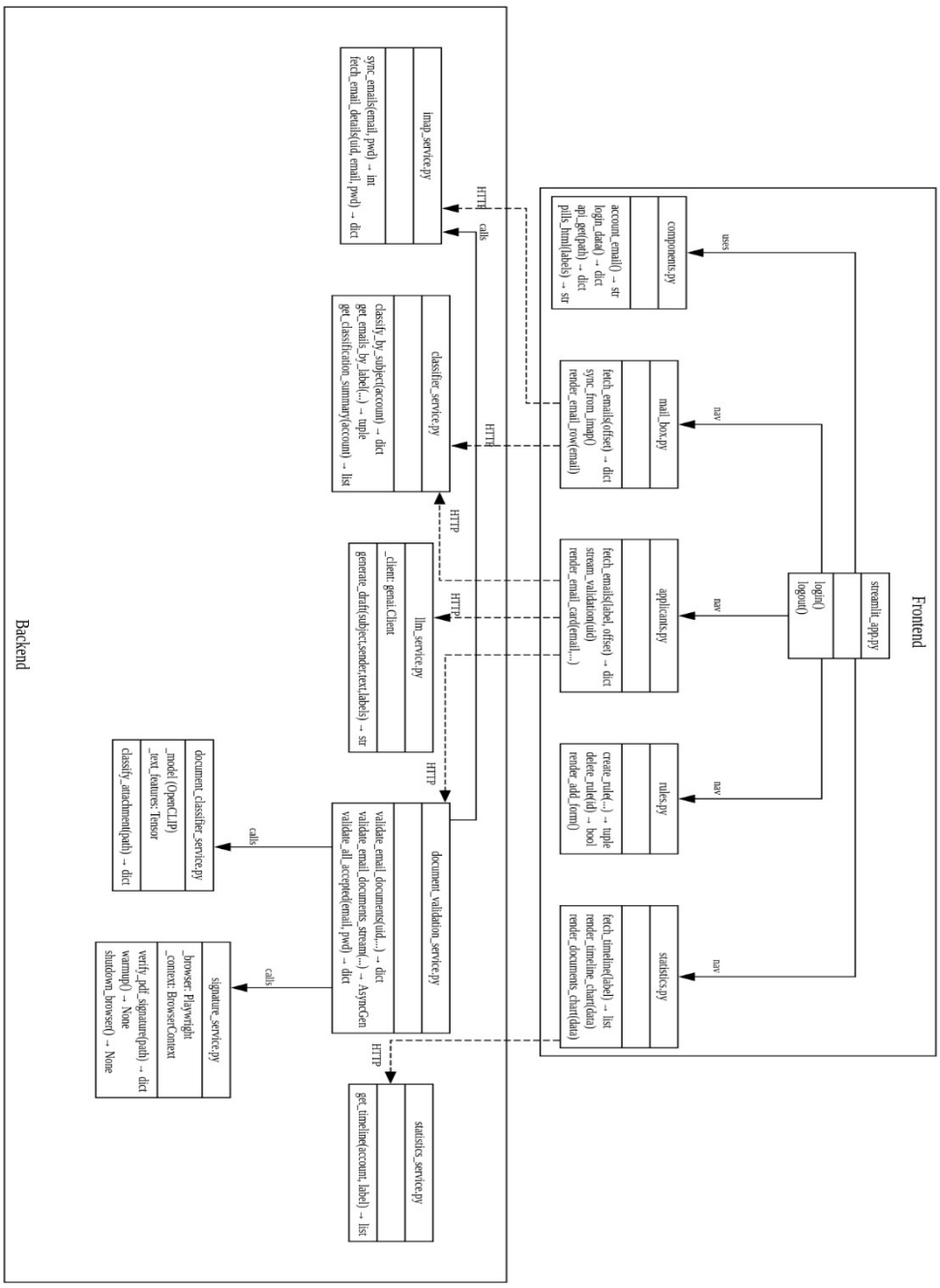
Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG

Діаграма класів

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.005 Д2
		№ докум.	Підпис	Дата	
Розробив		Гундарев Р. Д.			**Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG** Діаграма класів
Перевірив		Туганських О. А.			
Реценз.		Тарасенко-Клятченко О. В.			
Н. Контр.		Міщенко Л. Д.			
Затвердив		Волокита А. М.			

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21

ДОДАТОК В

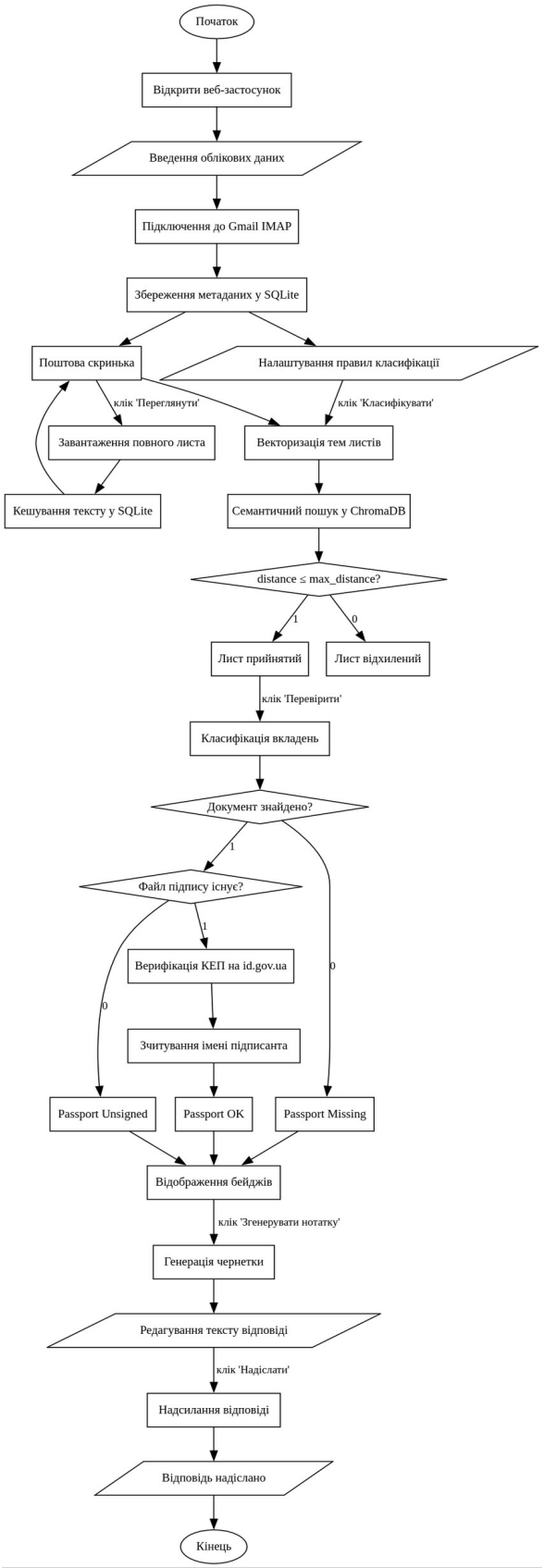
Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.006 ДЗ	
		№ докум.	Підпис	Дата		
Розробив		Гундарев Р. Д.			**Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG** Алгоритм дій програмного забезпечення	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Туганських О. А.				
Реценз.		Тарасенко-Клятченко О. В.				***КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-21***
Н. Контр.		Міщенко Л. Д.				
Затвердив		Волокита А. М.				

ДОДАТОК Г

Система автоматизації обробки листів абітурієнтів з використанням штучного інтелекту на основі архітектури RAG

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 25

Київ 2026 р

Повну кодову базу системи можна знайти в репозиторії GitHub: https://github.com/heereenveen/Emails-RAG-Classifier.

frontend/components.py

```
import os
import requests
import streamlit as st

BACKEND_URL = os.getenv("BACKEND_URL", "http://127.0.0.1:8000")
API_BASE = f"{BACKEND_URL}/api/v1"

def account_email() -> str:
    return st.session_state.get("login_data", {}).get("email_address",
"")
def login_data() -> dict | None:
    return st.session_state.get("login_data")

def body_box(html_text: str) -> None:
    st.markdown(
        f'<div style="background: #f9f9f9; padding: 15px; border-radius:
5px; '
        f'border: 1px solid #ddd; color: #333;">{html_text}</div>',
        unsafe_allow_html=True,
    )
 def pills_html(labels: list[str], badge_map: dict[str, tuple[str,
str]]) -> str:
      parts = []
      for label in labels:
          if label not in badge_map:
              continue
          color, text = badge_map[label]
          parts.append(
              f'<span style="background:{color};color:#fff;padding:2px
10px;'
              f'border-radius:999px;font-size:11px;font-weight:500;'
              f'display:inline-block;margin-top:4px;">{text}</span>'
          )
      return " ".join(parts)

def api_get(path: str, **params) -> dict | list | None:
        resp = requests.get(f"{API_BASE}{path}", params=params or None,
timeout=10)
        if resp.status_code == 200:
            return resp.json()
    return None
def fetch_rules() -> list[dict]:
    return api_get("/rules") or []
```

frontend/pages/applicants.py

```
    import json
    import requests
    import streamlit as st
    from components import API_BASE, account_email, api_get, body_box,
fetch_rules, login_data, pills_html

    LIMIT = 50
    REJECTED_LABEL = "Email Subject"

    DOCUMENT_BADGE = {
        "Passport OK": ("#28a745", "Паспорт ✓"),
        "Passport Unsigned": ("#0d6efd", "Паспорт ~"),
        "Passport Missing": ("#dc3545", "Паспорт ✗"),
        "Application OK": ("#28a745", "Заява ✓"),
        "Application Unsigned": ("#0d6efd", "Заява ~"),
        "Application Missing": ("#dc3545", "Заява ✗"),
        "Military OK": ("#28a745", "Резерв+ ✓"),
        "Military Unsigned": ("#0d6efd", "Резерв+ ~"),
        "Military Missing": ("#dc3545", "Резерв+ ✗"),
        "Photos OK": ("#28a745", "Фото ✓"),
        "Photos Missing": ("#dc3545", "Фото ✗"),
    }

    SENT_BADGE_HTML = (
        '<div style="background:#28a745;color:#fff;padding:6px 16px;'
        'border-radius:8px;text-align:center;font-size:14px;'
        'display:flex;align-items:center;justify-content:center;gap:6px">'
        '<span>✓</span><span>Відповідь надіслано</span></div>'
    )

    st.session_state.setdefault("classification_result", None)

    def fetch_emails(label: str, offset: int, include_passport_labels: bool
= False) -> dict | None:
        return api_get(
            "/classification/by_label",
            label=label,
            limit=LIMIT,
            offset=offset,
            include_passport_labels=include_passport_labels,
            account_email=account_email(),
        )
    def fetch_email_details(uid: str) -> dict | None:
        creds = login_data()
        if creds is None:
            return None
        try:
            resp = requests.post(f"{API_BASE}/{uid}/details", json=creds,
timeout=30)
            if resp.status_code == 200:
                return resp.json()
```

```
    st.error("Помилка завантаження")
        except Exception as e:
            st.error(f"Помилка: {e}")
        return None

    def generate_draft_request(uid: str) -> str | None:
        try:
            resp = requests.post(f"{API_BASE}/{uid}/draft", timeout=120)
            if resp.status_code == 200:
                return resp.json().get("draft", "")
            st.error(f"Помилка: {resp.status_code}")
        except Exception as e:
            st.error(f"Помилка зв'язку: {e}")
        return None

    def send_reply_request(uid: str, body: str) -> bool:
        creds = login_data()
        if creds is None:
            return False
        try:
            resp = requests.post(
                f"{API_BASE}/{uid}/reply",
                json={**creds, "body": body},
                timeout=30,
            )
            if resp.status_code == 200:
                return True
            st.error(f"Помилка: {resp.status_code} {resp.text}")
        except Exception as e:
            st.error(f"Помилка зв'язку: {e}")
        return False

    def render_pagination(offset_key: str, total: int):
        offset = st.session_state[offset_key]
        start = offset + 1
        end = min(offset + LIMIT, total)
        txt_col, prev_col, next_col = st.columns([8, 0.5, 0.5],
vertical_alignment="center")
        txt_col.write(f"Показано **{start}–{end}** з **{total}** листів")
        if prev_col.button("", icon=":material/arrow_back:",
disabled=(offset == 0), key=f"prev_{offset_key}"):
            st.session_state[offset_key] = max(0, offset - LIMIT)
            st.rerun()
        if next_col.button("", icon=":material/arrow_forward:",
disabled=(offset + LIMIT >= total), key=f"next_{offset_key}"):
            st.session_state[offset_key] = offset + LIMIT
            st.rerun()
    def render_email_header(email: dict, show_labels: bool, show_distance:
bool):
        c1, c2, c3, c4 = st.columns([2.5, 3.5, 1, 1])
        with c1:
            nickname = email.get("from_nickname", "")
            from_email = email.get("from_email", "")
            st.write(f"**{nickname}**" if nickname else f"**{from_email}**")
```

```
    if show_distance and distance is not None:
            c3.caption(f"Відстань: {distance:.3f}")
        c4.write(email.get("time", ""))

    def stream_validation(uid: str):
        creds = login_data()
        if creds is None:
            return
        with st.status("Перевірка документів...", expanded=True) as
verify_status:
            try:
                resp =
requests.post(f"{API_BASE}/{uid}/classify/documents/stream", json=creds,
stream=True, timeout=180)
                step_num = 0
                for raw_line in resp.iter_lines(decode_unicode=True):
                    if not raw_line or not raw_line.startswith("data: "):
                        continue
                    event = json.loads(raw_line[6:])
                    step = event.get("step")
                    message = event.get("message", "")
                    if step == "error":
                        st.error(message)
                        verify_status.update(label="Помилка перевірки",
state="error")
                        break
                    if step == "done":
                        labels = event.get("labels", [])
                        verify_status.update(label=f"Перевірено: {',
'.join(labels)}", state="complete", expanded=False)
                        break
                    step_num += 1
                    st.write(f"{step_num}. {message}")
                    if signer := event.get("signer"):
                        st.info(f"Підписант: {signer}")
                resp.close()
            except Exception as e:
                st.error(f"Помилка зв'язку: {e}")
                verify_status.update(label="Помилка зв'язку", state="error")
    def render_action_button(uid: str, key_prefix: str, email: dict) ->
tuple[bool, bool]:
        already_verified = any(lbl in DOCUMENT_BADGE for lbl in
email.get("labels", []))
        btn_key = f"btn_validate_{key_prefix}{uid}"
        disabled = login_data() is None
        draft_ready = bool(st.session_state.get(f"draft_{uid}"))
        sent_ready = st.session_state.get(f"reply_sent_{key_prefix}{uid}",
False)
        if not already_verified:
            clicked = st.button("Перевірити", key=btn_key,
icon=":material/verified:", disabled=disabled, use_container_width=True)
            return clicked, False
        if sent_ready:
            st.markdown(SENT_BADGE_HTML, unsafe_allow_html=True)
            return False, False
```

```
    clicked = st.button("Згенерувати нотатку", key=btn_key,
icon=":material/edit_note:", disabled=disabled, use_container_width=True)
        return False, clicked

    def render_attachments(uid: str, key_prefix: str, attachments: list):
        if not attachments:
            return
        st.markdown("<div style='margin-top:12px'></div>",
unsafe_allow_html=True)
        st.write("**Вкладення:**")
        for att in attachments:
            fname = att if isinstance(att, str) else att.get("filename")
            dl_url = f"{API_BASE}/{uid}/attachments/{fname}"
            st.download_button(label=f"{fname}",
data=requests.get(dl_url).content, file_name=fname, key=f"dl_{key_prefix}
{uid}_{fname}")

    def render_originals(uid: str, key_prefix: str):
        try:
            originals = requests.get(f"{API_BASE}/{uid}/originals",
timeout=5).json()
        except Exception:
            return
        if not originals:
            return
        try:
            meta = requests.get(f"{API_BASE}/{uid}/originals-meta",
timeout=5).json()
        except Exception:
            meta = {}
        st.write("**Оригінали:**")
        for fname in originals:
            display = fname.rsplit(".", 1)[0] if "." in fname else fname
            col_dl, col_signer = st.columns([3, 2])
            with col_dl:
                st.download_button(label=f"{display}",
data=requests.get(f"{API_BASE}/{uid}/originals/{fname}").content,
file_name=fname, key=f"orig_{key_prefix}{uid}_{fname}")
            if signer := meta.get(display):
                col_signer.caption(f"Підписав: {signer}")
    def render_draft_section(uid: str, key_prefix: str):
        draft = st.session_state.get(f"draft_{uid}")
        if not draft:
            return
        edited = st.text_area("Нотатка", value=draft, height=250,
key=f"draft_text_{key_prefix}{uid}")
        sent_key = f"reply_sent_{key_prefix}{uid}"
        if st.session_state.get(sent_key):
            return
        if st.button("Надіслати відповідь", key=f"send_reply_{key_prefix}
{uid}", icon=":material/send:", disabled=login_data() is None):
            if send_reply_request(uid, edited):
                st.session_state[sent_key] = True
                st.rerun()
```

frontend/pages/mail_box.py

```
import requests
import streamlit as st

from components import API_BASE, BACKEND_URL, body_box, login_data

EMAILS_API = f"{API_BASE}/emails"
LIMIT = 50

st.title("Поштова скринька", text_alignment="center")

if "email_offset_all" not in st.session_state:
    st.session_state.email_offset_all = 0

def fetch_emails(offset: int):
    creds = login_data() or {}
    try:
        response = requests.get(
            EMAILS_API,
            params={"limit": LIMIT, "offset": offset, "account_email":
creds.get("email_address", "")},
            timeout=10,
        )
        if response.status_code == 200:
            return response.json()
    except Exception:
        pass
    return None

def sync_from_imap():
    creds = login_data()
    if not creds:
        st.session_state["_toast"] = ("error", "Спочатку увійдіть в
систему.")
        return
    try:
 r = requests.post(f"{BACKEND_URL}/api/v1/login_imap", json=creds,
timeout=60)
        if r.status_code != 200:
            st.session_state["_toast"] = ("error", f"Помилка
синхронізації: {r.text}")
            return
        synced = r.json().get("synced", 0)
    except Exception as e:
        st.session_state["_toast"] = ("error", f"Помилка з'єднання:
{e}")
        return
    st.session_state["_toast"] = ("success", f"Синхронізовано {synced}
нових листів.")
```

```
    def render_toolbar(total: int, offset: int):
        start = offset + 1
        end = min(offset + LIMIT, total)
        sync_col, _, txt_col, prev_col, next_col = st.columns(
            [0.5, 6.5, 1, 0.5, 0.5], vertical_alignment="center"
        )
        sync_col.button(
            "", icon=":material/sync:", help="Оновити листи з пошти",
            on_click=sync_from_imap, key="sync_btn",
        )
        txt_col.write(f"{start}–{end} з {total:,}")
        if prev_col.button(
            "", icon=":material/arrow_back:",
            disabled=(offset == 0), key="prev",
        ):
            st.session_state.email_offset_all = max(0, offset - LIMIT)
            st.rerun()
        if next_col.button(
            "", icon=":material/arrow_forward:",
            disabled=(offset + LIMIT >= total), key="next",
        ):
            st.session_state.email_offset_all = offset + LIMIT
            st.rerun()
    ef fetch_email_details(uid: str) -> dict | None:
        creds = login_data()
        if creds is None:
            return None
        try:
            resp = requests.post(f"{API_BASE}/{uid}/details", json=creds,
timeout=30)
            if resp.status_code == 200:
                return resp.json()
            st.error("Помилка завантаження з сервера.")
        except Exception as e:
            st.error(f"Помилка зв'язку: {e}")
        return None


    def render_attachment_buttons(uid: str, attachments: list):
        if not attachments:
            return
        st.write("---")
        st.write("**Вкладення:**")
        for att in attachments:
            fname = att if isinstance(att, str) else att.get("filename")
            dl_url = f"{API_BASE}/{uid}/attachments/{fname}"
            st.download_button(
                label=f"{fname}",
                data=requests.get(dl_url).content,
                file_name=fname,
                key=f"dl_{uid}_{fname}",
            )
```

frontend/pages/rules.py

```
mport requests
import streamlit as st

from components import API_BASE, fetch_rules

RULES_API = f"{API_BASE}/rules"

st.title("Правила класифікації", text_alignment="center")

def delete_rule(rule_id: int) -> bool:
    try:
        return requests.delete(f"{RULES_API}/{rule_id}",
timeout=10).status_code == 204
    except Exception:
        return False

def create_rule(label: str, reference_query: str, max_distance: float)
-> tuple[bool, str]:
    try:
        r = requests.post(
            RULES_API,
            json={"label": label, "reference_query": reference_query,
"max_distance": max_distance},
            timeout=10,
        )
        if r.status_code == 201:
            return True, ""
        return False, r.json().get("detail", r.text)
    except Exception as e:
        return False, str(e)

def render_existing_rules(rules: list[dict]):
    if not rules:
        st.info("Правил немає. Додайте перше правило нижче.")
        return

    st.subheader("Існуючі правила")
    for rule in rules:
        col_label, col_query, col_dist, col_del = st.columns([2, 4, 1,
1])
        col_label.write(f"**{rule['label']}**")
        col_query.write(rule["reference_query"])
        col_dist.write(str(rule["max_distance"]))
        if col_del.button("", key=f"del_{rule['id']}"):
            if delete_rule(rule["id"]):
                st.success(f"Правило '{rule['label']}' видалено")
                st.rerun()
            else:
                st.error("Помилка видалення")
```

```
    def render_add_form():
         st.subheader("Додати правило")
         with st.form("add_rule"):
             label = st.text_input("Мітка", placeholder="Прийнято/Тема")
             query = st.text_input(
                 "Зразок теми",
                 placeholder="Бакалавр — бюджет. Шевченко Тарас Григорович",
             )
             distance = st.slider("Максимальна відстань", min_value=0.1,
max_value=1.0, value=0.5, step=0.05)

             if not st.form_submit_button("Додати",
use_container_width=True):
                 return
             if not label.strip() or not query.strip():
                 st.error("Заповніть мітку та зразок теми")
                 return

             ok, err = create_rule(label.strip(), query.strip(), distance)
             if ok:
                 st.success(f"Правило '{label}' додано")
                 st.rerun()
             else:
                 st.error(f"Помилка: {err}")


    render_existing_rules(fetch_rules())
    st.divider()
    render_add_form()
```

frontend/pages/statistics.py

```
    import altair as alt
    import pandas as pd
    import streamlit as st

    from components import account_email, api_get, fetch_rules

    ALL_DOCUMENT_LABELS = [label for group in DOC_GROUP_MAP.values() for
label in group]

    STATUS_COLOR = alt.Scale(
        domain=["OK", "Без підпису", "Відсутній"],
        range=["#28a745", "#0d6efd", "#dc3545"],
    )


    def fetch_timeline(label: str) -> list[dict]:
        return api_get("/statistics/timeline",
account_email=account_email(), label=label) or []


    def fetch_summary() -> list[dict]:
        return api_get("/classification/summary",
account_email=account_email()) or []
```

```
      def render_timeline_chart(timeline_data: list[dict]):
    st.subheader("Надходження листів")
    selected_series = st.pills(
        "Показати категорії листів:",
        SERIES_OPTIONS,
        selection_mode="multi",
        default=["Не перевірені", "Перевірено"],
        label_visibility="visible",
    )

    if not timeline_data:
        st.info("Немає даних для часового графіка.")
        return
    if not selected_series:
        st.caption("Оберіть хоча б одну категорію.")
        return

    df = pd.DataFrame(timeline_data)
    df["date"] = pd.to_datetime(df["date"])

    cols_to_show = [SERIES_COL[s] for s in selected_series]
    melted = df[["date"] + cols_to_show].melt(
        id_vars="date", var_name="series_key", value_name="Кількість"
    )
    melted["Категорія"] = melted["series_key"].map({v: k for k, v in
SERIES_COL.items()})

    color_scale = alt.Scale(
        domain=list(SERIES_COLOR.keys()),
        range=list(SERIES_COLOR.values()),
    )

    chart = (
        alt.Chart(melted)
        .mark_bar()
        .encode(
            x=alt.X("date:T", title="Дата", axis=alt.Axis(format="%d %b %Y",
labelAngle=-30)),
            y=alt.Y("Кількість:Q", title="Кількість листів"),
            color=alt.Color("Категорія:N", scale=color_scale, title="Категорія"),
            xOffset=alt.XOffset("Категорія:N"),
            tooltip=[
                alt.Tooltip("date:T", title="Дата", format="%d.%m.%Y"),
                alt.Tooltip("Категорія:N"),
                alt.Tooltip("Кількість:Q"),
            ],
        )
        .properties(width="container", height=320)
    )
    st.altair_chart(chart, use_container_width=True)
```

```
def render_documents_chart(summary_data: list[dict]):
st.subheader("Перевірка документів")

counts = {item["label"]: item["total"] for item in summary_data}
if not any(counts.get(lbl, 0) > 0 for lbl in ALL_DOCUMENT_LABELS):
    st.info(
        "Немає даних про перевірку документів. "
        "Використайте кнопку «Перевірити» у вкладці прийнятих листів."
    )
    return

doc_rows = [
    {"Документ": group, "Статус": status, "Кількість": counts.get(label, 0)}
    for group, label_map in DOC_GROUP_MAP.items()
    for label, status in label_map.items()
]
df_doc = pd.DataFrame(doc_rows)

chart = (
    alt.Chart(df_doc)
    .mark_bar()
    .encode(
        x=alt.X("Документ:N", title=None, sort=list(DOC_GROUP_MAP.keys())),
        y=alt.Y("Кількість:Q", title="Кількість листів"),
        color=alt.Color(
            "Статус:N",
            scale=STATUS_COLOR,
            title="Статус",
            sort=["ОК", "Без підпису", "Відсутній"],
        ),
        order=alt.Order("Статус:N", sort="descending"),
        tooltip=["Документ:N", "Статус:N", "Кількість:Q"],
    )
    .properties(width="container", height=320)
)
st.altair_chart(chart, use_container_width=True)

st.title("Статистика", text_alignment="center")

 rules = fetch_rules()
 summary_data = fetch_summary()

 rule_labels = [r["label"] for r in rules]
 selected_label = st.selectbox("Категорія:", ["Всі категорії"] + rule_labels,
 index=0)
 filter_label = "" if selected_label == "Всі категорії" else selected_label

 timeline_data = fetch_timeline(label=filter_label)

 if not timeline_data and not summary_data:
     st.info("Статистика відсутня. Запустіть класифікацію на сторінці
 «Абітурієнти».")
     st.stop()
```

frontend/streamlit_app.py

```
import os

import requests
import streamlit as st

st.set_page_config(layout="wide")

st.markdown(
    """<style>
    #MainMenu { display: none; }
    [data-testid="stMainMenu"] { display: none; }
    [data-testid="stDeployButton"] { display: none; }
    [data-testid="stToolbar"] > *:not([data-testid="stStatusWidget"])
{ display: none; }
    </style>""",
    unsafe_allow_html=True,
)

BACKEND_URL = os.getenv("BACKEND_URL", "http://127.0.0.1:8000")
LOGIN_API = f"{BACKEND_URL}/api/v1/login_imap"

if "role" not in st.session_state:
    st.session_state.role = None


def login():
    _, cent_co, _ = st.columns([2, 3, 2])

    with cent_co:
        with st.form("Вхід"):
            st.title("Вхід", text_alignment="center")

            email = st.text_input(
                label="Електронна пошта", placeholder="Адреса
електронної пошти", label_visibility="collapsed"
            )
            password = st.text_input(
                label="Пароль",
                placeholder="Пароль",
                type="password",
                max_chars=16,
                label_visibility="collapsed",
            )

            if st.form_submit_button("Увійти", width="stretch"):
                creds = {"email_address": email, "imap_password":
password}
                response = requests.post(LOGIN_API, json=creds)
```

```
     def logout():
     st.session_state.role = None
     st.rerun()

     role = st.session_state.role

     logout_page = st.Page(logout, title="Вийти", icon=":material/logout:")
     mail_box = st.Page(
        "pages/mail_box.py",
        title="Поштова скринька",
        icon=":material/stacked_email:",
        default=(role == "User"),
     )
     statistics = st.Page(
       "pages/statistics.py", title="Статистика",
icon=":material/query_stats:"
     )
     applicants = st.Page(
        "pages/applicants.py", title="Абітурієнти",
icon=":material/person_search:"
     )
     rules = st.Page(
        "pages/rules.py", title="Правила класифікації",
icon=":material/rule:"
     )

     account_pages = [logout_page]
     user_pages = [mail_box, applicants, rules, statistics]

     page_dict = {}

     if st.session_state.role == "User":
     page_dict["Сервіс"] = user_pages

     if len(page_dict) > 0:
        pg = st.navigation({"Account": account_pages} | page_dict)
     else:
        pg = st.navigation([st.Page(login)])

     pg.run()
```

services/imap_service.py

```
import json

from imap_tools import MailBox, AND

from app.config import GOOGLE_IMAP_SERVICE
from app.databases.sqlite_db import get_connection

def sync_emails(email_address: str, imap_password: str, limit: int =
500) -> int:
    with get_connection() as conn:
        row = conn.execute(
            "SELECT MAX(CAST(uid AS INTEGER)) FROM emails WHERE
account_email = ?",
            (email_address,),
        ).fetchone()
        max_uid = row[0] or 0

    with MailBox(GOOGLE_IMAP_SERVICE).login(
        email_address, imap_password, "INBOX"
    ) as mailbox:
        if max_uid > 0:
            messages = mailbox.fetch(
                AND(uid=f"{max_uid + 1}:*"),
                headers_only=True,
                bulk=100,
            )
        else:
            messages = mailbox.fetch(
                reverse=True, headers_only=True, limit=limit, bulk=100
            )

        empty_attachments = json.dumps([])
        rows_to_insert = []
        for msg in messages:
            nickname = msg.from_values.name if msg.from_values else ""
            email_addr = msg.from_values.email if msg.from_values else
msg.from_
            subject = msg.subject or ""

            if not email_addr and not subject:
                continue

            rows_to_insert.append(
                (msg.uid, email_address, nickname, email_addr,
msg.date_str, subject, empty_attachments)
            )

    if not rows_to_insert:
        return 0
```

```
with get_connection() as conn:
        cur = conn.executemany(
            """
                INSERT OR IGNORE INTO emails (uid, account_email,
from_nickname, from_email, time, subject, attachments)
                VALUES (?, ?, ?, ?, ?, ?, ?)
            """,
            rows_to_insert,
        )
        synced_count = cur.rowcount
        conn.commit()

    return synced_count

def fetch_email_details(uid: str, email_address: str, imap_password: str):

    with MailBox(GOOGLE_IMAP_SERVICE).login(
        email_address, imap_password, "INBOX"
    ) as mailbox:


        msg_gen = mailbox.fetch(AND(uid=uid))
        msg = next(msg_gen, None)

        if not msg:
            return None


        full_text = msg.html or msg.text or ""


        attachments_data = []
        for att in msg.attachments:
            attachments_data.append(
                {
                    "filename": att.filename,
                    "content_type": att.content_type,
                    "payload": att.payload,
                }
            )

        return {"text": full_text, "attachments": attachments_data}
```

services/document_validation_service.py

```
import json
import logging
import os
import sqlite3
import tempfile
from pathlib import Path
from typing import AsyncGenerator

from app.config import ATTACHMENTS_DIR
from app.databases.sqlite_db import get_connection
from app.labels import (
    CLIP_TO_CATEGORY,
    CLIP_TYPE_DISPLAY,
    DOC_CATEGORIES,
    DOCUMENT_LABELS,
    ORIGINAL_DISPLAY_NAMES,
    SUBJECT_ACCEPTED,
)
from app.services.document_classifier_service import classify_attachment
from app.services.imap_service import fetch_email_details
from app.services.signature_service import verify_pdf_signature

logger = logging.getLogger(__name__)

_IMAGE_SUFFIXES = frozenset((".pdf", ".jpg", ".jpeg", ".png", ".bmp",
".tiff", ".webp"))
_NO_SIG_CLIP_LABELS: frozenset[str] = frozenset(
    clip for cat in DOC_CATEGORIES if not cat.requires_signature for
clip in cat.clip_labels
)


def _ensure_attachments(uid: str, email_address: str, imap_password:
str) -> list[str]:
    att_dir = Path(ATTACHMENTS_DIR) / uid

    with get_connection() as conn:
        conn.row_factory = sqlite3.Row
        row = conn.execute(
            "SELECT attachments FROM emails WHERE uid = ?", (uid,)
        ).fetchone()

    saved_names = json.loads(row["attachments"] or "[]") if row else []

    if saved_names and att_dir.exists():
        return [str(att_dir / name) for name in saved_names if
(att_dir / name).exists()]

    details = fetch_email_details(uid, email_address, imap_password)
    if not details:
        logger.warning(f"Failed to fetch email {uid} from IMAP.")
        return []
```

```
 att_dir.mkdir(parents=True, exist_ok=True)
    saved_names = []
    for att in details["attachments"]:
        safe = os.path.basename(att["filename"])
        with open(att_dir / safe, "wb") as f:
            f.write(att["payload"])
        saved_names.append(safe)

    with get_connection() as conn:
        conn.execute(
            "UPDATE emails SET text = ?, attachments = ? WHERE uid = ?",
            (details["text"], json.dumps(saved_names), uid),
        )
        conn.commit()

    return [str(att_dir / name) for name in saved_names if (att_dir /
name).exists()]

def _replace_category_labels(uid: str, picked_labels: list[str]) -> None:
    placeholders = ",".join(["?"] * len(DOCUMENT_LABELS))
    with get_connection() as conn:
        conn.execute(
            f"DELETE FROM email_labels WHERE uid = ? AND label IN
({placeholders})",
            (uid, *DOCUMENT_LABELS),
        )
        conn.executemany(
            """INSERT OR IGNORE INTO email_labels (uid, label, distance,
synced_to_imap)
               VALUES (?, ?, NULL, 0)""",
            [(uid, label) for label in picked_labels],
        )
        conn.commit()


def _classify_content(content: bytes) -> dict:
    suffix = ".pdf" if content[:4] == b"%PDF" else ".bin"
    tmp_fd, tmp_path = tempfile.mkstemp(suffix=suffix)
    try:
        with os.fdopen(tmp_fd, "wb") as f:
            f.write(content)
        return classify_attachment(tmp_path)
    finally:
        try:
            os.unlink(tmp_path)
        except Exception:
            pass
```

```
def _save_original(uid: str, content: bytes, label: str, signer: str | None
= None) -> None:
    originals_dir = Path(ATTACHMENTS_DIR) / uid / "originals"
    originals_dir.mkdir(parents=True, exist_ok=True)
    valid_display_names = set(ORIGINAL_DISPLAY_NAMES.values()) |
{"Документ"}
    for f in originals_dir.iterdir():
        if f.suffix != ".signer" and f.stem not in valid_display_names:
            f.unlink(missing_ok=True)
    ext = ".pdf" if content[:4] == b"%PDF" else ".bin"
    display_name = ORIGINAL_DISPLAY_NAMES.get(label, "Документ")
    dest = originals_dir / f"{display_name}{ext}"
    dest.write_bytes(content)
    if signer:
        (originals_dir / f"{display_name}{ext}.signer").write_text(signer,
encoding="utf-8")

def _pick_labels(state: dict[str, dict[str, bool]]) -> list[str]:
    return sorted({
        cat.pick(state[cat.key]["found"], state[cat.key]["signed"])
        for cat in DOC_CATEGORIES
    })


def _mark_state(state: dict, clip_label: str, signed: bool) -> None:
    cat = CLIP_TO_CATEGORY.get(clip_label)
    if cat is None:
        return
    state[cat.key]["found"] = True
    if signed:
        state[cat.key]["signed"] = True


async def _process_p7s_file(
    uid: str, p7s_path: str, state: dict, info: dict
) -> AsyncGenerator[dict, None]:
    filename = os.path.basename(p7s_path)
    yield {"step": 2, "message": f'Знайдено підпис: "{filename}"'}

    yield {"step": 3, "message": "Валідація підпису на порталі..."}
    sig = await verify_pdf_signature(p7s_path)
    info["signature"] = {k: v for k, v in sig.items() if k != "content"}

    status_word = "успішно" if sig["valid"] else "невдало"
    yield {
        "step": 4,
        "message": f"Підпис {status_word} валідовано",
        "signer": sig.get("signer"),
        "valid": sig["valid"],
    }
```

```
yield {"step": 5, "message": "Завантаження оригіналу з порталу..."}
    content = sig.get("content")
    if not content:
        info["type"] = "extraction_failed"
        info["error"] = "Portal did not provide original document"
        yield {"step": 6, "message": "Портал не повернув оригінальний
документ"}
        return

    cls = _classify_content(content)
    info["type"] = cls["label"]
    info["confidence"] = cls["confidence"]
    info["scores"] = {k: round(v, 4) for k, v in cls["scores"].items()}
    _save_original(uid, content, cls["label"], sig.get("signer"))

    type_display = CLIP_TYPE_DISPLAY.get(cls["label"], cls["label"])
    yield {
        "step": 6,
        "message": f"Документ класифіковано: {type_display}",
    }

    _mark_state(state, cls["label"], signed=sig["valid"])
```

services/document_classifier_service.py

```
from pathlib import Path

import torch
import open_clip
from PIL import Image
import fitz


CLASS_KEYS = list(CLASS_PROMPTS.keys())

_model = None
_preprocess = None
_tokenizer = None
_text_features = None
_device = "cpu"


def _load_model():
    global _model, _preprocess, _tokenizer, _text_features
    if _model is not None:
        return
    model, _, preprocess = open_clip.create_model_and_transforms(
        "ViT-B-32", pretrained="laion2b_s34b_b79k"
    )
    tokenizer = open_clip.get_tokenizer("ViT-B-32")
    model.eval().to(_device)

    flat_prompts = []
    for key in CLASS_KEYS:
        flat_prompts.extend(CLASS_PROMPTS[key])
```

```
with torch.no_grad():
        tokens = tokenizer(flat_prompts).to(_device)
        text_feats = model.encode_text(tokens)
        text_feats = text_feats / text_feats.norm(dim=-1, keepdim=True)

    _model = model
    _preprocess = preprocess
    _tokenizer = tokenizer
    _text_features = text_feats

def _load_image(file_path: Path) -> Image.Image | None:

    suffix = file_path.suffix.lower()
    if suffix == ".pdf":
        try:
            doc = fitz.open(str(file_path))
            if doc.page_count == 0:
                return None
            page = doc.load_page(0)
            pix = page.get_pixmap(dpi=150)
            img = Image.frombytes("RGB", (pix.width, pix.height),
pix.samples)
            doc.close()
            return img
        except Exception:
            return None
    if suffix in (".jpg", ".jpeg", ".png", ".bmp", ".tiff", ".webp"):
        try:
            return Image.open(file_path).convert("RGB")
        except Exception:
            return None
    return None
def classify_attachment(file_path: str | Path) -> dict:

    file_path = Path(file_path)
    img = _load_image(file_path)
    if img is None:
        return {"label": "other", "confidence": 0.0, "scores": {}}

    _load_model()

    with torch.no_grad():
        img_tensor = _preprocess(img).unsqueeze(0).to(_device)
        image_feats = _model.encode_image(img_tensor)
        image_feats = image_feats / image_feats.norm(dim=-1, keepdim=True)
        sims = (image_feats @ _text_features.T).squeeze(0)


    scores: dict[str, float] = {}
    idx = 0
    for key in CLASS_KEYS:
        n = len(CLASS_PROMPTS[key])
        scores[key] = float(sims[idx : idx + n].mean().item())
        idx += n
```

```
 best_label = max(scores, key=scores.get)
 return {"label": best_label, "confidence": scores[best_label], "scores":
scores}
```

services/llm_service.py

```
import logging
import os
from google import genai
from google.genai.errors import ServerError

logger = logging.getLogger(__name__)

_client: genai.Client | None = None

SYSTEM_PROMPT = (
    "Ти — асистент приймальної комісії університету КПІ ім. Ігоря
Сікорського. "
    "Напиши коротку офіційну чернетку відповіді абітурієнту українською
мовою "
    "на основі наданих технічних міток класифікації. "
    "Не вигадуй інформацію поза межами міток. "
    "Якщо мітка вказує на відсутність або проблему з документом — повідом
про це. "
    "Якщо всі документи в порядку — підтверди отримання."
)


def _get_client() -> genai.Client:
    global _client
    if _client is None:
        api_key = os.environ.get("GEMINI_API_KEY")
        if not api_key:
            raise RuntimeError("GEMINI_API_KEY environment variable not
set")
        _client = genai.Client(api_key=api_key)
    return _client

def generate_draft(
    subject: str,
    sender: str,
    email_text: str,
    labels: list[str],
) -> str:
    client = _get_client()

    if not labels:
        raise ValueError("No labels provided — cannot generate anonymized
draft")

    user_msg = (
        "Мітки класифікації листа:\n"
        + "\n".join(f"- {label}" for label in labels)
        + "\n\nНапиши чернетку відповіді на основі цих міток."
    )
```

```
response = client.models.generate_content(
                model=model,
                contents=f"{SYSTEM_PROMPT}\n\n{user_msg}",
            )
return response.text.strip()
```

services/signature_service.py

```
import asyncio
from pathlib import Path

from playwright.async_api import async_playwright, Browser, BrowserContext,
TimeoutError as PWTimeout

_browser: Browser | None = None
_context: BrowserContext | None = None
_lock = asyncio.Lock()


async def _get_context() -> BrowserContext:
    global _browser, _context
    if _browser is None or not _browser.is_connected():
        pw = await async_playwright().start()
        _browser = await pw.chromium.launch(headless=True, args=["--no-
sandbox"])
        _context = None
    if _context is None:
        _context = await _browser.new_context()
    return _context


async def warmup() -> None:
    ctx = await _get_context()
    page = await ctx.new_page()
    try:
        await page.goto(VERIFY_WIDGET_URL, wait_until="networkidle")
        await page.wait_for_selector("#chooseFilesInput", state="attached",
timeout=30000)
    finally:
        await page.close()


async def shutdown_browser() -> None:
    global _browser, _context
    if _context is not None:
        await _context.close()
        _context = None
    if _browser is not None and _browser.is_connected():
        await _browser.close()
    _browser = None
```

```
async def verify_pdf_signature(file_path: str | Path, timeout_ms: int =
60000) -> dict:
    file_path = str(Path(file_path).resolve())

    async with _lock:
        ctx = await _get_context()
        page = await ctx.new_page()
        try:
            await page.goto(VERIFY_WIDGET_URL, wait_until="networkidle")
            await page.wait_for_selector("#chooseFilesInput",
state="attached", timeout=30000)

            await
page.locator("#chooseFilesInput").set_input_files(file_path)

            await page.wait_for_selector("#checkButton", state="visible",
timeout=15000)
            await page.click("#checkButton")

await page.wait_for_function(
                    """() => {
                        const result = document.querySelector('#result');
                        if (result && result.style.display !== 'none')
return true;
                        const err = document.querySelector('#errorBlock');
                        if (err && err.style.display !== 'none') return
true;
                        return false;
                    }""",
                    timeout=timeout_ms,
                )

save_btn = page.locator("#saveDataFileButton")
                await save_btn.wait_for(state="visible", timeout=10000)
                async with page.expect_download(timeout=15000) as dl_info:
                    await save_btn.click()
                download = await dl_info.value
                failure = await download.failure()
                if failure is None:
                    dl_path = await download.path()
                    if dl_path:
                        with open(dl_path, "rb") as f:
                            content = f.read()

            return {
                "signed": True,
                "valid": success,
                "details": full_text[:500],
                "signer": signer,
                "content": content,
            }
```

services/statistics_service.py

```
import sqlite3
from collections import defaultdict
from email.utils import parsedate_to_datetime

from app.databases.sqlite_db import get_connection
from app.labels import DOCUMENT_LABELS, SUBJECT_REJECTED


def _parse_date(time_str: str | None) -> str | None:
    if not time_str:
        return None
    try:
        return parsedate_to_datetime(time_str).strftime("%Y-%m-%d")
    except Exception:
        return None


def _load_emails_and_labels(account_email: str, label: str) ->
tuple[list, dict[str, set[str]]]:
    with get_connection() as conn:
        conn.row_factory = sqlite3.Row
        if label:
            email_rows = conn.execute(
                """SELECT e.uid, e.time FROM emails e
                   JOIN email_labels el ON el.uid = e.uid
                   WHERE e.account_email = ? AND el.label = ?""",
                (account_email, label),
            ).fetchall()
        else:
            email_rows = conn.execute(
                "SELECT uid, time FROM emails WHERE account_email = ?",
                (account_email,),
            ).fetchall()

        label_rows = conn.execute(
            """SELECT el.uid, el.label FROM email_labels el
               JOIN emails e ON el.uid = e.uid
               WHERE e.account_email = ?""",
            (account_email,),
        ).fetchall()

        uid_to_labels: dict[str, set[str]] = defaultdict(set)
        for row in label_rows:
            uid_to_labels[row["uid"]].add(row["label"])
        return email_rows, uid_to_labels
```

```
     def get_timeline(account_email: str, label: str = "") -> list[dict]:
         email_rows, uid_to_labels = _load_emails_and_labels(account_email,
label)
         doc_labels = set(DOCUMENT_LABELS)
         rule_labels_filter = doc_labels | {SUBJECT_REJECTED}

         buckets: dict[str, dict] = defaultdict(
             lambda: {"total": 0, "accepted": 0, "rejected": 0, "verified":
0, "unverified": 0}
         )

         for row in email_rows:
             date = _parse_date(row["time"])
             if not date:
                 continue

             labels = uid_to_labels.get(row["uid"], set())
             is_rejected = SUBJECT_REJECTED in labels
             is_accepted = bool(labels - rule_labels_filter)
             has_docs = bool(labels & doc_labels)

             bucket = buckets[date]
             bucket["total"] += 1
             bucket["rejected"] += int(is_rejected)
             bucket["accepted"] += int(is_accepted)
             bucket["verified"] += int(has_docs)
             bucket["unverified"] += int(is_accepted and not has_docs)

         return [{"date": d, **v} for d, v in sorted(buckets.items())]
```