**Технічний план розробки блоків B1 (SSOT-схеми) та B2 (Document Intake & Parsing)**

**B1: SSOT-схеми**

**Декомпозиція**

* **Моделі даних (Pydantic)**: Розширюємо файл core/schemas.py, що є **Single Source of Truth** для структур даних. Додаємо нові поля для підтримки мовних характеристик в сутностях, пов’язаних із текстовими фрагментами (spans). Зокрема, вводимо поля lang (ISO 639-1, напр. "en", "uk") та script (ISO 15924, напр. "Latn", "Cyrl") у відповідні моделі (наприклад, у модель **Span** або аналогічну структуру, що містить координати текстового фрагмента). Ці поля опціональні (мають тип Optional[str]), щоб зберегти зворотну сумісність. Якщо спеціальної моделі Span ще немає, варто створити Pydantic-модель TextSpan з полями: start: int, end: int, text: str, lang: Optional[str] = None, script: Optional[str] = None. Інакше – доповнити наявну структуру.
* **Подвійне представлення тексту**: Вводимо структури для **raw** та **normalized** тексту з вирівнюванням. У схеми документа (наприклад, модель ParsedDocument) додаємо поля для зберігання нормалізованого тексту та карти зміщень. Можливе поле normalized\_text: Optional[str] і offset\_map: Optional[List[int]], де offset\_map[i] = j означає, що символ з індексом *i* в normalized\_text відповідає символу з індексом *j* у вихідному *raw* тексті. Це дозволить точно співставляти позиції між нормалізованим і оригінальним документом, що критично для інтеграції з Word. Нові поля також робимо опціональними, щоб старі версії API не вимагали їх.
* **Класи та функції**: За потреби визначаємо допоміжні класи: наприклад, LanguageSpan (можливо синонім TextSpan) та DocumentMetadata. Функції, що формують вихідні об’єкти (як-от парсер документа або правила аналізу), будуть оновлені для заповнення нових полів, але інтерфейси залишаться незмінними. Логіку зворотної сумісності (модуль pipeline\_compat.py) оновимо мінімально – наприклад, якщо старі версії схем не мали поля lang, забезпечимо, що при конвертації старого формату в новий ці поля або заповнюються за замовчуванням (None), або відкидаються для старих клієнтів.
* **Версіонування API**: Підвищуємо мінорну версію API (v1.x → v1.y) для фіксації появи нових опційних полів. Явно не змінюємо URI endpoint’ів (залишаються /api/analyze, тощо), але у відповіді можемо додати заголовок X-API-Version: v1.y для прозорості. Старі поля не видаляємо, а якщо плануємо їх **депрекувати**, позначимо їх у схемах (через Deprecated = True у описі Pydantic-моделі або коментарем) та в OpenAPI. Додаємо у відповіді спеціальний заголовок (наприклад, Deprecation: true; Sunset=2026-02-\*\*) і логування при використанні застарілих полів, з періодом підтримки ~180 днів перед повним видаленням (sunset) згідно політики зворотної сумісності.

**Критерії прийняття**

* **Безперервна сумісність**: Існуючі endpoint’и повертають дані у старому форматі без помилок. Будь-які нові поля присутні лише як опціональні доповнення. Клієнти, що не очікують цих полів, повинні успішно ігнорувати їх. Наприклад, відповідь від /api/analyze містить звичний JSON ContractAnalysis з усіма старими полями; додатково кожен об’єкт Finding може містити поле span.lang та span.script, але якщо їх нема (null), клієнт все одно коректно опрацьовує відповідь. Схема відповіді суворо відповідає оновленим Pydantic-моделям.
* **Нові поля в відповіді**: Якщо документ містить текст кількома мовами/скриптами, в JSON-виході аналізу **для кожного знайденого фрагмента (span)** вказується мова та скрипт. *Приклад*: фрагмент знахідки:
* {
* "text": "Договір укладено ...",
* "start": 0,
* "end": 18,
* "lang": "uk",
* "script": "Cyrl"
* }

Інший фрагмент в тому ж документі:

{

"text": "This Agreement is ...",

"start": 100,

"end": 120,

"lang": "en",

"script": "Latn"

}

Таким чином, у масиві Findings для різних пунктів можуть з’явитися різні коди мов. Всі коди відповідають стандартам ISO 639-1 та ISO 15924.

* **Приклад JSON**: У загальному об’єкті ContractAnalysis може з’явитися секція document з інформацією про документ. *Приклад*:
* "document": {
* "content": "Hello “World” — Привіт \"Світ\"",
* "normalized\_text": "Hello \"World\" - Привіт \"Світ\"",
* "offset\_map": [0,1,2,3,4,5,6,...],
* "segments": [
* {"start": 0, "end": 13, "lang": "en", "script": "Latn"},
* {"start": 16, "end": thirty, "lang": "uk", "script": "Cyrl"}
* ]
* }

Тут вихідний контент містить англійський та український текст з різними лапками та тире; normalized\_text замінює *smart quotes* на звичайні " і довге тире на дефіс, карта offset\_map показує відповідність індексів (напр., offset\_map[13]=16 якщо в нормалізованому тексті менше символів через видалення невидимих знаків). segments позначає мовні фрагменти: перший сегмент англійський (Latn), другий український (Cyrl). **Примітка:** Нові поля (normalized\_text, offset\_map, segments) є опційними і можуть бути відсутні у відповіді, якщо не запитані або якщо документ мономовний.

* **Deprecated-поля**: Жодне існуюче поле не зникло. Якщо вирішено поступово прибрати застарілі поля (наприклад, старий формат span без вкладеного об’єкта), вони все ще присутні при ввімкненому режимі сумісності (через pipeline\_compat), але позначені як застарілі. Користувачі отримують попередження (HTTP-заголовок і повідомлення в логах) про те, що поле буде видалено через 180 днів. У MIGRATION.md описано, як перейти на новий формат.
* **Валідація схемою**: Вихід всіх основних endpoint’ів (/api/analyze, /api/gpt/draft, /api/suggest\_edits, /api/qa-recheck, /api/learning/\*) успішно проходить валідацію оновленими Pydantic-моделями. Автотести схем (test\_api\_contract\_ssot.py, test\_api\_schemas\_compat.py тощо) проходять без помилок.

**План тестування**

* **Юніт-тести моделей**: Додамо тестові кейси для Pydantic-моделей з новими полями. Перевіряємо, що модель Finding або Span коректно створюється як з заповненими lang/script, так і без них (None). Тестуємо серіалізацію в JSON: за відсутності значень поля не повинні з’являтися в вихідному JSON (що підтверджує опційність). Переконуємось, що при невалідних даних (наприклад, недопустимий код мови) схема відхиляє їх.
* **Фікстури даних**: Підготуємо приклади JSON (або використаємо існуючі фікстури контрактів), які відповідають старому формату і новому. Запустимо парсинг/валідацію через моделі: старі повинні проходити без змін, для нових – перевіримо заповнення нових полів. Також оновимо еталонні JSON-відповіді в інтеграційних тестах, додавши очікувані поля там, де потрібно.
* **Інтеграційні тести API**: Розширимо існуючі smoke-тести REST API.
  + Виклик /api/analyze з типовим документом: перевіряємо, що відповідь містить базові поля (Findings, Summary тощо) та **не містить** несподіваних змін (структура імена полів як раніше).
  + Якщо документ мультимовний (спеціально підготовлений), перевіряємо наявність lang/script в знайдених фрагментах і коректність їх значень.
  + Перевіряємо, що endpoints /api/gpt/draft, /api/suggest\_edits і інші, які використовують ті самі моделі, теж успішно працюють (вони могли опосередковано залежати від змін схем). Наприклад, /api/suggest\_edits може повертати зміни до тексту – переконаємось, що формат не змінений, а нові поля не заважають.
  + **Backward-compat**: Якщо існує спеціальний режим або флаг для старого формату (наприклад, v=1.0), пишемо тест, що при його використанні вихід точно відповідає старій схемі (нові поля відсутні) і що модуль pipeline\_compat правильно транслює дані. Також виконуємо тест на присутність заголовка Deprecation при використанні застарілих можливостей.
* **Тест сумісності версій**: Переконуємось, що пройшли тести на сумісність схем між версіями API (модульні тести test\_pipeline\_compat\_unit.py, test\_api\_schemas\_compat.py, які існують у проекті). Вони мають підтвердити, що *новий* формат може бути зведений до *старого* без втрати критичних даних і навпаки.
* **End-to-End**: Запускаємо повний аналіз на кількох контрактах через API в режимі реального використання (e2e тест): надсилаємо DOCX, отримуємо JSON, перевіряємо ключові поля. Особливо для багатомовного контракту перевіримо, що результати адекватні (наприклад, для українського тексту не спрацювали англомовні правила, але з’явилось зауваження що мова не підтримується, якщо таке передбачено). Відслідковуємо в логах повідомлення про депрекацію, якщо такі є.
* **Регресія**: Повторно проганяємо усі існуючі тести проекту. Очікується, що **всі** мають лишитись зеленими (включно зі test\_api\_contract\_ssot.py, який перевіряє узгодженість відповіді з схемою). Це гарантує, що ні формат, ні поведінка API не порушені.

**Ризики та пом’якшення**

* **Ризик 1: Злам сумісності API.** Будь-яка зміна схем може вплинути на клієнтів. *Пом’якшення:* Додаємо лише **опціональні** поля, не видаляючи нічого існуючого. Старі клієнти будуть отримувати структуру як раніше (нові поля просто ігноруються). На випадок, якщо зміни в логіці все ж торкнуться старого формату, підтримуємо дві версії через pipeline\_compat.py (за потреби). Також явно комунікуємо зміни: у відповіді та документації помічаємо нові поля і застарілі елементи (депрековані) з графіком sunset 180 днів.
* **Ризик 2: Неконсистентність схем.** Якщо не оновити якусь частину системи (наприклад, GPT-модуль чи SuggestEdits) відповідно до нових схем, дані можуть розсинхронізуватися. *Пом’якшення:* Дотримуємося SSOT – визначення структур лише в одному місці (core/schemas.py), всі інші модулі імпортують ці моделі. Після змін проганяємо всі тести, що покривають різні підсистеми, аби виявити розриви. В разі потреби – оновлюємо і їх (наприклад, якщо GPT-дто використовує копію класу Finding – замінюємо на імпорт із core).
* **Ризик 3: Неправильне визначення мови/скрипту.** Наприклад, фрагмент може містити змішані мови (англ. термін в укр. реченні) або символи, що вводять в оману алгоритм визначення. *Пом’якшення:* Використовуємо перевірену бібліотеку для визначення мов, яка повертає коди ISO 639-1. Якщо фрагмент короткий або двомовний – можна повертати мову основного тексту документа для нього або "un" (невизначено) в крайньому разі. Скрипт визначаємо на основі Юнікод-скриптів символів; у разі змішаних символів – беремо переважаючий скрипт або той, що відповідає визначеній мові (наприклад, для lang="uk" очікуємо script="Cyrl"). Ці нюанси документуємо. Потенційні помилки тут не ламають роботу системи, бо поля інформаційні; найгірший випадок – невідповідний код, що можна виправити в наступних патчах.
* **Ризик 4: Похибка вирівнювання offset’ів.** Якщо карта зміщень offset\_map буде некоректною, підсвічування тексту у Word (через координати) зіб’ється. *Пом’якшення:* Ретельно тестуємо на різних текстах (у т.ч. з емодзі, спецсимволами). В алгоритмі нормалізації уникаємо руйнівних змін довжини: наприклад, **не** видаляємо символи зовсім, окрім хіба що невидимих; заміну робимо на символи тієї ж довжини, якщо можливо (напр., лапки “ ” на " " – 1 символ на 1 символ). Якщо довжина змінюється (напр., видалення зайвого пробілу), відображаємо це точно в offset\_map. Додатково, забезпечуємо, що start/end знаходок відносяться до сирого тексту (raw), узгодженого з Word-документом. Це значить: після аналізу на нормалізованому тексті виконуємо зворотне відображення індексів на оригінал для запису в структуру Findings.
* **Ризик 5: Ускладнення тестів і підтримки.** Додавання нових полів може вимагати оновлення багатьох тестів і потенційно збільшить навантаження підтримки двох версій формату. *Пом’якшення:* Максимально автоматизуємо це через Codex: генеруємо патчі до тестів (більшість виправлень – додати lang/script у очікувані структури або відзначити їх як опціональні). Триматимемо обидві версії лише перехідний період; після sunset зайва складність зникне. Документацію (OpenAPI, MIGRATION.md) підтримуємо в актуальному стані, щоби зовнішні розробники знали про нові поля і план видалення старих.
* **Ризик 6: Проблеми з старими даними** (якщо зберігаються аналізи або кеші старого формату). *Пом’якшення:* Оскільки бекенд статично не зберігає результати (лише генерує при запиті), несумісність з історичними даними мінімальна. Якщо ж є збережені об’єкти навчання (/api/learning), їх схеми теж обновляємо міграційним скриптом або забезпечуємо читання старого формату через pipeline\_compat.

**Артефакти PR (Deliverables)**

* **Код**: Патч у вигляді unified diff, що містить зміни в основних файлах:
  + core/schemas.py – додані поля lang/script до моделей (напряму до Span або як вкладений TextSpan), можливе додання normalized\_text та offset\_map у модель документа. Зміни робимо акуратно, без перейменування існуючих полів.
  + engine/intake.py (або аналог парсера) – якщо потрібно, змінено щоб формувати об’єкт ParsedDocument з новими полями (це детальніше у B2).
  + engine/pipeline.py та pipeline\_compat.py – перевірено/доопрацьовано: у основному конвеєрі тепер використовується нова структура spans, а в compat-режимі вона транслюється назад (наприклад, у старому форматі могло бути finding.text і окремо finding.start, finding.end, а в новому – вкладений об’єкт; pipeline\_compat забезпечить при потребі розгортання або заповнення для старих клієнтів).
  + Моделі GPT (якщо є в gpt\_dto.py або interfaces.py) – імпортують оновлені схеми або додано аналогічні поля, щоби draft і інші функції не ламалися і могли при необхідності користуватися мовною міткою (наприклад, для вибору стилю відповіді).
  + Інші місця, де використовувалися індекси/текст: можливо, в Suggest Edits або QA Recheck – перевіряємо, чи не потрібно там врахувати normalized\_text або мовні атрибути (ймовірно ні, бо вони працюють з текстом контракту незалежно від мови).
* **Тести**: Оновлені файли тестів:
  + test\_api\_contract\_ssot.py, test\_api\_schemas\_compat.py – оновлено еталонні структури відповідей: додаємо відсутність/присутність нових полів, щоб тести пройшли.
  + Нові тести: test\_parser\_multilang.py (юнінти для визначення мови та нормалізації, хоча це більше B2), test\_api\_multilang\_intake.py (інтеграційний: надсилаємо мультимовний текст і перевіряємо JSON).
  + Всі існуючі тести проходять (CI покаже green). В PR включаємо звіт про це.
* **Документація**:
  + Оновлений файл **OpenAPI (swagger) спецификації** – у розділах схем даних (Schemas) додаємо описи нових полів lang, script у відповідних моделях (й зазначаємо deprecated: true для застарілих полів, якщо такі є). Версію API підвищуємо на мінор (v1.y). У коментарях OpenAPI або description до полів пояснюємо їх призначення (наприклад: *"Language code (ISO 639-1) of the text span"*).
  + **MIGRATION.md** – новий розділ, що описує зміни у версії v1.y. Зазначаємо, що додано поля lang/script у знайдки (Findings) та інші структури, старі поля (перелічити, якщо є) будуть вилучені через 6 місяців. Наводимо приклади, як адаптувати клієнтський код: наприклад, тепер span доступний як об’єкт з полями, замість окремих індексів (якщо так змінилося). Вказуємо дату sunset для deprecated-елементів.
  + **Приклади cURL** – додаємо в README або в коментар до PR приклади запитів. *Наприклад:*
  + curl -X POST -H "Content-Type: multipart/form-data" \
  + -F "file=@MultilangContract.docx" \
  + https://api.ourapp.com/v1/analyze

*Відповідь (фрагмент)*:

{

"findings": [

{

"id": 1,

"title": "Governing Law Missing",

"span": {

"text": "Даний контракт підпорядковується ...",

"start": 1024,

"end": 1100,

"lang": "uk",

"script": "Cyrl"

},

"suggestion": "Add governing law clause..."

},

...

],

"document": {

"normalized\_text": "...",

"offset\_map": [ ... ]

},

"version": "1.1"

}

Цей приклад демонструє, що у відповіді присутні нові поля (lang/script у span, секція document з normalized\_text) без порушення існуючої структури.

* **PR та коміти**: Файл патчу (наприклад, feature\_multilang\_spans.diff) із внесеними змінами. У описі PR наводимо план змін із пунктами (Conventional Commits), наприклад:
  + feat(core): add lang and script fields to text spans in schemas
  + feat(parser): implement language detection and offset mapping in intake engine
  + chore(tests): update and add tests for multi-language document support
  + docs(api): update OpenAPI spec and MIGRATION.md for v1.1  
    PR готовий до автоматичного злиття: після проходження CI (тести, лінтери) він може бути включений до main.

**B2: Document Intake & Parsing**

**Декомпозиція**

* **Парсер документів**: У модулі, відповідальному за зчитування документів (ймовірно contract\_review\_app/core/load\_docx\_text.py або engine/intake.py), реалізуємо покращений парсер. Його завдання – отримати *raw* контент з файлу (DOCX, PDF тощо) і підготувати структуру **ParsedDocument**. Парсер розбиває документ на логічні фрагменти (наприклад, абзаци, розділи) та витягує текст. Для DOCX використовуємо існуючий механізм (python-docx або lxml) – він вже повертає сплошний текст з розмітки.
* **Нормалізація тексту**: Після вилучення сирого тексту виконуємо нормалізацію контенту. Це включає:
  + Уніфікацію лапок та апострофів: перетворюємо “розумні” лапки («», “”) у стандартні ASCII-лапки ", а закриті апострофи – в '. Це усуває відмінності, які можуть завадити правилам пошуку шаблонів.
  + Обробку тире/дефісів: довгі тире (—) і середні (–) при необхідності замінюємо на простий дефіс - або на подвійний дефіс -- для консистентності, якщо цього вимагає логіка аналізу. Наприклад, послідовність "--" може використовуватися замість думок, але тут вирішуємо зробити всі тире однаковими.
  + Невидимі символи та пробіли: видаляємо zero-width space, нерозривні пробіли замінюємо на звичайні пробіли (або наявні подвійні пробіли скорочуємо до одного). Теж стосується кінцевих пробілів у рядках, табуляцій якщо вони не значущі.
  + Стандартизація регістру Unicode: застосовуємо NFC-нормалізацію до тексту, щоб всі комбіновані символи (акценти тощо) були в єдиній формі.
  + **Важливо:** Всі трансформації здійснюються так, щоб ми могли відстежити зміщення. Ми формуємо *normalized\_text* як новий рядок, паралельно вибудовуємо структуру **offset\_map**. Це можна зробити в одному проході по символах сирого тексту: для кожного символу вирішуємо, що додати до normalized\_text і як відобразити індекси. Наприклад, якщо на вході raw\_text[5] = "–" (ен-даш), а ми вирішили замінити на -, то в normalized\_text довжина збільшується на 1, і ми відображаємо індекс 5→5 (тобто символ змістився, але індекс співпадає позиційно). Якщо ж видаляємо якийсь символ (скажімо, zero-width), то просто пропускаємо його, і **тут** потрібно зафіксувати, що normalized\_text на цей індекс менше. Найпростіше – робимо список raw\_to\_norm де для кожного індексу сирого тексту зазначаємо відповідний індекс у нормалізованому. Так само, будуємо norm\_to\_raw (або користуємося raw\_to\_norm для оберненого пошуку) – саме ця структура буде збережена як offset\_map. На виході отримаємо дві версії тексту: raw\_text (по суті, теж можемо зберегти, хоча це вже є у Document.content) і normalized\_text.
* **Визначення мови та скрипту**: Реалізуємо компонент **Language Detection**, щоб розмітити витягнутий текст. Стратегія:
  + Аналізуємо текст документу по частинах. Найкраща гранулярність – рівень абзацу (paragraph) або навіть речення, але для простоти візьмемо абзаци. Кожен абзац, отриманий парсером, може розглядатися окремо: визначаємо мову цього абзацу. Якщо документ невеликий, можна і весь текст цілком – але тоді не побачимо вкраплень іншої мови. Отже, **ітерація по абзацам**: для кожного абзацу запускаємо функцію detect\_language(text) → код мови.
  + Бібліотека: використаємо, наприклад, langdetect або fastText модель для ISO 639-1. Для високої точності можна використовувати попередньо натреновану модель (наприклад, fasttext lid.176.bin). Враховуємо, що можуть бути короткі абзаци, де мовна модель не впевнена – тоді або залишаємо поле lang пустим, або успадковуємо мову від сусіднього контенту/загальної мови документа.
  + Визначення скрипту: на основі отриманої мови та символів абзацу. Часто мова однозначно визначає скрипт (наприклад, "uk" → Cyrl, "en" → Latn). Але, щоб бути формальними, можна пройтися по символах Unicode абзацу: якщо знаходимо букви з діапазону кирилиці – скрипт Cyrl; латиниці – Latn; присутність ієрогліфів – Hans або Hant (для китайської), арабські літери – Arab, тощо. Якщо текст містить змішані скрипти (скажімо, латиниця з кількома кириличними символами), можна обрати скрипт, що відповідає визначеній мові (бо кириличні вкраплення можуть бути випадковими, або навпаки).
  + Результатом буде набір **мовних сегментів**. Його можна представити, наприклад, як список словників {start\_char, end\_char, lang, script} відносно всього документу. *Приклад сегмента*: {"start": 0, "end": 42, "lang": "en", "script": "Latn"} для першого абзацу англійською; наступний сегмент для другого абзацу українською і т.д. Ці сегменти ми додаємо в ParsedDocument.segments.
  + Також додаємо поле основної мови документа, якщо потрібно (наприклад, найчастіше зустрічається мова, або мова першого значущого абзацу).
* **Інтеграція з конвеєром**: Модуль Intake/Parsing повинен видавати ParsedDocument, який включає: raw text, normalized\_text, offset\_map, segments із мовною розміткою. Потім Pipeline використовує ParsedDocument.normalized\_text для застосування правил аналізу (Findings). Необхідно адаптувати місця, де раніше міг використовуватися просто текст: тепер з об’єкта ParsedDocument дістанемо normalized\_text. Наприклад, engine.executor або legal\_rules.runner - вони можуть брати текст документа для пошуку патернів; там скоригуємо джерело тексту.
* **Збереження offset при аналізі**: Правила (business rules) при пошуку часто повертають позицію знайденого фрагмента. Якщо вони працюватимуть по нормалізованому тексту, їхній результат – індекси start\_norm, end\_norm. Ми маємо перетворити їх у індекси оригіналу перед формуванням відповіді. Це можна зробити в момент формування об’єкта Finding: беремо start\_raw = offset\_map[start\_norm], end\_raw = offset\_map[end\_norm]. Саме ці raw-координати записуємо в Finding.span.start/end. Таким чином, зовнішні клієнти (і Word-надбудова) отримають позиції, що відповідають вихідному документу, а не модифікованому тексту. (*Примітка:* Якщо структура Findings вже містить витяг text, його теж можна брати з сирого контенту по тих же координатах – щоб клієнт бачив оригінальні лапки та символи у виділеному тексті).
* **Span tagging у Findings**: Кожна знайдка буде збагачена мовною міткою. Оскільки ми вже знаємо, в якому сегменті вона знаходиться (маючи raw index), можемо при формуванні Finding вписати lang та script. Наприклад, якщо finding починається на 1024 символі, дивимося, до якого сегменту він належить (сегмент з діапазоном, що охоплює 1024) – і проставляємо відповідні lang/script у Finding.span. В результаті, правила не змінюються, але їхній output доповнюється цією інформацією, яку можна використовувати далі (UI може відобразити значок мови, або ж GPT-модуль може зважати на це при генеруванні рекомендацій іншою мовою).
* **Врахування emoji та спецсимволів**: Парсер має пропускати через себе емодзі та інші символи Unicode, не втрачаючи їх. В normalized\_text емодзі можна лишати без змін (вони не заважають аналізу, хіба що прибрати варіаційні селектори якщо є). В offset\_map кожен такий символ зазвичай відображається 1:1 (або декілька кодпоінтів Unicode як один видимий символ – але ми рахуємо індекси по кодпоінтах, тож якщо в raw був комплексний символ з кількома code units, ми трактуємо їх як окремі індекси; втім, це тонкість, яку приймаємо як стандарт). Головне – не дропати emoji, щоб не зрушувати позиції непередбачувано. Тестуємо, що рядки з емодзі проходять без винятків і правильно мапляться.
* **Архітектурні зв’язки**: Переконуємося, що новий ParsedDocument сумісний з існуючим потоком:
  + В orchestrator чи api/app.py – де зчитувався файл, тепер отримуємо ParsedDocument і передаємо далі.
  + Якщо десь використовувалися глобальні змінні на кшталт current\_doc\_text – замінюємо на об’єкт.
  + Решта системи (LLM, SuggestEdits) може брати дані з Findings, тож для них зміна прозора (вони просто отримають трохи більше даних в JSON).
  + **Performance**: операції визначення мови та нормалізації лінійні від розміру тексту, що прийнятно навіть для великих контрактів. Якщо документ >100 стор., треба перевірити, чи не стане визначення мови вузьким місцем; можна оптимізувати, визначаючи мову не кожного абзацу, а груп абзаців, або припиняти детекцію якщо вже явно визначено одну мову. Але для першої версії зробимо повний розбір.

**Критерії прийняття**

* **Підтримка багатомовних документів**: Система коректно обробляє документ, що містить більше однієї мови (напр., англійську та українську в різних частинах). *Acceptance test:* подаємо на вхід DOCX з двома абзацами: перший англійською *"The Agreement is concluded in two languages."*, другий українською *"Цей контракт укладено двома мовами."*. На виході /api/analyze отримуємо Findings для кожного правила. Якщо якісь правила спрацьовують (наприклад, відсутність положення, яке стосується обох мовних версій), перевіряємо, що у відповідних Finding.span:
  + lang = "en", script = "Latn" для фрагментів з першого абзацу;
  + lang = "uk", script = "Cyrl" для фрагментів з другого.  
    Навіть якщо правила для українського тексту ще не реалізовані і знайдки відсутні, сам факт успішного проходження аналізу без помилок уже критерій прийняття. Система не падає і не видає помилок кодування Unicode.
* **Коректність мапування індексів**: Для кожної знайдки перевіряється, що виділений текст відповідає оригіналу. *Приклад:* якщо у відповіді span.start = 50, span.end = 60, то фрагмент document.content[50:60] точно співпадає з span.text і відповідає тому місцю в DOCX. При цьому document.normalized\_text[offset\_map[50]:offset\_map[60]] (через карту) дорівнює нормалізованій версії цього ж фрагменту. Всі findigns, сформовані по нормалізованому тексту, правильно відображаються назад. Особливо це важливо, якщо в тексті були “розумні” лапки чи інші символи: наприклад, правило знайшло щось між лапками, то span.text має містити ті самі **вигнуті** лапки, як у оригіналі, хоча в normalized\_text вони прямі.
* **Відсутність деградації на моно-мовних документах**: На звичайному англомовному контракті, який раніше підтримувався, результати /api/analyze ідентичні до змін (окрім можливого додавання порожніх lang полів). Критично, щоб не знизилась якість правил: нормалізація не повинна зламати існуючі пошукові шаблони. *Acceptance:* проганяємо кілька англійських контрактів – вихідні Findings збігаються з раніше отриманими (ті самі зауваження на тих самих місцях).
* **Emoji та спеціальні символи**: Якщо документ містить emoji або символи інших писемностей, вони відображаються в результатах без помилки. *Приклад:* текст *"All good ✅. Все добре ✅."* – очікуємо, що в normalized\_text галочка ✅ залишиться (або заміниться на ідентичний символ, не на пустоту), і що offset\_map довжину рахує правильно. Відповідь API буде містити цей символ у span.text якщо він у фрагменті. Ніяких UnicodeEncodeError чи втрати даних в логах бути не повинно.
* **Логування і заголовки**: Сервіс при обробці документа пише в лог ключові кроки (для дебагу): визначені основні мови документа, час виконання визначення мов. Якщо ввімкнено debug-режим, можна побачити offset\_map або статистику нормалізації (скільки символів замінено/видалено). Клієнт у відповіді отримує заголовок X-Parsed-Languages: en,uk (наприклад, перелік знайдених мов) – це необов’язково, але корисно. Наявність такого заголовка чи аналогічного вважаємо плюсом (Acceptance nice-to-have).
* **Документованість**: Код парсера та нормалізації супроводжується коментарями, а у README/Developer Docs описано, як працює мовна розмітка. В OpenAPI документації на рівні схем зазначено, що document.segments показує мови фрагментів, а span.lang/script – мова знайдки. Критерій прийняття – наявність цих описів, щоб команди, що використовують API, розуміли нові поля.

**План тестування**

* **Юніт-тести парсера**: Розробляємо модульні тести для функцій normalize\_text і detect\_language (умовні назви).
  + Для normalize\_text(raw\_text): беремо спеціальні рядки і перевіряємо, що вихід відповідає очікуваному. Наприклад:
    - Вхід: "Hello\u00A0World" (Hello&non-breaking-space;World) → вихід: "Hello World" та відображення індексів [0,1,...,5, **пропуск 6**,7,...] де видно, що NBSP на позиції 5 зник, і всі подальші символи зсунулись на -1 у normalized.
    - Вхід: "Quote: “Test”." → вихід: "Quote: \"Test\"." (лапки замінено), перевіряємо що довжина та відображення 1:1 (усі символи окрім лапок однакові, лапки теж 1 символ → довжина рядка незмінна, offset\_map[i]=i для всіх i).
    - Вхід з емодзі: "Cool 👍!" → вихід той самий "Cool 👍!" (offset\_map[i]=i для всіх, бо нічого не змінено).
    - Вхід зі змішаними дефісами: "Non\u2011breaking-hyphen" (\u2011 – нерозривний дефіс) → вихід: "Non-breaking-hyphen" (звичайний дефіс), offset\_map показує зміщення позиції на -1 після заміни, якщо вирішимо вилучити символ або замінимо на один символ – тоді зміщення нема).
  + Для detect\_language(text): перевіряємо на відомих прикладах:
    - "This is an English sentence." -> "en".
    - "Це українське речення." -> "uk".
    - Дуже короткий текст, напр. "Hello" -> може повернути "en" або навіть "un" якщо невпевнено; у тесті допускаємо обидва варіанти, головне щоб не кидало винятків.
    - Текст зі змішаними мовами, напр. "Hello і привіт" -> можливо, залежно від бібліотеки, визначиться якась одна; тут перевіряємо, що або "en" або "uk", і скрипт відповідно Latn/Cyrl (ми можемо окремо протестувати функцію визначення скрипту: подати рядок "ABC123" -> Latn, "Абвгд" -> Cyrl, "Test тест" (мікс) -> результат, наприклад, Latn якщо беремо перший скрипт, або None – залежно від нашої реалізації).
  + Для функції побудови offset\_map: можемо виділити її окремо (хоча це побічний продукт normalize\_text). Протестуємо на кількох випадках, що відображення правильне (як описано для normalize\_text).
* **Інтеграційні тести парсингу**: Створюємо у contract\_review\_app/tests/ кілька нових fixture-файлів:
  + multilang.docx: документ, що містить, напр., заголовок англійською, абзац українською, потім знов англійською. Також вставимо туди emoji та різні лапки для комплексності.
  + emoji.docx: невеличкий документ спеціально з різними емодзі, символами (&, ©, ™) для перевірки стійкості.
  + mixed\_script.docx: текст, в якому навмисно перемішані латинські та кириличні символи (наприклад, слово, де одна літера замінена схожою з іншого алфавіту) – щоб побачити, як це відобразиться.
  + Проганяємо парсер на цих файлах (можна викликати внутрішню функцію, що він використовує). Перевіряємо:
    - ParsedDocument.content містить очікуваний сирий текст (з мінімальними технічними артефактами, типу відсутні контрольні символи Word).
    - normalized\_text не порожній і відрізняється від content лише у потрібних місцях (порівнюємо вручну або через очікуваний фрагмент).
    - offset\_map довжиною = len(normalized\_text). Вибірково перевіряємо кілька пар індексів з мапи на коректність.
    - segments список не порожній. Кількість сегментів відповідає кількості абзаців (або об’єднаних фрагментів) у документі. Кожен сегмент має ненулеву довжину (end > start) і валідні коди lang/script (строки по 2 і 4 символи відповідно, або None). Перевіряємо, що текст між start і end у content дійсно належить цій мові (наприклад, беремо сегмент з lang "uk" – витягуємо відповідний шматок content і впевнюємося, що там переважно кирилиця).
  + Інтегруємо цей parsed output у Pipeline: викликаємо функцію аналізу правил на ParsedDocument і перевіряємо, що вона виконується без помилок і повертає Findings. Для multilang.docx: якщо є специфічні правила під англ. право – вони можуть не знайти нічого в укр. тексті, але головне, щоб Pipeline не кинув exception в процесі. Перевіряємо, що на виході Findings містять lang/script згідно з сегментами.
* **Інтеграційні тести API**:
  + Виконуємо запит через тестовий клієнт API: /api/analyze з файлом multilang.docx. Очікування: статус 200, контент-тип application/json. В тілі – список findings. Перевіряємо, що для відомих нам мовних фрагментів присутні коректні поля lang/script. Якщо, наприклад, документ двомовний і якесь правило спрацювало на обох частинах (скажімо, відсутній розділ Definitions і в англ, і в укр тексті) – будемо мати дві знайдки, одну з "lang":"en", іншу з "lang":"uk". Ці поля перевіряємо в тесті.
  + Перевіряємо заголовки відповіді (якщо додали X-Parsed-Languages, Version тощо).
  + Перевіряємо, що час відповіді не значно зріс (можна засікти, але це скоріше non-functional тест).
  + Викликаємо також /api/qa-recheck чи /api/suggest\_edits на результат попереднього аналізу (або в рамках orchestrator) – очікуємо, що вони відпрацюють нормально. (Їх логіка майже не змінюється, але, наприклад, suggest\_edits може використовувати span для накладання правок – тут критично, щоб span.start/end все ще відповідали raw-тексту Word, що ми забезпечили).
* **Тестування Word-інтеграції вручну**: Хоч автотестів на OfficeJS нема, варто провести мануальний тест: відкрити Word, завантажити надбудову, проаналізувати двомовний документ. Перевірити, що надбудова підсвічує знайдені місця правильно. Особливо – якщо є emoji або різні лапки, вони повинні коректно виділятися. Це підтвердить, що наша offset\_map + використання raw координат успішно працює у реальній інтеграції (в межах наших можливостей; повноцінно автоматизувати Word важко, але такий експеримент додасть впевненості).
* **Регресійне тестування**: Проганяємо **всі** існуючі тести застосунку, не тільки ті, що ми додали. Очікуємо, що вони пройдуть (це вже було у B1). Особливу увагу на test\_doc\_type\_backcompat.py, test\_pipeline\_compat\_unit.py – вони можуть виявити, якщо ми щось забули щодо старих форматів. За необхідності – доповнюємо compat-шар (можливо, додавши видалення normalized\_text перед поверненням відповіді, якщо старий формат цього не очікує).

**Ризики та пом’якшення**

* **Ризик 1: Хибне визначення мови.** Автоматичне визначення може помилятися, особливо на коротких фразах або спеціальних юридичних формулюваннях (які можуть бути схожі в різних мовах). *Пом’якшення:* По-перше, використовуємо комбінований підхід: якщо детектор мови має низьку впевненість (менше певного порогу) – можемо позначати сегмент як невизначений (lang=None) або спадково призначати мову документа. По-друге, допускаємо, що точність не 100%, бо наслідки невеликі (помилка в полі lang не критична для логіки правил наразі). В майбутньому можна додати явне зазначення мови користувачем (наприклад, якщо відомо, що контракт двомовний англ/укр, одразу це врахувати). Також можна розширити тестовий набір фраз, щоб настроїти детектор під юридичні тексти (додати доменні стоп-слова або примусово фільтрувати результат: якщо текст містить багато кирилиці – навіть при сумнівах віддавати "uk").
* **Ризик 2: Нестандартні скрипти і символи.** Якщо документ містить, скажімо, текст арабською або ієрогліфи, а наша система наразі не орієнтована на них, можливі помилки (наприклад, неправильно визначиться код скрипту, або взагалі правила нічого не знайдуть і виникне питання підтримки). *Пом’якшення:* Архітектура вже передбачає мультискриптовість, тож технічно додати нову мову – лише справа навчання правил. Для парсера: тести з арабським та китайським текстом теж варто спробувати (напр., додати у fixture рядок арабською). Якщо виявиться, що detect\_language повертає не ISO 639-1 (деякі бібліотеки можуть дати ISO 639-3 чи BCP-47) – обробляємо цей випадок (мапимо код на 639-1 або беремо перші 2 літери). У випадку символів, які не належать жодному алфавіту (скажімо, математичні символи, символи валют) – вони не впливають на визначення мови, але можуть вплинути на нормалізацію (ми їх, втім, не чіпаємо).
* **Ризик 3: Зсув індексів через непередбачені трансформації.** Можливо, у Word-документі є приховані елементи (поле, коментар), які ми витягнули або пропустили у тексті. Це може призвести до розбіжності між нашим raw\_text і реальним текстом, який бачить користувач у Word (тобто offset\_map не допоможе, бо текст інший). *Пом’якшення:* Переконатися, що метод витягнення тексту ігнорує невидимий контент Word (common practice – python-docx зазвичай повертає тільки видимий текст). Для повної впевненості – у Word-надбудові, якщо є API, яке дозволяє отримати position/range за текстовим збігом, використовувати його, а не покладатися лише на індекси. Але це поза нашим кодом. З нашого боку – можемо логувати довжину тексту і кількість абзаців, щоб за аномаліями зрозуміти, чи все витяглось. Якщо, наприклад, Word-надбудова покаже зміщення, доведеться відладити конкретний випадок і, можливо, зберігати більше структур (наприклад, відповідність індексів абзаців).
* **Ризик 4: Продуктивність.** Додаткові кроки (нормалізація, визначення мов) збільшують час аналізу. Якщо контракт дуже великий, це може бути відчутно. *Пом’якшення:* Перевіряємо ефективність: всі алгоритми O(n). FastText модель для language detection дуже швидка (мсек на речення). Нормалізація – простий прохід по тексту (можливо, Python-loop, але можна оптимізувати через .translate() частково, хоча там складно відобразити offset\_map). Якщо побачимо, що аналіз ~50-100 сторінок займає надто довго, можна зробити: а) опціонально вимикати детекцію мов (напр., прапор ?lang=false в API), б) кешувати результати для однакових абзаців, в) написати критичні місця на Cython або використати SIMD для сканування символів. Попередньо, очікуємо приріст часу не більше 5-10% для середнього документа.
* **Ризик 5: Вплив на LLM-модуль.** Якщо GPT-модуль тепер отримає в контексті Finding з lang="uk", чи не спричинить це, що він спробує згенерувати відповідь українською? *Пом’якшення:* Залежить від реалізації. Якщо ми цього не хочемо – можна поки що не передавати поле lang у prompt. Але можливо, це навіть бажано – GPT-драфт міг би враховувати мову. У будь-якому разі, перевіримо тести GPT (вони є з mock) – вони не повинні зламатись від додаткового поля (DTO просто його проігнорує). Отже ризик невеликий.
* **Ризик 6: Помилки в pipeline\_compat.** Додаючи нові поля, треба не забути про backward-compat. *Пом’якшення:* В модулі pipeline\_compat.py (або аналог) напишемо конверсію: якщо клієнт вимагає старий формат, ми можемо **не включати** document секцію і lang/script поля. Це можна зробити через Pydantic-модель з exclude\_none/exclude\_unset або явною обробкою словника. Тестом ми це вже покрили, тож вважатимемо, що ризик адресовано.
* **Ризик 7: Управління версіями і розгортання.** Після внесення змін потрібен плавний реліз. *Пом’якшення:* Налаштовуємо feature-flag чи конфігурацію для ввімкнення мовного аналізу. Можна спочатку випустити приховано (наприклад, якщо середовища staging, то тестуємо там). Також, якщо API версіонується, не забуваємо оновити версію в CI/CD (в Docker image тегах тощо). Це організаційний ризик, який знімається уважним плануванням релізу (blue-green деплой, можливість відкоту на v1.x, оскільки схема сумісна).

**Артефакти PR (Deliverables)**

* **Код і дифф**: Основні зміни будуть в таких файлах:
  + core/intake.py (або core/load\_docx\_text.py / utils/doc\_loader.py): нові функції normalize\_text() та detect\_language(), або інтеграція їх у існуючі. Код, що формує об’єкт ParsedDocument, змінено: додаємо заповнення normalized\_text, offset\_map, segments. Приклад diff-секції:
  + def parse\_document(file\_bytes) -> ParsedDocument:
  + raw\_text = extract\_text\_from\_docx(file\_bytes)
  + - return ParsedDocument(content=raw\_text)
  + + normalized\_text, offset\_map = normalize\_text(raw\_text)
  + + segments = detect\_language\_segments(raw\_text, offset\_map)
  + + return ParsedDocument(content=raw\_text, normalized\_text=normalized\_text,
  + + offset\_map=offset\_map, segments=segments)

Де detect\_language\_segments повертає список сегментів з мовою і скриптом.

* + core/schemas.py: модель ParsedDocument доповнюється полями (як зазначено в B1). Модель Finding.span тепер може бути не просто Tuple[start,end], а об’єкт Span з полями (start,end,text,lang,script). В diff це буде видно як нові класи або поля зі схемами Optional. Наприклад:
  + + class Span(BaseModel):
  + + start: int
  + + end: int
  + + text: str
  + + lang: Optional[str] = None
  + + script: Optional[str] = None
  + +
  + class Finding(BaseModel):
  + - start: int
  + - end: int
  + - text: str
  + + span: Span
  + ...

*(Примітка: Якщо вирішимо не змінювати структуру Finding радикально, можемо залишити старі поля start/end/text для сумісності, а нові додати окремо. Але оптимальніше зробити вкладений Span і у pipeline\_compat розгорнути його назад для старих клієнтів.)*

* + engine/pipeline.py та pipeline\_compat.py: зміни для використання нового ParsedDocument. В pipeline.py, де викликаються правила:
  + parsed = parse\_document(file)
  + - findings = run\_rules(parsed.content)
  + + findings = run\_rules(parsed.normalized\_text)

і далі, перед віддачею:

- result.findings = findings

+ result.findings = []

+ for f in findings:

+ # map normalized indices to raw

+ f.span.start = parsed.offset\_map[f.span.start]

+ f.span.end = parsed.offset\_map[f.span.end]

+ f.span.text = parsed.content[f.span.start: f.span.end]

+ f.span.lang = find\_segment\_lang(parsed.segments, f.span.start)

+ f.span.script = find\_segment\_script(parsed.segments, f.span.start)

+ result.findings.append(f)

А в pipeline\_compat.py, якщо старий формат, робимо зворотне:

if legacy\_mode:

- for f in result.findings: del f["span"]

- # possibly re-add f["start"], etc from span

+ for f in result.findings:

+ f["text"] = f["span"]["text"]

+ f["start"] = f["span"]["start"]

+ f["end"] = f["span"]["end"]

+ del f["span"]

(Це псевдокод дифу, справжній буде залежати від реалізації).

* + Тести:
    - Нові файли test\_normalize.py, test\_language\_detect.py з тестами, про які говорилося.
    - test\_parser\_intake.py з інтеграційним тестом на ParsedDocument.
    - Оновлення існуючих: test\_app\_contract.py можливо чекатиме старі поля, тож або перемикаємо його на новий формат, або якщо endpoint ще віддає старий формат за замовчанням – лишаємо як є.
    - В будь-якому разі, test\_api\_contract\_ssot.py оновлюємо еталон, щоб включав новий вкладений span або нові поля (залежно від рішення).
  + CI/інфраструктура: Можливо, додати встановлення пакета для language detection (якщо не був). У requirements.txt/pyproject.toml з’явиться, наприклад, langdetect або fasttext. Це теж частина deliverables (оновлені залежності).
* **Документація та приклади**:
  + **OpenAPI**: Специфікація оновлена відповідно до нової структури Finding і ParsedDocument. Будуть нові визначення компонентів, наприклад, схема Span з полями. У вихідних JSON-схемах для /api/analyze показані ці оновлення. Документацію щодо параметрів запиту не змінюємо (вони ті самі, файл або текст на вході), але в описі відповіді з’являться поля document.normalized\_text тощо, помічені як optional.
  + **MIGRATION.md**: Описуємо, що тепер API підтримує багато мов. В розділі “Changes in 1.y” вказуємо:
    - “Parsing now normalizes text and identifies languages/scripts of content. New optional fields in response: findings[].span.lang/script, document.normalized\_text etc. No action needed for clients, as these are additive. If your integration parses the returned JSON, ensure it can handle the nested span object (previously separate fields). The old flat fields remain for now under compatibility mode.”
    - Якщо ми ввели вкладений Span і прибрали старі поля з нової версії – це breaking, але за умовою ми цього уникнемо. Тож напишемо, що старі поля start, end, text **поки** доступні, але застарілі, і буде повний перехід на вкладений span.
    - Розписуємо timeline: “fields X, Y deprecated, to be removed after 2026-02-...”.
    - Наводимо приклад до/після JSON, щоб розробники бачили різницю.
  + **Приклади використання**: Оновлюємо README або API Reference, додаючи приклад запиту аналізу мультимовного документа і приклад відповіді. Відображаємо, як виглядає знайдка з мовним тегом. Також, у документації до Word-надбудови (якщо є dev docs) зазначимо, що результати аналізу тепер містять координати, які точно відповідають тексту, і що надбудова може їх використовувати для виділення (можливо, це вже було, але підтвердимо покращення).
  + **Конвенції комітів**: У описі PR перераховані коміти (як згадувалося в B1). Codex автоматично підготує PR, ми переконаємось що там є:
    - feat: language-aware parsing in intake (multi-language support)
    - feat: add offset mapping for span alignment
    - test: add parser fixtures for multi-language
    - docs: update API docs for new minor version
    - ... (коміти відповідають зробленим змінам).  
      Ці коміти згруповано в PR, CI підтверджує їх проходження. Patch-файл patch\_intake\_multilang.diff додається як артефакт (для перегляду змін).
* **Приклади cURL**:
  + Наводимо приклад запиту аналізу, щоб підкреслити, що нічого не змінилося в способі виклику:
  + curl -H "Authorization: Bearer <TOKEN>" -F "file=@Contract\_multi.docx" https://api.ourapp.com/v1.1/analyze

(якщо версію в URL не використовуємо, то просто /v1/analyze, версію можна через header).  
Отримуємо код 200 і JSON.

* + Для ілюстрації, приклад фрагменту відповіді:
  + {
  + "summary": { ... },
  + "findings": [
  + {
  + "id": "NoDefinitions",
  + "title": "Definitions section missing",
  + "severity": "medium",
  + "span": {
  + "text": "В цьому контракті відсутній розділ визначень...",
  + "start": 500,
  + "end": 530,
  + "lang": "uk",
  + "script": "Cyrl"
  + },
  + "suggestion": "Додайте розділ «Визначення термінів»..."
  + },
  + {
  + "id": "NoDefinitions",
  + "title": "Definitions section missing",
  + "severity": "medium",
  + "span": {
  + "text": "This contract does not contain a Definitions section...",
  + "start": 480,
  + "end": 530,
  + "lang": "en",
  + "script": "Latn"
  + },
  + "suggestion": "Add a Definitions section to clearly define terms..."
  + }
  + ],
  + "document": {
  + "content": "...",
  + "normalized\_text": "...",
  + "offset\_map": [...],
  + "segments": [
  + {"start":0,"end":...,"lang":"en","script":"Latn"},
  + {"start":...,"end":...,"lang":"uk","script":"Cyrl"}
  + ]
  + }
  + }

Тут показано, що одне правило (відсутність розділу визначень) дало дві знайдки – для укр та англ частин окремо, з відповідними мовними мітками. Це демонструє роботу мультимовності. document.segments показує діапазони символів для двох мов.

* + Такий приклад буде додано до **Postman Collection** або API документації для наочності.
* **Автоматизація Codex**: Після написання плану та специфікацій, інтегрований Codex (AI pair programmer) використовує ці вимоги для генерування коду. Він створить гілку з змінами, включаючи модифіковані файли і нові тести, згенерує патч. Ми перевіримо PR, внесемо виправлення при необхідності, і після схвалення – зіллємо. Усі артефакти (код, тести, документація) будуть частиною одного PR для функціоналу B1+B2, готового до деплою після проходження CI.