Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалавр
(ступінь вищої освіти)

на тему СТВОРЕННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ДЛЯ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ СТРУКТУРОВАНИХ ДОКУМЕНТІВ ЗА ДОПОМОГОЮ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ
DESIGN OF AN INTELLIGENT SYSTEM FOR THE AUTOMATED GENERATION OF STRUCTURED DOCUMENTS USING LARGE LANGUAGE MODELS

Виконав(ла): студент(ка) 4 курсу, групи КНТ-222
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Комп'ютерні науки

РУБАН М.М.
(ПРІЗВИЩЕ та ініціали)
Керівник ТАБУНЩИК Г.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент ШИТІКОВА О.В.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____Сергій СУББОТІН
"____" _____2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

РУБАНА Матвія Михайловича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей. Design of an Intelligent System for the Automated Generation of Structured Documents Using Large Language Models

керівник проєкту (роботи) к.т.н., професор, ТАБУНЩИК Галина Володимирівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 07 " квітня 2026 року № 139
2. Строк подання студентом проєкту (роботи) 10 червня 2026 року
3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Огляд предметної галузі. 2. Огляд програмного забезпечення та проєктування системи. 3. Програмна реалізація інтелектуальної системи генерації структурованих документів. 4. Тестування, експлуатація та експериментальна перевірка інтелектуальної системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ТАБУНЩИК Г.В., професор		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2-3 тижні	Розділ 1
3	Вибір мовних моделей, середовища виконання та	3 тиждень	Розділ 2
	технологій розробки.		
4	Проєктування архітектури інтелектуальної системи.	4 тиждень	Розділ 2
5	Реалізація програмних компонентів системи.	5-6 тижні	Розділ 3
6	Тестування та експериментальне дослідження	7-8 тижні	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	9 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	10 тиждень	
9	Захист роботи.	10 тиждень	

Студент(ка) _____ Матвій РУБАН
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Галина ТАБУНЩИК
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 164 с., 7 табл., 2 рис., 2 дод., 57 джерел.

ВАЛІДАЦІЯ ДАНИХ, ВЕЛИКІ МОВНІ МОДЕЛІ, ДЖЕРЕЛЬНА ПРОСТЕЖУВАНІСТЬ, ІНТЕЛЕКТУАЛЬНА СИСТЕМА, ЛОКАЛЬНИЙ ПРОГРАМНИЙ КОНВЕЄР, ПОШУКОВО-ДОПОВНЕНА ГЕНЕРАЦІЯ, СТРУКТУРОВАНІ ДОКУМЕНТИ.

Об’єкт дослідження — методи та засоби опрацювання даних великими мовними моделями.

Предмет дослідження — створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей.

Мета роботи — покращення обробки табличних даних за рахунок використання великих мовних моделей.

Матеріали, методи та технічні засоби: аналіз цифрових і табличних документних джерел, рядкова лінеаризація таблиць, пошуково-доповнена генерація, формальна валідація структурованого результату, джерельна перевірка та експериментальне тестування; Python, Ollama, FAISS, JSON Schema і Pydantic.

Результати. Розроблено локальний програмний конвеєр для приймання цифрових і підготовлених джерел, вилучення текстових і табличних даних, лінеаризації таблиць, фрагментації з метаданими, добору джерельного контексту, формування запиту до локальної мовної моделі та перевірки кандидатного JSON-результату за формальними й джерельними критеріями. Реалізація формує валідований машинозчитуваний результат, журнали, метрики та звітні артефакти.

Висновки. Розроблена система демонструє контрольований підхід до перетворення цифрових документних і табличних джерел у структуроване машинозчитуване подання в межах локального прототипу. Мовна модель виконує

роль проміжного компонента, а прийнятність результату визначається зовнішнім програмним контуром перевірки.

Галузь використання – автоматизоване опрацювання технічних, звітних, табличних і слабкоформалізованих документів у навчальних, дослідних та прикладних інформаційних системах.

ABSTRACT

Explanatory note to the bachelor's qualification thesis: 164 pages, 7 tables, 2 figures, 2 appendices, 57 sources.

DATA VALIDATION, INTELLIGENT SYSTEM, LARGE LANGUAGE MODELS, LOCAL SOFTWARE PIPELINE, RETRIEVAL-AUGMENTED GENERATION, SOURCE TRACEABILITY, STRUCTURED DOCUMENTS.

The object of the study is methods and tools for processing data using large language models.

The subject of the study is the design of an intelligent system for the automated generation of structured documents using large language models.

The purpose of the work is to improve the processing of tabular data by using large language models.

Materials, methods, and technical tools: analysis of digital and tabular document sources, row-wise table linearization, retrieval-augmented generation, formal validation of structured results, source verification, and experimental testing; Python, Ollama, FAISS, JSON Schema, and Pydantic.

Results. A local software pipeline has been developed for ingesting digital and prepared sources, extracting textual and tabular data, linearizing tables, creating metadata-enriched chunks, selecting source context, constructing prompts for a local language model, and validating a candidate JSON result against formal and source-based criteria. The implementation produces a validated machine-readable result, logs, metrics, and reporting artifacts.

Conclusions. The developed system demonstrates a controlled approach to transforming digital document and tabular sources into a structured machine-readable representation within a local prototype. The language model acts as an intermediate component, while result acceptance is determined by an external software validation pipeline.

The field of use is automated processing of technical, reporting, tabular, and semi-structured documents in educational, research, and applied information systems.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API — програмний інтерфейс застосунку;

CSV — текстовий табличний формат із розділенням значень комами;

DOCX — формат текстових документів Microsoft Word;

FAISS — бібліотека для пошуку близьких векторних представлень;

HTTP — протокол передавання гіпертекстових даних;

JSON — машинозчитуваний формат подання структурованих даних;

JSONL — формат запису JSON-документів по одному об’єкту в рядку;

LLM — велика мовна модель;

MVP — мінімально життєздатний продукт;

OCR — оптичне розпізнавання символів;

PDF — формат електронного документа;

RAG — пошуково-доповнена генерація;

TXT — текстовий файловий формат;

XLSX — формат електронних таблиць Microsoft Excel;

YAML — формат конфігураційних файлів;

СКБД — система керування базами даних.

ВСТУП

Автоматизоване формування структурованих документів є актуальним завданням для систем, що працюють із технічними, табличними та слабкоформалізованими джерелами. У таких документах значення реквізиту часто залежить від локального контексту, назви таблиці, заголовка колонки, одиниці вимірювання, сторінки, рядка або іншої службової ознаки. Під час ручного опрацювання такі зв’язки зберігаються завдяки експертному розумінню документа, однак під час автоматизації вони легко втрачаються. Це призводить до помилок вилучення, неповних структурованих результатів, втрати джерельної прив’язки та складності подальшої перевірки.

Великі мовні моделі можуть формувати структуровані відповіді на основі природномовних інструкцій і підготовленого контексту. Водночас вони не є джерелом технічної істини й можуть повертати правдоподібні, але неперевірені або некоректно прив’язані до джерела значення. Тому для практичного використання великої мовної моделі у задачах формування структурованих документів потрібен не автономний виклик моделі, а контрольований програмний конвеєр. Такий конвеєр має приймати цифрові або підготовлені документні джерела, вилучати текстові й табличні дані, зберігати метадані, добирати релевантний джерельний контекст, обмежувати модель наданими фрагментами, формувати кандидатний JSON-об’єкт і перевіряти його за формальними та джерельними критеріями.

Актуальність роботи зумовлена потребою зменшення ручної трудомісткості під час опрацювання табличних і слабкоформалізованих документних даних та підвищення контрольованості результатів, сформованих за участю великих мовних моделей. Особливого значення набувають механізми пошуково-доповненої генерації, формальної валідації, типізованого контролю та джерельної простежуваності, оскільки саме вони дають змогу відокремити кандидатний результат моделі від прийнятого структурованого результату системи.

Об’єкт дослідження – методи та засоби опрацювання даних великими мовними моделями.

Предмет дослідження – створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей.

Мета роботи — покращення обробки табличних даних за рахунок використання великих мовних моделей.

Для досягнення поставленої мети у роботі вирішувалися такі завдання:

а) проаналізувати проблеми автоматизованого опрацювання структурованих, табличних і слабкоформалізованих документних джерел та визначити обмеження автономного використання великих мовних моделей;

б) обґрунтувати вибір локальних мовних моделей, середовища виконання і технологічного стеку для реалізації мінімально життєздатного продукту;

в) спроєктувати архітектуру програмного конвеєра для приймання документних джерел, вилучення текстових і табличних даних, рядкової лінеаризації таблиць, нормалізації проміжного подання, фрагментації з метаданими, добору джерельного контексту, формування запиту, локального інференсу, валідації, джерельної перевірки і звітності;

г) реалізувати програмні компоненти системи, включно з модулями вилучення даних, побудови фрагментів, векторного та резервного лексичного пошуку, формування запиту, взаємодії з локальною мовною моделлю, валідації й журналювання;

д) провести експериментальну перевірку реалізованого MVP на контрольних тестових сценаріях, оцінити проходження синтаксичного розбору JSON, валідації JSON Schema, Pydantic, перевірки джерельної відповідності поля source_excerpt, фіксації атрибута source_anchor, типових помилок та меж застосування системи.

Методи роботи охоплюють аналіз науково-технічних джерел, інженерне проєктування програмного забезпечення, модульну реалізацію локального Python-конвеєра, пошуково-доповнену генерацію, векторний і лексичний добір

контексту, формальну перевірку структурованого результату за JSON Schema, типізовану валідацію Pydantic, перевірку джерельного фрагмента та експериментальне тестування на підготовлених документних сценаріях.

Практичне значення роботи полягає у створенні локального прототипу системи, що реалізує контрольований цикл перетворення цифрових і підготовлених документних джерел у валідований машинозчитуваний структурований результат із джерельними прив’язками, службовими статусами, результатами перевірок, метриками та звітними артефактами. У межах роботи підтримуються джерела TXT, CSV, XLSX, DOCX і PDF з текстовим шаром; функції оптичного розпізнавання символів, обробка сканованих і рукописних джерел, повна реконструкція табличної та сторінкової геометрії, графічний інтерфейс користувача, база даних, багатокористувацький режим і промисловий DOCX/PDF-експорт розглядаються як напрями подальшого розвитку.

Структура роботи відображає послідовність розв’язання поставлених завдань. У першому розділі розглянуто предметну область, обмеження традиційної обробки документних даних і роль великих мовних моделей у формуванні структурованих результатів. У другому розділі обґрунтовано вибір моделей, середовища виконання, технологічного стеку та архітектури системи. У третьому розділі описано програмну реалізацію компонентів локального конвеєра. У четвертому розділі наведено методику й результати експериментальної перевірки, а також обмеження реалізованого MVP.

1 ОГЛЯД ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної області автоматичної генерації структурованих документів

Автоматична генерація структурованих документів у межах цієї роботи розглядається як задача контрольованого перетворення документної інформації зі збереженням структури, контексту, документної ролі та походження даних. Об’єктом аналізу є технічний документ, у якому зміст формується через взаємозв’язок реквізитів, полів, таблиць, числових параметрів, одиниць вимірювання, службових позначень, приміток і джерельного контексту. Такий документ не можна коректно обробляти як плоский текстовий масив, оскільки окреме значення набуває технічного змісту лише у зв’язку з параметром, умовами отримання, місцем у документі та документною роллю відповідного фрагмента. У межах інтелектуального аналізу документів зазначена проблема розглядається ширше за класичне опрацювання тексту, що зумовлено складною топологією джерел, які містять текстові, структурні, табличні та логічно-візуальні елементи [1].

1.1.1 Структуровані технічні документи як об’єкти автоматизованої обробки

Структурований технічний документ не є лінійним текстовим масивом, а становить систему взаємопов’язаних інформаційних блоків. Документи такого класу поєднують поля, реквізити, заголовки, логічні розділи, табличні блоки, числові параметри, одиниці вимірювання, примітки та службові позначення. Кожен елемент виконує визначену документну роль і набуває технічного змісту лише в комбінації з іншими елементами. Наприклад, ізольоване числове значення в таблиці без назви фізичної величини й одиниці вимірювання не може бути коректно інтерпретоване як технічний факт; дата без реквізитної прив’язки до номера документа не встановлює часові межі; висновок без посилання на

методику та результати унеможливлює верифікацію і таким чином втрачає доказову силу. Таким чином, структурованість технічного документа визначається не лише наявністю візуального шаблону, а й зв’язками між параметрами, їхнім розташуванням та правилами інтерпретації даних [2].

1.1.2 Табличні, реквізитні та текстові дані в технічних документах

Технічна документація зазвичай має змішану інформаційну природу. Таблиці концентрують числові параметри, результати вимірювань, допустимі межі, одиниці вимірювання та підсумкові значення. Реквізити формують ідентифікаційний контекст: номер документа, дату, назву об’єкта, виконавця, статус або версію редакції. Текстові фрагменти доповнюють ці дані умовами застосування, методичними обмеженнями, винятками та висновками. Специфікації обладнання, акти, накладні, протоколи вимірювань, реєстри та звіти випробувань можуть одночасно містити всі зазначені типи даних [1]. У напівструктурованих формах окрему складність створює залежність змісту від просторового розташування на сторінці, структури та ієрархії рядків і колонок [2], [3]. Просторова впорядкованість елементів робить ці зв’язки візуально зрозумілими для фахівця, однак під час автоматизованої обробки їх потрібно явно перетворювати на машинозчитувані структури.

1.1.3 Міграція технічних даних і ризик втрати контексту

У технічному документообігу дані часто переходять між різними формами джерел: від первинних журналів контролю до протоколів вимірювань, від технічних специфікацій до зведених звітів, від актів до реєстрів або підсумкових форм. Таке перенесення є не механічним копіюванням символів, а трансформацією документного подання. Розлогий опис може стискатися до окремого поля, рядок таблиці — перетворюватися на числовий індекс, масив вимірювань — узагальнюватися в текстовий блок. Основним ризиком такої

міграції є деградація або повна втрата контексту на ідентифікаційному, параметричному, структурному або джерельному рівні. Значення може втратити приналежність до документа, версії, параметра, одиниці вимірювання, рядка, стовпця або первинного фрагмента. У такому разі документ може залишатися зовні впорядкованим, але його перевірюваність знижується [4]. Типові приклади втрати контексту наведено в таблиці 1.1.

Таблиця 1.1 – Приклади втрати контексту під час перетворення документних даних

Тип джерела	Приклад даних	Що може втрачатися	Наслідок	Вимога до системи
Специфікація обладнання	Значення напруги “220 В”	Назва параметра, режим застосування, тип значення	Значення може бути віднесене до іншої технічної характеристики	Зберігати зв’язок значення з параметром, одиницею та розділом
Протокол вимірювань	Лінійний розмір дефекту “5,4 мм”	Умова контролю, допустима межа, дата вимірювання	Неможливо перевірити відповідність показника допустимій межі	Фіксувати умови отримання параметра та джерельний фрагмент
Таблиця параметрів	Числове значення в окремій комірці	Заголовок рядка, назва стовпця, примітка	Значення втрачає табличну прив’язку	Зберігати відомості про рядок, стовпець, заголовок і примітку
Акт або накладна	Номер, дата, виконавець	Реквізитна належність до конкретного документа	Дані не дають змоги однозначно ідентифікувати джерело	Зберігати реквізити та документну прив’язку
Звіт про випробування	Підсумковий висновок	Методика, обмеження, вихідні числові дані	Висновок має завершену форму, але не має перевірюваної підстави	Пов’язувати висновок із джерельними фрагментами

1.1.4 Слабкоформалізовані джерела та обмеження традиційної автоматизації

Складність автоматизованого аналізу технічної інформації підсилюється тим, що значна частина цифрових документів орієнтована на людське сприйняття,

а не на інтерпретацію програмними засобами. Візуально подібні елементи у файлі можуть бути представлені як структурована таблиця, набір абзаців із символами табуляції, вирівняні текстові рядки або мати змішане подання. Візуальна впорядкованість сторінки не гарантує явної логічної структури у внутрішньому поданні документа, тому під час лінійного зчитування тексту просторові й табличні зв’язки стають неявними або руйнуються [3]. Традиційні підходи на основі фіксованих правил, регулярних виразів або координатних шаблонів є придатними лише за стабільної структури джерела. Зміна назви поля, поява додаткового стовпця, перенесення примітки або інший порядок блоків підвищують ризик помилкового вилучення, пропуску значень або неповного зіставлення даних. Ручне перенесення залишається адаптивним, однак потребує значних часових витрат і підвищує ризик людських помилок під час роботи з великими масивами числових і реквізитних даних.

1.1.5 Потреба в інтелектуальній системі контрольованого формування структурованих документів

Аналіз предметної області демонструє потребу в інтелектуальній системі, орієнтованій на роботу з різноманітними цифровими й підготовленими джерелами, вилучення релевантних текстових, табличних, реквізитних та інших фрагментів, збереження документного контексту та формування результату в заданій структурі. Головною умовою є зміщення фокусу з довільного створення нового змісту на контрольоване перетворення наявної джерельної інформації. Сформоване поле вихідного об’єкта має зберігати зв’язок із фрагментом джерела, з якого воно походить. Це підвищує перевірюваність результату й дає змогу виявляти випадки, коли значення формально потрапило до структури, але втратило параметр, одиницю вимірювання, реквізит або контекст застосування. У цьому інженерному контексті велика мовна модель може розглядатися як керований компонент перетворення документного контексту у структуроване машинозчитуване подання, але не як автономне джерело технічної істини.

1.2 Архітектура та принципи роботи великих мовних моделей

Побудова системи автоматичної генерації структурованих документів потребує визначення того, як великі мовні моделі опрацьовують документний контекст і які межі випливають із їхньої архітектури. У межах цієї роботи LLM розглядається не як автономний автор технічного документа, а як керований структурувальний компонент, що працює з підготовленим контекстом і формує кандидатний результат для подальшого програмного контролю.

1.2.1 Місце великих мовних моделей у задачах документного аналізу

Великі мовні моделі працюють із текстовими послідовностями та сформульованими завданнями. У документному аналізі їхня роль пов’язана не з безпосереднім опрацюванням PDF, DOCX, сторінки або форми, а з інтерпретацією підготовленого текстового подання документа. Після такого подання модель може визначати змістові фрагменти, зіставляти близькі формулювання, узагальнювати описові частини, нормалізувати текст і формувати початкове структуроване подання інформації. Водночас технічний документ залишається системою реквізитів, параметрів, одиниць вимірювання, табличних зв’язків і джерельних фрагментів, тому мовна інтерпретація не замінює контроль документної структури та коректності даних.

1.2.2 Transformer, токенізація та внутрішнє подання тексту у великих мовних моделях

Архітектурною основою сучасних великих мовних моделей є Transformer, який забезпечує обробку послідовностей і формування контекстно залежних внутрішніх представлень [5]. Перед обробкою текст перетворюється на послідовність токенів. Токен не завжди дорівнює слову: це може бути слово, частина слова, окремий символ, окрема цифра, знак пунктуації або службовий

елемент, тому технічне позначення, дата, одиниця вимірювання або числове значення можуть бути подані кількома елементами. Модель працює не з документом у первинній сторінковій чи табличній формі, а з його токенізованим текстовим поданням, де один і той самий фрагмент може інтерпретуватися по-різному залежно від сусідніх фрагментів.

1.2.3 Механізм самоуваги та врахування контекстних зв'язків у текстовій послідовності

Базовий принцип функціонування математичної логіки моделей класу Transformer спирається на механізм самоуваги. Він реалізує встановлення асоціативних зв'язків між дискретними токенами та дає можливість інтерпретувати кожен елемент послідовності в межах доступного контексту [5].

Для технічних документів це важливо, оскільки значення може залежати від назви параметра, одиниці вимірювання, заголовка таблиці, примітки або попередніх висновків. Водночас самоувага працює з токенізованою послідовністю, а не з документом як явно заданою системою сторінок, рядків, колонок і реквізитів. Якщо такі зв'язки втрачено під час підготовки тексту, модель не гарантує їх правильного відновлення.

1.2.4 Лінеаризація документної структури та контекстне вікно

Оскільки LLM опрацьовує текстову послідовність, складна структура документа має бути лінеаризована. Таблиці, реквізитні блоки, примітки й вкладені фрагменти подаються як послідовність текстових елементів. Це робить документ придатним для подання в мовну модель, але може послаблювати зв'язки «рядок–стовпець», «параметр–значення», «одиниця–число» або «примітка–фрагмент». Формально значення може залишатися в тексті, але його документна роль уже не буде відповідати вихідній формі. Додатковою межею є контекстне вікно, яке обмежує обсяг токенів, доступних моделі в одному запиті. До цього

обсягу входить не тільки документний матеріал, а й інструкція, службові пояснення, приклади очікуваного формату, додаткові обмеження та сама відповідь моделі. Навіть передавання значного обсягу контексту не гарантує рівномірного використання всіх релевантних фрагментів.

1.2.5 Авторегресивна генерація та імовірнісна природа відповіді

Генеративна мовна модель формує відповідь авторегресивно: кожен наступний токен залежить від вхідного запиту, контексту та вже сформованої частини відповіді [5]. Модель обирає продовження на основі імовірнісного розподілу можливих токенів, але це не означає довільності відповіді, оскільки вибір обмежується контекстом, інструкцією, мовними закономірностями та параметрами декодування. Тому результат не є детермінованим документним об’єктом, створеним за формальним алгоритмом. Модель може сформувати граматично правильну й структурно впорядковану відповідь, але це не гарантує відповідності кожного поля вихідному документу, правильній одиниці вимірювання або джерельному фрагменту.

1.2.6 Інструкційне керування, промпт і структуроване виведення даних

Промпт задає завдання, джерельний контекст, очікуваний формат відповіді та обмеження на використання інформації. У задачах формування документів він може вимагати відповідь у вигляді переліку полів, ієрархічного об’єкта або кандидатного JSON-об’єкта. Структуроване виведення даних зменшує варіативність форми й полегшує подальшу перевірку, однак не доводить фактичну правильність значень. Інструкційне керування спрямовує модель, але не замінює формальну валідацію.

1.2.7 Користь великих мовних моделей і межа між мовною переконливістю та фактичною достовірністю

У задачах автоматичної генерації структурованих документів LLM доцільно використовувати не як автономний механізм встановлення технічних фактів, а як компонент роботи з текстовою формою підготовленого контексту: зіставлення фрагментів, нормалізації формулювань, узагальнення описових частин і формування кандидатного JSON-об’єкта. Її вихід може бути мовно переконливим, але це не дорівнює технічній достовірності. Значення, одиниці вимірювання, реквізити й висновки мають перевірятися позамодельними засобами. Великі мовні моделі є корисними для автоматичної генерації структурованих документів як керований структурувальний компонент, але розглянуті архітектурні й генеративні властивості LLM не дозволяють ототожнювати вихідні дані моделі з технічно достовірним документом через імовірнісний механізм перетворення контексту, який у технічному завданні потребує обмежень і зовнішнього контролю.

1.3 Обмеження застосування великих мовних моделей у задачах генерації технічних документів

Автономне використання LLM у задачах технічного документообігу створює ризики, що не обмежуються неточним формулюванням або стилістичною похибкою. Після підготовки документного контексту LLM може сформувати зв’язну відповідь і подати її у заданій структурі, однак такий результат не можна ототожнювати з перевіреним технічним документом. У межах цієї роботи вихід моделі має оцінюватися не за мовною завершеністю відповіді, а за відповідністю вихідним даним, збереженням структурної організації документа та коректністю технічної інтерпретації. Результат розглядається як кандидатний та потребує подальшого програмного контролю структури, типів, прикладних правил і джерельної відповідності.

1.3.1 Розбіжність між мовною зв’язністю та технічною коректністю документа

Мовна зв’язність відповіді не є доказом технічної коректності. Модель може сформувати граматично правильний, стилістично цілісний і формально впорядкований текст, який зовні відповідає очікуваному документному жанру. Проте технічний документ оцінюється не за мовною переконливістю, а за відповідністю значень джерелу, правильністю одиниць вимірювання, повнотою реквізитів, узгодженістю параметрів і можливістю перевірити походження даних. Тому мовна зв’язність, фактична достовірність і джерельна відповідність мають розглядатися як різні рівні якості результату [7].

1.3.2 Непідтверджені твердження, числові значення та ризик технічних галюцинацій

Одним із ключових ризиків є генерація непідтвердженого змісту. У технічній документації це може проявлятися як вигаданий параметр, неточна одиниця вимірювання, неправильно узагальнений висновок, доданий реквізит або числове значення, якого немає у джерелі. Такі помилки можуть зберігати правдоподібну форму й не виявлятися під час поверхневого читання. У дослідженнях генерації природної мови такі явища розглядаються як галюцинації, тобто відхилення сформованого змісту від джерела або фактичної основи [8]. Для технічного документа це означає потребу в політиці недостатності даних: за браку підтвердження модель не повинна домислювати відсутній зміст.

1.3.3 Обмеження контексту та відтворення табличних і логічних зв’язків

Навіть за наявності підготовленого контексту модель не гарантує повного й рівномірного використання всіх релевантних фрагментів. Контекстне вікно

обмежує обсяг доступної інформації, а довгі послідовності можуть ускладнювати використання фрагментів, розташованих у середині контексту [6]. Для технічних документів це створює ризик втрати зв’язку між числом і параметром, значенням і одиницею вимірювання, рядком таблиці й заголовком стовпця, реквізитом і відповідним фрагментом. У випадку табличних даних додатковою проблемою є те, що структуроване двовимірне подання після лінеаризації стає текстовою послідовністю, у якій частина логічних зв’язків може бути неявною [9]. В результаті інформаційний елемент може бути використаний без урахування семантичної та топологічної специфіки, від якої залежить коректність технічного висновку.

1.3.4 Нестабільність структурованого виведення даних та чутливість до промпта

Структуроване виведення даних зменшує варіативність форми відповіді, але не усуває імовірнісної природи моделі. Навіть якщо промпт задає очікувані поля, порядок елементів і формат кандидатного JSON-об’єкта, модель може пропустити обов’язкове поле, змінити назву атрибута, додати непередбачене значення або порушити вкладеність. Якість такого виведення залежить від формулювання промпта, порядку подання контексту, опису очікуваного формату та обмежень у завданні. Дослідження структурованого виведення даних показують, що сам формат відповіді потребує формальних обмежень і перевірки, а не лише текстової інструкції [10], [11]. Тому інструкційне керування не може замінювати схему даних, зовнішню перевірку й детермінований контроль сформованого виведення даних.

1.3.5 Відсутність вбудованої верифікації та джерельної простежуваності

LLM не має вбудованого механізму формальної перевірки кожного сформованого поля щодо джерела. Навіть наявність посилання, цитати або джерельного фрагмента не гарантує, що відповідь повністю узгоджена з цим фрагментом. Проблема джерельної атрибуції полягає в тому, що модель може надати правдоподібну прив'язку, але її коректність потребує окремої перевірки [4]. Для технічних документів це означає потребу в зовнішньому контролі: перевірці структури через схему даних, прикладній валідації правил, контролі типів і дослівному зіставленні значущих полів із джерельними фрагментами, зокрема через механізм контролю поля source_excerpt як джерельного фрагмента.

1.3.6 Конфіденційність і перехід до керованого конвеєрного підходу

Додатковим обмеженням для технічної документації є потенційна чутливість документних джерел. Виробничі звіти, специфікації, протоколи вимірювань, акти або реєстри можуть містити службові, технічні чи організаційно значущі дані. Передавання таких матеріалів до хмарних сервісів може створювати ризики для конфіденційності, контролю доступу та дотримання внутрішніх політик організації [12]. Конфіденційність у цьому контексті пов'язана не лише з місцем запуску моделі, а й із загальною керованістю процесу: доступом до документів, збереженням проміжних даних, контролем експорту та можливістю перевірити шлях проходження інформації через систему.

Застосування LLM має враховувати не лише якість відповіді, а й контроль середовища обробки, обсяг переданого контексту та політику роботи з недостатніми даними. Якщо джерело не містить потрібного значення або зв'язок між фрагментами є неоднозначним, модель не повинна домислювати зміст; результат має фіксувати недостатність даних. Відповідно, обмеження LLM не заперечують її використання, але вимагають керованого конвеєрного підходу, у

якому добір релевантного контексту, структуроване виведення даних, перевірка JSON-схеми, прикладна валідація, перевірка джерельної простежуваності і політика недостатності даних виконують роль зовнішніх механізмів контролю кандидатного результату.

1.4 Аналіз технології пошуково-доповненої генерації у задачах обробки документів

Архітектурні обмеження великих мовних моделей зумовлюють потребу в механізмах, які пов’язують формування відповіді з фрагментами джерельних документів. Пошуково-доповнена генерація є підходом, за якого модель формує відповідь не лише на основі внутрішніх параметричних знань, а з урахуванням попередньо відібраного документного контексту. Це зменшує ризик генерації без джерельної опори, проте не усуває потребу в подальшій перевірці сформованого кандидатного результату [13], [14].

1.4.1 Концептуальна сутність пошуково-доповненої генерації та потреба в джерельному контексті

За стандартного функціонування LLM відповідь формується на основі інформації, закодованої у параметрах моделі, та тексту поточного запиту. Для технічної документації цього недостатньо, оскільки значення, реквізити, одиниці вимірювання й висновки мають бути пов’язані з конкретним документом. Пошуково-доповнена генерація (Retrieval-Augmented Generation, RAG) використовується не для розширення параметричних знань великої мовної моделі, а для обмеження генерації конкретним документним контекстом. Перед генерацією система відбирає частини документа, пов’язані із запитом або цільовим елементом майбутнього результату, після чого LLM формує відповідь на основі цього обмеженого контексту [13]. У такій схемі модель працює не лише

з узагальненим знанням про предметну область, а з фрагментами, які можуть бути перевірені щодо джерела.

1.4.2 Базова логіка конвеєра пошуково-доповненої генерації та семантичний пошук

Конвеєр пошуково-доповненої генерації починається з підготовки джерел, що охоплює отримання документного джерела, вилучення тексту, табличних даних та інших структурних елементів змісту. Після цього виконуються попередня обробка, очищення, формування метаданих, сегментація та індексація фрагментів, яка включає побудову векторних подань. Подальший пошук працює не з документом як неподільним файлом, а з детермінованим набором проіндексованих сегментів, релевантність яких оцінюється під час формування контексту для передавання у велику мовну модель.

Векторні подання необхідні для подальшого семантичного пошуку, який дозволяє знаходити фрагменти не лише за буквальним збігом слів, а й за змістовою близькістю між запитом і документним фрагментом [14]. Для технічної документації такий підхід є важливим, коли потрібна інформація

сформульована в документі інакше, ніж у запиті. Наприклад, однакові параметри можуть описуватися через різні формулювання, скорочення або описи.

1.4.3 Вплив сегментації та метаданих на збереження документного контексту

Якість пошуково-доповненої генерації залежить від того, як документ поділено на фрагменти. Текстовий сегмент надмірного обсягу може містити інформаційний шум, кілька різних параметрів або нерелевантні службові блоки. Сегмент недостатнього обсягу може розірвати зв’язок між параметром, значенням, одиницею вимірювання, заголовком таблиці або пояснювальною приміткою. Тому сегментація має зберігати контекст, достатній для інтерпретації

даних. Метадані, зокрема відомості про джерело, сторінку, розділ, тип фрагмента або табличне походження, підвищують контрольованість подальшого добору й створюють основу для джерельної простежуваності [15].

1.4.4 Формування джерельного контексту для великої мовної моделі

Після пошуку релевантних фрагментів формується контекст, придатний для передавання в LLM. До нього входять не всі знайдені дані, а відібрана добірка, що відповідає завданню, обмеженню контекстного вікна й очікуваній структурі відповіді. На цьому етапі важливими є усунення дублювань, відсікання другорядних блоків, збереження джерельних прив’язок і контроль обсягу контексту. Якщо контекст містить надмірний шум, суперечливі фрагменти або не містить потрібного доказового фрагмента, модель може сформувати зв’язну, але неперевірювану відповідь [15].

1.4.5 Обмеження базової пошуково-доповненої генерації та недостатність лише векторної близькості

Пошуково-доповнена генерація не гарантує автоматичної коректності контексту та відповіді. Серед типових точок відмови виділяють неправильний добір фрагментів, неповний або надлишковий контекст, пропущені докази та залежність від якості запиту [15]. Опора лише на векторну близькість не завжди означає документну релевантність: фрагмент може бути семантично схожим, але належати до іншого розділу, іншого параметра або іншої версії документа. Окремим чинником є нерівномірне використання фрагментів у довгому контексті, зокрема коли релевантна інформація розташована в його середині [6]. Тому результат пошуку має розглядатися як кандидатний контекст, а не як гарантовано правильне джерельне підґрунтя.

1.4.6 Розширені підходи до добору контексту: гібридний пошук, повторне ранжування та фільтрація за метаданими

Для технічної документації базовий векторний пошук доцільно доповнювати механізмами керованого добору контексту. До них належать гібридний пошук, повторне ранжування знайдених фрагментів та фільтрація за метаданими [14].

Гібридний пошук складається з семантичного й лексичного компонентів. Семантичний пошук забезпечує вилучення змістовно близьких інформаційних фрагментів. Натомість лексичний компонент спрямований на точну ідентифікацію формальних елементів джерела: кодів, назв параметрів, одиниць вимірювання, номерів пунктів і реквізитів. Повторне ранжування (переранжування) використовується для вторинної оцінки кандидатних фрагментів і уточнення їхнього порядку відповідно до запиту. Фільтрація за метаданими обмежує простір пошуку певним документом, розділом, таблицею, типом блока або іншою структурною ознакою.

Такі механізми не замінюють валідацію, але зменшують ризик передавання до мовної моделі нерелевантного, неповного або надлишкового контексту. У задачах документної генерації контроль цього етапу є важливим, оскільки будь-яка помилка на етапі добору джерела може трансформуватися у викривлення підсумкового структурованого результату.

1.4.7 Пошуково-доповнена генерація як засіб підвищення перевірюваності та її залежність від попередньої підготовки документів

Пошуково-доповнена генерація підвищує перевірюваність результату, оскільки відповідь може бути пов’язана з конкретними фрагментами джерела, а не лише з внутрішніми знаннями моделі. Проте така перевага залежить від якості вилучення тексту, лінеаризації таблиць, сегментації, метаданих, добору контексту й подальшої перевірки. Наявність джерельного фрагмента або посилання не

гарантує, що сформоване поле правильно підтверджене цим фрагментом [4], [16]. Пошуково-доповнена генерація є необхідним архітектурним шаром для роботи з документними джерелами, проте вона залишається недостатньою без валідації, журналювання та контролю джерельної відповідності.

1.5 Особливості обробки табличних і слабкоформалізованих джерел

Пошуково-доповнена генерація залежить від якості фрагментів, які передаються моделі як джерельний контекст. Для суцільного тексту сегментація переважно пов’язана з межами абзаців і змістових блоків. Для таблиць, форм і слабкоформалізованих документів опрацювання ускладнюється, оскільки технічне значення формується не лише послідовністю слів, а й просторовими, табличними, реквізитними та службовими зв’язками. Тому такі джерела потребують окремого аналізу перед визначенням вимог до системи.

1.5.1 Таблиця як логічна структура технічних даних

Таблиця в технічному документі є двовимірною матрицею відношень між даними, а не простим набором рядків тексту. Значення в комірці набуває змісту через зв’язок із рядком, стовпцем, заголовком, групою параметрів, одиницею вимірювання та приміткою. Ізольоване числове значення не забезпечує повноти інформації, якщо невідомо, до якого параметра воно належить і в яких умовах отримане. Тому задачі опрацювання табличної інформації в документах пов’язані не лише з вилученням текстового шару, а й із локалізацією, структурою та контекстом елементів [2], [17].

1.5.2 Просторові, реквізитні та одиничні залежності технічного значення

Зміст певного значення часто залежить від просторового розташування та реквізитного оточення. Один і той самий числовий показник може мати різний зміст залежно від заголовка таблиці, номера документа, версії, дати, підпису, одиниці вимірювання або пояснювальної примітки. У формах і напівструктурованих документах ці зв’язки часто задаються макетом: положенням поля, близькістю до підпису, групуванням блоків або візуальною ієрархією. Під час перетворення такого документа на послідовний текст частина цих залежностей може втрачатися або ставати неявною [3].

1.5.3 Обмеження лінійного вилучення тексту та ускладнення слабкоформалізованих джерел

Лінійне вилучення тексту не гарантує збереження документної структури. Слабкоформалізовані документи посилюють цю проблему через неоднорідність подання. Документ може містити абзаци, таблиці, реквізити, службові позначення, примітки й вкладені блоки, які візуально зрозумілі для людини, але після текстового вилучення перетворюються на послідовність фрагментів без явних структурних зв’язків. У слабкоформалізованих джерелах однакові за змістом поля можуть мати різні назви, різне розташування або частково змінений порядок. Це ускладнює формування якісних фрагментів для індексації та подальшого добору контексту [1], [2].

1.5.4 Розмежування вилучення тексту та машинозчитуваного розуміння документа

Вилучення текстового шару та машинозчитуване розуміння документа належать до різних рівнів обробки. Текст може бути доступним для опрацювання,

проте його наявність не гарантує коректного встановлення зв’язків між значеннями, параметрами, одиницями вимірювання й фрагментами джерела. Для великих мовних моделей і пошуково-доповненої генерації це означає, що якість відповіді залежить не лише від наявності тексту, а й від того, чи збережено зв’язки, потрібні для інтерпретації фрагмента. Саме цей рівень підготовки створює основу для логічної сегментації, добору релевантного контексту, простежуваності джерел і подальшої перевірки сформованого результату. Дослідження роботи моделей із табличними даними показують, що формат подання таблиці та порядок елементів впливають на стабільність її обробки як структурованого джерела [9].

1.5.5 Попередня обробка з урахуванням макета й табличної структури перед індексацією

Перед індексацією табличні й слабкоформалізовані джерела потребують попередньої обробки з урахуванням їхньої структури. Така обробка має зберігати не лише текст, а й мінімально необхідні відомості про походження фрагмента, його належність до таблиці, рядка, стовпця, розділу або реквізитного блоку. Без цього знайдений фрагмент може бути семантично близьким, але втратити документну прив’язку, потрібну для правильного заповнення структурованого результату. На цьому рівні доцільним є не повне відновлення сторінкової геометрії, а контрольоване збереження зв’язків, достатніх для подальшої генерації й перевірки [3]. Реалізація цього принципу покладається на поєднання процесів попередньої обробки з урахуванням макета сторінки та внутрішньої логіки таблиць. Перший підхід забезпечує фіксацію просторової організації й неявних залежностей слабкоформалізованих блоків, тоді як другий — структурує топологію матриць даних та ієрархію їхніх заголовків. У результаті формується проміжне машинозчитуване подання документа, що зменшує сегментацію інформації та запобігає втраті критично важливого технічного контексту.

1.5.6 Логічна сегментація, таблично-орієнтоване фрагментування і проміжне машинозчитуване подання

У межах MVP доцільним компромісом є контрольована рядкова лінеаризація табличних даних. Такий підхід перетворює рядок таблиці на текстовий фрагмент із частковим збереженням зв’язків між значеннями, заголовками й одиницями вимірювання. Він полегшує сегментацію, індексацію та подальший добір контексту, але не дорівнює повній реконструкції таблиці: не відновлює повну сітку, об’єднані комірки, топологію сторінки або розташування елементів. Табличні й слабкоформалізовані джерела потребують обробки з урахуванням структури або контрольованої лінеаризації перед пошуком контексту й генерацією.

1.6 Вимоги до інтелектуальної системи генерації структурованих документів

1.6.1 Перехід від аналізу обмежень до системних вимог

Вимоги до інтелектуальної системи генерації структурованих документів випливають з аналізу предметної області, обмежень великих мовних моделей, пошуково-доповненої генерації та табличних джерел. Така система має бути орієнтована не на автономне створення технічного змісту, а на контрольоване перетворення цифрових і підготовлених документних джерел у структурований результат. Основними вимогами є збереження документного контексту, добір релевантних фрагментів, формування кандидатного JSON-об’єкта, програмна перевірка структури та контроль джерельної відповідності [4], [15].

1.6.2 Функціональні вимоги до конвеєра обробки документів

Функціонально система має підтримувати послідовний конвеєр обробки документних даних: надходження цифрового або підготовленого джерела,

вилучення текстових і табличних фрагментів, контрольовану лінеаризацію таблиць, нормалізацію, сегментацію, формування метаданих, добір і підготовку джерельного контексту, передавання контексту до мовної моделі, формування кандидатного структурованого результату, його перевірку та створення звітних артефактів. Така послідовність потрібна для того, щоб модель оперувала не довільним текстом, а підготовленим і обмеженим джерельним контекстом. Окремий інструмент вилучення, пошуку або генерації не закриває всю задачу, оскільки документний результат потребує поєднання вилучення, добору контексту, генерації, валідації та джерельної простежуваності.

1.6.3 Вимоги до простежуваності та метаданих

Система має зберігати джерельну простежуваність кожного значущого поля, реквізиту, числового значення або сформованого висновку. Для цього фрагменти джерела мають супроводжуватися метаданими про документ, сторінку, розділ, таблицю, рядок, стовпець, реквізитний блок або інший доступний контекст походження. Вихідний результат має містити джерельні прив’язки, зокрема атрибут source_anchor як джерельний якір та поле source_excerpt як дослівне джерельне обґрунтування, достатні для ручної або автоматизованої перевірки. Наявність посилання або цитати не є доказом фактичної правильності, тому джерельна прив’язка має перевірятися окремо [4].

1.6.4 Вимоги до структурованого виходу, валідації та контролю недостатності даних

Вихід системи має бути структурованим і машинозчитуваним. Кандидатний JSON-об’єкт має відповідати наперед визначеній схемі, містити обов’язкові поля, очікувані типи даних, статуси обробки та ознаку недостатності даних. Перевірка JSON-схеми забезпечує контроль структури, типів і допустимих полів, а прикладна валідація має уточнювати правила предметної області [18],

[19]. При цьому формальна валідність не дорівнює фактичній істинності. Якщо джерело не містить потрібного значення або зв’язок між фрагментами неоднозначний, система має фіксувати статус insufficient_data, а не передавати до результату модельне припущення [10].

1.6.5 Нефункціональні вимоги та межі застосування системи

Нефункціональні вимоги мають охоплювати локальність, відтворюваність, журналювання, ресурсну обмеженість і контроль меж MVP. Система має бути придатною для роботи в локальному середовищі MacBook Air M1 з 8 ГБ уніфікованої пам’яті, що обмежує розмір локальних моделей, обсяг контексту, паралельність ресурсомістких етапів і доцільність складних компонентів промислового рівня. У межах MVP підтримуються цифрові або підготовлені джерела без оптичного розпізнавання символів: TXT, CSV, XLSX, DOCX і PDF з текстовим шаром. Повна реконструкція сітки таблиці, об’єднаних комірок, геометрії сторінки, скановані PDF, інтерфейс користувача, база даних та експорт для промислового використання не входять до меж базової системи. Для відтворюваності мають зберігатися конфігурації, журнали запусків, результати перевірок, метрики та звітні файли.

Для узагальнення сформованих функціональних, валідаційних, джерельних і нефункціональних вимог у таблиці 1.2 наведено порівняння основних класів підходів до автоматизованої обробки структурованих і частково структурованих документів. Таблиця показує, що окремі інструменти закривають лише частину завдання, тоді як для локального контрольованого рішення потрібне поєднання добору контексту, структурованого виведення даних, формальної валідації, джерельної простежуваності та фіксації меж застосування.

Таблиця 1.2 – Порівняння підходів до автоматизованої обробки структурованих і частково структурованих документів

Підхід / шар обробки	Що забезпечує	Основне обмеження	Висновок
Вилучення за правилами та шаблонами	Заповнення полів за наперед заданими правилами	Чутливість до зміни макета, назв полів і структури джерела	Недостатньо для слабкоформалізованих документів
Інструменти вилучення вмісту з файлів	Визначення типу файла, вилучення тексту та базових метаданих	Не зберігають повні логічні зв'язки таблиць і реквізитів	Придатні як початковий шар вилучення, але не як повна система
Хмарні системи інтелектуальної обробки документів	Оптичне розпізнавання, вилучення полів, таблиць і структури макета	Залежність від хмарного сервісу, зовнішнього API та політик обробки даних	Корисний клас рішень, але не повністю відповідає локальному контрольованому рішенню
Фреймворки пошуково-доповненої генерації	Організація завантаження джерел, фрагментації, добору контексту й генерації	Не гарантують правильність добору контексту, фактичну коректність і валідацію результату	Корисні як інфраструктурний шар, але потребують контрольованої конфігурації
Локальне середовище виконання великих мовних моделей	Локальний запуск моделі та формування структурованого результату	Не виконує вилучення документів, валідацію й джерельну перевірку самостійно	Потрібне як модельний шар, але не як автономна система
Шар валідації	Перевірка JSON, типів, обов'язкових полів і прикладних правил	Не доводить фактичну істинність значень і не відновлює структуру джерела	Необхідний після генерації, але має працювати разом із джерельним контекстом
Спеціалізований локальний конвеєр із пошуково-доповненою генерацією	Поєднання вилучення, лінеаризації, добору контексту, кандидатного JSON, валідації та звітності	Межі MVP: відсутність оптичного розпізнавання символів, повної реконструкції таблиць і геометрії сторінки	Узгоджується із завданням контрольованого локального рішення

Примітка. Таблицю складено на основі джерел [4], [15], [18]-[27]. Розглянуті інструменти подано як приклади класів рішень, а не як компоненти розроблюваної системи.

Порівняння показує, що жоден окремий клас інструментів не забезпечує одночасного вилучення, контекстного добору, структурованого виведення даних, формальної валідації та джерельної простежуваності в межах локального контрольованого MVP. Це обґрунтовує потребу в програмному конвеєрі, де

формальна перевірка структури й типів доповнюється зовнішнім контролем джерельного походження кожного значущого елемента.

1.7 Постановка завдання розробки програмного забезпечення

1.7.1 Загальна постановка завдання розробки

На основі аналізу предметної області, обмежень великих мовних моделей, пошуково-доповненої генерації, табличних джерел та системних вимог завдання роботи полягає у створенні інтелектуальної системи у вигляді контрольованого програмного конвеєра для автоматизованого формування структурованого машинозчитуваного результату з цифрових і підготовлених документних джерел. У межах MVP таким технічним результатом є валідований JSON-об’єкт із джерельними прив’язками, службовими статусами та артефактами перевірки.

Конвеєр має забезпечувати перетворення підготовленого джерельного контексту в кандидатні машинозчитувані дані з подальшою формальною та джерельною перевіркою, виключаючи неконтрольоване породження технічного тексту мовною моделлю. Отриманий результат не має замінювати експертну оцінку технічної достовірності інформації, а виступає засобом підвищення контрольованості та автоматизації попереднього формування структурованих даних.

1.7.2 Об’єкт, предмет і мета роботи

Об’єктом дослідження є методи та засоби опрацювання даних великими мовними моделями.

Предметом роботи є створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей.

Метою роботи є покращення обробки табличних даних за рахунок використання великих мовних моделей.

1.7.3 Основні завдання розробки програмного забезпечення

Для досягнення поставленої мети необхідно виконати такі завдання:

— проаналізувати предметну область автоматичної генерації структурованих документів;

— визначити обмеження великих мовних моделей у задачах технічної документації;

— розглянути принципи пошуково-доповненої генерації, структурованого виведення даних і джерельної простежуваності;

— сформувати функціональні та нефункціональні вимоги до контрольованої системи;

— обґрунтувати технологічний стек і архітектуру програмного забезпечення;

— спроєктувати конвеєр обробки документних даних;

— реалізувати базовий MVP-конвеєр обробки документних даних;

— провести тестування коректності структурованого результату, валідації та джерельної перевірюваності;

— оцінити результати перевірки й визначити межі застосування розробленого рішення.

1.7.4 Вхідні, проміжні та вихідні дані системи

Вхідними даними системи є цифрові або підготовлені документні джерела без оптичного розпізнавання символів: TXT, CSV, XLSX, DOCX і PDF з текстовим шаром. До проміжних даних належать нормалізований текст, лінеаризовані табличні рядки, фрагменти документа, метадані, векторні подання фрагментів, знайдений джерельний контекст і кандидатний JSON-об’єкт. Наявність проміжних представлень дає змогу не передавати документ до мовної моделі як нерозділений текстовий масив, а працювати з підготовленими фрагментами зі збереженими контекстними та джерельними ознаками. Вихідними

даними є валідований JSON-результат, службові статуси обробки, ознаки недостатності даних, результати перевірок, метрики та звітні артефакти, придатні для подальшого аналізу й відтворення запусків. Окремим допустимим результатом обробки є фіксація недостатності або неоднозначності даних без домислювання відсутніх значень.

1.7.5 Обмеження, критерії успішності та зв’язок із подальшим проєктуванням

Межі MVP передбачають роботу лише з цифровими або попередньо підготовленими джерелами. Скановані PDF, оптичне розпізнавання символів, рукописні документи, повна реконструкція сітки таблиць, об’єднані комірки, геометрія сторінки, графічний інтерфейс користувача, база даних і експорт для промислового використання не входять до базового обсягу. Критеріями успішності є оброблення визначених типів джерел, формування кандидатного JSON-об’єкта, проходження синтаксичного розбору, перевірки JSON-схеми та прикладної валідації, виконання перевірки джерельного фрагмента у полі source_excerpt, коректна фіксація недостатності даних у неповних або неоднозначних випадках, фіксація результатів тестів за статусами проходження/непроходження і наявність артефактів відтворюваності.

1.8 Висновки за розділом 1

У першому розділі обґрунтовано, що автоматична генерація структурованих документів є комплексним завданням контрольованого перетворення цифрових і підготовлених документних джерел у структуроване машинозчитуване подання, а не процесом неконтрольованого модельного породження технічного тексту. Показано, що технічний зміст документа залежить від збереження зв’язків між реквізитами, таблицями, числовими значеннями, одиницями вимірювання, метаданими та джерельними фрагментами.

Визначено, що велика мовна модель може використовуватися як керований компонент перетворення підготовленого контексту, тоді як автономно сформований нею результат не є прийнятним технічним рішенням без зовнішньої формальної та джерельної перевірки.

Розглянуто роль пошуково-доповненої генерації як механізму добору контексту, що зменшує ризик формування тексту без опори на джерело, проте автоматично не гарантує коректності вихідних даних. Встановлено, що табличні та слабкоформалізовані джерела потребують попередньої структуризації або контрольованої лінеаризації перед етапами добору контексту та формування результату мовною моделлю. На основі проведеного аналізу визначено вимоги до системи, сформовано постановку завдання розробки контрольованого MVP-конвеєра та обґрунтовано перехід до огляду програмного забезпечення, вибору технологічного стеку й проєктування архітектури системи у наступному розділі.

2 ОГЛЯД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ПРОЄКТУВАННЯ СИСТЕМИ

Вибір мовної моделі для системи автоматичного формування структурованих документів є багатокритеріальним процесом, який має враховувати її параметри, середовище інференсу, цільовий формат результату, апаратні обмеження та зовнішній контур післягенераційної перевірки. У розробленому конвеєрі велика мовна модель формує кандидатний JSON-об’єкт на основі підготовленого джерельного контексту. Придатність такого результату визначається не моделлю, а подальшою автоматизованою перевіркою синтаксису, відповідності структурі, сумісності типів даних, виконання політики недостатності даних та джерельної простежуваності. Аналітичний огляд у цьому підрозділі обмежено класом компактних, локально доступних моделей із відкритими вагами та платформами розгортання, придатними для функціонування відтворюваного MVP у заданому обчислювальному середовищі [10], [28].

2.1 Огляд моделей із відкритим вихідним кодом та платформ локального розгортання

Добір мовної моделі для системи автоматичного формування структурованих документів виконується не як пошук найпотужнішої LLM загального призначення, а як інженерне узгодження моделі, середовища виконання, формату виведення даних, ресурсних обмежень і контуру післягенераційної перевірки. У базовому комплексі програмних засобів мовна модель є керованим компонентом, який формує кандидатний JSON-об’єкт на основі підготовленого джерельного контексту. Придатність результату визначається не фактом генерації, а подальшим програмним контролем структури, типів, політики недостатності даних і джерельної прив’язки. Тому огляд у цьому підрозділі обмежується компактними локально доступними моделями та платформами запуску, придатними для відтворюваного MVP у заданому апаратному середовищі.

2.1.1 Критерії оцінювання великих мовних моделей для локального формування структурованих документів

Для локального формування структурованих документів мовна модель оцінюється за функціональною придатністю до роботи в автоматизованому програмному конвеєрі, а не за загальними діалоговими чи стилістичними властивостями. Основними інженерними критеріями є локальна доступність ваг моделі, стабільне завершення процесу інференсу, повернення одного синтаксично коректного JSON-об’єкта, відповідність цільовій структурі JSON Schema та проходження типізованої перевірки Pydantic [10], [18], [19]. Програмний контур не має приймати сторонні або невалідовані поля у структурі результату. Окремим критерієм є здатність моделі коректно заповнювати обов’язкові атрибути джерельної простежуваності: джерельний якір source_anchor та дослівний фрагмент оригінального документа source_excerpt. За фіксованого значення temperature=0.0 модель повинна демонструвати стабільну поведінку, а в разі відсутності, розмитості або суперечливості даних у наданому контексті — відображати цей стан через службові маркери null, insufficient_data, ambiguous і validation_notes, уникаючи генерації припущень. Для технічної документації також важлива стабільна робота з кирилицею, числовими значеннями, одиницями вимірювання, реквізитами та лінеаризованими рядками таблиць, що відокремлює формальну придатність структурованого виведення від загальної правдоподібності тексту [7], [8].

2.1.2 Ресурсні обмеження локального інференсу та вплив цільового апаратного середовища на вибір моделей

Цільовим експлуатаційним середовищем для розгортання MVP визначено MacBook Air M1 з 8 ГБ уніфікованої пам’яті під керуванням операційної системи macOS. Така конфігурація зменшує залежність архітектури від зовнішніх API, але обмежує граничний розмір моделей, довжину контекстного вікна, паралельність

виконання процесів і складність компонентів середовища виконання [12], [28]. Загальний обсяг уніфікованої пам’яті одночасно розподіляється між операційною системою, активними Python-процесами оркестрації, локальним середовищем інференсу, моделлю формування векторних представлень, векторним індексом FAISS та проміжними структурами даних [29]. Через це вибір моделі не може спиратися лише на її потенційну якість у тестах, відірваних від цільового обладнання. Пріоритет надається компактним локальним моделям, контрольованому розміру контекстного вікна з базовим обмеженням num_ctx=4096, послідовному виконанню ресурсоємних етапів обробки та відмові від надлишкових серверних компонентів. Ресурсна придатність моделі визначається її здатністю стабільно повертати структуровану відповідь, придатну до машинної перевірки в повному програмному конвеєрі.

2.1.3 Цільовий клас моделей для базового комплексу програмних засобів

Цільовим класом для розроблюваного комплексу програмних засобів визначено компактні локальні мовні моделі інструктивного типу, зокрема моделі інструктивного й діалогового типів, здатні виконувати чіткі інструкції та підтримувати структуроване виведення даних. У межах MVP обґрунтованим є застосування моделей у діапазоні від 3 до 4 мільярдів параметрів або близьких до них за рівнем використання ресурсів локальних конфігурацій. Зазначені компоненти не розглядаються як альтернатива великим хмарним системам, проте їх застосування достатнє для перевірки інженерної гіпотези роботи: обмежена за розміром локальна модель може виконувати функцію генератора кандидатного результату, якщо її інтегровано в програмний контур із зовнішніми механізмами добору контексту за допомогою пошуково-доповненої генерації, керованим формуванням запиту та багаторівневою післягенераційною валідацією [13], [14], [18], [19]. Таким чином, головним критерієм вибору є не максимальна кількість

параметрів, а відтворювана керованість, ресурсна прийнятність і низький ризик збоїв процесу інференсу моделей у межах локального MVP.

2.1.4 Формування експериментального набору локальних тегів моделей

Для локального оцінювання системи визначено експериментальний набір тегів моделей, зафіксований у конфігураційному файлі configs/models.yaml як фактичне підтвердження програмної бази репозиторію. До складу набору включено чотири конфігурації, розгорнуті в локальному середовищі інференсу: llama3.2:3b з програмним ключем llama_3b, gemma4:e2b з програмним ключем gemma_e2b, qwen3.5:4b з програмним ключем qwen_4b та gemma4:e4b з програмним ключем gemma_e4b. Для забезпечення порівнюваності для всіх тегів встановлено однакові базові параметри генерації: temperature=0.0 та num_ctx=4096. У межах архітектури системи локальний тег llama3.2:3b використовується як основна модель для подальшої перевірки, тоді як gemma4:e2b визначено як comparison_model. Інші конфігурації залишено як додаткові. Зазначені ідентифікатори використовуються як локальні теги середовища виконання і не претендують на роль глобальних оціночних маркерів upstream-моделей. Такий набір є контрольованою вибіркою для перевірки здатності кількох локальних моделей формувати кандидатний JSON у межах одного стеку, одного апаратного середовища та однієї політики валідації.

2.1.5 Обґрунтування виключення ресурсоємних, міркувальних і дублювальних моделей з активного набору

Обмеження активного набору моделей зумовлене потребою зменшити ризики апаратних збоїв та специфікою інженерної задачі структурованого вилучення. Ресурсоємні моделі великого розміру, архітектури з розгорнутим внутрішнім міркуванням та дублювальні варіанти однієї функціональної ролі не

включаються до активного набору, якщо вони збільшують час інференсу, навантаження на пам’ять, ризик помилок локального API або ймовірність відповіді, непридатної для синтаксичного розбору JSON. Процес покрокового міркування може генерувати значну кількість службових токенів, що підвищує навантаження на контекстний бюджет і ускладнює дотримання вимоги JSON-only відповіді [10], [11]. Оскільки цільовим результатом роботи моделі є формування кандидатного структурованого JSON-об’єкта, надмірна генеративна свобода може призводити до появи супровідного природномовного тексту або порушення структури відповіді. Виключення таких архітектур не означає їх загальної непридатності. Воно лише фіксує межу проєктування поточного MVP, де пріоритет надається відтворюваності, послідовному виконанню етапів конвеєра та стабільності роботи на цільовому обладнанні.

2.1.6 Огляд платформ локального розгортання та обґрунтування вибору Ollama як середовища виконання

Програмні засоби для локального розгортання великих мовних моделей відрізняються за архітектурною роллю: одні виконують функції низькорівневого середовища інференсу, інші надають графічну оболонку, локальний сервер або API для програмної інтеграції. Для розроблюваного MVP критично важливими є наявність локального API-сервера, керування параметрами генерації, можливість фіксації модельних тегів і інтеграція з Python-рівнем через HTTP-запити. На основі цих вимог як базове середовище інференсу обрано Ollama. У структурі системи Ollama виконує строго обмежені функції локального середовища виконання та API-сервера, а не роль засобу вилучення даних, пошуково-доповненої генерації, валідації або звітності [27], [29], [30]. Порівняльну характеристику платформ наведено в таблиці 2.1.

Таблиця 2.1 – Порівняльна характеристика локальних середовищ виконання LLM

Платформа	Основна роль	Придатність і обмеження	Висновок
Ollama	Локальне середовище виконання та API-сервер для запуску локальних тегів моделей через HTTP-запити з Python-рівня.	Придатна для MVP завдяки локальному запуску, фіксації модельних тегів і керуванню параметрами генерації. Не виконує вилучення даних, пошуково-доповнену генерацію, валідацію, перевірку джерел або звітність.	Обрано як базовий вузол локальної генерації кандидатного JSON-об’єкта.
llama.cpp	Низькорівневе локальне середовище інференсу для запуску моделей у відповідних форматах.	Придатне як технічна альтернатива для локального запуску, зокрема в ресурсно обмежених середовищах. Потребує додаткового налаштування середовища виконання та не формує готовий контур валідації.	Залишено як порівняльне середовище виконання без інтеграції в ядро MVP.
LM Studio	Настільне середовище та локальний API-сервер для запуску локальних моделей.	Придатне як порівняльний контекст для локального запуску моделей і програмної інтеграції через локальний сервер, REST API та сумісні API-інтерфейси. Не використовується у фактичному Python-конвеєрі MVP.	Не обрано як базовий компонент системи, але враховано як локальну альтернативу.

Примітка. Таблицю складено на основі джерел [27], [29]-[32]. Ollama подано як компонент розроблюваної системи, а llama.cpp і LM Studio — як порівняльні приклади локальних платформ запуску LLM.

2.1.7 Методика локального оцінювання якості, стабільності структурованого виведення даних та ресурсної придатності моделей

Методика локального оцінювання орієнтована на визначення інженерної придатності моделей до інтеграції у сформований конвеєр обробки даних, а не на аналіз їх загальної лінгвістичної переваги. Оцінювання здійснюється шляхом подання зіставних за структурою документів на вхід єдиного шаблону інструкцій, що обмежує відповідь вимогами заданої JSON-схеми. Протокол оцінювання фіксує технічні показники: факт успішного завершення операції інференсу на рівні API, наявність кандидатного об’єкта у відповіді, успішність синтаксичного розбору JSON, проходження валідації за JSON Schema та Pydantic, наявність атрибута source_anchor_present, а також результат точного збігу поля

source_excerpt [18], [19], [33]. Крім того, фіксуються параметри elapsed time, tokens/sec та поля помилок error fields як елементи майбутнього експериментального аналізу без наведення числових результатів у цьому розділі. Методика розділяє синтаксичну некоректність відповіді, формальну невідповідність структурі схеми та невиконання критерію джерельної простежуваності, що дозволяє не змішувати структурну валідність із фактичною істинністю даних [7], [8].

2.2 Огляд засобів пошуково-доповненої генерації, індексації та формування структурованого виведення даних

Обмежена роль мовної моделі як локального генератора кандидатного JSON-об’єкта зумовлює потребу в зовнішньому програмному контурі для підготовки даних, добору контексту та післягенераційної перевірки. У межах обраної архітектури локальна модель не виконує самостійно вилучення вмісту, збереження табличних зв’язків, семантичний пошук, валідацію або контроль джерельної простежуваності. Ці операції делегуються компонентам конвеєра: зчитуванню вхідних форматів, нормалізації, рядковій лінеаризації таблиць, фрагментації, векторному індексуванню, добору контексту, валідації, журналюванню та звітності.

2.2.1 Пошуково-доповнена генерація як підхід до обмеження мовної моделі джерельним контекстом

Пошуково-доповнена генерація забезпечує інженерне розділення функцій добору ймовірно релевантного контексту та формування відповіді моделі [13], [14]. Модель отримує не повний документний масив, а фрагменти технічного тексту, інтегровані в запит. Це зменшує залежність результату від її параметричних знань і підвищує перевірюваність походження значень. Водночас пошуково-доповнена генерація не гарантує автоматичної фактичної коректності

результату й не усуває імовірнісну природу LLM [15]. У розроблюваному конвеєрі вона виконує функцію контекстного обмеження перед побудовою кандидатного JSON-об’єкта.

2.2.2 Інструменти вилучення текстових і табличних даних із цифрових документних джерел

Якість структурованого результату залежить від коректності первинного вилучення текстових і табличних даних із джерел. Межі проєктування MVP обмежуються цифровими або підготовленими джерелами без застосування оптичного розпізнавання символів: файлами TXT, CSV, XLSX, DOCX та PDF із текстовим шаром [1]. Для їх опрацювання релевантними є спеціалізовані бібліотеки документного синтаксичного розбору: PyMuPDF для PDF-документів, python-docx для DOCX, pandas та openpyxl для CSV і XLSX [34], [35]-[37]. Виключення сканованих матеріалів і зображень фіксує межі MVP і дозволяє зосередити аналіз на контрольованому перетворенні цифрових джерел.

2.2.3 Збереження структурного контексту таблиць і лінеаризація табличних даних

Обробка табличних даних у слабкоформалізованій документації ускладнюється втратою змісту комірок поза зв’язком із заголовками стовпців, рядків або документом [3], [9]. Для збереження цього контексту в MVP використовується рядкова лінеаризація: кожен рядок таблиці трансформується в окремий інформаційний фрагмент table_chunk як послідовність пар «назва стовпця: значення комірки». Для забезпечення простежуваності фрагмент супроводжується метаданими про джерело, таблицю, рядок, стовпці та тип вмісту. Підхід є компромісом MVP: він не відновлює повну геометрію сторінки чи сітку таблиці [17], але зберігає реквізитні зв’язки, необхідні для семантичного пошуку та включення табличних даних до запиту.

2.2.4 Засоби векторного представлення, семантичного пошуку та локальної індексації фрагментів

Добір контексту вимагає перетворення фрагментів на щільні векторні представлення, придатні для оцінювання семантичної близькості [38], [39]. У межах MVP визначено використання моделі sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2, нормалізованих векторів і локального кешування. Локальне індексування здійснюється через FAISS з індексом IndexFlatIP в оперативній пам’яті та не потребує серверної векторної бази даних [40]. Семантичний пошук доповнюється резервним лексичним каналом для точної ідентифікації числових маркерів, ідентифікаторів, одиниць вимірювання та реквізитів [41], [42]. Подібність векторних представлень відображає семантичну близькість фрагментів, але не є перевіркою фактичної істинності змісту.

2.2.5 Підходи до оркестрації конвеєра: спеціалізовані фреймворки та власна Python-реалізація

Для побудови систем пошуково-доповненої генерації існують спеціалізовані фреймворки оркестрації, які надають абстракції для індексації, керування запитами та агентних сценаріїв [25]-[26]. Ці інструменти можуть бути доцільними для масштабних систем, однак для локального MVP частина абстракцій є надлишковою. У цій роботі обґрунтовано власну Python-реалізацію оркестраційного шару з явним розмежуванням етапів зчитування, лінеаризації, фрагментації, добору контексту, побудови запиту, валідації, журналювання та звітності. Такий підхід забезпечує прозорість обчислень, спрощує трасування помилок і не переносить контроль логіки на зовнішній фреймворк.

2.2.6 Засоби формальної валідації структурованого виведення даних та контролю типів

Кандидатний JSON-об’єкт моделі розглядається як проміжний артефакт і підлягає обов’язковій перевірці програмним контуром [33]. Валідаційний каскад складається з чотирьох етапів: синтаксичного розбору JSON, перевірки відповідності структурі за допомогою JSON Schema, типізованого контролю через Pydantic та перевірки дослівного збігу поля source_excerpt [18], [19]. JSON Schema задає обов’язкові поля, типи, вкладені структури, статуси, source_anchor і обмеження щодо сторонніх полів. Pydantic забезпечує перевірку типів і прикладних правил. Невдале проходження дослівного збігу трактується як невиконання критерію джерельної відповідності, а не як автоматичне доведення хибності факту [7], [43], [44]. Формальна валідність структури не є тотожною фактичній істинності значень.

2.2.7 Формати проміжного подання, журналювання, метрик і експериментальної звітності

Контрольований конвеєр вимагає уніфікації форматів проміжних даних, журналювання станів і формування звітності. Для збереження параметрів конфігурацій використовується YAML [45]. Записи запусків, трасування помилок валідації та фіксація використаних фрагментів здійснюються у форматах JSON або JSONL [46]. Для подання часових і обчислювальних метрик використовується формат CSV, а підсумкові технічні звіти формуються у вигляді Markdown-документів [47], [48]. Таке поєднання форматів підвищує відтворюваність запусків, підтримує аналіз помилок і створює доказову основу для подальшого експериментального аналізу системи.

2.3 Обґрунтування вибору технологічного стеку для базового комплексу програмних засобів

На основі аналізу локальних моделей, середовищ інференсу та засобів пошуково-доповненої генерації визначено технологічний стек MVP. Його призначення полягає у створенні локального програмного конвеєра без надлишкової інфраструктури, у якому велика мовна модель функціонує як керований компонент формування кандидатного JSON-об’єкта, а придатність результату встановлюється зовнішнім програмним контуром. Обрані компоненти враховують апаратні межі MacBook Air M1 з 8 ГБ уніфікованої пам’яті, підтримують роботу з цифровими документними джерелами та не переносять етапи контролю на мовну модель.

2.3.1 Принцип технологічного мінімалізму та критерії відбору компонентів

Вибір стеку підпорядковано принципу технологічного мінімалізму: до комплексу включено лише компоненти, необхідні для зчитування документів, нормалізації контексту, локального інференсу, формування структурованого результату, валідації та звітності. Критеріями відбору є локальна придатність, прозорість проміжних обчислень, контроль контекстного бюджету, помірне споживання пам’яті та сумісність із послідовним виконанням ресурсоємних операцій. Надлишкові клієнт-серверні сервіси, важкі фреймворки оркестрації та закриті хмарні API не входять до активного набору засобів.

2.3.2 Вибір Python і локального ізольованого середовища виконання

Python обрано як основний інтеграційний, координаційний і контрольний рівень системи [49]. На цьому рівні реалізовано оркестрацію конвеєра: зчитування вхідних даних, нормалізацію, рядкову лінеаризацію таблиць, фрагментацію, добір

контексту, побудову запиту, HTTP-взаємодію з локальним API-сервером через requests, каскад післягенераційної валідації та формування звітів [50]. Локальне середовище venv ізолює залежності проєкту від системних компонентів операційної системи [51], а файл requirements.txt фіксує перелік програмних пакетів [52].

2.3.3 Вибір Ollama та активного набору моделей для локального інференсу

Ollama обрано як локальне середовище виконання великих мовних моделей та API-сервер для програмної взаємодії з оркестраційним контуром системи [27], [29], [30]. У конфігураційному файлі configs/models.yaml зафіксовано локальні теги моделей: llama3.2:3b як основний кандидат, gemma4:e2b як модель порівняння (comparison_model), а також qwen3.5:4b і gemma4:e4b як некореневі варіанти. Базові параметри інференсу обмежено значеннями temperature=0.0 та num_ctx=4096. Роль моделей звужено до генерації кандидатного JSON-об’єкта, тоді як журналювання та контроль залишаються на Python-рівні.

2.3.4 Вибір інструментів вилучення, нормалізації та лінеаризації документних даних

Програмне опрацювання вхідних файлів орієнтоване на цифрові джерела без застосування оптичного розпізнавання символів [1]. Вилучення текстового шару з PDF реалізовано через PyMuPDF [34], з документів DOCX — через python-docx [35], а опрацювання табличних масивів у форматах CSV та XLSX покладено на pandas [36] та openpyxl [37]. Для збереження реквізитних зв’язків структурованих даних використовується рядкова лінеаризація за принципом «рядок у текст». Обмеження такого способу табличного подання деталізовано під час опису реалізації вхідного рівня та меж функціональності системи.

2.3.5 Вибір засобів векторного представлення, індексації та добору релевантного контексту

Семантичний канал добору контексту побудовано на основі моделі формування векторних представлень sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 [38], [39] та локального індексу FAISS з конфігурацією IndexFlatIP [40]. Для зменшення залежності від мережевих ресурсів передбачено роботу з локальним офлайн-кешем векторних ваг. Семантичний пошук доповнено резервним лексичним каналом пошуку для точнішої ідентифікації числових маркерів, ідентифікаторів, кодів та реквізитів [41], [42]. Обчислення виконуються в оперативній пам’яті без розгортання серверних систем керування базами даних (СКБД).

2.3.6 JSON, JSON Schema і Pydantic як основа контрольованого формування структурованого результату

JSON (RFC 8259) прийнято як проміжну форму машинозчитуваного подання й обміну даними в конвеєрі [33]. Первинний синтаксичний розбір контролює коректність структури об’єкта, після чого jsonschema перевіряє відповідність результату контракту JSON Schema [18], [53]. Подальшу прикладну валідацію правил і маркування недостатності даних покладено на типізовані моделі Pydantic [19]. Критерієм суворої відповідності джерелу є алгоритм дослівного порівняння значень поля source_excerpt. Цей каскад контролює структуру та типи даних, але не доводить фактичної істинності значень.

2.3.7 Розмежування реалізованого стеку та перспективних інфраструктурних розширень

Зафіксований стек MVP доповнюється легковаговими текстовими форматами конфігурації, трасування та звітності. YAML використовується для

збереження параметрів конфігурацій [45]. Трасування помилок валідації та пооб'єктне журналювання станів запусків реалізується у форматах JSON або JSONL [46]. Збір часових та обчислювальних метрик здійснюється у форматі CSV [47], а підсумкові звіти формуються як Markdown-документи [48]. Автоматизована перевірка працездатності модулів підтримується pytest [54]. Поточний стек MVP свідомо обмежено локальним послідовним конвеєром із файловими артефактами; промислові інфраструктурні розширення винесено за межі базового комплексу. Це задає рамку проєктування архітектури конвеєра за шаблоном «конвеєри та фільтри» [55]. Зведену характеристику технологічного стеку MVP наведено в таблиці 2.2.

Таблиця 2.2 – Зведена характеристика технологічного стеку MVP

Рівень стеку	Обрані засоби	Інженерна роль у конвеєрі	Межі використання
Локальне виконання моделі	Ollama; локальні теги моделей	Запуск LLM через локальний API-сервер і формування кандидатного JSON-об'єкта	Не виконує добір контексту, валідацію, журналювання або звітність
Оркестрація та середовище	Python, requests, venv, requirements.txt	Координація етапів конвеєра, HTTP-взаємодія з Ollama, ізоляція залежностей	Орієнтовано на локальний послідовний режим, без промислової інфраструктури
Вилучення й підготовка документів	PyMuPDF, python-docx, pandas, openpyxl, рядкова лінеаризація	Зчитування цифрових PDF, DOCX, CSV, XLSX та перетворення таблиць у текстові фрагменти	Без оптичного розпізнавання символів, сканованих PDF, рукописних матеріалів і повної реконструкції таблиць
Добір джерельного контексту	sentence-transformers, FAISS, резервний лексичний пошук	Формування векторних представлень, локальний семантичний пошук і добір точних технічних маркерів	Подібність фрагментів не доводить фактичну істинність знайденого змісту
Валідація, журналювання і звітність	JSON, JSON Schema, jsonschema, Pydantic, верифікація поля source_excerpt, YAML, JSONL, CSV, Markdown, pytest	Перевірка структури й типів кандидатного результату, контроль джерельного фрагмента, фіксація параметрів, метрик і звітів	Формальна валідність не замінює експертну оцінку; звітність не є промисловим моніторингом

Примітка. Таблицю складено на основі джерел [18], [19], [27], [33], [38],[40]-[42], [46]-[48] та доказів репозиторію. Компоненти подано як фактичний стек MVP або як службові засоби його конфігурації, перевірки й звітності.

2.4 Проєктування архітектури системи генерації структурованих документів

Проєктування архітектури системи автоматизованої генерації структурованих документів базується на конвеєрному опрацюванні даних за шаблоном «Конвеєри та фільтри» (Pipes and Filters) [55]. Такий підхід дає змогу розділити процес перетворення слабкоформалізованих джерел на послідовність функціональних компонентів, взаємодія між якими координується програмним рівнем на базі Python. Основна архітектурна позиція полягає в обмеженні ролі локальної великої мовної моделі. Модель не розглядається як автономний автор технічного тексту або джерело істини, а інтегрується в конвеєр як керований компонент мовно-структурного перетворення підготовленого контексту в проміжну кандидатну форму. Усі суміжні етапи — від приймання файлів до багаторівневої перевірки й верифікації походження даних — виконуються зовнішнім програмним контуром. Загальну схему конвеєра генерації структурованого документного результату представлено на рисунку 2.1.

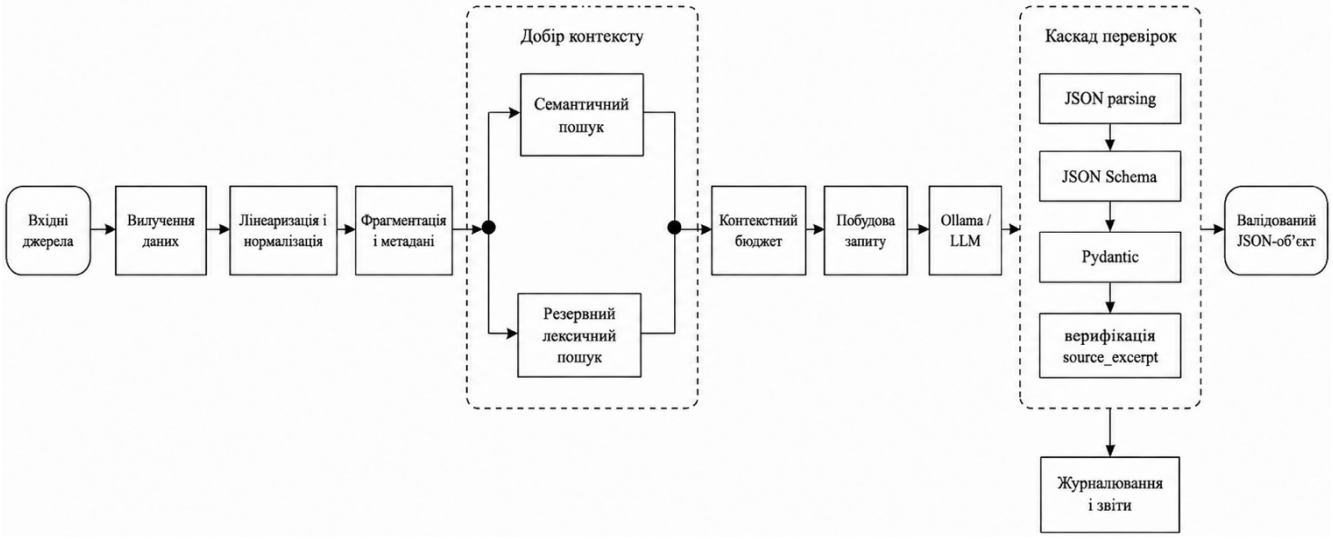


Рисунок 2.1 – Структурно-функціональна схема конвеєра генерації структурованого документного результату

2.4.1 Концептуальна структурно-функціональна схема конвеєра перетворення документних даних

Концептуальна схема системи проєктується як орієнтований інформаційний потік станів даних від вхідного файла до валідованого машинозчитуваного результату [13], [14]. Програмний контур послідовно об’єднує рівні приймання та нормалізації, індексування й двоканального добору контексту, локального інференсу мовної моделі, каскадної валідації та звітності. Така декомпозиція забезпечує трасованість обчислень і дозволяє послідовно виконувати ресурсоємні операції без надмірного навантаження на уніфіковану пам’ять цільового середовища [15]. Загальну структурно-функціональну схему

2.4.2 Рівень приймання, ідентифікації та попередньої обробки вхідних файлів

Рівень приймання виконує роль вхідного шлюзу системи, забезпечує первинну ідентифікацію типів джерел, перевірку розширень файлів та фільтрацію вхідного потоку відповідно до меж MVP. Система спроєктована для роботи з цифровими або підготовленими текстовими й табличними документами, що мають текстовий шар [1]. На цьому етапі архітектурно обмежується передавання сканованих зображень, фотографій документів та рукописних матеріалів, які потребують залучення систем оптичного розпізнавання символів. Мета фільтрації полягає в тому, щоб не переносити похибки графічного розпізнавання в контур перевірки текстових і табличних даних. На основі розширення файла .txt, .csv, .xlsx, .docx або .pdf система активує відповідний спосіб зчитування даних із збереженням базового кодування символів UTF-8.

2.4.3 Формування проміжного машинозчитуваного представлення документа

Після вилучення вмісту першоджерела за допомогою профільних інструментів PyMuPDF, python-docx, pandas та openpyxl дані трансформуються у проміжне машинозчитуване представлення, спільне для підтримуваних форматів [34]-[37]. Для неструктурованих текстових джерел представлення формується у вигляді лінійного нормалізованого тексту, у якому зберігаються числові значення, коди, реквізити та одиниці вимірювання. Для табличних масивів застосовується рядкова лінеаризація [3], [9]. Кожен рядок таблиці перетворюється на текстову послідовність пар «назва стовпця: значення комірки». Це уніфікує табличний вміст із лінійним текстом для фрагментації та пошуку без повної реконструкції сітки таблиці, геометрії сторінки чи об’єднаних комірок, що виходить за межі MVP [17].

2.4.4 Проєктування системи джерельних прив’язок і метаданих для забезпечення простежуваності

Джерельна простежуваність значень результату є необхідною умовою контрольованої автоматизованої обробки документів [43]. Для цього на рівні проміжного представлення проєктується система метаданих походження інформації. Кожен інформаційний фрагмент на етапі нормалізації супроводжується службовими даними: ідентифікатором документа, типом файла, індексом сторінки або назвою табличного аркуша, номером рядка та типом вмісту. У цільовій вихідній структурі для забезпечення простежуваності передбачено два зв’язані атрибути: джерельний якір source_anchor і дослівний джерельний фрагмент source_excerpt [44]. Атрибут source_anchor виконує роль логічного маркера походження, а поле source_excerpt містить дослівний фрагмент із вхідного контексту, який надалі використовується як об’єкт програмної перевірки.

2.4.5 Логічна сегментація документа та формування фрагментів для пошуку

Логічна сегментація (chunking) призначена для поділу проміжного представлення документа на фрагменти, придатні для векторного кодування, лексичного пошуку та подальшого включення до контексту запиту. Згідно з конфігураційним файлом configs/retrieval.yaml, для лінійного текстового шару сегментація реалізується за алгоритмом ковзного вікна з фіксованими параметрами: chunk_size_chars=600 символів і chunk_overlap_chars=80 символів. Перекриття знижує ризик руйнування реквізитних і змістових зв’язків на межах фрагментів. Для табличних масивів базовою одиницею сегментації є один лінеаризований рядок із відповідними метаданими. На виході цього рівня формується масив фрагментів для індексації та добору контексту.

2.4.6 Двоканальний механізм добору контексту: семантичний пошук і резервний лексичний пошук

Добір релевантного контексту спроєктовано як двоканальний механізм, що поєднує семантичний та лексичний канали пошуку. Семантичний канал кодує фрагменти у щільні вектори за допомогою моделі sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 [38], [39] та виконує пошук у локальному індексі FAISS з конфігурацією IndexFlatIP [40]. Межа семантичного добору визначена параметром semantic_top_k=4 фрагменти. Резервний лексичний канал виконує точний рядковий пошук за числовими кодами, номерами та технічними абревіатурами, добираючи до keyword_max_chunks=2 фрагментів [41]. Результати обох каналів об’єднуються, проходять дедуплікацію та обмежуються лімітами max_context_chunks=6 фрагментів і max_context_chars=6000 символів відповідно. Такий бар’єр зменшує ризик перевищення вікна інференсу, але не гарантує повної релевантності кожного фрагмента [15], [42].

2.4.7 Рівень локального інференсу та формування кандидатного структурованого результату

Рівень локального інференсу забезпечує HTTP-взаємодію оркестратора з локальним сервером Ollama [30]. На основі відібраного масиву фрагментів модуль конструктора запитів формує запит із системними інструкціями, джерельним контекстом, описом цільової схеми виведення даних та прикладом коректної структури JSON. Функція мовної моделі обмежена: вона виконує мовно-структурне перетворення наданого контексту у фіксовані машинозчитувані поля [27]. Отриманий від Ollama текстовий об’єкт розглядається як кандидатний JSON-об’єкт, тобто проміжний технічний артефакт без статусу перевіреного документа. Його подальше приймання можливе лише після проходження каскаду перевірок на Python-рівні.

2.4.8 Валідація, журналювання, метрики та контур фінального контролю результатів

Контур фінального контролю приймає або відхиляє кандидатний JSON-об’єкт після генерації. Валідація реалізується як послідовний чотирирівневий каскад. Перший рівень здійснює синтаксичний розбір об’єкта, блокуючи відповіді з порушеною JSON-структурою [33]. Другий рівень перевіряє артефакт за допомогою JSON Schema та jsonschema, контролюючи наявність обов’язкових ключів і типи полів [18], [53]. Третій рівень виконує валідацію за допомогою моделей Pydantic, перевіряючи прикладні правила та статуси недостатності даних [19]. Четвертий рівень реалізує перевірку source_excerpt — дослівне порівняння фрагмента source_excerpt із вхідним контекстом документа. Розбіжність трактується як невиконання критерію джерельного походження, а не як автоматичне доведення хибності факту [7]. Рівні та межі контролю деталізовано у таблиці 2.3.

Таблиця 2.3 – Рівні перевірки кандидатного JSON-результату

Шар перевірки	Що перевіряє	Чого не перевіряє	Роль у системі
Синтаксичний розбір JSON	Коректність JSON-структури та придатність відповіді до машинного опрацювання	Відповідність схемі, типам і фактичній істинності значень	Відсікає синтаксично непридатну відповідь
JSON Schema / jsonschema	Наявність обов'язкових полів, допустимі типи, вкладені структури та обмеження схеми	Семантичну правильність і фактичну істинність значень	Перевіряє відповідність результату структурному контракту
Pydantic	Типізовану модель даних і прикладні правила перевірки	Джерельну наявність фрагмента та експертну правильність змісту	Виконує прикладну валідацію кандидатного об'єкта
source_excerpt	Дослівну наявність зазначеного джерельного фрагмента у вхідному контексті	Повну семантичну правильність або достатність інтерпретації джерела	Забезпечує суворий контроль джерельної простежуваності
Журналювання і звітність	Статуси перевірок, типи помилок, параметри запуску та звітні артефакти	Відсутність дефектів і промислову готовність системи	Фіксує слід виконання для аудиту та подальшого тестування
Зовнішня експертна оцінка	Технічну правильність, нормативну відповідність та істинність документа	Не є автоматизованою програмною перевіркою в межах MVP	Залишається зовнішньою межею прийняття документного результату

2.5 Стратегії формування запитів, добору джерельного контексту та керування інструкціями

Стратегія формування запитів у спроєктованій системі виконує роль керувального інтерфейсу між підсистемою добору джерельного контексту, локальною мовною моделлю та каскадом програмної валідації. Вона визначає, які фрагменти передаються до моделі, як задаються інструкції та як відповідь має узгоджуватися з цільовою JSON-структурою. Запит не є самостійним механізмом гарантування фактичної або синтаксичної коректності результату: він лише створює умови для формування кандидатного JSON-об'єкта, який надалі перевіряється на формальному, типізованому та джерельному рівнях [10], [11].

2.5.1 Промпт як керувальний інтерфейс між пошуком, мовною моделлю та валідацією

Промпт розташовується між етапом добору контексту та локальним інференсом, тому його функція полягає в обмеженні простору відповіді моделі. Він поєднує відібрані фрагменти, вимоги до структури результату, правило використання лише наданих джерел і вимогу повернення одного кандидатного JSON-об’єкта. Такий підхід підвищує керованість генерації, але не усуває потреби в післягенераційній перевірці [6], [14].

2.5.2 Класифікація запитів за складністю та типом документної операції

Запити доцільно класифікувати за типом документної операції. Базовий рівень охоплює вилучення окремого реквізиту або числового значення. Складнішими є запити на групу пов’язаних полів, обробку лінеаризованого table chunk, агрегацію кількох фрагментів і побудову вкладеної JSON-структури. Такий розподіл потрібний для узгодження обсягу контексту, очікуваного формату відповіді та подальшої валідації [9].

2.5.3 Контекстне обмеження моделі джерельними фрагментами та правило використання лише наданих даних

Ключовим обмеженням інструкцій є використання лише наданого джерельного контексту. Модель не повинна доповнювати відсутні значення параметричними знаннями, екстраполяцією або правдоподібними припущеннями. Якщо потрібне значення відсутнє у відібраних фрагментах, результат має штатно відображати недостатність даних. Це зменшує ризик генерації без опори на джерело, але не замінює каскад зовнішньої програмної перевірки [7], [8].

2.5.4 Інструкції для обробки лінеаризованих таблиць, технічних маркерів та одиниць вимірювання

Опрацювання слабкоформалізованих джерел потребує окремих інструкцій для лінеаризованих таблиць, технічних маркерів, чисел та одиниць вимірювання. Лінеаризований рядок має сприйматися як послідовність пар «назва стовпця: значення комірки», а не як довільний текст. Числові значення, коди, одиниці вимірювання й ідентифікатори мають зберігати вихідний формат запису, оскільки їхнє переформулювання ускладнює джерельну простежуваність [9], [41], [42].

2.5.5 Обробка інформаційної недостатності, неоднозначності та конфлікту джерельних даних

Стратегія формування запитів має передбачати штатну поведінку за умов неповноти, неоднозначності або конфлікту джерельних даних. Якщо шуканий реквізит не знайдено, модель має позначити відповідне поле як null або insufficient_data, а не генерувати припущення. Якщо різні фрагменти містять суперечливі значення, результат має зберігати ознаку ambiguous або пояснення у validation_notes для подальшої програмної та експертної перевірки.

2.5.6 Узгодження інструкцій, цільової JSON-структури та програмної перевірки результату

Інструкції запиту, приклад очікуваного JSON, JSON Schema, Pydantic-модель і верифікація поля source_excerpt мають бути узгоджені між собою. Запит орієнтує модель на поля, типи й службові статуси, які надалі перевіряються програмним контуром. JSON Schema задає структурний контракт, Pydantic забезпечує контроль типів і прикладних правил, а значення поля source_excerpt перевіряється на дослівний збіг джерельного фрагмента з вхідним контекстом

[18], [19], [33], [42], [43]. У межах MVP це реалізовано у файлах шаблонів запитів та підтверджується схемою результату.

2.5.7 Межі інженерії промптів і потреба в детермінованому контролі після генерації

Інженерія запитів підвищує керованість локального інференсу, проте не забезпечує повного контролю над відповіддю мовної моделі. Навіть за чітких інструкцій модель може повернути синтаксично непридатний JSON, пропустити обов'язкове поле, змінити джерельний фрагмент або некоректно інтерпретувати контекст. Тому остаточне прийняття згенерованого артефакту виконується зовнішнім програмним контуром: синтаксичний розбір JSON, валідація JSON Schema і Pydantic і верифікація поля source_excerpt [7], [10], [11].

2.6 Вимоги до апаратного та програмного забезпечення

Вимоги до апаратного та програмного забезпечення визначають експлуатаційний контур, у якому можливий відтворюваний запуск системи як локального MVP. Вони випливають із обраного технологічного стеку, архітектури конвеєра та стратегії керування LLM через джерельний контекст і структуроване виведення даних. Підрозділ фіксує умови запуску, локального кешування, конфігурацій, журналювання й файлової звітності без переходу до результатів експериментальної перевірки.

2.6.1 Апаратні обмеження цільового локального середовища

Цільовим апаратним середовищем MVP є MacBook Air M1 з 8 ГБ уніфікованої пам'яті під керуванням macOS. Така конфігурація визначає ресурсні межі системи, оскільки спільна пам'ять одночасно використовується операційною системою, Python-конвеєром, моделлю векторних представлень, локальним

індексом, середовищем виконання Ollama і локальною LLM. Тому архітектура орієнтується на компактні моделі, локальну індексацію в оперативній пам’яті та послідовне виконання ресурсоємних етапів.

2.6.2 Програмна екосистема для локального запуску компонентів системи

Програмна екосистема базується на macOS, Python, локальному середовищі venv, файлі requirements.txt та Ollama як локальному середовищі виконання/API-сервері [29], [49], [51], [52]. Python координує етапи конвеєра, HTTP-взаємодію з Ollama через requests, валідацію, збір метрик і формування звітів [50]. Допоміжні бібліотеки забезпечують роботу з цифровими документами, векторними представленнями, JSON Schema, Pydantic, YAML-конфігураціями та тестовими перевірками [45], [54].

2.6.3 Вимоги до локального кешування моделей, конфігурацій і тестових артефактів

Відтворюваний запуск потребує локальної доступності модельних тегів, кешу векторних представлень, конфігурацій, схем і шаблонів запитів. Локальні теги Ollama мають бути підготовлені до інференсу, а модель векторних представлень — доступна з локального кешу, що зменшує залежність від повторного мережевого завантаження. Конфігурації моделей, зчитування й добору контексту зберігаються у YAML-форматі, а схеми, шаблони запитів, тестові вхідні дані та еталонні відповіді логічно відокремлюються від приватних сирих документів.

2.6.4 Організація файлової структури, журналів, метрик і результатів експериментів

Файлова організація MVP розмежовує конфігурації, схеми, запити, вхідні дані, еталонні відповіді, журнали запусків, метрики та звітні артефакти. Для конфігурацій використовується YAML, для записів запусків і проміжних артефактів — JSONL, для компактних метрик — CSV, для пояснювальних звітів — Markdown [45]-[48]. У журналах доцільно фіксувати модель, конфігурацію, ідентифікатор джерела, статус запуску, зареєстровані помилки, відібрані фрагменти та статус джерельного фрагмента без наведення числових результатів у цьому розділі.

2.6.5 Локальність виконання, конфіденційність і межі масштабування системи

Локальне виконання зменшує залежність від зовнішніх сервісів і спрощує контроль проходження документних даних. Водночас локальність не є гарантією конфіденційності без окремої політики доступу, керування правами, шифрування та експлуатаційного адміністрування [12]. У межах MVP система орієнтована на однокористувацький послідовний режим, локальне середовище виконання, локальний індекс, локальні конфігурації та файлову звітність. Серверне масштабування, багатокористувацький режим, оптичне розпізнавання символів, промисловий моніторинг, база даних і графічний інтерфейс користувача не входять до поточного MVP. Узагальнені вимоги до середовища запуску наведено в таблиці 2.4.

Таблиця 2.4 – Вимоги до середовища запуску MVP

Вимоги	Умови	Роль у MVP	Обмеження
Апаратне середовище	MacBook Air M1, 8 ГБ уніфікованої пам’яті, macOS	Визначає ресурсні межі локального запуску	Не призначено для багатокористувацького або серверного навантаження
Python-середовище	Python, venv, requirements.txt, requests	Координація конвеєра, ізоляція залежностей, HTTP-взаємодія з Ollama	Не є промисловою інфраструктурою
Середовище виконання LLM	Ollama, локальні теги моделей	Запуск локальних моделей і повернення відповіді LLM	Не виконує добір контексту, валідацію або звітність
Пошукові артефакти	Кеш векторних представлень, FAISS-індекс, YAML-конфігурації	Локальний добір контексту та фіксація параметрів запуску	Потребує попередньої підготовки й перевірки кешу
Схеми та запити	JSON Schema, шаблони запитів, еталонні відповіді	Фіксація структури результату, інструкцій і очікуваної форми відповіді	Не замінює післягенераційну програмну перевірку
Журнали й звітність	JSONL, CSV, Markdown	Фіксація запусків, помилок, метрик і технічних звітів	Не є промисловим моніторингом

Примітка. Таблицю складено на основі джерел [12], [29], [49]-[52], [54], [45]-[48] та файлів репозиторію: README, project_full_map_clean.txt, mvp_verification_summary.md, configs, schemas, prompts, reports.

2.7 Висновки за розділом 2

У розділі 2 обґрунтовано локальні мовні моделі, середовище виконання та технологічний стек MVP для системи автоматичного формування структурованих документів. Визначено, що LLM у цій системі не є автономним автором документа, а виконує роль керованого компонента, який формує кандидатний JSON-об’єкт на основі підготовленого джерельного контексту.

Спроєктовано архітектуру програмного конвеєра, що охоплює зчитування цифрових і підготовлених джерел, нормалізацію, лінеаризацію таблиць, фрагментацію з метаданими, добір контексту, побудову запиту, локальний інференс, каскадну валідацію, верифікацію значень поля source_excerpt, журналювання і звітність. Окремо пояснено роль стратегії запитів як проміжної ланки між добором контексту, Ollama та програмним контуром перевірки.

Зафіксовано вимоги до локального середовища запуску та межі MVP: цифрові або підготовлені джерела без оптичного розпізнавання символів, однокористувацький послідовний режим, локальні моделі, локальний індекс і файлові звітні артефакти.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ГЕНЕРАЦІЇ СТРУКТУРОВАНИХ ДОКУМЕНТІВ

3.1 Структура програмного проєкту та середовище розробки

Програмну реалізацію інтелектуальної системи виконано як локальний Python-проєкт із розмежуванням програмної логіки, конфігурацій, схем валідації, інструкцій керування, тестових матеріалів, запускових процедур і звітних артефактів. Така організація підтримує кероване виконання MVP-конвеєра без змішування коду, статичних налаштувань і результатів запусків. Репозиторій відображає послідовність обробки, за якої цифрові або підготовлені документні джерела проходять приймання, нормалізацію, фрагментацію, добір контексту, локальний інференс, формування кандидатного JSON-об’єкта, програмну перевірку, контроль джерельного фрагмента та фіксацію звітних артефактів. В таблиці 3.1 наведено відповідність етапів реалізованого конвеєра основним програмним модулям і їхній ролі в системі.

Таблиця 3.1 – Відповідність етапів конвеєра програмним модулям

Етап роботи	Основні модулі	Роль у системі
Вилучення й нормалізація документних даних	ingestion, preprocessing	Перетворення вхідних файлів у підготовлене текстове й табличне подання
Фрагментація документного вмісту	chunking	Поділ підготовлених даних на фрагменти з ознаками походження
Добір джерельного контексту	retrieval	Пошук релевантних фрагментів для подальшого передавання моделі
Формування запиту до моделі	prompting, prompts	Поєднання інструкції, схеми відповіді та відібраного контексту
Перевірка кандидатного JSON-результату	schemas, validation, scripts	Контроль структури, типів і джерельної відповідності JSON-об’єкта

Далі наведено організацію каталогів, середовища залежностей і запускові процедури, що забезпечують виконання зазначених етапів.

3.1.1 Організація каталогів репозиторію та структура програмних пакетів

Основну програмну логіку зосереджено в пакеті src/docgen/, який реалізує компоненти конвеєра. Внутрішня структура пакета побудована за принципом розподілу відповідальності між етапами обробки. Підпакет ingestion забезпечує приймання документних джерел, preprocessing — нормалізацію й лінеаризацію табличного вмісту, chunking — логічну сегментацію, retrieval — індексацію та добір контексту, prompting — формування запитів до мовної моделі, schemas — Pydantic-моделі результату, validation — післягенераційний контроль кандидатного JSON-об’єкта. Допоміжні засоби винесено до utils. Такий поділ зменшує ризик змішування функцій між рівнями обробки даних.

Керувальні елементи винесено за межі основного пакета. Каталог configs/ містить YAML-конфігурації моделей, приймання джерел і добору контексту [45]. Каталог schemas/ зберігає mvp_output.schema.json як формальний контракт JSON Schema для цільового об’єкта [18]. Каталог prompts/ містить системні інструкції, завдання для формування JSON і приклад очікуваної структури. Запускові процедури розміщено в scripts/, контрольні матеріали — у test_inputs/, test_outputs/smoke/ і data/gold/expected_json/, а технічні звіти — у reports/. Підсумовуючи викладене, структура репозиторію відокремлює код, конфігурації, правила перевірки, тестові матеріали й артефакти виконання.

3.1.2 Ізольоване середовище залежностей та керування конфігураціями

Локальне функціонування системи реалізовано в ізольованому Python-середовищі venv, що зменшує залежність від глобальних пакетів операційної системи та підвищує відтворюваність запусків [49], [51]. Файл requirements.txt фіксує перелік сторонніх бібліотек, потрібних для HTTP-взаємодії з Ollama, зчитування документів, векторного пошуку, формальної валідації та тестових

процедур [52]. Ізоляція середовища не скасовує потреби в попередній підготовці локальних моделей і кешу векторних представлень, але спрощує контроль складу залежностей.

Декларативне керування параметрами конвеєра реалізовано через YAML-конфігурації [45]. Файл configs/models.yaml фіксує локальні теги моделей, їхні ролі, температуру генерації та базовий розмір контексту. Конфігурація configs/retrieval.yaml визначає модель векторних представлень, режим локального кешу, тип індексу FAISS, нормалізацію векторів і ліміти добору контексту. Параметри chunk_size_chars=600, chunk_overlap_chars=80, semantic_top_k=4, keyword_max_chunks=2, max_context_chunks=6 і max_context_chars=6000 використовуються як фіксовані налаштування MVP, а не як універсально оптимальні значення. Файл configs/ingestion.yaml задає підтримувані формати, виключені джерела, політику рядкової лінеаризації таблиць і послідовний режим ресурсоємних етапів.

3.1.3 Запускові скрипти та керування виконанням конвеєра

Каталог scripts/ є технічним інтерфейсом для запуску окремих процедур конвеєра та підготовки артефактів перевірки. Скрипт cache_embedding_model.py використовується для попереднього кешування моделі векторних представлень. run_retrieval_dry_run.py дає змогу перевірити добір контексту без виклику мовної моделі. validate_mvp_output.py виконує післягенераційну перевірку JSON-об’єкта, а за наявності джерельного тексту — контроль source_excerpt на точний дослівний збіг із вхідним фрагментом.

Скрипти run_synthetic_dry_runs.py і run_format_dry_runs.py призначені для контрольних проходжень на підготовлених синтетичних і форматних вхідних даних. create_format_fixtures.py формує контрольні файли підтримуваних форматів. Наскрізне виконання координує run_benchmark.py: він об’єднує читання конфігурацій, підготовку контексту, HTTP-взаємодію з локальним Ollama API через requests, отримання відповіді, валідацію й запис артефактів [50]. У

межах розділу 3 ці скрипти розглядаються як засоби реалізації й диспетчеризації конвеєра; кількісні результати їх виконання належать до розділу 4.

3.2 Реалізація вилучення та нормалізації документних даних

Вхідний рівень системи реалізовано як набір процедур для приймання цифрових або підготовлених документних джерел і приведення їх до спільного проміжного подання. На цьому етапі конвеєр не створює нову інформацію, не виконує семантичне збагачення й не інтерпретує зміст документа. Його функція обмежена вилученням доступного текстового й табличного вмісту, первинною нормалізацією та перетворенням різнорідних джерел у записи, придатні для сегментації, формування метаданих, добору контексту і побудови запиту до мовної моделі. Реалізація охоплює TXT, synthetic TXT, CSV, XLSX, DOCX і PDF із текстовим шаром. Форматні обмеження вхідного рівня деталізовано у п. 3.7.1.

3.2.1 Обробка цифрових PDF-документів із текстовим шаром

Обробку PDF-джерел реалізовано для документів із придатним електронним текстовим шаром. Для відкриття PDF і доступу до сторінок використовується PyMuPDF [34]. Якщо відповідна залежність недоступна, обробка PDF завершується помилкою цього етапу без переходу до оптичного розпізнавання символів або альтернативного способу вилучення. Текст отримується посторінково через текстове представлення сторінки; порожні сторінки не формують подальших записів, а непорожній вміст зберігається як запис із content_type=text.

Службові ознаки PDF-запису містять source_type=pdf, page і ocr=false. Разом із базовими полями document_id, source_path, source_name, source_type і content ці дані створюють первинну прив’язку вилученого тексту до вихідного файла. У межах MVP PDF розглядається як цифровий контейнер із доступним текстом, а не як зображення сторінки. Система не розпізнає скани, не аналізує

координати блоків, не відновлює геометрію сторінки й не реконструює повну табличну структуру PDF.

3.2.2 Вилучення текстового й табличного вмісту з DOCX-документів

Обробку DOCX-документів реалізовано через python-docx, що забезпечує доступ до абзаців і таблиць документа [35]. Якщо залежність недоступна, обробка DOCX завершується помилкою відповідного етапу. Текстові абзаци зчитуються послідовно, нормалізуються й зберігаються як записи з content_type=text. Для кожного непорожнього абзацу в metadata фіксуються source_type=docx і paragraph_index, що створює службову прив'язку до структури вихідного документа без аналізу стилів, колонтитулів або внутрішнього XML-представлення.

Табличний вміст DOCX обробляється через індекси таблиць і рядків. Комірки кожного рядка перетворюються на словникове представлення з технічними назвами column_1, column_2 тощо. Далі застосовується спільна процедура лінеаризації, яка формує текстовий запис із content_type=table. У metadata такого запису фіксуються source_type=docx, table_index, row_index і перелік column_names. Реалізація не відновлює стилі, вкладені таблиці, зображення, коментарі або складне форматування.

3.2.3 Лінеаризація табличних даних CSV і XLSX

Для CSV і XLSX реалізовано перетворення двовимірних табличних структур у рядкове текстове подання. CSV-файли зчитуються засобами pandas, а за відсутності цієї бібліотеки передбачено резервне зчитування через стандартний модуль csv [36], [47]. XLSX-файли зчитуються через pandas із використанням openpyxl; при недоступності потрібних залежностей система повідомляє про неможливість обробки цього формату [36], [37]. Для обох форматів назви колонок

приводяться до текстового вигляду, а значення рядків передаються до спільної процедури лінеаризації.

Лінеаризація виконується за правилом “рядок у текст”: кожен рядок перетворюється на послідовність пар «назва колонки: значення», розділених крапкою з комою. Порожні назви колонок або порожні значення не включаються до підсумкового тексту. Для CSV у metadata зберігаються source_type=csv, row_index і column_names; для XLSX додатково фіксується sheet_name. Такий спосіб подання ізолює подальший конвеєр від безпосередньої роботи з табличною сіткою й водночас зберігає зв’язок між реквізитом і його значенням у межах рядка. Реалізація не відновлює об’єднані комірки, багаторівневі заголовки, формули, стилі й геометричну топологію аркуша.

3.2.4 Нормалізація вилученого тексту та стандартизація проміжного подання

Нормалізація є завершальним етапом вхідного рівня перед сегментацією. Вона застосовується до тексту з TXT, PDF і DOCX-абзаців, а також до назв колонок і значень комірок під час лінеаризації. Процедура обробляє відсутні значення як порожній текст, замінює службовий маркер BOM і нерозривні пробіли, уніфікує послідовності пробільних символів та обрізає зайві пробіли на межах рядка. Це стандартизує технічне подання без змістового переписування джерела.

Після нормалізації різнорідні вхідні дані зводяться до спільної структури запису з полями document_id, source_path, source_name, source_type, content, content_type і metadata. Для TXT формується один текстовий запис; якщо шлях до джерела містить ознаку synthetic, content_type позначається як synthetic. Для PDF і текстових абзаців DOCX формуються записи типу text, для CSV, XLSX і DOCX-таблиць — записи типу table. На цьому рівні система не створює кінцеві source_anchor і не виконує добір контексту, але формує стандартизовані записи для подальшої фрагментації й джерельної простежуваності.

3.3 Реалізація внутрішнього представлення даних, метаданих і фрагментів

Після вилучення та нормалізації документні дані не передаються далі як суцільний текстовий масив. Реалізація формує проміжне машинозчитуване представлення, у якому текстові блоки, лінеаризовані табличні рядки та службові ознаки джерела існують як записи з єдиною базовою структурою. Це дає змогу перейти від різнорідних файлових форматів до контрольованих одиниць обробки, придатних для сегментації, маркування метаданими та добору релевантного контексту. На цьому рівні не виконується векторизація, ранжування, генерація відповіді або перевірка кандидатного JSON-об’єкта.

3.3.1 Проміжне машинозчитуване представлення документних даних

Початковою одиницею внутрішнього представлення є запис, сформований після приймання й нормалізації джерела. Кожен запис містить document_id, source_path, source_name, source_type, content, content_type і metadata. Поле content зберігає нормалізований текстовий блок або лінеаризований табличний рядок, а content_type розрізняє звичайний текст, synthetic-вхід і табличне подання. Завдяки цьому подальші етапи працюють не з об’єктами PDF, DOCX, CSV або XLSX безпосередньо, а з уніфікованими записами.

Текстові й табличні дані залишаються логічно розділеними. PDF-сторінки й DOCX-абзаци представлені як записи з текстовим вмістом, а CSV, XLSX і таблиці DOCX — як лінеаризовані записи типу table. У metadata зберігаються доступні ознаки походження: для PDF — сторінка й ознака ocr=false; для DOCX-абзаців — індекс абзацу; для DOCX-таблиць — індекси таблиці й рядка; для CSV і XLSX — індекс рядка, назви колонок, а для XLSX також назва аркуша. Це представлення ще не є фрагментною моделлю, але фіксує зв’язок між вилученим вмістом і джерелом.

3.3.2 Логічне розбиття текстових і табличних даних на фрагменти

Перетворення проміжних записів на фрагменти реалізовано в модулі src/docgen/chunking/chunk_builder.py. Параметри сегментації завантажуються з configs/retrieval.yaml: chunk_size_chars=600, chunk_overlap_chars=80, min_chunk_chars=80. Ці значення визначають максимальний розмір фрагмента, перекриття між суміжними текстовими фрагментами й мінімальну довжину змістового фрагмента. Вони є фіксованими параметрами MVP, а не універсально оптимальними значеннями для всіх документів.

Для текстових записів застосовується символьне розбиття з перекриттям. Якщо content не перевищує встановлений розмір, запис утворює один фрагмент. Для довших текстів система формує послідовність інтервалів char_start і char_end, намагаючись завершувати фрагмент на межі пробільного символу. Перекриття зменшує ризик втрати значення на межі розбиття, але не є семантичним аналізом документа. Для content_type=table діє окреме правило: лінеаризований рядок не розбивається, якщо його довжина не перевищує chunk_size_chars, і зберігається як один фрагмент зі стратегією table_row. Якщо табличний запис довший, застосовується загальна символьна процедура, що є обмеженням реалізованого MVP.

3.3.3 Формування метаданих фрагментів і зв’язку з джерелом

Кожен сформований фрагмент зберігає базові поля джерельного запису й отримує власні службові ознаки. До структури фрагмента входять document_id, source_path, source_name, source_type, chunk_id, chunk_index, content, content_type, metadata, source_excerpt, char_start і char_end. Ідентифікатор chunk_id формується за шаблоном document_id:chunk:chunk_index. Поле chunk_index задає порядок фрагмента, а char_start і char_end фіксують його позиційні межі в межах нормалізованого запису.

Метадані фрагмента утворюються через успадкування metadata вихідного запису з додаванням параметрів сегментації: source_record_index, original_content_type, chunking_strategy, chunk_size_chars і chunk_overlap_chars. Для текстових фрагментів стратегія позначається як char_window, а для коротких табличних рядків — як table_row. Поле source_excerpt дорівнює вмісту сформованого фрагмента й виконує роль джерельного витягу для добору контексту, побудови запиту та подальшої перевірки. У цьому шарі ще не створюється кінцевий source_anchor для кандидатного JSON-об’єкта; формується набір фрагментів і службових ознак, потрібних для подальшої прив’язки.

3.3.4 Початкова простежуваність даних на етапі сегментації

Початкова простежуваність закладається до звернення до мовної моделі. Вона ґрунтується на тому, що кожен фрагмент містить chunk_id, source_excerpt, базові поля джерела та metadata, успадковані з попереднього етапу. Для PDF це зберігає прив’язку до сторінки, для DOCX — до абзацу або рядка таблиці, для CSV і XLSX — до рядка, назв колонок і, за наявності, аркуша. Такі службові ознаки не доводять фактичну правильність майбутнього результату, але підвищують перевірюваність походження фрагментів.

Сегментація відокремлює технічну простежуваність від генеративної частини системи. Мовна модель надалі працює не з повним документом, а з обмеженим набором фрагментів, що мають ідентифікатори, службові ознаки та джерельні витяги. З огляду на це, можна стверджувати, що простежуваність починається не після генерації кандидатного JSON-об’єкта, а на етапі побудови фрагментів.

3.4 Реалізація архітектурного шару пошуково-доповненої генерації: індексація, семантичний і резервний лексичний пошук

Шар пошуково-доповненої генерації реалізовано як окремий програмний рівень між фрагментацією документних даних і побудовою запиту до мовної моделі. Він не є функцією Ollama та не виконується самою LLM. Його завдання полягає в доборі обмеженого набору джерельних фрагментів, які передаються до модуля формування запиту як контрольований контекст. Вхідними даними цього рівня є фрагменти з полями chunk_id, content, content_type, metadata і source_excerpt. Виходом є дедуплікований набір контекстних елементів, отриманий через семантичний пошук за векторними представленнями та резервний лексичний пошук за точними технічними маркерами.

3.4.1 Локальне формування векторних представлень фрагментів

Формування векторних представлень реалізовано в модулі src/docgen/retrieval/embeddings.py. У конфігурації configs/retrieval.yaml як модель векторизації визначено sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 [38], [39]. Для локального MVP використовується параметр local_files_only=true, а підготовку кешу винесено до scripts/cache_embedding_model.py. Якщо модель або бібліотека sentence-transformers недоступні локально, цей етап завершується помилкою з вимогою попереднього кешування або зміни режиму завантаження.

Для векторизації використовується список текстів із поля content сформованих фрагментів незалежно від того, походять вони з текстових блоків чи лінеаризованих таблиць. Модель формує числові вектори, які приводяться до формату, сумісного з подальшою індексацією. У реалізації передбачено нормалізацію векторів перед семантичним пошуком. Це узгоджується з використанням FAISS IndexFlatIP, оскільки пошук у такій конфігурації

виконується через внутрішній добуток нормалізованих векторних представлень, що відповідає косинусній близькості для нормалізованих векторів.

3.4.2 Побудова локального індексу FAISS для семантичного пошуку

Локальний семантичний пошук реалізовано в модулі src/docgen/retrieval/faiss_index.py. Для індексації використовується FAISS IndexFlatIP, що відповідає параметру index_type=IndexFlatIP у configs/retrieval.yaml [40]. Індекс створюється в оперативній пам’яті, оскільки в конфігурації задано persist_indexes=false. У межах MVP це означає відсутність окремої серверної векторної бази даних і збережених індексів між незалежними запусками.

Перед додаванням до індексу вектори перевіряються як двовимірний масив, а їхня кількість має відповідати кількості фрагментів. Якщо активовано normalize_embeddings=true, вектори додатково нормалізуються перед передаванням до FAISS. Після визначення розмірності створюється індекс, до якого додаються нормалізовані векторні представлення. Пошук виконується за одним вектором запиту, а кількість семантичних результатів обмежується semantic_top_k=4. FAISS повертає позиції векторів і значення близькості, але не є базою знань, валідатором змісту або механізмом доказу правильності фрагмента.

3.4.3 Зв’язування векторних ідентифікаторів із метаданими фрагментів

Числовий результат FAISS не використовується самостійно. Для відновлення повного контекстного фрагмента формується metadata_map, у якому кожному vector_id відповідає копія вихідного фрагмента. Ідентифікатор вектора визначається порядком додавання векторних представлень до індексу, тому коректність зв’язування залежить від відповідності між порядком списку фрагментів і порядком рядків у матриці векторів.

Після пошуку кожна знайдена позиція зіставляється з елементом metadata_map. На виході формується результат із chunk_id, content, metadata, retrieval_type=semantic і retrieval_score. Далі результат FAISS поєднується з вихідним списком фрагментів через chunk_id, що дає змогу відновити document_id, source_type, content_type і source_excerpt. Завдяки цьому рівень добору контексту повертає не абстрактний вектор, а контекстний об’єкт із текстом і службовими ознаками походження. Такий зв’язок потрібний для формування запиту, фіксації використаних фрагментів і подальшої перевірки джерельного збігу.

3.4.4 Реалізація резервного лексичного пошуку для технічних маркерів

Резервний лексичний пошук реалізовано в модулі src/docgen/retrieval/keyword_fallback.py. Його функція полягає не в заміні семантичного пошуку, а в зменшенні ризику пропуску точних технічних маркерів, які можуть бути слабко представлені у векторному просторі. До таких маркерів належать числові значення, одиниці вимірювання, стандарти, ідентифікатори, коди й окремі службові слова. У конфігурації configs/retrieval.yaml цей контур увімкнено через keyword_fallback.enabled=true, а кількість результатів обмежено keyword_max_chunks=2. Поєднання лексичного та семантичного добору відповідає інженерній практиці комбінування точних і семантичних механізмів пошуку [41], [42].

Пошук починається з нормалізації запиту: текст переводиться до нижнього регістру, службові пробіли уніфікуються, а за регулярними шаблонами виділяються групи number_units, standards, identifiers, numbers, units і words. Далі система перевіряє наявність відібраних термінів у нормалізованому вмісті кожного фрагмента. Знайдені фрагменти отримують retrieval_type=keyword_hint, список matched_terms, keyword_score і match_count. Реалізація не використовує TF-IDF, BM25, стемінг, Elasticsearch або окрему лексичну базу даних; резервний канал обмежений прямим пошуком нормалізованих маркерів у тексті фрагментів.

3.4.5 Об’єднання, дедуплікація та обмеження знайденого контексту

Завершальний етап рівня добору контексту реалізовано в модулі src/docgen/retrieval/retriever.py. Спочатку семантичні результати обмежуються semantic_top_k, а лексичні — keyword_max_chunks. Далі за chunk_id формується словник для відновлення повного фрагмента з базового списку.

Об’єднання результатів виконується за стабільним ідентифікатором chunk_id, що відповідає параметру deduplicate_by=chunk_id. Якщо один і той самий фрагмент знайдено семантичним і лексичним каналами, retrieval_type змінюється на both (обидва), а знайдені лексичні маркери додаються до matched_terms без дублювання. Якщо фрагмент знайдено лише лексичним каналом, він зберігає тип keyword_hint. Після об’єднання застосовується обмеження контексту: не більше max_context_chunks=6 фрагментів і не більше max_context_chars=6000 символів сумарного текстового вмісту. Такий механізм не гарантує повноту знайденого контексту, але формує контрольований набір джерельних фрагментів для побудови запиту до мовної моделі.

3.5 Реалізація взаємодії з мовною моделлю та керування інструкціями

Після добору джерельного контексту система переходить до взаємодії з локальною мовною моделлю. Цей етап реалізовано як керований проміжний рівень між пошуково-доповненою генерацією та програмною валідацією. Python-рівень не передає моделі весь документ і не очікує від неї фінально оформленого документа. До локального середовища виконання передається сформований запит, що містить системну інструкцію, завдання, обмежений джерельний контекст, JSON Schema-контракт і приклад очікуваної структури. Результатом роботи моделі є raw_output, який за наявності придатної JSON-структури розглядається як кандидатний JSON-об’єкт і передається на подальший контроль.

3.5.1 Локальна взаємодія з середовищем виконання мовних моделей

Взаємодію з мовною моделлю реалізовано через Ollama, яке в межах системи виконує роль локального API-сервера для запуску вибраної LLM [27], [30]. Звернення здійснюється з Python-рівня через HTTP-запит бібліотекою requests [50]. У наскрізному сценарії scripts/run_benchmark.py використовується endpoint http://localhost:11434/api/generate. До тіла запиту передаються назва моделі, сформований запит, режим stream=false, JSON Schema як параметр format і блок options із параметрами temperature та num_ctx.

Активна модель визначається через configs/models.yaml. Під час запуску службовий model_key зіставляється з локальним Ollama-тегом, наприклад llama3.2:3b або gemma4:e2b, після чого цей тег передається до Ollama як значення model. Ollama не виконує вилучення документних даних, добір контексту, валідацію JSON або перевірку джерельного фрагмента. Його функція обмежена запуском локальної мовної моделі та поверненням текстового результату для подальшої обробки Python-рівнем.

3.5.2 Формування інструкцій на основі системного завдання, джерельного контексту та цільової структури

Формування запиту до мовної моделі реалізовано в модулі src/docgen/prompting/prompt_builder.py. Функція build_prompt поєднує системну інструкцію з prompts/system/mvp_extraction_system.txt, шаблон завдання з prompts/tasks/extract_mvp_output_json.txt, приклад структури з prompts/templates/mvp_output_example.json і JSON Schema-контракт schemas/mvp_output.schema.json. Таке поєднання синхронізує інструкційний рівень, приклад очікуваного об’єкта та правила подальшої перевірки.

Джерельний контекст формується з відібраних фрагментів. Для кожного фрагмента до запиту додаються chunk_id, доступні document_id, source_type, content_type, retrieval_type, retrieval_score, matched_terms, metadata і текстовий

вміст. Для тексту контексту пріоритет має source_excerpt, а за його відсутності використовується content. Якщо фрагменти не знайдено, до запиту вставляється службове повідомлення про відсутність джерельного контексту з вимогою повертати null і статус insufficient_data для відсутніх даних. Конструктор запитів формує не довільний текстовий запит, а контрольований інтерфейс між відібраним контекстом і локальною LLM.

3.5.3 Параметри інференсу та стабілізація генерації структурованого результату

Параметри інференсу передаються до Ollama через тіло HTTP-запиту. У реалізованому сценарії запит містить stream=false, поле format зі схемою mvp_output.schema.json і блок options, у якому задаються temperature та num_ctx. Базові значення беруться з configs/models.yaml; для моделей у конфігурації встановлено temperature=0.0 і num_ctx=4096. Це відповідає орієнтації MVP на зменшення варіативності відповіді та обмеження контексту в локальному середовищі.

Використання temperature=0.0 не гарантує правильність або повну відтворюваність результату, але зменшує варіативність генерації. Передавання JSON Schema через параметр format задає структурне обмеження для відповіді моделі на рівні локального середовища виконання Ollama [27]. Водночас це не замінює подальший програмний контроль. Тому результат LLM трактується як проміжний кандидатний JSON-об’єкт, прийнятність якого визначається не фактом отримання відповіді, а каскадом перевірок: синтаксичний розбір JSON, JSON Schema, Pydantic і контроль source_excerpt.

3.5.4 Обробка помилок локального інференсу та неповних відповідей моделі

Реалізація відокремлює успішне отримання відповіді від технічних і структурних збоїв локального інференсу. HTTP-запит до Ollama виконується з обмеженням часу очікування, після чого відповідь перевіряється через raise_for_status. Якщо локальний сервер недоступний, модель не завантажена, середовище виконання повертає HTTP-помилку або виникає інший виняток, система не підміняє результат порожнім JSON і не продовжує обробку як успішну. Для такого сценарію формується контрольований error result.

У структурі помилки фіксуються порожній raw_output, негативні значення json_parse_ok, schema_valid_ok і pydantic_ok, тип помилки у validation_error_type та повідомлення у validation_error_message. Для успішної відповіді Ollama зберігається сирий текст raw_output, кількість символів відповіді, службові метрики середовища виконання за наявності, а також шлях до збереженого raw_output. Якщо модель повертає текст без придатного JSON або без відповідності схемі, це не виправляється на етапі інференсу, а передається до процедур валідації.

3.6 Реалізація валідації, журналювання, метрик і джерельної простежуваності результатів

Після отримання відповіді мовної моделі система не приймає її як завершений результат. Текстовий raw_output передається до програмного контуру перевірки, а технічні збої інференсу оформлюються як контрольований error result. Контрольний контур послідовно визначає машинну розбірність, структурну відповідність, прикладну коректність типів, дотримання політики недостатності даних і наявність дослівної джерельної прив’язки. Цей рівень фіксує основну інженерну межу MVP: мовна модель формує лише кандидатний JSON-об’єкт, а прийнятність результату встановлюється зовнішніми Python-процедурами.

Валідація не доводить фактичну технічну істинність значень, але відсікає синтаксично, структурно й джерельно неприйнятні відповіді. Узагальнену послідовність післягенераційного контролю кандидатного JSON-об’єкта подано на рисунку 3.1.

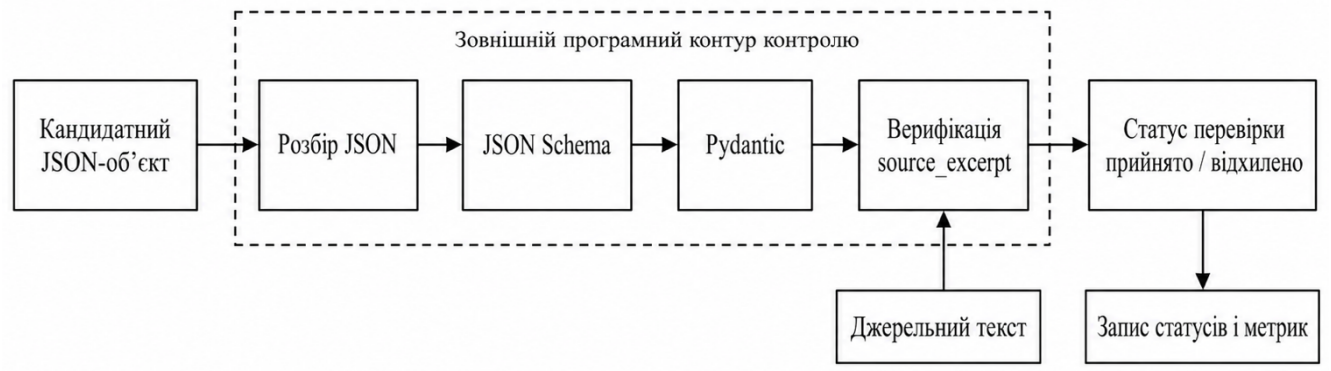


Рисунок 3.1 – Каскад програмної перевірки кандидатного JSON-результату

3.6.1 Синтаксичний розбір кандидатного JSON-об’єкта

Першим рівнем контролю є синтаксичний розбір відповіді мовної моделі. У модулі src/docgen/validation/mvp_validator.py відповідь передається до parse_json_text, яка виконує розбір через json.loads. На цьому етапі не оцінюється зміст, повнота реквізитів або відповідність джерелу; перевіряється лише те, чи може відповідь бути інтерпретована як коректний JSON-об’єкт відповідно до загального формату JSON [33].

У разі помилки розбору формується результат із json_parse_ok=false, schema_valid_ok=false, pydantic_ok=false, validation_error_type=json_parse_error, повідомленням у validation_error_message і значенням parsed_output=None. Такий механізм відокремлює машинно розбірний кандидатний JSON-об’єкт від тексту, який не може бути переданий до структурної перевірки. Рівень синтаксичного розбору JSON не виправляє відповідь моделі й не реконструює пошкоджений об’єкт; він переводить синтаксичну непридатність у формалізований статус помилки.

3.6.2 Перевірка структури результату засобами JSON Schema

Якщо JSON-розбір виконано успішно, наступним рівнем є структурна перевірка за schemas/mvp_output.schema.json. У реалізації схема завантажується функцією load_schema, після чого validate_against_json_schema застосовує Draft202012Validator із бібліотеки jsonschema [18], [53]. Цей рівень перевіряє наявність обов'язкових верхньорівневих полів schema_version, input_id, document, entities, parameters, conclusion, warnings і processing_metadata, а також структуру вкладених об'єктів, типи значень, допустимі статуси та заборону неочікуваних полів через additionalProperties=false.

JSON Schema виконує роль структурного фільтра перед прикладною перевіркою. Вона визначає, чи відповідає кандидатний об'єкт формальному контракту, але не підтверджує фактичну правильність значень і не встановлює їхню джерельну достовірність. Якщо схема виявляє помилку, результат отримує json_parse_ok=true, schema_valid_ok=false, pydantic_ok=false і validation_error_type=json_schema_error.

3.6.3 Детермінований контроль типів і додаткових полів засобами Pydantic

Після проходження JSON Schema кандидатний об'єкт передається до Pydantic-рівня через MVPOutput.model_validate. Внутрішні моделі src/docgen/schemas/mvp_output.py задають програмне представлення результату: DocumentInfo, Entity, Parameter, Conclusion, WarningItem, ProcessingMetadata, SourceAnchor і загальну модель MVPOutput. Усі ці моделі успадковують StrictModel з режимом extra="forbid", тому зайві поля не приймаються. Pydantic забезпечує типізовану перевірку прикладної моделі даних [19].

Pydantic-перевірка відрізняється від JSON Schema тим, що працює з Python-моделлю системи та містить додаткові правила. Для Parameter і Conclusion використано model_validator, який контролює політику недостатності даних: якщо

статус дорівнює insufficient_data, відповідне значення має бути null. У разі помилки формується validation_error_type=pydantic_error. Успішне проходження цього рівня означає відповідність прикладним обмеженням MVP, але не є доказом технічної істинності згенерованих значень.

3.6.4 Перевірка політики недостатності даних

Політика недостатності даних реалізована як частина прикладного контролю кандидатного JSON-об’єкта. Вона потрібна для розмежування ситуацій, у яких модель має достатній джерельний фрагмент для заповнення поля, і ситуацій, у яких потрібна інформація відсутня або неоднозначна. У схемі результату передбачено статуси extracted, insufficient_data, ambiguous і not_applicable, а інструкції вимагають не вигадувати відсутні реквізити, числові значення, одиниці вимірювання, дати або висновки без джерельної опори.

На рівні Pydantic ця політика конкретизована для параметрів і висновку. Якщо параметр має status="insufficient_data", поле value повинно бути null; якщо висновок має такий самий статус, поле result також повинно бути null. Відсутність значення не прирівнюється до помилки, якщо вона явно представлена як передбачений стан результату. Помилкою є ситуація, коли модель одночасно заявляє недостатність даних і заповнює поле конкретним значенням.

3.6.5 Сувора перевірка фрагмента джерела та контроль джерельної простежуваності

За наявності parsed_output у запусковому сценарії окремо фіксуються два аспекти джерельної простежуваності: наявність атрибута source_anchor і дослівна наявність поля source_excerpt в тексті, переданого для перевірки. Функція source_anchors_present перевіряє, чи представлені атрибути source_anchor як словникові об’єкти для сутностей, параметрів і висновку. Цей статус фіксує наявність структурної джерельної прив’язки, але не оцінює її змістову істинність.

Суворіша перевірка реалізована функцією source_excerpts_found_in_input. Вона збирає всі значення поля source_excerpt з атрибутів source_anchor у масивах entities, parameters і в об’єкті conclusion, після чого кожний непорожній фрагмент перевіряється на дослівну наявність в переданому для перевірки тексті. У run_benchmark.py цей результат фіксується в полі source_excerpt_found_in_input. Перевірка не використовує нечітке зіставлення, семантичну близькість або часткове переформулювання. Значення true означає, що всі зібрані джерельні витяги дослівно знайдені у тексті, а false — невиконання суворого критерію джерельної відповідності, не автоматичну хибність факту [43], [44].

3.6.6 Запис результатів, метрик і звітних артефактів

Результати виконання зберігаються у структурованих артефактах, придатних для подальшого аналізу в розділі 4. У scripts/run_benchmark.py для кожного запуску формується запис із технічними ідентифікаторами експерименту, моделлю, вхідним файлом, версією запиту, назвою і версією схеми, режимом добору контексту, списком retrieved_chunk_ids, типами retrieval, параметрами temperature і num_ctx, шляхом до raw_output і статусами валідації. Сирий текст відповіді зберігається в raw_outputs, що відокремлює первинне виведення даних моделі від агрегованих метрик.

Машинозчитувані результати записуються у results.jsonl, де кожний рядок відповідає окремому запуску [46]. Компактний табличний зріз формується у metrics.csv з полями elapsed_sec, tokens_per_second, output_char_count, json_parse_ok, schema_valid_ok, pydantic_ok, source_anchor_present_ok, source_excerpt_found_in_input, validation_error_type і validation_error_message [47]. Для пояснювальних звітів використовується Markdown [48]. У межах цього підрозділу ці артефакти розглядаються як механізм журналювання й підготовки даних, а не як підстава для висновків про якість моделей або продуктивність системи.

3.7 Обмеження реалізації та межі функціональності базового комплексу програмних засобів

Реалізований базовий комплекс програмних засобів має визначені межі MVP, пов’язані з підтримуваними форматами, способом табличного подання, локальним інференсом і відсутністю промислової інфраструктури. Ці межі задають контрольований обсяг дипломної реалізації, у межах якого описано локальний програмний конвеєр: вилучення, нормалізація, сегментація, добір контексту, побудова запиту, локальний інференс, формування кандидатного JSON-об’єкта, валідація та запис технічних артефактів. Підрозділ фіксує різницю між фактично реалізованим MVP і перспективними напрямами розширення системи.

3.7.1 Межі підтримуваних вхідних форматів і відмова від оптичного розпізнавання

Система працює з цифровими або підготовленими джерелами у форматах TXT, synthetic TXT, CSV, XLSX, DOCX і PDF із текстовим шаром. Обробка сканованих документів, зображень, рукописного тексту та PDF без придатного текстового шару не входить до меж MVP. Оптичне розпізнавання символів не реалізовано, тому система не перетворює графічні сторінки на текст. Це обмежує область застосування підготовленими цифровими джерелами, але спрощує контроль вхідних даних і зменшує ризик некерованих помилок приймання.

3.7.2 Обмеження табличної обробки та відсутність повної реконструкції табличної сітки

Таблична обробка в MVP реалізована через рядкову лінеаризацію. Рядок таблиці перетворюється на текстове подання з парами «назва колонки: значення», що дає змогу зберігати реквізитний зв’язок у межах рядка та передавати

табличний вміст у спільний текстовий контур. Водночас система не виконує повну реконструкцію табличної сітки, геометрії сторінки, об’єднаних комірок, багаторівневих заголовків, формул Excel, стилів і складної табличної топології. Лінеаризація є практичним компромісом MVP між збереженням ключових пар «поле — значення» та ресурсною простотою реалізації.

3.7.3 Обмеження локального інференсу на цільовому апаратному середовищі

Локальний запуск системи орієнтовано на MacBook Air M1 із 8 ГБ уніфікованої пам’яті під керуванням macOS. Це визначає використання компактних локальних моделей через Ollama, послідовне виконання ресурсоємних етапів, локальний індекс FAISS в оперативній пам’яті та обмеження розміру контексту через num_ctx=4096. Реалізація не подається як універсально масштабована LLM-система для великих корпусів або промислових навантажень. Для окремих моделей, більших контекстів або нестабільних локальних конфігурацій можливі технічні збої середовища виконання, їхній кількісний аналіз представлений в наступному розділі.

3.7.4 Перспективні напрями розширення функціональності системи

Подальший розвиток системи може охоплювати оптичне розпізнавання символів для сканованих документів, повнішу реконструкцію макета й таблиць, експорт у DOCX за шаблонами, графічний інтерфейс користувача, багатокористувацький режим, серверне розгортання, рольовий доступ, постійну векторну базу даних, моніторинг і розширені тестові набори. Ці напрями не входять до фактичної реалізації MVP і не повинні трактуватися як приховані можливості поточної системи. Вони визначають межу переходу від локального базового комплексу програмних засобів до розширеного прикладного рішення.

3.8 Висновки за розділом 3

У розділі 3 описано фактичну реалізацію базового комплексу програмних засобів для контрольованого формування структурованого документного результату з цифрових і підготовлених джерел. Реалізацію оформлено як локальний Python-проєкт із відокремленими зонами коду, конфігурацій, схем, запитів, тестових матеріалів і звітних артефактів. Вхідний рівень забезпечує приймання підтримуваних форматів, нормалізацію тексту й лінеаризацію таблиць без оптичного розпізнавання символів і повної реконструкції макета.

Після вилучення дані перетворюються на фрагменти з метаданими, chunk_id і source_excerpt, що створює основу для добору джерельного контексту. Шар пошуково-доповненої генерації реалізовано через локальні векторні представлення, індекс FAISS IndexFlatIP, резервний лексичний пошук, об’єднання, дедуплікацію та обмеження контексту. Взаємодія з мовною моделлю виконується через локальне середовище виконання Ollama, а результат моделі розглядається лише як кандидатний JSON-об’єкт.

Завершальний контроль реалізовано через перевірку JSON, JSON Schema, Pydantic, політики недостатності даних, source_excerpt, журналювання, метрики й артефакти results.jsonl, metrics.csv і raw_outputs. Розділ 3 фіксує не експериментальну ефективність системи, а технічну готовність реалізованого MVP до перевірки в розділі 4.

4 ТЕСТУВАННЯ, ЕКСПЛУАТАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

4.1 Мета, об’єкт і умови тестування системи

4.1.1 Мета експериментальної перевірки локального конвеєра обробки документних даних

Метою експериментальної перевірки є встановлення працездатності реалізованого локального конвеєра обробки документних даних у межах MVP. Перевіряється не автономна здатність великої мовної моделі створювати технічний документ, а контрольований шлях від цифрового або підготовленого джерела до кандидатного JSON-об’єкта, його формальної валідації, джерельної перевірки, журналювання та звітності. Така постановка узгоджується з інженерною логікою роботи: мовна модель є проміжним компонентом, а прийняття або відхилення результату виконується програмним контуром. У межах цієї роботи покращення обробки табличних даних оцінюється не лише за фактом отримання відповіді від мовної моделі, а за здатністю системи зберігати табличні реквізити у машинозчитуваному JSON-поданні, проходити синтаксичну, схемну й прикладну перевірку, фіксувати джерельні прив’язки та коректно відхиляти неповні або джерельно непідтверджені результати.

Експериментальна частина має зафіксувати, чи здатна система відтворювано виконувати вилучення даних, лінеаризацію таблиць, нормалізацію, фрагментацію, добір контексту, побудову запиту, локальний інференс, синтаксичний розбір JSON, валідацію JSON Schema та Pydantic, верифікацію поля source_excerpt і формування звітних артефактів. Методика оцінювання враховує підходи до перевірки систем пошуково-доповненої генерації та типові точки відмов таких систем [15], [16].

4.1.2 Об’єкт перевірки: стабільність структурованого виведення даних, валідація та джерельна простежуваність

Об’єктом перевірки є повний програмний конвеєр: імпорт даних, лінеаризація таблиць, нормалізація, сегментація, побудова векторних представлень, добір контексту, формування запиту, виклик Ollama, отримання кандидатного JSON-об’єкта, синтаксичний розбір, валідація JSON Schema і Pydantic, верифікація поля source_excerpt, журналювання, розрахунок метрик і формування звітів. Основна увага приділяється машинній розбірності результату, проходженню детермінованих перевірок і джерельній простежуваності. Наявність source_anchor фіксує спробу прив’язати результат до джерела, а source_excerpt_found_in_input=true означає проходження суворого дослівного збігу. Значення source_excerpt_found_in_input=false не є автоматичним доказом хибності факту; воно означає невиконання верифікації походження джерела. Таке розмежування важливе, оскільки коректність відповіді, джерельна вірність і формальна валідність не є тотожними властивостями [43], [44].

4.1.3 Умови проведення тестування та обмеження цільового локального середовища

Тестування проводиться в локальному середовищі MacBook Air M1 з 8 ГБ уніфікованої пам’яті під керуванням macOS. Це середовище визначає вибір компактних локальних моделей, послідовне виконання ресурсоємних етапів, використання Ollama як локального середовища виконання LLM та застосування FAISS IndexFlatIP як локального індексу в оперативній пам’яті. Python-рівень працює у віртуальному середовищі та відповідає за приймання даних, добір контексту, виклик моделі, валідацію, метрики й звіти. Конфігураційними умовами є temperature=0.0, num_ctx=4096, локальний кеш моделі векторних представлень, нормалізовані векторні представлення, обмежений контекстний бюджет і резервний лексичний пошук. Перевірка охоплює цифрові або підготовлені

джерела без оптичного розпізнавання символів: TXT, synthetic TXT, CSV, XLSX, DOCX і PDF із текстовим шаром. Скановані документи, рукописні матеріали, повна реконструкція таблиць, графічний інтерфейс користувача, база даних і промисловий експорт документів не входять до меж MVP.

4.2 Формування тестового набору документів і еталонних відповідей

4.2.1 Корпус синтетичних і підготовлених технічних документів

Тестовий набір сформовано як контрольований корпус синтетичних і підготовлених документних джерел для перевірки основних сценаріїв роботи конвеєра. До нього входять синтетичні TXT-файли з технічними протоколами, форматні фікстури CSV, XLSX, DOCX і PDF з текстовим шаром, а також табличні джерела для перевірки рядкової лінеаризації. Кожний файл виконує окрему експериментальну функцію: перевіряє базове вилучення реквізитів, роботу з пропущеними значеннями, конфліктними даними, контекстним шумом, повторюваними секціями або таблично-орієнтованими параметрами. Корпус не претендує на репрезентацію всіх можливих технічних документів. Його функція — забезпечити відтворюваний набір контрольних вхідних даних для перевірки реалізованого MVP у межах підтримуваних форматів. Такий підхід узгоджується з вимогою відтворюваності машинного навчання та програмних експериментів через фіксацію коду, даних, конфігурацій і артефактів перевірки [56], [57].

4.2.2 Сценарії з базовими, пропущеними, конфліктними, зашумленими та табличними даними

Сценарії тестування побудовано навколо ризиків автоматизованого формування структурованого результату з документних джерел. Базовий сценарій перевіряє вилучення реквізитів, параметрів і підсумкового висновку з відносно простого технічного протоколу. Сценарій із пропущеними значеннями перевіряє політику недостатності даних і використання null замість довільного заповнення

відсутніх полів. Конфліктний сценарій фіксує поведінку за наявності взаємно неузгоджених значень. Зашумлені й довші вхідні дані перевіряють стійкість до повторів, службових фрагментів, попередніх і фінальних значень. Таблично-орієнтовані сценарії перевіряють збереження зв'язків між назвами колонок, значеннями, одиницями вимірювання та джерельними фрагментами після лінеаризації. Усі ці сценарії перевіряють не "якість тексту" як таку, а працездатність структурованого конвеєра та його контрольних бар'єрів.

4.2.3 Структура еталонних відповідей у форматі JSON

Еталонні відповіді подано у форматі JSON відповідно до контракту структурованого результату MVP. Їхня функція полягає в заданні контрольної структури полів, типів, обов'язкових секцій і прикладних правил для підготовленого піднабору тестових вхідних даних. Еталонний JSON узгоджується зі схемою mvp_output.schema.json і містить блоки schema_version, input_id, document, entities, parameters, conclusion, warnings і processing_metadata. Для стану вилучених значень використовуються статуси extracted, insufficient_data, ambiguous і not_applicable. Поля source_anchor, fragment_id і source_excerpt забезпечують перевірку наявності джерельних фрагментів у кандидатному JSON-об'єкті. JSON у цьому контексті є машинозчитуваним поданням і контрактом перевірки, а не готовим документним файлом. Формат JSON та валідація за JSON Schema обґрунтовують машинну розбірність і структурну перевірку, але не доводять фактичну істинність значень [18], [33], [53].

4.2.4 Маніфест тестових вхідних даних і зв'язок зі сценаріями перевірки

Маніфести тестових даних фіксують атрибути вхідних файлів і їхній зв'язок зі сценаріями перевірки. Для синтетичних фікстур вони задають input_id, шлях до джерела, тип, складність, призначення, ознаки пропусків, конфліктів,

табличної структури, шуму та використання в контрольованих прогонах. Для форматних фікстур додатково реєструється тип джерела: CSV, XLSX, DOCX або PDF із текстовим шаром. Зіставлення тестових вхідних даних з еталонними JSON-об’єктами і звітними артефактами забезпечується структурою каталогів, input_id та експериментальними скриптами. Підготовка форматних фікстур виконується через create_format_fixtures.py, а локальна підготовка моделі векторних представлень підтримується cache_embedding_model.py. У межах розділу 4 ці матеріали розглядаються як контрольована основа відтворюваної перевірки, а не як повний корпус усіх можливих технічних документів.

4.3 Тестові сценарії, фази перевірки та критерії успішності

4.3.1 Наскрізна програма виконання конвеєра для контрольних документів

Наскрізна програма перевірки передбачає проходження документа через реалізований конвеєр із фіксацією критичних етапів. Джерело завантажується підсистемою приймання даних, проходить нормалізацію, табличну лінеаризацію за потреби, сегментацію на фрагменти та збагачення метаданими. Далі виконуються побудова векторних представлень, семантичний пошук у FAISS, резервний лексичний добір технічних маркерів, об’єднання знайденого контексту, дедуплікація та формування запиту. Після виклику Ollama отриманий кандидатний JSON-об’єкт передається на синтаксичний розбір JSON, валідацію JSON Schema та Pydantic і верифікацію source_excerpt. Інтеграційним контуром наскрізного запуску є run_benchmark.py. Контрольовані прогони використовуються для ізольованої перевірки окремих рівнів без обов’язкового локального інференсу. Завершальним етапом є запис метрик, статусів помилок, журналів і звітних файлів. Така організація відповідає практиці фіксації експериментальних артефактів для подальшої перевірки та повторного аналізу [56], [57].

4.3.2 Розподіл перевірки на первинне порівняння моделей, приймальну перевірку та тестування на ускладнених сценаріях

Експериментальна перевірка поділяється на три фази. Перша фаза використовується як первинний відбір локальних моделей, доступних через Ollama, і перевіряє здатність моделі повертати машинно розбірний структурований JSON у межах поточного запиту й контракту валідації. Друга фаза виконує приймальну перевірку компонентів і відтворюваності: модульні тести, контрольовані прогони синтетичних і форматних фікстур, первинні перевірки, офлайн-доступність кешу моделі векторних представлень і обмежені наскрізні запуски основного кандидата. Третя фаза застосовується для ускладнених сценаріїв: зашумлених, довших, таблично-орієнтованих і конфліктних вхідних даних. Такий поділ розмежовує первинну придатність моделей, приймальну працездатність компонентів і поведінку системи в складніших умовах. Він не є повним промисловим benchmark і не дає універсальної оцінки моделей поза реалізованим локальним MVP.

4.3.3 Критерії успішності синтаксичної, структурної, логічної та джерельної перевірки

Успішність запуску оцінюється послідовними критеріями, що відповідають етапам конвеєра. Синтаксичний критерій передбачає завершення запуску без збою локального виконання або API і наявність JSON, який може бути розібраний без json_parse_error. Структурний критерій передбачає відповідність кандидатного JSON-об’єкта схемі mvp_output.schema.json, включно з обов’язковими секціями, типами полів і забороною неочікуваних властивостей. Прикладний критерій перевіряється через Pydantic, зокрема через політику недостатності даних. Джерельний критерій охоплює наявність атрибута source_anchor і проходження верифікації поля source_excerpt. Запуск може бути синтаксично й структурно валідним, але не пройти сувору джерельну перевірку.

Такий випадок фіксується як обмеження простежуваності, а не як повний збій усього конвеєра. Розмежування цих рівнів узгоджується з практикою окремого оцінювання структурної валідності, вірності та джерельної прив’язки в RAG-системах [15], [16], [43], [44].

4.4 Метрики оцінювання якості, продуктивності та простежуваності даних

Метрики експериментальної перевірки формалізують критерії успішності та переводять їх у машинно фіксовані показники проходження етапів конвеєра. Їхня функція полягає не в доведенні повної технічної правильності результату, а в стандартизованій фіксації машинної розбірності, структурної відповідності, прикладної валідності, джерельної простежуваності, часу виконання та статусів локального інференсу. Основними джерелами значень є results.jsonl і metrics.csv, де зберігаються поля json_parse_ok, schema_valid_ok, pydantic_ok, source_anchor_present_ok, source_excerpt_found_in_input, validation_error_type, elapsed_sec і tokens_per_second. Формати JSONL, CSV і Markdown застосовуються для журналювання запусків, компактного подання метрик і пояснювальних звітів [46], [47], [48].

4.4.1 Метрики синтаксичної та структурної коректності JSON-об’єкта

Синтаксична коректність фіксується полем json_parse_ok. Його позитивне значення означає, що відповідь моделі може бути програмно розібрана як JSON-об’єкт і передана на наступний рівень перевірки. Якщо виникає json_parse_error, подальші етапи JSON Schema і Pydantic не застосовуються, оскільки машинно придатний кандидатний JSON-об’єкт не сформовано. Структурна коректність фіксується полем schema_valid_ok. Вона означає відповідність об’єкта схемі mvp_output.schema.json, включно з обов’язковими секціями, типами, вкладеними структурами й допустимим набором полів. Ці метрики не оцінюють фактичну

правильність значень і не підтверджують джерельну достатність відповіді. Вони лише показують, чи пройшов результат два перші машинні бар'єри: синтаксичний і структурний. Така інтерпретація узгоджується з роллю JSON як формату обміну даними та JSON Schema як засобу структурної валідації [18], [33], [53].

4.4.2 Метрики проходження валідації та політики недостатності даних

Прикладна валідність фіксується полем pydantic_ok. Це поле показує, чи відповідає кандидатний JSON-об'єкт Python-моделям даних і додатковим правилам MVP. На цьому рівні перевіряються типи значень, обов'язкові поля, вкладені структури, службові статуси й політика недостатності даних. Зокрема, якщо параметр має status="insufficient_data", поле value має бути null; для висновку з таким самим статусом поле result також має бути null. Помилка pydantic_error означає порушення прикладних правил після проходження синтаксичного розбору JSON і валідації JSON Schema. Проходження валідації Pydantic не означає автоматичної технічної правильності результату. Воно підтверджує відповідність об'єкта формальній прикладній моделі системи. Такий підхід відповідає ролі Pydantic як інструмента типізованого контролю даних і прикладної валідації [19].

4.4.3 Метрики джерельної простежуваності та суворого збігу фрагмента джерела

Джерельна простежуваність оцінюється окремо від синтаксичної, структурної та прикладної валідності. Поле source_anchor_present_ok фіксує наявність службових об'єктів source_anchor у сутностях, параметрах і висновку. Воно показує, що модель сформувала структурну джерельну прив'язку, але не доводить дослівний збіг із джерелом. Для цього використовується поле source_excerpt_found_in_input. Його позитивне значення означає, що всі непорожні source_excerpt знайдені як точні підрядки у вхідному тексті,

переданому для перевірки. Негативне значення означає невиконання верифікації source_excerpt. Воно не є автоматичним доказом хибності факту, оскільки причина може полягати в переформулюванні, обрізанні, зміні пробілів або наслідках нормалізації. Джерельну простежуваність тому потрібно інтерпретувати як окремий критерій прийнятності, а не як повну експертну перевірку змісту [43], [44].

4.4.4 Метрики швидкодії та ресурсної придатності локального інференсу

Продуктивність локального інференсу оцінюється через відсутність збоїв локального виконання або API, elapsed_sec, tokens_per_second, HTTPError, timeout, empty output і validation_error_type. Показник elapsed_sec характеризує тривалість модельного HTTP-виклику через локальний Ollama API у конкретному запуску. Показник tokens_per_second розраховується на основі службових полів, які повертає Ollama, якщо вони доступні. Технічні помилки відокремлюються від структурних: HTTP 500 або timeout означає збій локального виконання чи API, тоді як json_parse_error означає отримання відповіді, непридатної для синтаксичного розбору JSON. Ці метрики характеризують конкретне середовище MacBook Air M1 з 8 ГБ уніфікованої пам’яті, конкретні локальні теги моделей, Ollama, конфігурації та тестові сценарії. Вони не є універсальною оцінкою продуктивності моделей поза межами цього MVP.

4.5 Результати первинного порівняння моделей і приймальної перевірки конвеєра обробки даних

4.5.1 Результати первинного порівняння локальних моделей

Первинне порівняння локальних моделей у першій фазі виконувало роль практичного фільтра для подальшої перевірки. Оцінювалася не загальна якість моделей, а їхня здатність у межах поточного локального конвеєра повертати

машинно розбірний кандидатний JSON-об’єкт, що проходить JSON Schema і Pydantic. Перевірка виконувалася в режимі пошуково-доповненої генерації MVP-стеку, зафіксованому в метриках як rag_mvp, при temperature=0.0 на трьох синтетичних вхідних даних: protocol_basic_measurement.txt, protocol_missing_values.txt і protocol_conflicting_values.txt.

За результатами першої фази llama3.2:3b пройшла синтаксичний розбір JSON, JSON Schema і Pydantic у всіх трьох запусках. У всіх трьох відповідях також зафіксовано source_anchor. Суворий дослівний збіг source_excerpt підтвердився лише в одному з трьох запусків. Модель gemma4:e2b мала один структурно валідний запуск із трьох, а два інші завершилися HTTP 500. Модель gemma4:e4b у всіх трьох запусках завершилася HTTP 500 і не дійшла до етапів валідації. Модель qwen3.5:4b у трьох контрольних запусках не сформувала машинно розбірний JSON і була зафіксована зі статусом json_parse_error. Ці результати не є універсальним рейтингом моделей, але обґрунтовують перехід llama3.2:3b до наступних перевірок як основна модель для подальшої перевірки у межах поточного MVP.

4.5.2 Результати приймальної перевірки компонентів і наскрізного виконання конвеєра

Приймальна перевірка другої фази була спрямована на відокремлення працездатності інфраструктурного Python-рівня від варіативності локального інференсу. Перевірялися модульні тести, контрольовані прогони синтетичних і форматних фікстур, первинна перевірка валідації, офлайн-доступність кешу моделі векторних представлень і обмежені наскрізні запуски основного кандидата. Фінальна перевірка clean submission зафіксувала pytest: 62 успішних запуски, 3 попередження. Попередження пов’язані з застарілими елементами залежностей і не супроводжувалися відмовою тестів. Синтетичні контрольовані прогони завершилися як 20 успішно пройдено, 0 завершено з помилкою, форматні контрольовані прогони — як 15 успішно пройдено, 0 завершено з помилкою.

Початковий контроль підсистем підтвердив різні коди завершення: коректний випадок із відповідним джерелом завершився exit=0, некоректний JSON — exit=1, а коректний JSON із невідповідним джерелом — exit=1. Ці результати підтверджують працездатність детермінованих компонентів конвеєра, але не доводять загальну якість мовної моделі.

4.5.3 Узагальнення якісних, часових і ресурсних показників базової конфігурації

Обмежений набір наскрізних запусків другої фази уточнив поведінку інференсу після проходження неінференсних перевірок. Для конфігурації llama_3b було збережено п’ять запусків на контрольних вхідних даних, включно з базовим, неповним, конфліктним, таблично-подібним і зашумленим сценаріями. Усі п’ять запусків пройшли синтаксичний розбір JSON, JSON Schema і Pydantic; у всіх п’яти зафіксовано source_anchor. Суворий дослівний збіг source_excerpt підтвердився у двох із п’яти запусків. Отже, базова конфігурація з llama3.2:3b відтворювано формувала структурно прийнятний кандидатний JSON-об’єкт у цих п’яти збережених запусках, але джерельна простежуваність з точним збігом залишилася обмеженням.

Опційне порівняння gemma_e2b у другій фазі показало нижчу стабільність локального виконання: з трьох запусків один пройшов JSON/Schema/Pydantic і мав source_anchor, два завершилися HTTP 500, а суворий дослівний збіг source_excerpt не був підтверджений у жодному запуску. Додаткові перевірки чутливості до режиму добору контексту, num_ctx і повторюваності табличного-подібного сценарію показали, що структурні бар’єри проходили стабільніше, ніж сувора джерельна перевірка. Часові та ресурсні показники розглядаються як характеристики конкретних локальних запусків, а не як апаратно-незалежний профіль моделей.

4.6 Результати тестування на ускладнених сценаріях і порівняльного аналізу обраних моделей

Третя фаза експериментальної перевірки була спрямована на оцінювання поведінки локального MVP-конвеєра в ускладнених документних умовах. Порівнювалися дві конфігурації: llama3.2:3b як основний кандидат і gemma4:e2b як порівняльна модель. Аналіз охоплював завершеність запусків, синтаксичний розбір JSON, JSON Schema, Pydantic, верифікацію атрибута source_anchor та поля source_excerpt, збій HTTP/середовища виконання, json_parse_error, elapsed_sec і tokens_per_second. До тестового набору входили protocol_table_heavy_acceptance.txt, protocol_multi_section_noise.txt, protocol_table_like_complex.txt, protocol_noise_mixed.txt і protocol_conflicting_values.txt.

4.6.1 Поведінка системи у сценаріях з дефіцитом і конфліктом даних

Сценарії з дефіцитом, неоднозначністю або конфліктом даних перевіряли не здатність моделі створювати повний текст, а проходження структурного контракту, Pydantic-правил і джерельної перевірки. У protocol_conflicting_values.txt модель llama3.2:3b завершила запуск, пройшла синтаксичний розбір JSON, JSON Schema і Pydantic, а також сформувала source_anchor. Водночас source_excerpt_found_in_input мав значення false, тобто суворий дослівний збіг із джерельним контекстом не був підтверджений. У protocol_table_heavy_acceptance.txt, де наявні таблично-орієнтовані дані та пропуски, llama3.2:3b також пройшла формальні й прикладні перевірки, але не пройшла верифікацію source_excerpt. Для gemma4:e2b у protocol_conflicting_values.txt було отримано структурно валідний результат із source_anchor, але без дослівного збігу; у protocol_table_heavy_acceptance.txt запуск завершився HTTP 500. Це свідчить про те, що у дефіцитних і конфліктних умовах структурна валідність не збігається з джерельною прийнятністю.

4.6.2 Результати перевірки на зашумлених, довших і таблично-орієнтованих вхідних даних

Зашумлені, довші й таблично-орієнтовані вхідні дані перевіряли поведінку конвеєра за наявності службових фрагментів, повторів, псевдотабличних структур і параметрів, розміщених у менш очевидних частинах документа. Для llama3.2:3b усі п’ять запусків третьої фази завершилися успішно з погляду синтаксичного розбору JSON, JSON Schema і Pydantic. У кожному з них було зафіксовано source_anchor. Єдиний позитивний результат верифікації source_excerpt для цієї моделі отримано на protocol_table_like_complex.txt. Інші чотири ускладнені вхідні дані завершилися формально валідними JSON-об’єктами, але без суворого дослівного збігу source_excerpt.

Для gemma4:e2b результати були менш стабільними: protocol_noise_mixed.txt і protocol_conflicting_values.txt пройшли JSON/Schema/Pydantic та містили source_anchor; protocol_table_like_complex.txt завершився json_parse_error через незавершений рядок; protocol_multi_section_noise.txt і protocol_table_heavy_acceptance.txt завершилися HTTP 500. Наведені результати демонструють, що у стресових сценаріях llama3.2:3b зберегла формальну структурованість результатів, тоді як gemma4:e2b продемонструвала поєднання збоїв локального виконання або API, json_parse_error і відсутності точного дослівного збігу.

4.6.3 Оцінка ролі резервного лексичного пошуку при доборі технічних маркерів

Запуски третьої фази виконувалися в режимі MVP-стеку пошуково-доповненої генерації з retrieval_mode=rag_mvp. Конфігурація поєднувала семантичний добір через FAISS із резервним лексичним пошуком технічних маркерів. У звітних артефактах поле keyword_fallback_used мало значення true для збережених спроб третьої фази, а retrieval_types містили semantic, both (обидва) і,

для таблично-насиченого сценарію, keyword_hint. Це підтверджує, що резервний лексичний пошук був задіяний у доборі контексту для ускладнених вхідних даних.

Його роль полягає у зменшенні ризику пропуску коротких технічних маркерів, кодів, числових значень, одиниць вимірювання та табличних реквізитів. Водночас keyword_fallback_used=true не гарантує проходження верифікації source_excerpt. Для llama3.2:3b резервний лексичний пошук був задіяний у всіх п’яти запусках третьої фази, але точний збіг підтвердився лише в одному випадку. Відповідно, лексичний канал доповнює добір контексту, але не замінює валідацію source_excerpt і не доводить джерельну прийнятність кандидатного JSON-об’єкта [41], [42].

4.6.4 Порівняння стабільності структурованого виведення даних та ресурсної придатності обраних моделей

Порівняння обраних моделей у третій фазі підтвердило проміжний висновок: llama3.2:3b є практично придатнішою базовою конфігурацією для поточного локального MVP, тоді як gemma4:e2b зберігає значення порівняльної конфігурації. Для llama3.2:3b зафіксовано 5/5 завершених запусків, 5/5 проходжень синтаксичного розбору JSON, JSON Schema і Pydantic, 5/5 випадків із source_anchor і 1/5 позитивний результат перевірки source_excerpt. Середній час модельного HTTP-виклику становив приблизно 52,8 с, а середня швидкість генерації — близько 17,0 tokens_per_second.

Для gemma4:e2b із п’яти спроб дві пройшли JSON/Schema/Pydantic і містили source_anchor, дві завершилися HTTP 500, а одна — json_parse_error. Суворий дослівний збіг source_excerpt не був підтверджений у жодній спробі. Для двох структурно успішних запусків середній elapsed_sec становив приблизно 335,6 с, а середній tokens_per_second — близько 15,2. Ці результати характеризують поведінку конкретного локального середовища, поточного контракту формування запиту й набору ускладнених сценаріїв. Вони не є

загальною оцінкою моделей поза межами цього конвеєра. Основним обмеженням третьої фази виявилася не відсутність добору контексту, а нестабільність проходження верифікації source_excerpt.

4.7 Аналіз типових помилок, обмежень джерельної простежуваності та меж застосування системи

Результати трьох фаз показали, що помилки конвеєра доцільно систематизувати за рівнем виникнення: синтаксичний розбір JSON, структурна перевірка, прикладна валідація, джерельна простежуваність і локальне виконання модельного етапу. Такий поділ дає змогу відокремити помилки форми кандидатного JSON-об’єкта від помилок прикладної логіки, джерельного підтвердження та відмов локального інференсу. Подібна класифікація відповідає інженерній логіці оцінювання RAG-систем, у яких окремо розглядають релевантність контексту, коректність формату, помилки генерації та місця збою системного конвеєра [15], [16].

4.7.1 Систематизація синтаксичних, структурних і валідаційних помилок

Першу групу становлять синтаксичні помилки, які фіксуються як json_parse_error. Вони виникають тоді, коли відповідь моделі не може бути розібрана як JSON через обірвану структуру, некоректне екранування, незавершений рядок або сторонній текст. У такому випадку JSON Schema і Pydantic не застосовуються, оскільки машинно придатний кандидатний JSON-об’єкт не сформований. Другу групу становлять структурні помилки, пов’язані з невідповідністю schemas/mvp_output.schema.json: відсутністю обов’язкових секцій, некоректними типами, порушенням вкладеності або появою неочікуваних властивостей. Третю групу формують Pydantic-помилки, які виникають після синтаксичного розбору та структурної перевірки, але свідчать про порушення

прикладних правил моделі даних MVP. Каскад валідації не виправляє помилки модельного етапу; він переводить їх у контрольовані машинні статуси, що блокують автоматичне приймання непридатного результату [18], [19], [33], [53].

4.7.2 Аналіз помилок інформаційної недостатності, конфліктів і непідтверджених значень

Окремий клас ризиків пов’язаний з інформаційною недостатністю, неоднозначністю або суперечливістю джерельного контексту. У таких умовах кандидатний JSON-об’єкт може залишатися синтаксично й структурно придатним, але окремі значення не мають достатньої джерельної основи. Для зменшення цього ризику застосовується політика недостатності даних: якщо параметр має status="insufficient_data", поле value повинно бути null; для висновку з таким самим статусом поле result також має бути null. Порушення цієї відповідності відхиляється на рівні Pydantic. Конфліктні сценарії створюють іншу проблему: модель може сформувати валідний об’єкт і source_anchor, але не забезпечити source_excerpt, який проходить сувору перевірку. Тому дефіцитні поля, конфліктні значення та непідтверджені source_excerpt аналізуються окремо від помилок JSON-форми. Програмний рівень фіксує ці ризики, але не замінює експертну оцінку складного технічного документа.

4.7.3 Обмеження суворого збігу фрагмента джерела та проблема джерельної простежуваності

Найбільш вираженим обмеженням експериментальної перевірки стала нестабільність суворого дослівного збігу source_excerpt із джерельним контекстом. Наявність source_anchor фіксує структурну спробу прив’язки результату до фрагмента, але не гарантує, що текст витягу буде дослівно знайдений у джерелі. Значення source_excerpt_found_in_input=true означає точну наявність заявленого фрагмента до джерельного тексту. Значення false означає

непроходження цього критерію й не прирівнюється автоматично до змістової хибності факту. Невідповідність може бути пов’язана з мікроперефразуванням, зміною пунктуації, пробілів або регістру, обрізанням фрагмента, вибором узагальненого формулювання чи наслідками нормалізації й лінеаризації таблиць. У межах MVP не застосовувалися нечітке зіставлення, відстань Левенштейна, токенне перекриття або семантична перевірка джерела. Це робить критерій source_excerpt контрольованим, але чутливим до дрібних текстових відхилень [43], [44].

4.7.4 Ресурсні обмеження локального виконання та межі масштабування

Ресурсні межі системи визначаються локальним середовищем MacBook Air M1 з 8 ГБ уніфікованої пам’яті, Ollama, фіксованими параметрами контексту та послідовним виконанням ресурсоємних етапів. Статуси HTTP 500, timeout, empty output і json_parse_error фіксуються як експлуатаційні показники конкретного локального запуску, а не як універсальні властивості моделей. Без окремих низькорівневих логів такі відмови не подаються як доведений наслідок дефіциту пам’яті або іншої конкретної апаратної причини. Додатковими межами є num_ctx=4096, max_context_chars=6000 і max_context_chunks=6, які зменшують навантаження, але обмежують обсяг контексту, доступного моделі. Функціональні межі пов’язані з відсутністю оптичного розпізнавання символів, підтримкою лише цифрових або підготовлених джерел, PDF тільки з текстовим шаром і спрощеною рядковою лінеаризацією таблиць. Ці межі не дають підстав переносити результати на промислове масштабування без окремого проєктування й тестування.

4.8 Висновки за розділом 4

Експериментальна перевірка підтвердила працездатність реалізованого комплексу програмних засобів у межах локального MVP, визначеного тестового набору та зафіксованих конфігурацій. Перевірка охопила первинний відбір моделей, приймальні інфраструктурні тести, первинні перевірки, контрольовані прогони, офлайн-перевірку кешу векторних представлень і наскрізні запуски на ускладнених сценаріях. У межах перевірених сценаріїв система виконувала шлях від підготовленого джерела до кандидатного JSON-об’єкта, його програмної валідації, джерельної перевірки та формування звітних артефактів.

Детерміновані інфраструктурні компоненти конвеєра пройшли приймальні перевірки: pytest завершився як 62 успішні перевірки та 3 попередження, синтетичні контрольовані прогони — 20 успішно пройдено, без відмов, форматні контрольовані прогони — 15 успішно пройдено, без відмов, а первинні перевірки повернули очікувані коди завершення. У третій фазі llama3.2:3b пройшла синтаксичний розбір JSON, JSON Schema і Pydantic у 5/5 ускладнених запусків, тоді як gemma4:e2b мала 2/5 структурно валідних запусків, два HTTP 500 і один json_parse_error. Це обґрунтовує використання llama3.2:3b як базової конфігурації саме в межах поточного MVP.

Основним підтвердженим обмеженням залишилося нестабільне проходження верифікації поля source_excerpt. Атрибут source_anchor формується стабільніше, але не є еквівалентом точного джерельного збігу. Додаткові межі пов’язані з відсутністю оптичного розпізнавання символів, спрощеною лінеаризацією таблиць, відсутністю реконструкції табличної геометрії та ресурсними обмеженнями локального середовища. Подальший розвиток потребує окремої реалізації та перевірки розширеної джерельної валідації, обробки сканованих документів, складніших таблиць, ширшого корпусу вхідних даних і промислового журналювання.

ВИСНОВКИ

З метою покращення обробки табличних даних за рахунок використання великих мовних моделей вирішено практичне завдання створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей. У ході роботи показано, що автономний виклик великої мовної моделі не є достатнім для прийняття технічного результату, оскільки відповідь моделі може бути синтаксично прийнятною, але структурно неповною, джерельно непідтвердженою або непридатною для подальшого машинного контролю. Тому основою розробки став локальний програмний конвеєр, у якому мовна модель виконує роль керованого компонента формування кандидатного JSON-об’єкта, а прийнятність результату визначається зовнішнім програмним контуром.

У першому розділі проаналізовано предметну область автоматичного формування структурованих документів, особливості табличних і слабкоформалізованих джерел, а також обмеження традиційної обробки документних даних і автономного використання великих мовних моделей. Встановлено, що для технічних і табличних документів важливо зберігати зв’язок між значеннями, назвами полів, одиницями вимірювання, табличними рядками, фрагментами джерела та службовими метаданими. Це обґрунтувало потребу у контрольованій системі, яка не лише формує структурований результат, а й забезпечує його перевірюваність, джерельну простежуваність і фіксацію станів обробки.

У другому розділі обґрунтовано вибір локальних мовних моделей, середовища виконання й технологічного стеку MVP. Визначено, що система має працювати в межах локального середовища MacBook Air M1 з 8 ГБ уніфікованої пам’яті, використовувати компактні локальні моделі через Ollama, Python-рівень для оркестрації, FAISS для локального векторного пошуку, резервний лексичний пошук для технічних маркерів, JSON Schema і Pydantic для формального та прикладного контролю. Спроєктовано архітектуру конвеєра, яка охоплює

приймання документних джерел, вилучення текстових і табличних даних, рядкову лінеаризацію таблиць, нормалізацію проміжного подання, фрагментацію з метаданими, добір джерельного контексту, побудову запиту, локальний інференс, каскадну валідацію, верифікацію source_excerpt, журналювання і звітність.

У третьому розділі описано програмну реалізацію системи як локального Python-проєкту з відокремленими зонами коду, конфігурацій, схем, шаблонів запитів, тестових матеріалів і звітних артефактів. Реалізовано підтримку цифрових або попередньо підготовлених джерел без оптичного розпізнавання символів у форматах TXT, CSV, XLSX, DOCX і PDF з текстовим шаром. Вхідний рівень системи виконує вилучення текстових і табличних даних, рядкову лінеаризацію таблиць, нормалізацію проміжного подання та формування фрагментів із метаданими. Шар добору контексту реалізовано через локальні векторні представлення, FAISS IndexFlatIP, резервний лексичний пошук, об’єднання, дедуплікацію та обмеження контекстного бюджету. Взаємодія з великою мовною моделлю виконується через локальне середовище Ollama, а результат моделі передається на програмну перевірку як кандидатний JSON-об’єкт.

У четвертому розділі проведено тестування та експериментальну перевірку реалізованого MVP. Перевірка охопила модульні тести, синтетичні й форматні контрольовані прогони, початкові верифікації, офлайн-перевірку кешу моделі векторних представлень, порівняльний аналіз локальних моделей і запуски на ускладнених сценаріях. Фінальна перевірка програмної бази зафіксувала 62 успішні pytest-тести за наявності 3 попереджень, 20 успішних синтетичних контрольованих прогонів без відмов, 15 успішних форматних контрольованих прогонів без відмов, а також очікувані коди завершення первинних перевірок для валідного, некоректного та джерельно невідповідного випадків. У порівняльних запусках модель llama3.2:3b стабільніше проходила синтаксичний розбір JSON, перевірку за JSON Schema та Pydantic-валідацію в межах заданого локального MVP, тоді як суворий дослівний збіг поля source_excerpt залишився основним обмеженням джерельної простежуваності.

Практичний результат роботи полягає у створенні локального прототипу програмної системи, що демонструє контрольований підхід до автоматичної генерації структурованих документних результатів за участю великої мовної моделі. Система не формує готовий DOCX/PDF-документ у межах поточної реалізації, а створює валідований JSON-об’єкт із джерельними прив’язками, службовими статусами, результатами перевірок, метриками та звітними артефактами. Такий результат може бути використаний як основа для подальшого формування готового документа, інтеграції з іншими інформаційними системами або розширення механізмів контролю джерельної відповідності.

Межі застосування розробленого MVP визначаються підтримкою лише цифрових або підготовлених документних джерел без оптичного розпізнавання символів, відсутністю повної реконструкції складної табличної сітки, об’єднаних комірок і геометрії сторінки, а також локальним характером виконання без графічного інтерфейсу користувача, бази даних, багатокористувацького режиму та промислового експорту документів. Подальший розвиток системи доцільно спрямувати на розширення корпусу тестових документів, удосконалення перевірки джерельних фрагментів, підтримку складніших таблиць, реалізацію експорту готових документних файлів і створення інтерфейсу користувача як окремого етапу після стабілізації MVP.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Cui L., Xu Y., Lv T., Wei F. Document AI : Benchmarks, Models and Applications. arXiv:2111.08609. 2021. DOI: https://doi.org/10.48550/arXiv.2111.08609. URL: https://arxiv.org/abs/2111.08609 (дата звернення: 10.05.2026).

2. Šimsa Š., Šulc M., Uřičář M. et al. DocILE Benchmark for Document Information Localization and Extraction. Document Analysis and Recognition – ICDAR 2023 : 17th International Conference, San José, CA, USA, August 21–26, 2023, Proceedings, Part II. Lecture Notes in Computer Science. Cham : Springer, 2023. Vol. 14188. P. 147–166. DOI: https://doi.org/10.1007/978-3-031-41679-8_9. URL: https://arxiv.org/abs/2302.05658 (дата звернення: 10.05.2026).

3. Lee C.-Y., Li C.-L., Dozat T. et al. FormNet : Structural Encoding beyond Sequential Modeling in Form Document Information Extraction. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics. Dublin : Association for Computational Linguistics, 2022. P. 3735–3754. DOI: https://doi.org/10.18653/v1/2022.acl-long.260. URL: https://aclanthology.org/2022.acl-long.260/ (дата звернення: 10.05.2026).

4. Gao T., Yen H., Yu J., Chen D. Enabling Large Language Models to Generate Text with Citations. Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Singapore : Association for Computational Linguistics, 2023. P. 6465–6488. DOI: https://doi.org/10.18653/v1/2023.emnlp-main.398. URL: https://aclanthology.org/2023.emnlp-main.398/ (дата звернення: 10.05.2026).

5. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need. Advances in Neural Information Processing Systems. 2017. Vol. 30. URL: https://arxiv.org/abs/1706.03762 (дата звернення: 10.05.2026).

6. Liu N. F., Lin K., Hewitt J. et al. Lost in the Middle : How Language Models Use Long Contexts. Transactions of the Association for Computational Linguistics. 2024. Vol. 12. P. 157–173. DOI: https://doi.org/10.1162/tacl_a_00638. URL: https://aclanthology.org/2024.tacl-1.9/ (дата звернення: 10.05.2026).

7. Maynez J., Narayan S., Bohnet B., McDonald R. On Faithfulness and Factuality in Abstractive Summarization. Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, 2020. P. 1906–1919. DOI: https://doi.org/10.18653/v1/2020.acl-main.173. URL: https://aclanthology.org/2020.acl-main.173/ (дата звернення: 10.05.2026).

8. Ji Z., Lee N., Frieske R. et al. Survey of Hallucination in Natural Language Generation. ACM Computing Surveys. 2023. Vol. 55, no. 12. Article 248. P. 1–38. DOI: https://doi.org/10.1145/3571730. URL: https://arxiv.org/abs/2202.03629 (дата звернення: 10.05.2026).

9. Sui Y., Zhou M., Zhou M., Han S., Zhang D. Table Meets LLM : Can Large Language Models Understand Structured Table Data? A Benchmark and Empirical Study. Proceedings of the ACM International Conference on Web Search and Data Mining. New York : Association for Computing Machinery, 2024. P. 645–654. DOI: https://doi.org/10.1145/3616855.3635752. URL: https://arxiv.org/abs/2305.13062 (дата звернення: 10.05.2026).

10. Geng S., Cooper H., Moskal M. et al. JSONSchemaBench : A Rigorous Benchmark of Structured Outputs for Language Models. arXiv:2501.10868. 2025. DOI: https://doi.org/10.48550/arXiv.2501.10868.
URL: https://arxiv.org/abs/2501.10868 (дата звернення: 10.05.2026).

11. Geng S., Josifoski M., Peyrard M., West R. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Singapore : Association for Computational Linguistics, 2023. P. 10932–10952. DOI: https://doi.org/10.18653/v1/2023.emnlp-main.674. URL: https://aclanthology.org/2023.emnlp-main.674/ (дата звернення: 10.05.2026).

12. Takabi H., Joshi J. B. D., Ahn G.-J. Security and Privacy Challenges in Cloud Computing Environments. IEEE Security & Privacy. 2010. Vol. 8, no. 6. P. 24–31. DOI: https://doi.org/10.1109/MSP.2010.186. URL: https://doi.org/10.1109/MSP.2010.186 (дата звернення: 10.05.2026).

13. 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. URL: https://arxiv.org/abs/2005.11401 (дата звернення: 10.05.2026).

14. Gao Y., Xiong Y., Gao X. et al. Retrieval-Augmented Generation for Large Language Models : A Survey. arXiv:2312.10997. 2023. DOI: https://doi.org/10.48550/arXiv.2312.10997.
URL: https://arxiv.org/abs/2312.10997 (дата звернення: 10.05.2026).

15. Barnett S., Kurniawan S., Thudumu S., Brannelly Z., Abdelrazek M. Seven Failure Points When Engineering a Retrieval Augmented Generation System. Proceedings of the 2024 IEEE/ACM International Conference on AI Engineering – Software Engineering for AI. New York : Association for Computing Machinery, 2024. P. 194–199. DOI: https://doi.org/10.1145/3644815.3644945. URL: https://arxiv.org/abs/2401.05856 (дата звернення: 10.05.2026).

16. Es S., James J., Espinosa-Anke L., Schockaert S. RAGAS : Automated Evaluation of Retrieval Augmented Generation. Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics : System Demonstrations. St. Julian’s : Association for Computational Linguistics, 2024. P. 150–158. DOI: https://doi.org/10.18653/v1/2024.eacl-demo.16. URL: https://aclanthology.org/2024.eacl-demo.16/ (дата звернення: 10.05.2026).

17. Smock B., Pesala R., Abraham R. PubTables-1M : Towards Comprehensive Table Extraction from Unstructured Documents. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022. P. 4634–4642. DOI: https://doi.org/10.1109/CVPR52688.2022.00459. URL: https://openaccess.thecvf.com/content/CVPR2022/html/Smock_PubTables-1M_Towards_Comprehensive_Table_Extraction_From_Unstructured_Documents_CVPR_2022_paper.html (дата звернення: 10.05.2026).

18. Wright A., Andrews H., Hutton B. JSON Schema Validation : A Vocabulary for Structural Validation of JSON. JSON Schema Draft 2020-12. 2022. URL: https://json-schema.org/draft/2020-12/json-schema-validation (дата звернення: 10.05.2026).

19. Pydantic Documentation. Pydantic. URL: https://docs.pydantic.dev/latest/ (дата звернення: 10.05.2026).

20. Apache Tika. Apache Software Foundation. URL: https://tika.apache.org/ (дата звернення: 10.05.2026).

21. Form Parser. Google Cloud Document AI. URL: https://cloud.google.com/document-ai/docs/form-parser (дата звернення: 10.05.2026).

22. Amazon Textract. Tables and document response. Amazon Web Services. URL: https://docs.aws.amazon.com/textract/latest/dg/how-it-works-tables.html; URL: https://docs.aws.amazon.com/textract/latest/dg/document-response.html (дата звернення: 10.05.2026).

23. Azure AI Document Intelligence. Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/ (дата звернення: 10.05.2026).

24. LangChain RAG and Structured Output. LangChain Documentation. URL: https://docs.langchain.com/oss/python/langchain/rag; URL: https://docs.langchain.com/oss/python/langchain/structured-output (дата звернення: 10.05.2026).

25. LlamaIndex Documentation. LlamaIndex. URL: https://developers.llamaindex.ai/python/framework/; URL: https://developers.llamaindex.ai/python/framework/module_guides/loading/documents_and_nodes/ (дата звернення: 10.05.2026).

26. Haystack Documentation. deepset. URL: https://docs.haystack.deepset.ai/docs/intro; URL: https://docs.haystack.deepset.ai/reference/validators-api (дата звернення: 10.05.2026).

27. Ollama Structured Outputs. Ollama Documentation. URL: https://docs.ollama.com/capabilities/structured-outputs (дата звернення: 10.05.2026).

28. Touvron H., Lavril T., Izacard G. et al. LLaMA : Open and Efficient Foundation Language Models. arXiv:2302.13971. 2023. DOI: https://doi.org/10.48550/arXiv.2302.13971.
URL: https://arxiv.org/abs/2302.13971 (дата звернення: 10.05.2026).

29. FAQ. Ollama Documentation. URL: https://ollama.readthedocs.io/en/faq/ (дата звернення: 10.05.2026).

30. Ollama API Reference. Ollama Documentation. URL: https://ollama.readthedocs.io/en/api/ (дата звернення: 10.05.2026).

31. llama.cpp. ggml-org. GitHub. URL: https://github.com/ggml-org/llama.cpp (дата звернення: 10.05.2026).

32. LM Studio as a Local LLM API Server. LM Studio Documentation. URL: https://lmstudio.ai/docs/developer/core/server (дата звернення: 10.05.2026).

33. Bray T. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. RFC Editor, 2017. DOI: https://doi.org/10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259 (дата звернення: 10.05.2026).

34. Text. PyMuPDF Documentation. URL: https://pymupdf.readthedocs.io/en/latest/recipes-text.html (дата звернення: 10.05.2026).

35. python-docx documentation. python-docx Documentation. URL: https://python-docx.readthedocs.io/en/latest/ (дата звернення: 10.05.2026).

36. Package overview. pandas Documentation. URL: https://pandas.pydata.org/docs/getting_started/overview.html (дата звернення: 10.05.2026).

37. openpyxl documentation. openpyxl Documentation. URL: https://openpyxl.readthedocs.io/en/stable/ (дата звернення: 10.05.2026).

38. Semantic Search. Sentence Transformers Documentation. URL: https://www.sbert.net/examples/sentence_transformer/applications/semantic-search/README.html (дата звернення: 10.05.2026).

39. Reimers N., Gurevych I. Sentence-BERT : Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods

in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. Hong Kong : Association for Computational Linguistics, 2019. P. 3982–3992. DOI: https://doi.org/10.18653/v1/D19-1410. URL: https://aclanthology.org/D19-1410/ (дата звернення: 10.05.2026).

40. FAISS Documentation. Meta / FAISS. URL: https://faiss.ai/index.html (дата звернення: 10.05.2026).

41. Thakur N., Reimers N., Rücklé A. et al. BEIR : A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. arXiv:2104.08663. 2021. DOI: https://doi.org/10.48550/arXiv.2104.08663. URL: https://arxiv.org/abs/2104.08663 (дата звернення: 10.05.2026).

42. Lin S.-C., Lin J. A Dense Representation Framework for Lexical and Semantic Matching. arXiv:2206.09912. 2022. DOI: https://doi.org/10.48550/arXiv.2206.09912.
URL: https://arxiv.org/abs/2206.09912 (дата звернення: 10.05.2026).

43. Qi J., Sarti G., Fernández R. Model Internals-based Answer Attribution for Trustworthy Retrieval-Augmented Generation. Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2024. P. 6037–6053. DOI: https://doi.org/10.18653/v1/2024.emnlp-main.347. URL: https://aclanthology.org/2024.emnlp-main.347/ (дата звернення: 10.05.2026).

44. Wallat J., Heuss M., de Rijke M. Correctness is not Faithfulness in RAG Attributions. arXiv:2412.18004. 2024. DOI: https://doi.org/10.48550/arXiv.2412.18004. URL: https://arxiv.org/abs/2412.18004 (дата звернення: 10.05.2026).

45. YAML Ain’t Markup Language version 1.2, revision 1.2.2. YAML Language Development Team. 2021. URL: https://yaml.org/spec/1.2.2/ (дата звернення: 10.05.2026).

46. JSON Lines. jsonlines.org. URL: https://jsonlines.org/ (дата звернення: 10.05.2026).

47. Shafranovich Y. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. RFC Editor, 2005.

DOI: https://doi.org/10.17487/RFC4180. URL: https://www.rfc-editor.org/info/rfc4180 (дата звернення: 10.05.2026).

48. MacFarlane J. CommonMark Spec, version 0.31.2. CommonMark. 2024. URL: https://spec.commonmark.org/0.31.2/ (дата звернення: 10.05.2026).

49. What is Python? Executive Summary. Python Software Foundation. URL: https://www.python.org/doc/essays/blurb/ (дата звернення: 10.05.2026).

50. Requests Quickstart. Requests Documentation. URL: https://requests.readthedocs.io/en/latest/user/quickstart/ (дата звернення: 10.05.2026).

51. venv — Creation of virtual environments. Python Documentation. URL: https://docs.python.org/3/library/venv.html (дата звернення: 10.05.2026).

52. Requirements File Format. pip Documentation / Python Packaging Authority. URL: https://pip.pypa.io/en/stable/reference/requirements-file-format/ (дата звернення: 10.05.2026).

53. jsonschema documentation. jsonschema project. URL: https://python-jsonschema.readthedocs.io/en/stable/ (дата звернення: 10.05.2026).

54. pytest documentation. pytest. URL: https://docs.pytest.org/en/stable/ (дата звернення: 10.05.2026).

55. Pipes and Filters pattern. Microsoft Azure Architecture Center. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters (дата звернення: 10.05.2026).

56. Pineau J., Vincent-Lamarre P., Sinha K. et al. Improving Reproducibility in Machine Learning Research : A Report from the NeurIPS 2019 Reproducibility Program. Journal of Machine Learning Research. 2021. Vol. 22, no. 164. P. 1–20. URL: https://jmlr.org/papers/v22/20-303.html (дата звернення: 10.05.2026).

57. Artifact Review and Badging – Current. Association for Computing Machinery. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (дата звернення: 10.05.2026).

ДОДАТОК А
Текст програми

А.1 Текст файлу src/docgen/ingestion/loaders.py

```
from __future__ import annotations

import csv
from pathlib import Path
from typing import Any

from src.docgen.preprocessing.normalization import normalize_text
from src.docgen.preprocessing.table_linearization import linearize_table_row


SUPPORTED_EXTENSIONS = {".txt", ".csv", ".xlsx", ".pdf", ".docx"}


def load_document(path: str | Path) -> list[dict[str, Any]]:
    source_path = Path(path)
    if not source_path.exists():
        raise FileNotFoundError(f"Input document not found: {source_path}")

    suffix = source_path.suffix.lower()
    if suffix not in SUPPORTED_EXTENSIONS:
        raise ValueError(f"Unsupported input extension: {suffix or '<none>'}")

    if suffix == ".txt":
        return _load_txt(source_path)
    if suffix == ".csv":
        return _load_csv(source_path)
    if suffix == ".xlsx":
        return _load_xlsx(source_path)
    if suffix == ".pdf":
        return _load_pdf(source_path)
    if suffix == ".docx":
        return _load_docx(source_path)

    raise ValueError(f"Unsupported input extension: {suffix or '<none>'}")


def _base_record(
    source_path: Path,
    content: str,
    content_type: str,
    metadata: dict[str, Any] | None = None,
) -> dict[str, Any]:
    return {
```

```
        "document_id": source_path.stem,
        "source_path": str(source_path),
        "source_name": source_path.name,
        "source_type": source_path.suffix.lower().lstrip("."),
        "content": content,
        "content_type": content_type,
        "metadata": metadata or {},
    }


def _read_text_with_fallback(source_path: Path) -> str:
    try:
        return source_path.read_text(encoding="utf-8")
    except UnicodeDecodeError:
        return source_path.read_text()


def _load_txt(source_path: Path) -> list[dict[str, Any]]:
    content = normalize_text(_read_text_with_fallback(source_path))
    if not content:
        return []

    content_type = "synthetic" if "synthetic" in source_path.parts else "text"
    return [
        _base_record(
            source_path=source_path,
            content=content,
            content_type=content_type,
            metadata={"source_type": "txt"},
        )
    ]


def _load_csv(source_path: Path) -> list[dict[str, Any]]:
    try:
        import pandas as pd  # type: ignore[import-not-found]
    except ImportError:
        return _load_csv_with_stdlib(source_path)

    dataframe = pd.read_csv(source_path)
    column_names = [str(column) for column in dataframe.columns]
    records: list[dict[str, Any]] = []

    for row_index, row in dataframe.iterrows():
        row_values = {
```

```
            column: _empty_to_none(row[column])
            for column in column_names
        }
        content = linearize_table_row(row_values)
        if not content:
            continue
        records.append(
            _base_record(
                source_path=source_path,
                content=content,
                content_type="table",
                metadata={
                    "source_type": "csv",
                    "row_index": int(row_index),
                    "column_names": column_names,
                },
            )
        )

    return records


def _load_csv_with_stdlib(source_path: Path) -> list[dict[str, Any]]:
    records: list[dict[str, Any]] = []
    with source_path.open("r", encoding="utf-8", newline="") as file:
        reader = csv.DictReader(file)
        column_names = list(reader.fieldnames or [])

        for row_index, row in enumerate(reader):
            content = linearize_table_row(row)
            if not content:
                continue
            records.append(
                _base_record(
                    source_path=source_path,
                    content=content,
                    content_type="table",
                    metadata={
                        "source_type": "csv",
                        "row_index": row_index,
                        "column_names": column_names,
                    },
                )
            )
```

```
    return records


def _load_xlsx(source_path: Path) -> list[dict[str, Any]]:
    try:
        import pandas as pd  # type: ignore[import-not-found]
    except ImportError as exc:
        raise RuntimeError(
            "XLSX ingestion requires pandas and openpyxl to be installed."
        ) from exc

    try:
        sheets = pd.read_excel(source_path, sheet_name=None, engine="openpyxl")
    except ImportError as exc:
        raise RuntimeError(
            "XLSX ingestion requires pandas and openpyxl to be installed."
        ) from exc

    records: list[dict[str, Any]] = []
    for sheet_name, dataframe in sheets.items():
        column_names = [str(column) for column in dataframe.columns]

        for row_index, row in dataframe.iterrows():
            row_values = {
                column: _empty_to_none(row[column])
                for column in column_names
            }
            content = linearize_table_row(row_values)
            if not content:
                continue
            records.append(
                _base_record(
                    source_path=source_path,
                    content=content,
                    content_type="table",
                    metadata={
                        "source_type": "xlsx",
                        "sheet_name": str(sheet_name),
                        "row_index": int(row_index),
                        "column_names": column_names,
                    },
                )
            )

    return records
```

```
def _load_pdf(source_path: Path) -> list[dict[str, Any]]:
    try:
        import fitz  # type: ignore[import-not-found]
    except ImportError as exc:
        raise RuntimeError(
            "PDF ingestion requires PyMuPDF to be installed. OCR is not supported."
        ) from exc

    records: list[dict[str, Any]] = []
    with fitz.open(source_path) as document:
        for page_index, page in enumerate(document, start=1):
            content = normalize_text(page.get_text("text"))
            if not content:
                continue
            records.append(
                _base_record(
                    source_path=source_path,
                    content=content,
                    content_type="text",
                    metadata={
                        "source_type": "pdf",
                        "page": page_index,
                        "ocr": False,
                    },
                )
            )

    return records


def _load_docx(source_path: Path) -> list[dict[str, Any]]:
    try:
        from docx import Document  # type: ignore[import-not-found]
    except ImportError as exc:
        raise RuntimeError(
            "DOCX ingestion requires python-docx to be installed."
        ) from exc

    document = Document(source_path)
    records: list[dict[str, Any]] = []

    for paragraph_index, paragraph in enumerate(document.paragraphs):
        content = normalize_text(paragraph.text)
```

```
        if not content:
            continue
        records.append(
            _base_record(
                source_path=source_path,
                content=content,
                content_type="text",
                metadata={
                    "source_type": "docx",
                    "paragraph_index": paragraph_index,
                },
            )
        )

    for table_index, table in enumerate(document.tables):
        for row_index, row in enumerate(table.rows):
            values = {
                f"column_{column_index + 1}": cell.text
                for column_index, cell in enumerate(row.cells)
            }
            content = linearize_table_row(values)
            if not content:
                continue
            records.append(
                _base_record(
                    source_path=source_path,
                    content=content,
                    content_type="table",
                    metadata={
                        "source_type": "docx",
                        "table_index": table_index,
                        "row_index": row_index,
                        "column_names": list(values.keys()),
                    },
                )
            )

    return records


def _empty_to_none(value: Any) -> Any:
    try:
        import pandas as pd  # type: ignore[import-not-found]
    except ImportError:
        pd = None
```

```
    if pd is not None and pd.isna(value):
        return None
    return value
```

А.2 Текст файлу src/docgen/preprocessing/normalization.py

```
from __future__ import annotations

import re


def normalize_text(text: object) -> str:
    if text is None:
        return ""

    normalized = str(text).replace("\ufeff", " ")
    normalized = normalized.replace("\xa0", " ")
    normalized = re.sub(r"\s+", " ", normalized)
    return normalized.strip()
```

А.3 Текст файлу src/docgen/preprocessing/table_linearization.py

```
from __future__ import annotations

from collections.abc import Mapping
from typing import Any

from src.docgen.preprocessing.normalization import normalize_text


def linearize_table_row(row: Mapping[str, Any]) -> str:
    parts: list[str] = []

    for column_name, value in row.items():
        column = normalize_text(column_name)
        cell = normalize_text(value)
        if not column or not cell:
            continue
        parts.append(f"{column}: {cell}")

    return "; ".join(parts)
```

А.4 Текст файлу src/docgen/chunking/chunk_builder.py

```
from __future__ import annotations
```

```
from pathlib import Path
from typing import Any

import yaml


DEFAULT_CHUNK_SIZE_CHARS = 600
DEFAULT_CHUNK_OVERLAP_CHARS = 80
DEFAULT_MIN_CHUNK_CHARS = 80
DEFAULT_CONFIG_PATH = Path("configs/retrieval.yaml")


def build_chunks(
    records: list[dict[str, Any]],
    chunk_size_chars: int = DEFAULT_CHUNK_SIZE_CHARS,
    chunk_overlap_chars: int = DEFAULT_CHUNK_OVERLAP_CHARS,
    min_chunk_chars: int = DEFAULT_MIN_CHUNK_CHARS,
) -> list[dict[str, Any]]:
    _validate_chunking_params(
        chunk_size_chars=chunk_size_chars,
        chunk_overlap_chars=chunk_overlap_chars,
        min_chunk_chars=min_chunk_chars,
    )

    chunks: list[dict[str, Any]] = []

    for record_index, record in enumerate(records):
        content = str(record.get("content") or "")
        if not content.strip():
            continue

        content_type = record.get("content_type") or "text"
        if content_type == "table" and len(content) <= chunk_size_chars:
            chunk_index = len(chunks)
            chunks.append(
                _make_chunk(
                    record=record,
                    record_index=record_index,
                    chunk_index=chunk_index,
                    content=content,
                    char_start=0,
                    char_end=len(content),
                    chunk_size_chars=chunk_size_chars,
                    chunk_overlap_chars=chunk_overlap_chars,
```

```
                chunking_strategy="table_row",
            )
        )
        continue

    spans = _split_text_spans(
        content=content,
        chunk_size_chars=chunk_size_chars,
        chunk_overlap_chars=chunk_overlap_chars,
        min_chunk_chars=min_chunk_chars,
    )

    for char_start, char_end in spans:
        chunk_content = content[char_start:char_end].strip()
        if not chunk_content:
            continue
        chunk_index = len(chunks)
        chunks.append(
            _make_chunk(
                record=record,
                record_index=record_index,
                chunk_index=chunk_index,
                content=chunk_content,
                char_start=char_start,
                char_end=char_end,
                chunk_size_chars=chunk_size_chars,
                chunk_overlap_chars=chunk_overlap_chars,
                chunking_strategy="char_window",
            )
        )

return chunks


def load_chunking_config(
    config_path: str | Path = DEFAULT_CONFIG_PATH,
) -> dict[str, int]:
    path = Path(config_path)
    with path.open("r", encoding="utf-8") as file:
        data = yaml.safe_load(file) or {}

    chunking = data.get("chunking") or {}
    return {
        "chunk_size_chars": int(
            chunking.get("chunk_size_chars", DEFAULT_CHUNK_SIZE_CHARS)
```

```
        ),
        "chunk_overlap_chars": int(
            chunking.get("chunk_overlap_chars",
DEFAULT_CHUNK_OVERLAP_CHARS)
        ),
        "min_chunk_chars": int(
            chunking.get("min_chunk_chars", DEFAULT_MIN_CHUNK_CHARS)
        ),
    }


def _make_chunk(
    record: dict[str, Any],
    record_index: int,
    chunk_index: int,
    content: str,
    char_start: int,
    char_end: int,
    chunk_size_chars: int,
    chunk_overlap_chars: int,
    chunking_strategy: str,
) -> dict[str, Any]:
    document_id = str(record.get("document_id") or "document")
    metadata = dict(record.get("metadata") or {})
    metadata.update(
        {
            "source_record_index": record_index,
            "original_content_type": record.get("content_type"),
            "chunking_strategy": chunking_strategy,
            "chunk_size_chars": chunk_size_chars,
            "chunk_overlap_chars": chunk_overlap_chars,
        }
    )

    return {
        "document_id": document_id,
        "source_path": record.get("source_path"),
        "source_name": record.get("source_name"),
        "source_type": record.get("source_type"),
        "chunk_id": f"{document_id}:chunk:{chunk_index}",
        "chunk_index": chunk_index,
        "content": content,
        "content_type": record.get("content_type"),
        "metadata": metadata,
        "source_excerpt": content,
```

```
        "char_start": char_start,
        "char_end": char_end,
    }


def _split_text_spans(
    content: str,
    chunk_size_chars: int,
    chunk_overlap_chars: int,
    min_chunk_chars: int,
) -> list[tuple[int, int]]:
    if len(content) <= chunk_size_chars:
        return [(0, len(content))]

    spans: list[tuple[int, int]] = []
    start = 0
    content_length = len(content)

    while start < content_length:
        raw_end = min(start + chunk_size_chars, content_length)
        end = _prefer_whitespace_boundary(content, start, raw_end, min_chunk_chars)

        if end <= start:
            end = raw_end

        spans.append((start, end))

        if end >= content_length:
            break

        next_start = max(0, end - chunk_overlap_chars)
        while next_start < content_length and content[next_start].isspace():
            next_start += 1

        if next_start <= start:
            next_start = end
        start = next_start

    return spans


def _prefer_whitespace_boundary(
    content: str,
    start: int,
    raw_end: int,
```

```
    min_chunk_chars: int,
) -> int:
    if raw_end >= len(content) or content[raw_end - 1].isspace():
        return raw_end

    minimum_end = start + min_chunk_chars
    for index in range(raw_end, minimum_end, -1):
        if content[index - 1].isspace():
            return index - 1

    return raw_end


def _validate_chunking_params(
    chunk_size_chars: int,
    chunk_overlap_chars: int,
    min_chunk_chars: int,
) -> None:
    if chunk_size_chars <= 0:
        raise ValueError("chunk_size_chars must be greater than zero.")
    if chunk_overlap_chars < 0:
        raise ValueError("chunk_overlap_chars must be zero or greater.")
    if chunk_overlap_chars >= chunk_size_chars:
        raise ValueError("chunk_overlap_chars must be smaller than chunk_size_chars.")
    if min_chunk_chars <= 0:
        raise ValueError("min_chunk_chars must be greater than zero.")
```

A.5 Текст файлу src/docgen/retrieval/keyword_fallback.py

```
from __future__ import annotations

import re
from typing import Any


NUMBER_PATTERN = re.compile(r"(?<![\w])\d+(?:[.,]\d+)?(?![\w])")
STANDARD_PATTERN = re.compile(r"\b(?:iso|gost|dstu|дсту|гост)\b",
re.IGNORECASE)
IDENTIFIER_PATTERN = re.compile(
    r"\b(?=[a-za-яіїєґ0-9_/-]*[a-za-яіїєґ])"
    r"(?=[a-za-яіїєґ0-9_/-]*\d)"
    r"[a-za-яіїєґ0-9]+(?:[-/_][a-za-яіїєґ0-9]+)*\b",
    re.IGNORECASE,
)
WORD_PATTERN = re.compile(r"\b[a-za-яіїєґ]{4,}\b", re.IGNORECASE)
```

```
UNIT_PATTERN = re.compile(
    r"(?<![\w])(?:"
    r"v|в|a|а|kw|квт|w|вт|mm|мм|cm|см|m|м|kg|кг|%|°c|°c|c|c"
    r")(?![\w])",
    re.IGNORECASE,
)
NUMBER_WITH_UNIT_PATTERN = re.compile(
    r"(?<![\w])\d+(?:[.,]\d+)?\s*"
    r"(?:v|в|a|а|kw|квт|w|вт|mm|мм|cm|см|m|м|kg|кг|%|°c|°c|c|c)(?![\w])",
    re.IGNORECASE,
)


def normalize_text(text: str) -> str:
    normalized = str(text).lower()
    normalized = normalized.replace("\ufeff", " ").replace("\xa0", " ")
    normalized = re.sub(r"\s+", " ", normalized)
    return normalized.strip()


def extract_query_markers(query: str) -> dict[str, list[str]]:
    normalized_query = normalize_text(query)

    number_units                                                    =
_unique(NUMBER_WITH_UNIT_PATTERN.findall(normalized_query))
    numbers = _unique(NUMBER_PATTERN.findall(normalized_query))
    units = _unique(UNIT_PATTERN.findall(normalized_query))
    standards = _unique(STANDARD_PATTERN.findall(normalized_query))
    identifiers = _unique(IDENTIFIER_PATTERN.findall(normalized_query))
    words = _unique(WORD_PATTERN.findall(normalized_query))

    return {
        "number_units": number_units,
        "numbers": numbers,
        "units": units,
        "standards": standards,
        "identifiers": identifiers,
        "words": words,
    }


def keyword_search(
    chunks: list[dict[str, Any]],
    query: str,
```

```
    max_chunks: int = 2,
) -> list[dict[str, Any]]:
    if max_chunks <= 0:
        return []

    terms = _flatten_markers(extract_query_markers(query))
    if not terms:
        return []

    scored_results: list[dict[str, Any]] = []

    for original_index, chunk in enumerate(chunks):
        content = str(chunk.get("content") or "")
        normalized_content = normalize_text(content)
        matched_terms = [
            term for term in terms
            if _term_matches(term, normalized_content)
        ]

        if not matched_terms:
            continue

        match_count = len(matched_terms)
        result = {
            "chunk_id": chunk.get("chunk_id"),
            "content": content,
            "metadata": chunk.get("metadata") or {},
            "retrieval_type": "keyword_hint",
            "matched_terms": matched_terms,
            "keyword_score": match_count,
            "match_count": match_count,
            "_original_index": original_index,
        }
        scored_results.append(result)

    scored_results.sort(
        key=lambda result: (
            -int(result["match_count"]),
            str(result.get("chunk_id") or ""),
            int(result["_original_index"]),
        )
    )

    limited_results = scored_results[:max_chunks]
    for result in limited_results:
```

```
        result.pop("_original_index", None)
    return limited_results


def _term_matches(term: str, normalized_content: str) -> bool:
    if not term:
        return False
    return normalize_text(term) in normalized_content


def _flatten_markers(markers: dict[str, list[str]]) -> list[str]:
    ordered_terms: list[str] = []
    for key in ["number_units", "standards", "identifiers", "numbers", "units", "words"]:
        ordered_terms.extend(markers.get(key, []))
    return _unique(ordered_terms)


def _unique(values: list[str]) -> list[str]:
    seen: set[str] = set()
    unique_values: list[str] = []

    for value in values:
        normalized = normalize_text(value)
        if not normalized or normalized in seen:
            continue
        seen.add(normalized)
        unique_values.append(normalized)

    return unique_values
```

А.6 Текст файлу src/docgen/retrieval/retriever.py

```
from __future__ import annotations

from typing import Any

from src.docgen.retrieval.keyword_fallback import keyword_search


def retrieve_context(
    chunks: list[dict[str, Any]],
    query: str,
    semantic_results: list[dict[str, Any]] | None = None,
    keyword_results: list[dict[str, Any]] | None = None,
    semantic_top_k: int = 4,
```

```
    keyword_max_chunks: int = 2,
    max_context_chunks: int = 6,
    max_context_chars: int = 6000,
) -> list[dict[str, Any]]:
    semantic_items = list(semantic_results or [])[:semantic_top_k]
    keyword_items = (
        list(keyword_results)
        if keyword_results is not None
        else keyword_search(chunks, query, max_chunks=keyword_max_chunks)
    )
    keyword_items = keyword_items[:keyword_max_chunks]

    chunk_lookup = {
        chunk.get("chunk_id"): chunk
        for chunk in chunks
        if chunk.get("chunk_id")
    }

    merged: dict[str, dict[str, Any]] = {}
    ordered_ids: list[str] = []

    for result in semantic_items:
        chunk_id = result.get("chunk_id")
        if not chunk_id:
            continue
        base = _build_context_item(result, chunk_lookup.get(chunk_id))
        base["retrieval_type"] = "semantic"
        if "retrieval_score" in result:
            base["retrieval_score"] = result["retrieval_score"]
        merged[chunk_id] = base
        ordered_ids.append(chunk_id)

    for result in keyword_items:
        chunk_id = result.get("chunk_id")
        if not chunk_id:
            continue

        if chunk_id in merged:
            merged_item = merged[chunk_id]
            merged_item["retrieval_type"] = "both"
            if "retrieval_score" in result and "retrieval_score" not in merged_item:
                merged_item["retrieval_score"] = result["retrieval_score"]
            if "matched_terms" in result:
                merged_item["matched_terms"] = _deduplicate_terms(
                    list(merged_item.get("matched_terms") or [])
```

```
                + list(result.get("matched_terms") or [])
            )
            continue

        item = _build_context_item(result, chunk_lookup.get(chunk_id))
        item["retrieval_type"] = result.get("retrieval_type") or "keyword_hint"
        if "matched_terms" in result:
            item["matched_terms"] = _deduplicate_terms(
                list(result.get("matched_terms") or [])
            )
        if "retrieval_score" in result:
            item["retrieval_score"] = result["retrieval_score"]
        merged[chunk_id] = item
        ordered_ids.append(chunk_id)

    return _apply_context_budget(
        [merged[chunk_id] for chunk_id in ordered_ids],
        max_context_chunks=max_context_chunks,
        max_context_chars=max_context_chars,
    )


def _build_context_item(
    result: dict[str, Any],
    source_chunk: dict[str, Any] | None,
) -> dict[str, Any]:
    source = source_chunk or {}
    item = {
        "chunk_id": result.get("chunk_id") or source.get("chunk_id"),
        "content": result.get("content", source.get("content", "")),
        "metadata": result.get("metadata", source.get("metadata") or {}),
    }

    for key in ["document_id", "source_type", "content_type", "source_excerpt"]:
        value = result.get(key, source.get(key))
        if value is not None:
            item[key] = value

    if "matched_terms" in result:
        item["matched_terms"] = _deduplicate_terms(
            list(result.get("matched_terms") or [])
        )
    if "retrieval_score" in result:
        item["retrieval_score"] = result["retrieval_score"]
```

```
    return item


def _apply_context_budget(
    items: list[dict[str, Any]],
    max_context_chunks: int,
    max_context_chars: int,
) -> list[dict[str, Any]]:
    if max_context_chunks <= 0 or max_context_chars <= 0:
        return []

    selected: list[dict[str, Any]] = []
    used_chars = 0

    for item in items:
        content_length = len(str(item.get("content") or ""))
        if len(selected) >= max_context_chunks:
            break
        if used_chars + content_length > max_context_chars and selected:
            continue
        if used_chars + content_length > max_context_chars:
            continue
        selected.append(item)
        used_chars += content_length

    return selected


def _deduplicate_terms(terms: list[str]) -> list[str]:
    seen: set[str] = set()
    result: list[str] = []
    for term in terms:
        normalized = str(term)
        if normalized in seen:
            continue
        seen.add(normalized)
        result.append(normalized)
    return result
```

А.7 Текст файлу src/docgen/prompting/prompt_builder.py

```
from __future__ import annotations

from pathlib import Path
from typing import Any
```

```
DEFAULT_SCHEMA_PATH = Path("schemas/mvp_output.schema.json")
EMPTY_CONTEXT_MESSAGE = (
    "No retrieved source context is available. Return null values with "
    "status=\"insufficient_data\" for absent data."
)


def load_text(path: str | Path) -> str:
    return Path(path).read_text(encoding="utf-8")


def format_retrieved_context(chunks: list[dict[str, Any]]) -> str:
    if not chunks:
        return EMPTY_CONTEXT_MESSAGE

    formatted_chunks: list[str] = []
    for index, chunk in enumerate(chunks, start=1):
        lines = [
            f"[Retrieved chunk {index}]",
            f"chunk_id: {chunk.get('chunk_id')}",
        ]

        for key in ["document_id", "source_type", "content_type", "retrieval_type"]:
            value = chunk.get(key)
            if value is not None:
                lines.append(f"{key}: {value}")

        if chunk.get("retrieval_score") is not None:
            lines.append(f"retrieval_score: {chunk['retrieval_score']}")

        matched_terms = chunk.get("matched_terms")
        if matched_terms:
            lines.append(f"matched_terms: {', '.join(str(term) for term in matched_terms)}")

        metadata = chunk.get("metadata") or {}
        if metadata:
            lines.append(f"metadata: {_format_metadata(metadata)}")

        content = chunk.get("source_excerpt") or chunk.get("content") or ""
        lines.append("content:")
        lines.append(str(content))
        formatted_chunks.append("\n".join(lines))
```

```
    return "\n\n".join(formatted_chunks)


def build_prompt(
    query: str,
    retrieved_chunks: list[dict[str, Any]],
    system_prompt_path: str,
    task_prompt_path: str,
    output_example_path: str | None = None,
) -> dict[str, Any]:
    system_prompt = load_text(system_prompt_path)
    task_template = load_text(task_prompt_path)
    output_example = load_text(output_example_path) if output_example_path else ""
    schema_text = load_text(DEFAULT_SCHEMA_PATH)
    retrieved_context = format_retrieved_context(retrieved_chunks)

    source_context = (
        f"User query/task:\n{query.strip()}\n\n"
        f"Retrieved source context:\n{retrieved_context}"
    ).strip()

    task_prompt = (
        task_template
        .replace("{{ source_context }}", source_context)
        .replace("{{ json_schema }}", schema_text)
        .replace("{{ output_example }}", output_example)
    )
    full_prompt = f"{system_prompt}\n\n{task_prompt}".strip()

    return {
        "system_prompt": system_prompt,
        "task_prompt": task_prompt,
        "retrieved_context": retrieved_context,
        "full_prompt": full_prompt,
        "prompt_metadata": {
            "retrieved_chunk_ids": [
                chunk.get("chunk_id")
                for chunk in retrieved_chunks
                if chunk.get("chunk_id")
            ],
            "retrieval_modes": _unique_ordered(
                [
                    str(chunk.get("retrieval_type"))
                    for chunk in retrieved_chunks
                    if chunk.get("retrieval_type")
```

```
            ]
        ),
        "system_prompt_path": str(system_prompt_path),
        "task_prompt_path": str(task_prompt_path),
        "output_example_path": str(output_example_path)
        if output_example_path
        else None,
    },
}


def _format_metadata(metadata: dict[str, Any]) -> str:
    return "; ".join(
        f"{key}={value}"
        for key, value in sorted(metadata.items(), key=lambda item: str(item[0]))
    )


def _unique_ordered(values: list[str]) -> list[str]:
    seen: set[str] = set()
    result: list[str] = []
    for value in values:
        if value in seen:
            continue
        seen.add(value)
        result.append(value)
    return result
```

A.8 Текст файлу src/docgen/schemas/mvp_output.py

```
from __future__ import annotations

from typing import Literal

from pydantic import BaseModel, ConfigDict, Field, model_validator


Status = Literal["extracted", "insufficient_data", "ambiguous", "not_applicable"]
WarningCode = Literal[
    "missing_value",
    "ambiguous_source",
    "unsupported_value",
    "conflicting_source",
    "format_warning",
]
```

```
class StrictModel(BaseModel):
    model_config = ConfigDict(extra="forbid")


class SourceAnchor(StrictModel):
    fragment_id: str | None
    source_excerpt: str | None


class DocumentInfo(StrictModel):
    document_type: str | None
    document_title: str | None
    document_number: str | None
    document_date: str | None
    language: str | None


class Entity(StrictModel):
    entity_type: str | None
    name: str | None
    identifier: str | None
    role: str | None
    source_anchor: SourceAnchor
    validation_notes: str | None


class Parameter(StrictModel):
    field_name: str = Field(min_length=1)
    label: str | None
    value: str | int | float | bool | None
    unit: str | None
    status: Status
    source_anchor: SourceAnchor
    validation_notes: str | None

    @model_validator(mode="after")
    def validate_missing_data_policy(self) -> "Parameter":
        if self.status == "insufficient_data" and self.value is not None:
            raise ValueError(
                "Parameters with status='insufficient_data' must use value=null."
            )
        return self
```

```
class Conclusion(StrictModel):
    result: str | None
    status: Status
    source_anchor: SourceAnchor
    validation_notes: str | None

    @model_validator(mode="after")
    def validate_missing_data_policy(self) -> "Conclusion":
        if self.status == "insufficient_data" and self.result is not None:
            raise ValueError(
                "Conclusion with status='insufficient_data' must use result=null."
            )
        return self


class WarningItem(StrictModel):
    code: WarningCode
    message: str
    field_name: str | None


class ProcessingMetadata(StrictModel):
    model: str | None
    runtime: Literal["ollama"]
    timestamp: str | None


class MVPOutput(StrictModel):
    schema_version: Literal["1.0"]
    input_id: str = Field(min_length=1)
    document: DocumentInfo
    entities: list[Entity]
    parameters: list[Parameter]
    conclusion: Conclusion
    warnings: list[WarningItem]
    processing_metadata: ProcessingMetadata
```

A.9 Текст файлу src/docgen/validation/mvp_validator.py

```
from __future__ import annotations

import json
from dataclasses import dataclass
from pathlib import Path
```

```
from typing import Any

from jsonschema import Draft202012Validator
from pydantic import ValidationError

from src.docgen.schemas.mvp_output import MVPOutput


DEFAULT_SCHEMA_PATH = Path("schemas/mvp_output.schema.json")


@dataclass
class MVPValidationResult:
    json_parse_ok: bool
    schema_valid_ok: bool
    pydantic_ok: bool
    validation_error_type: str | None
    validation_error_message: str | None
    parsed_output: dict[str, Any] | None


def load_schema(schema_path: str | Path = DEFAULT_SCHEMA_PATH) -> dict[str,
Any]:
    with Path(schema_path).open("r", encoding="utf-8") as f:
        return json.load(f)


def parse_json_text(raw_text: str) -> dict[str, Any]:
    return json.loads(raw_text)


def validate_against_json_schema(
    data: dict[str, Any],
    schema: dict[str, Any],
) -> None:
    validator = Draft202012Validator(schema)
    errors = sorted(validator.iter_errors(data), key=lambda e: list(e.path))
    if errors:
        first_error = errors[0]
        path = ".".join(str(part) for part in first_error.path)
        if path:
            raise ValueError(f"{path}: {first_error.message}")
        raise ValueError(first_error.message)
```

```
def validate_mvp_output(
    raw_text: str,
    schema_path: str | Path = DEFAULT_SCHEMA_PATH,
) -> MVPValidationResult:
    try:
        parsed = parse_json_text(raw_text)
    except json.JSONDecodeError as exc:
        return MVPValidationResult(
            json_parse_ok=False,
            schema_valid_ok=False,
            pydantic_ok=False,
            validation_error_type="json_parse_error",
            validation_error_message=str(exc),
            parsed_output=None,
        )

    try:
        schema = load_schema(schema_path)
        validate_against_json_schema(parsed, schema)
    except Exception as exc:
        return MVPValidationResult(
            json_parse_ok=True,
            schema_valid_ok=False,
            pydantic_ok=False,
            validation_error_type="json_schema_error",
            validation_error_message=str(exc),
            parsed_output=parsed,
        )

    try:
        model = MVPOutput.model_validate(parsed)
    except ValidationError as exc:
        return MVPValidationResult(
            json_parse_ok=True,
            schema_valid_ok=True,
            pydantic_ok=False,
            validation_error_type="pydantic_error",
            validation_error_message=str(exc),
            parsed_output=parsed,
        )
    except Exception as exc:
        return MVPValidationResult(
            json_parse_ok=True,
            schema_valid_ok=True,
            pydantic_ok=False,
```

```
            validation_error_type="pydantic_error",
            validation_error_message=str(exc),
            parsed_output=parsed,
        )

    return MVPValidationResult(
        json_parse_ok=True,
        schema_valid_ok=True,
        pydantic_ok=True,
        validation_error_type=None,
        validation_error_message=None,
        parsed_output=model.model_dump(),
    )


def source_excerpts_found_in_input(
    parsed_output: dict[str, Any],
    source_text: str,
) -> bool:
    excerpts: list[str] = []

    for entity in parsed_output.get("entities", []):
        anchor = entity.get("source_anchor") or {}
        excerpt = anchor.get("source_excerpt")
        if excerpt:
            excerpts.append(excerpt)

    for parameter in parsed_output.get("parameters", []):
        anchor = parameter.get("source_anchor") or {}
        excerpt = anchor.get("source_excerpt")
        if excerpt:
            excerpts.append(excerpt)

    conclusion_anchor = (parsed_output.get("conclusion") or {}).get("source_anchor") or
{}
    conclusion_excerpt = conclusion_anchor.get("source_excerpt")
    if conclusion_excerpt:
        excerpts.append(conclusion_excerpt)

    return bool(excerpts) and all(excerpt in source_text for excerpt in excerpts)
```

A.10 Текст файлу scripts/run_benchmark.py

```
import argparse
import csv
```

```
import json
import sys
import time
from datetime import datetime
from pathlib import Path
from typing import Any

import requests
import yaml

PROJECT_ROOT = Path(__file__).resolve().parents[1]
if str(PROJECT_ROOT) not in sys.path:
    sys.path.insert(0, str(PROJECT_ROOT))

from src.docgen.chunking.chunk_builder import build_chunks
from src.docgen.ingestion.loaders import load_document
from src.docgen.prompting.prompt_builder import build_prompt
from src.docgen.retrieval.embeddings import SentenceTransformerEmbedder
from src.docgen.retrieval.faiss_index import FaissChunkIndex
from src.docgen.retrieval.keyword_fallback import keyword_search
from src.docgen.retrieval.retriever import retrieve_context
from src.docgen.utils.resource_monitor import take_resource_snapshot
from src.docgen.validation.mvp_validator import (
    source_excerpts_found_in_input,
    validate_mvp_output,
)


CONFIG_PATH = PROJECT_ROOT / "configs" / "models.yaml"
RETRIEVAL_CONFIG_PATH = PROJECT_ROOT / "configs" / "retrieval.yaml"
SYSTEM_PROMPT_PATH = PROJECT_ROOT / "prompts" / "system" /
"mvp_extraction_system.txt"
TASK_PROMPT_PATH = PROJECT_ROOT / "prompts" / "tasks" /
"extract_mvp_output_json.txt"
OUTPUT_EXAMPLE_PATH = PROJECT_ROOT / "prompts" / "templates" /
"mvp_output_example.json"
SCHEMA_PATH = PROJECT_ROOT / "schemas" / "mvp_output.schema.json"
INPUT_DIR = PROJECT_ROOT / "test_inputs" / "synthetic"
EXPERIMENTS_DIR = PROJECT_ROOT / "experiments"

PROMPT_VERSION = "mvp_v1"
SCHEMA_NAME = "mvp_output.schema.json"
SCHEMA_VERSION = "1.0"

CSV_FIELDNAMES = [
```

```
    "experiment_id",
    "run_id",
    "model_key",
    "model",
    "input_id",
    "input_file",
    "prompt_version",
    "schema_name",
    "schema_version",
    "retrieval_mode",
    "retrieved_chunk_ids",
    "retrieval_types",
    "retrieval_scores",
    "keyword_fallback_used",
    "prompt_metadata",
    "temperature",
    "num_ctx",
    "elapsed_sec",
    "tokens_per_second",
    "total_duration_ns",
    "load_duration_ns",
    "prompt_eval_count",
    "prompt_eval_duration_ns",
    "eval_count",
    "eval_duration_ns",
    "output_char_count",
    "json_parse_ok",
    "schema_valid_ok",
    "pydantic_ok",
    "source_anchor_present_ok",
    "source_excerpt_found_in_input",
    "validation_error_type",
    "validation_error_message",
    "raw_output_saved_path",
]


def load_prompt(input_text: str, schema_text: str, output_example_text: str) -> str:
    system_prompt = SYSTEM_PROMPT_PATH.read_text(encoding="utf-8")
    task_prompt_template = TASK_PROMPT_PATH.read_text(encoding="utf-8")

    task_prompt = (
        task_prompt_template
        .replace("{{ source_context }}", input_text)
        .replace("{{ json_schema }}", schema_text)
```

```
        .replace("{{ output_example }}", output_example_text)
    )

    return f"{system_prompt}\n\n{task_prompt}".strip()


def load_retrieval_config() -> dict[str, Any]:
    if not RETRIEVAL_CONFIG_PATH.exists():
        return {}
    return   yaml.safe_load(RETRIEVAL_CONFIG_PATH.read_text(encoding="utf-8"))
or {}


def build_direct_prompt(
    input_text: str,
    schema_text: str,
    output_example_text: str,
) -> tuple[str, dict[str, Any]]:
    return load_prompt(
        input_text=input_text,
        schema_text=schema_text,
        output_example_text=output_example_text,
    ), {
        "retrieval_mode": "none",
        "retrieved_chunk_ids": [],
        "retrieval_types": [],
        "retrieval_scores": [],
        "keyword_fallback_used": False,
        "prompt_metadata": {},
    }


def build_retrieval_prompt(
    input_path: Path,
    input_text: str,
    requested_mode: str,
    retrieval_config: dict[str, Any],
    embedder: Any | None = None,
    index_factory: Any | None = None,
) -> tuple[str, dict[str, Any]]:
    chunking_config = retrieval_config.get("chunking") or {}
    retrieval_settings = retrieval_config.get("retrieval") or {}
    embedding_config = retrieval_config.get("embedding_model") or {}
    faiss_config = retrieval_config.get("faiss") or {}
```

```
    records = load_document(input_path)
    chunks = build_chunks(
        records,
        chunk_size_chars=int(chunking_config.get("chunk_size_chars", 600)),
        chunk_overlap_chars=int(chunking_config.get("chunk_overlap_chars", 80)),
        min_chunk_chars=int(chunking_config.get("min_chunk_chars", 80)),
    )
    if not chunks:
        raise RuntimeError(f"No retrieval chunks were built for input: {input_path}")

    keyword_max_chunks = int(retrieval_settings.get("keyword_max_chunks", 2))
    keyword_results = keyword_search(
        chunks,
        input_text,
        max_chunks=keyword_max_chunks,
    )
    semantic_top_k = int(retrieval_settings.get("semantic_top_k", 4))

    semantic_results: list[dict[str, Any]] = []
    if requested_mode == "rag_mvp":
        semantic_results = build_semantic_results(
            chunks=chunks,
            query=input_text,
            semantic_top_k=semantic_top_k,
            embedding_config=embedding_config,
            faiss_config=faiss_config,
            embedder=embedder,
            index_factory=index_factory,
        )

    context_chunks = retrieve_context(
        chunks=chunks,
        query=input_text,
        semantic_results=semantic_results,
        keyword_results=keyword_results,
        semantic_top_k=semantic_top_k,
        keyword_max_chunks=keyword_max_chunks,
        max_context_chunks=int(retrieval_settings.get("max_context_chunks", 6)),
        max_context_chars=int(retrieval_settings.get("max_context_chars", 6000)),
    )

    prompt_result = build_prompt(
        query="Extract the MVP structured output JSON from the retrieved source
context.",
        retrieved_chunks=context_chunks,
```

```
        system_prompt_path=str(SYSTEM_PROMPT_PATH),
        task_prompt_path=str(TASK_PROMPT_PATH),
        output_example_path=str(OUTPUT_EXAMPLE_PATH),
    )

    retrieval_mode = "rag_mvp" if requested_mode == "rag_mvp" else "keyword"
    prompt_metadata = prompt_result["prompt_metadata"]
    return prompt_result["full_prompt"], {
        "retrieval_mode": retrieval_mode,
        "retrieved_chunk_ids": prompt_metadata.get("retrieved_chunk_ids", []),
        "retrieval_types": prompt_metadata.get("retrieval_modes", []),
        "retrieval_scores": [
            chunk.get("retrieval_score")
            for chunk in context_chunks
            if chunk.get("retrieval_score") is not None
        ],
        "keyword_fallback_used": bool(keyword_results),
        "prompt_metadata": prompt_metadata,
    }


def build_semantic_results(
    chunks: list[dict[str, Any]],
    query: str,
    semantic_top_k: int,
    embedding_config: dict[str, Any],
    faiss_config: dict[str, Any],
    embedder: Any | None = None,
    index_factory: Any | None = None,
) -> list[dict[str, Any]]:
    if semantic_top_k <= 0:
        return []

    model_name = embedding_config.get("model_name")
    if not model_name and embedder is None:
        raise RuntimeError(
            "retrieval_mode='rag_mvp' requires embedding_model.model_name "
            "in configs/retrieval.yaml."
        )

    normalize_embeddings = bool(faiss_config.get("normalize_embeddings", True))
    active_embedder = embedder or SentenceTransformerEmbedder(
        str(model_name),
        normalize=normalize_embeddings,
        local_files_only=bool(embedding_config.get("local_files_only", True)),
```

```
        cache_folder=embedding_config.get("cache_folder"),
    )
    chunk_texts = [str(chunk.get("content") or "") for chunk in chunks]
    chunk_embeddings = active_embedder.encode_texts(chunk_texts)
    query_embedding = active_embedder.encode_texts([query])

    active_index = (
        index_factory(normalize_embeddings=normalize_embeddings)
        if index_factory is not None
        else FaissChunkIndex(normalize_embeddings=normalize_embeddings)
    )
    active_index.build(chunks, chunk_embeddings)
    return active_index.search(query_embedding, top_k=semantic_top_k)


def source_anchors_present(parsed_output: dict[str, Any] | None) -> bool | None:
    if parsed_output is None:
        return None

    for entity in parsed_output.get("entities", []):
        if not isinstance(entity.get("source_anchor"), dict):
            return False

    for parameter in parsed_output.get("parameters", []):
        if not isinstance(parameter.get("source_anchor"), dict):
            return False

    conclusion = parsed_output.get("conclusion") or {}
    if not isinstance(conclusion.get("source_anchor"), dict):
        return False

    return True


def calculate_tokens_per_second(data: dict[str, Any]) -> float | None:
    eval_count = data.get("eval_count")
    eval_duration = data.get("eval_duration")

    if not eval_count or not eval_duration:
        return None

    return eval_count / (eval_duration / 1_000_000_000)


def run_model(
```

```
    model_name: str,
    temperature: float,
    num_ctx: int,
    prompt: str,
    schema: dict[str, Any],
    source_text: str,
) -> dict[str, Any]:
    resource_before = take_resource_snapshot()
    started_at = time.time()

    response = requests.post(
        "http://localhost:11434/api/generate",
        json={
            "model": model_name,
            "prompt": prompt,
            "stream": False,
            "format": schema,
            "options": {
                "temperature": temperature,
                "num_ctx": num_ctx,
            },
        },
        timeout=600,
    )

    elapsed_sec = time.time() - started_at
    resource_after = take_resource_snapshot()
    response.raise_for_status()

    data = response.json()
    raw_output = data.get("response", "")
    validation_result = validate_mvp_output(raw_output, SCHEMA_PATH)

    source_excerpt_ok = None
    if validation_result.parsed_output is not None:
        source_excerpt_ok = source_excerpts_found_in_input(
            validation_result.parsed_output,
            source_text,
        )

    tokens_per_second = calculate_tokens_per_second(data)

    return {
        "elapsed_sec": round(elapsed_sec, 3),
        "tokens_per_second": round(tokens_per_second, 3)
```

```
        if tokens_per_second
        else None,
        "total_duration_ns": data.get("total_duration"),
        "load_duration_ns": data.get("load_duration"),
        "prompt_eval_count": data.get("prompt_eval_count"),
        "prompt_eval_duration_ns": data.get("prompt_eval_duration"),
        "eval_count": data.get("eval_count"),
        "eval_duration_ns": data.get("eval_duration"),
        "output_char_count": len(raw_output),
        "json_parse_ok": validation_result.json_parse_ok,
        "schema_valid_ok": validation_result.schema_valid_ok,
        "pydantic_ok": validation_result.pydantic_ok,
        "source_anchor_present_ok": source_anchors_present(
            validation_result.parsed_output
        ),
        "source_excerpt_found_in_input": source_excerpt_ok,
        "validation_error_type": validation_result.validation_error_type,
        "validation_error_message": validation_result.validation_error_message,
        "raw_output": raw_output,
        "ollama_metrics": {
            "total_duration": data.get("total_duration"),
            "load_duration": data.get("load_duration"),
            "prompt_eval_count": data.get("prompt_eval_count"),
            "prompt_eval_duration": data.get("prompt_eval_duration"),
            "eval_count": data.get("eval_count"),
            "eval_duration": data.get("eval_duration"),
        },
        "resource_before": resource_before,
        "resource_after": resource_after,
    }


def write_raw_output(
    raw_output_dir: Path,
    run_id: str,
    raw_output: str,
) -> str:
    raw_output_dir.mkdir(parents=True, exist_ok=True)
    output_path = raw_output_dir / f"{run_id}.json"
    output_path.write_text(raw_output, encoding="utf-8")
    return str(output_path.relative_to(PROJECT_ROOT))


def build_error_result(error: Exception) -> dict[str, Any]:
    return {
```

```
        "elapsed_sec": None,
        "tokens_per_second": None,
        "total_duration_ns": None,
        "load_duration_ns": None,
        "prompt_eval_count": None,
        "prompt_eval_duration_ns": None,
        "eval_count": None,
        "eval_duration_ns": None,
        "output_char_count": 0,
        "json_parse_ok": False,
        "schema_valid_ok": False,
        "pydantic_ok": False,
        "source_anchor_present_ok": None,
        "source_excerpt_found_in_input": None,
        "validation_error_type": type(error).__name__,
        "validation_error_message": str(error),
        "raw_output": "",
        "ollama_metrics": {},
        "resource_before": None,
        "resource_after": None,
    }


def select_models(
    config: dict[str, Any],
    requested_model_key: str | None,
) -> list[tuple[str, dict[str, Any]]]:
    models = list(config["models"].items())
    if requested_model_key is None:
        return models

    selected = [
        (model_key, model_config)
        for model_key, model_config in models
        if model_key == requested_model_key
    ]
    if not selected:
        raise ValueError(f"Model key not found in {CONFIG_PATH}: {requested_model_key}")
    return selected


def select_input_files(requested_input_file: str | None) -> list[Path]:
    if requested_input_file:
        input_path = INPUT_DIR / requested_input_file
```

```
        if not input_path.exists():
            raise FileNotFoundError(f"Input file not found: {input_path}")
        return [input_path]

    input_files = sorted(INPUT_DIR.glob("*.txt"))
    if not input_files:
        raise RuntimeError(f"No .txt files found in {INPUT_DIR}")
    return input_files

def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument("--experiment-name", default=None)
    parser.add_argument("--model-key", default=None)
    parser.add_argument("--input-file", default=None)
    parser.add_argument("--temperature", type=float, default=None)
    parser.add_argument("--num-ctx", type=int, default=None)
    parser.add_argument(
        "--retrieval-mode",
        choices=["none", "keyword", "rag_mvp"],
        default="none",
        help=(
            "Optional pre-LLM retrieval path. Default 'none' preserves the "
            "existing direct-context benchmark behavior. 'rag_mvp' uses "
            "embeddings, FAISS semantic retrieval, keyword fallback, and merge."
        ),
    )
    args = parser.parse_args()

    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    experiment_id = args.experiment_name or f"{timestamp}_model_benchmark"
    experiment_dir = EXPERIMENTS_DIR / experiment_id
    raw_output_dir = experiment_dir / "raw_outputs"
    experiment_dir.mkdir(parents=True, exist_ok=True)

    config = yaml.safe_load(CONFIG_PATH.read_text(encoding="utf-8"))
    schema_text = SCHEMA_PATH.read_text(encoding="utf-8")
    schema = json.loads(schema_text)
    output_example_text = OUTPUT_EXAMPLE_PATH.read_text(encoding="utf-8")
    retrieval_config = load_retrieval_config()

    selected_models = select_models(config, args.model_key)
    input_files = select_input_files(args.input_file)

    results_jsonl_path = experiment_dir / "results.jsonl"
    metrics_csv_path = experiment_dir / "metrics.csv"
```

```
    csv_rows = []
    run_index = 0

    for model_key, model_config in selected_models:
        model_name = model_config["name"]
        temperature = (
            args.temperature
            if args.temperature is not None
            else model_config["temperature"]
        )
        num_ctx = args.num_ctx if args.num_ctx is not None else model_config["num_ctx"]

        for input_path in input_files:
            run_index += 1
            input_id = input_path.stem
            safe_model_name = model_name.replace(":", "_").replace("/", "_")
            run_id = f"{run_index:04d}_{model_key}_{safe_model_name}_{input_id}"

            input_text = input_path.read_text(encoding="utf-8")
            if args.retrieval_mode == "none":
                prompt, retrieval_metadata = build_direct_prompt(
                    input_text=input_text,
                    schema_text=schema_text,
                    output_example_text=output_example_text,
                )
            else:
                prompt, retrieval_metadata = build_retrieval_prompt(
                    input_path=input_path,
                    input_text=input_text,
                    requested_mode=args.retrieval_mode,
                    retrieval_config=retrieval_config,
                )

            print(
                f"Running model={model_name} input={input_path.name} "
                f"retrieval_mode={retrieval_metadata['retrieval_mode']}"
            )

            try:
                result = run_model(
                    model_name=model_name,
                    temperature=temperature,
                    num_ctx=num_ctx,
```

```
                prompt=prompt,
                schema=schema,
                source_text=input_text,
            )
            raw_output_saved_path = write_raw_output(
                raw_output_dir,
                run_id,
                result["raw_output"],
            )
        except Exception as error:
            result = build_error_result(error)
            raw_output_saved_path = ""

        record = {
            "experiment_id": experiment_id,
            "run_id": run_id,
            "model_key": model_key,
            "model": model_name,
            "input_id": input_id,
            "input_file": input_path.name,
            "prompt_version": PROMPT_VERSION,
            "schema_name": SCHEMA_NAME,
            "schema_version": SCHEMA_VERSION,
            **retrieval_metadata,
            "temperature": temperature,
            "num_ctx": num_ctx,
            "raw_output_saved_path": raw_output_saved_path,
            **result,
        }

        with results_jsonl_path.open("a", encoding="utf-8") as file:
            file.write(json.dumps(record, ensure_ascii=False) + "\n")

        csv_rows.append({field: record.get(field) for field in CSV_FIELDNAMES})

    with metrics_csv_path.open("w", encoding="utf-8", newline="") as file:
        writer = csv.DictWriter(file, fieldnames=CSV_FIELDNAMES)
        writer.writeheader()
        writer.writerows(csv_rows)

    print(f"Saved JSONL results to: {results_jsonl_path}")
    print(f"Saved CSV metrics to: {metrics_csv_path}")


if __name__ == "__main__":
```

```
main()
```

ДОДАТОК Б
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів

Дипломна кваліфікаційна робота бакалавра

Тема: «Створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей»

Виконав ст. гр. КНТ-222 Рубан Матвій Михайлович
Керівник к.т.н., професор Табунщик Галина Володимирівна

Запоріжжя, 2026

1

Рисунок Б.1 — Слайд презентації

Мета, об’єкт, предмет і задачі роботи

Об’єкт — методи та засоби опрацювання даних великими мовними моделями.

Предмет — створення інтелектуальної системи для автоматичної генерації структурованих документів за допомогою великих мовних моделей.

Мета — покращення обробки табличних даних за рахунок використання великих мовних моделей.

Задачі:

– проаналізувати проблеми опрацювання документних джерел;

– обґрунтувати вибір локальних моделей, середовища виконання та технологій;

– спроєктувати архітектуру програмного конвеєра;

– реалізувати програмні компоненти системи;

– провести експериментальну перевірку MVP.

2

Рисунок Б.2 — Слайд презентації

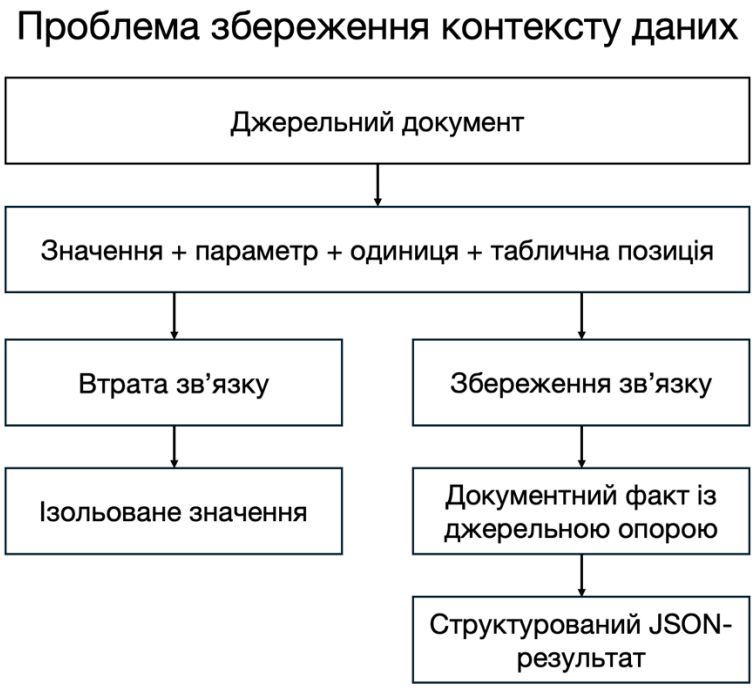


Рисунок Б.3 — Слайд презентації

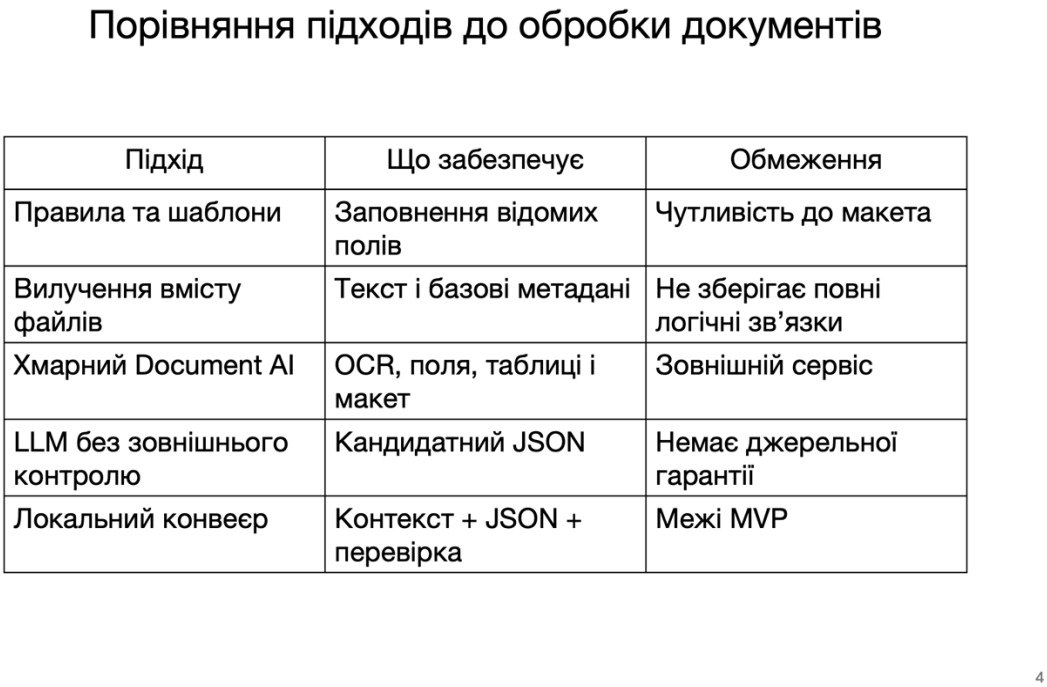


Рисунок Б.4 — Слайд презентації

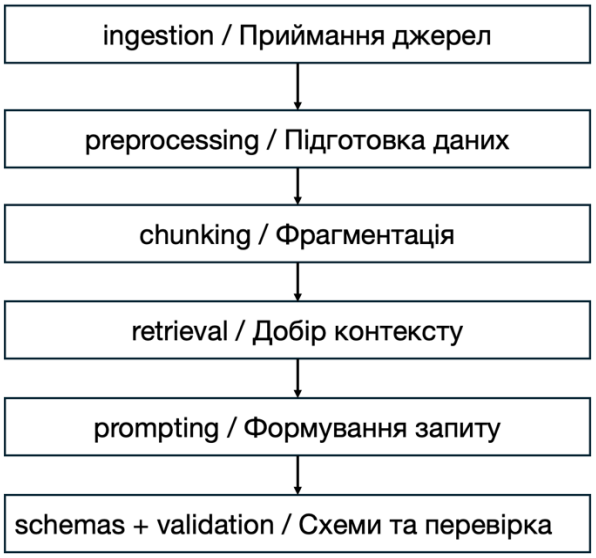
Обґрунтування вибору реалізації

Рівень рішення	MVP	Альтернативи
Вилучення документів	PyMuPDF / python-docx	Cloud Document AI, Azure Document Intelligence
Опрацювання таблиць	pandas / openpyxl / CSV-імпорт; рядок → «колонка: значення»; row_index, column_names	Docling / TableFormer / розбір з урахуванням макета
Добір контексту	FAISS + резервний лексичний пошук	LangChain / LlamaIndex / Haystack / векторна БД
Виконання LLM	Ollama + локальна модель	llama.cpp / vLLM / Hugging Face TGI / OpenAI API / Claude API / Gemini API
Приймання результату	JSON Schema + Pydantic + source_excerpt + source_anchor	Структуроване виведення / виклик функцій / декодування за граматичними обмеженнями
Джерельна відповідність	source_anchor + source_excerpt	Цитування в RAG / фільтрація за метаданими / генерація посилань

5

Рисунок Б.5 — Слайд презентації

Структура програмного проєкту



6

Рисунок Б.6 — Слайд презентації

Рисунок Б.7 — Слайд презентації

Рисунок Б.8 — Слайд презентації

Рисунок Б.9 — Слайд презентації

Результати експериментальної перевірки

Рівень перевірки	Обсяг	Результат
Модульні тести	pytest	62 / 62, 3 попередження без відмов
Сухі прогони	synthetic + format	20 / 20 + 15 / 15
Первинні сценарії	valid / invalid / mismatch	очікувані exit-коди
Локальний інференс	llama3.2:3b	JSON + Schema + Pydantic 5/5
Джерельна перевірка	source_anchor / source_excerpt	source_anchor стабільний; source_excerpt нестабільний

10

Рисунок Б.10 — Слайд презентації

Висновки

Результат	Підтвердження	Межі MVP
Локальний програмний конвеєр	Реалізовано шлях джерело → JSON → звіт	Без GUI, БД і промислового експорту
Контекстна обробка документів	Фрагменти, метадані, добір контексту	Без OCR і повної реконструкції макета
Перевірка результату	JSON Schema, Pydantic, source_excerpt, source_anchor	source_excerpt нестабільний у складних випадках
Експериментальна перевірка	Модульні тести 62 / 62; сухі прогони 20 / 20 + 15 / 15; модельна перевірка 5 / 5	результати дійсні в межах MVP
Практичне значення	валідований JSON з джерельними прив'язками	основа для подальшого DOCX/PDF-експорту

11

Рисунок Б.11 — Слайд презентації