Bilkent University
Department of Computer Science

CS 492 - Senior Design Project II
Project Final Report

Agreemind (T2516)

Group Members:
Ata Oğuz - 22202453
Ata Soykal - 22202290
Can Polat Bülbül - 22203369
Edip Emre Dönger - 22201531
Emir Görgülü - 22202834

Supervisor: Hamdi Dibeklioğlu
Instructors: Mert Bıçakçı, İlker Burak Kurt

1. Introduction

People regularly enter into agreements of many kinds, including online Terms of Service, privacy policies, banking agreements, subscription contracts, non-disclosure agreements, rental documents, warranty policies, employment terms, and freelance contracts. These documents often define important financial, personal, and professional responsibilities. However, they are usually written in dense legal language, contain long and highly structured clauses, and place important rights, obligations, and deadlines deep inside the document. As a result, non-expert users may accept unfavorable conditions without fully understanding their consequences, overlook time-sensitive actions, or fail to recognize clauses that significantly affect their rights.

The problem continues after a document is signed or accepted. Agreements may be revised over time, and users often have no practical mechanism for identifying what changed between two versions or whether a new version introduces additional risks. Users may also store many agreements across different platforms, making it difficult to search them, compare them, or remember obligations such as renewal dates, cancellation periods, payment deadlines, or notice requirements. Therefore, the project is not limited to one-time contract summarization; it addresses the broader lifecycle of personal agreement management. Existing enterprise contract analysis tools such as Ironclad, Kira Systems, and Luminance mainly target corporate legal teams rather than individual users [1], [2], [3]. Crowdsourced transparency projects such as ToS;DR and Open Terms Archive provide useful public agreement summaries, but they do not analyze arbitrary private documents uploaded by the user [4], [5].

Agreemind is designed as an intelligent consumer-facing legal document assistant that helps users understand, organize, compare, and act upon their agreements. The system accepts documents through multiple client surfaces: a web application, a native mobile application, and a browser extension. Uploaded or extracted agreement text is processed by a backend analysis pipeline that performs document type classification, clause-level risk detection, rights extraction, timing and deadline extraction, risk scoring, optional LLM-based refinement, and optional clause rewriting. The resulting analysis is presented to the user through structured summaries, risk findings, rights, reminders, and document-level metadata.

Since the detailed design stage, Agreemind has evolved from a proposed multi-client legal analysis platform into an implemented system with specialized analysis engines and end-to-end document management features. The final system supports different contract domains through separate engines for non-disclosure agreements, Turkish banking contracts, English banking contracts, Terms of Service or web agreements, and general contracts. This domain-specific architecture allows the system to apply more appropriate models and rules instead of relying on a single generic classifier for all documents. The analysis pipeline also combines deterministic NLP and machine learning components with optional LLM refinement, providing a balance between reproducibility, cost control, and improved explanation quality.

The final architecture consists of three client surfaces connected to a FastAPI backend. The web application is implemented with React Native and Expo Router running in the browser through react-native-web. The mobile application is implemented as a standalone React Native CLI application with tab-based navigation and includes features such as camera-based document scanning, OCR-based analysis, background analysis status polling, local completion notifications, alerts, folder management, document type override, clause rewrite display, and global vault chat. The browser extension is implemented with Vite, React, and TypeScript, and allows users to detect legal agreement pages, extract readable webpage text using Readability.js, submit the content to the backend, and view analysis results directly from the extension popup.

The backend is implemented using FastAPI and is organized into versioned API routes for authentication, documents, folders, and chat. The backend manages user authentication, document upload, text extraction, background analysis, RAG ingestion, version comparison, document storage metadata, and result retrieval. Long-running document analysis is executed asynchronously through FastAPI background tasks so that upload requests can return quickly while the frontend polls for analysis and ingestion status. Persistent data is stored in PostgreSQL with pgvector support for embeddings, while document chunks are indexed for hybrid vector and full-text retrieval.

A major implemented capability of the system is the retrieval-augmented generation chat system. During ingestion, each document is split into sentence-aware overlapping chunks, embedded using OpenAI text-embedding-3-small, and stored in PostgreSQL with pgvector. At query time,

Agreemind combines vector similarity search and PostgreSQL full-text search using Reciprocal Rank Fusion, expands retrieved chunks with neighboring context, and uses GPT-4o to answer user questions over stored agreements. The response includes both the generated answer and source metadata so that users can relate the answer back to the relevant document excerpts.

The final system also includes version comparison and highlighted difference reporting. Documents can be linked through version chains using parent document references, version numbers, and latest-version flags. Users can compare two documents through a dedicated backend endpoint, and the comparison service identifies textual differences as well as heuristic changes involving dates, money amounts, and document structure. This supports the original project goal of helping users understand how agreements change over time.

The project was tested using a comprehensive automated pytest-based integration test suite. The final test suite contains twenty report-level test cases implemented as thirty-nine individual test functions. These tests cover authentication, document upload, text analysis, risk span generation, highlighted PDF retrieval, folder operations, version comparison, browser extension submission, validation failures, performance behavior, corrupted PDF handling, duplicate registration, logout behavior, cross-user data isolation, and resilience for non-analyzable extension content. All thirty-nine tests pass in the implemented test environment.

This final report presents the completed state of Agreemind. It first refines the system requirements according to the implemented product, then explains the final architecture, implementation details, testing process, maintenance plan, ethical and professional responsibilities, teamwork process, acquired knowledge, and future work. The purpose of the report is to provide a self-contained description of the system as implemented, including the design decisions made after the detailed design report and the final status of the project.

2. Requirements Details

The requirements of Agreemind were refined throughout the development process as the system evolved from the initial design into a fully implemented multi-platform legal agreement analysis system. The final requirements reflect the implemented behavior of the web application, mobile application, browser extension, backend services, analysis pipeline, RAG chat system, version

comparison system, and document management features. The requirements are divided into functional and non-functional requirements. Functional requirements describe the services and behaviors that the system provides to users, while non-functional requirements describe the quality attributes and constraints that guide the implementation.

2.1. Functional Requirements

2.1.1. User authentication and data ownership

The system shall allow users to register, log in, and access protected features through JWT-based authentication. Every protected backend operation shall resolve the authenticated user before executing, and all document, folder, analysis, chunk, and comparison queries shall be scoped to the current user. The system shall reject invalid credentials, duplicate registrations, and attempts by one user to access another user's documents.

2.1.2. Multi-source agreement submission

The system shall allow users to submit agreements through the web application, mobile application, browser extension, pasted text input, PDF upload, and mobile camera scanning. The backend shall validate inputs before processing, including rejecting empty or whitespace-only text and rejecting unsupported file uploads. Uploaded PDFs and scanned images shall be converted into analyzable text before entering the common analysis pipeline.

2.1.3. Background document processing and status tracking

The system shall process long-running analysis and ingestion tasks in the background so that upload requests return quickly. Each document shall maintain separate analysis_status and ingestion_status fields, together with error fields for failed processing. Clients shall be able to poll these statuses, display progress or failure states, and notify mobile users when background analysis completes.

2.1.4. Contract type detection and specialized analysis

The system shall automatically classify documents into supported contract types, including NDA, English banking, Turkish banking, Terms of Service, and general contracts. Users shall

also be able to override the detected type before submission. Based on the selected type, the backend shall route the document to the appropriate specialized engine rather than using a single generic model for all agreements.

2.1.5. Risk, rights, and timing extraction

The system shall analyze agreement text at clause or semantic-atom level to detect risky clauses, user rights, and time-sensitive obligations. Risk findings shall include categories, severity, confidence, rationale, and source spans. Rights findings shall describe actionable entitlements and conditions. Timing findings shall identify deadlines, renewal periods, payment obligations, effective dates, and notice windows, with normalized duration information when possible.

2.1.6. LLM refinement, risk scoring, and clause rewriting

The system shall support an optional second-stage LLM refinement process that improves deterministic risk, rights, and timing findings when API keys are available. The system shall compute an aggregate document risk score using severity and confidence values and provide a per-category breakdown. When the user enables clause rewrites, the system shall generate risk-reducing alternative clauses and projected post-rewrite severity values for risky findings.

2.1.7. Traceable analysis and document highlighting

The system shall preserve absolute character offsets for analysis findings so that results can be traced back to the original contract text. These spans shall be used to support highlighted PDF viewing and evidence-based result inspection. If an engine returns evidence text without valid offsets, the backend shall attempt to recover the span through string-search fallback to maintain reliable source grounding. The mobile application shall also support viewing the original uploaded document or extracted document text inside the app, including inline highlights for risks and rights where available.

2.1.8. Vault, folders, alerts, and version comparison

The system shall provide a personal vault where users can store, retrieve, search, and organize analyzed agreements. Users shall be able to create folders, move documents between folders, view extracted alerts, persist deadlines across mobile sessions, export deadlines to the native

calendar, and compare any two selected documents or document versions. Document comparison shall validate ownership and report textual, date, money, and structural changes between the selected documents.

2.1.9. Mobile preferences, detector configuration, and app security

The mobile application shall provide configurable user preferences for appearance, notifications, privacy and security, analysis defaults, and calendar behavior. It shall also allow users to configure the background agreement detector, including detection sensitivity, alert style, detected document categories, flagged danger types, and ignored applications. The mobile app shall support PIN-lock and auto-lock behavior to protect sensitive agreement data on shared or unattended devices.

2.1.10. Retrieval-augmented chat over stored agreements

The system shall allow users to ask natural-language questions across their stored agreements. Documents shall be chunked, embedded, and stored in PostgreSQL with pgvector. At query time, the backend shall combine vector similarity search and full-text search using Reciprocal Rank Fusion, expand retrieved chunks with neighboring context, and generate grounded answers with source metadata.

2.1.11. Local and API-independent operation

The system shall support a local operation mode in which document analysis can run without external AI API calls. When local mode is enabled, deterministic transformer-based engines shall continue to run locally, while LLM-dependent features such as summarization, RAG answer generation, and generative text services shall use a locally hosted Ollama-compatible model instead of OpenAI or Google services. The frontend shall allow users to toggle this mode and route requests to the configured local backend.

2.1.12. Automated testing and validation coverage

The system shall include automated integration tests covering the major implemented workflows, including authentication, document upload, text analysis, browser extension submission, folder operations, version comparison, validation failures, corrupted document handling, performance

behavior, logout behavior, and cross-user data isolation. The final test suite shall be executable with pytest and shall verify that the implemented system satisfies the major functional requirements.

2.2. Non-Functional Requirements

2.2.1. Usability

The system shall present legal analysis results in a form understandable to non-expert users. Summaries, risk findings, rights, reminders, and rewrite suggestions shall be organized clearly with readable labels, risk levels, and evidence text. The web, mobile, and extension interfaces shall support different user workflows while preserving a consistent interpretation of the same backend analysis results.

2.2.2. Security

The system shall protect user accounts and private documents through password hashing, JWT-based authentication, and user-scoped backend queries. A user shall not be able to retrieve, compare, modify, or view another user's documents or analysis results. Sensitive document workflows shall be designed with explicit ownership checks at the API and database-query level. On mobile, PIN-lock and auto-lock features further protect sensitive documents when the device is shared, lost, or left unattended.

2.2.3. Privacy

The system shall treat uploaded agreements and generated analysis results as user-owned private data. Documents, extracted text, analysis outputs, embeddings, and chat context shall only be used to provide the requested system functionality. Since legal documents may contain personal, financial, or contractual information, the system shall avoid unnecessary exposure of document content and shall communicate that outputs are informational rather than legal advice.

2.2.4. Reliability

The system shall handle expected failure cases without crashing or producing misleading results. Invalid credentials, duplicate registrations, empty text submissions, unsupported files, corrupted

PDFs, missing comparison parameters, non-existent documents, and non-analyzable webpage content shall return controlled error states. Failed analysis or ingestion shall be reflected through status and error fields instead of being silently ignored.

2.2.5. Performance

The system shall keep user-facing upload interactions responsive by offloading long-running analysis, OCR, embedding generation, clause rewriting, and ingestion tasks to background processing. Performance is improved through hybrid PDF extraction, where digital PDFs are parsed directly with PyMuPDF and OCR is used only for scanned pages. The analysis pipeline also uses batched GPU inference for transformer-based risk classification, concurrent execution of summarization and contract analysis, and built-in timing instrumentation for identifying slow pipeline stages. For the implemented test suite, upload response behavior is verified against a response-time threshold of 1.5 seconds.

2.2.6. Maintainability

The system shall be structured into modular backend services, API routers, analysis engines, client screens, and data models so that individual components can be modified without rewriting the entire application. Separate engines for NDA, Turkish banking, English banking, Terms of Service, and general contracts shall make it possible to update domain-specific logic independently.

2.2.7. Extensibility

The architecture shall allow additional document types, risk labels, ML models, rule-based extractors, LLM providers, and client features to be added later. The document router, engine abstraction, LLM client abstraction, RAG service, and structured result schema shall support future expansion without requiring fundamental redesign of the system.

2.2.8. Portability

The system shall run across multiple client environments, including web browsers, native mobile platforms, and Chromium-based browsers through the extension. The backend, frontend,

database, and pgAdmin services shall be orchestrated through Docker Compose to support reproducible local deployment. In addition, the system shall support fully local operation through Ollama-based LLM fallback, allowing the main analysis and chat workflows to run on a single machine without depending on OpenAI or Google API availability.

2.2.9. Traceability

The system shall link generated findings back to the original agreement text using spans, evidence text, section information, and source metadata. Risk findings, rights, reminders, chat answers, and comparison results shall remain grounded in the underlying document content so that users can inspect the basis of system outputs instead of relying only on generated summaries.

2.2.10. Testability

The system shall be testable through automated integration tests that exercise backend workflows under controlled conditions. Heavy ML, OCR, and vector database dependencies may be mocked in the test environment to make tests deterministic and executable during development. The final implemented suite shall verify both successful workflows and negative cases such as validation failures, corrupted inputs, and unauthorized access.

3. Final Architecture and Design Details

3.1. Overall System Architecture

The final architecture of Agreemind follows a layered and modular design that separates the system into client, backend/API, core processing, storage, and external AI/model service layers. This structure preserves the architectural direction introduced in the detailed design report while updating it to match the final implemented system. In the final version, Agreemind is no longer limited to web and mobile access; it also includes a browser extension for analyzing legal agreement pages directly from the browser. The backend remains the central coordination point for authentication, document management, analysis execution, vault operations, version comparison, and document-based chat. The final architecture is shown in Fig. 1.

The final architecture also supports two execution modes: cloud mode and local mode. In cloud mode, external providers such as OpenAI or Google Gemini may be used for summarization, LLM refinement, embeddings, clause rewriting, and RAG answer generation. In local mode, the deterministic transformer engines still run locally as before, while LLM-dependent services are redirected to an Ollama-compatible local model runtime. This allows Agreemind to operate without external AI API calls and improves privacy, portability, and resilience against provider availability or pricing changes.

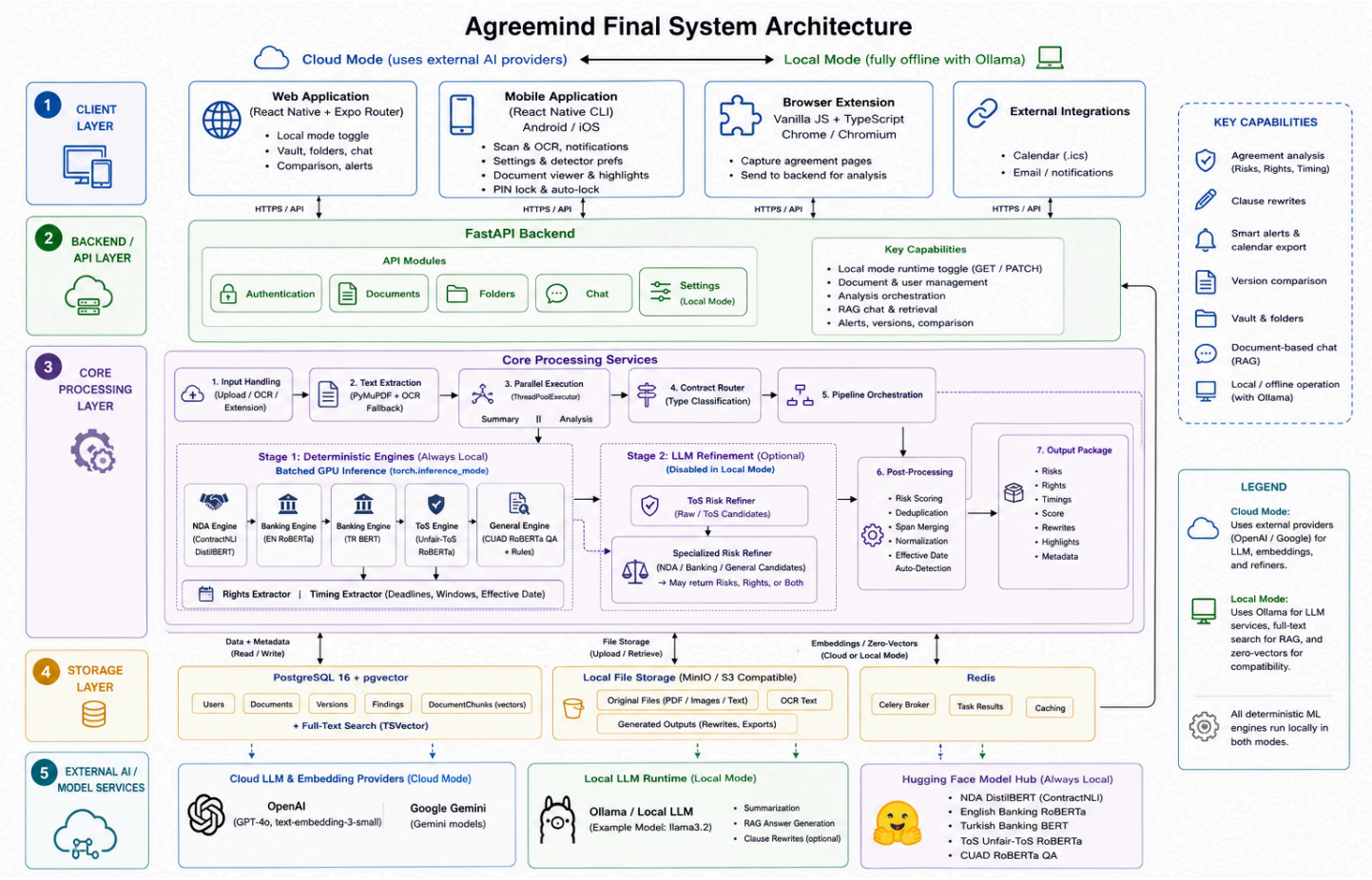

Figure 1: Agreemind Final System Architecture

At a high level, all three clients communicate with a shared FastAPI backend through authenticated HTTP requests. The backend exposes versioned API routes for authentication, documents, folders, and chat. Long-running tasks such as document analysis, OCR, embedding generation, RAG ingestion, and clause rewriting are handled through background processing so

that client interactions remain responsive. The backend also enforces ownership checks and input validation before accessing or modifying user-specific data.

The core processing layer contains the main intelligence of the system. Submitted agreements are converted into text, classified by document type, segmented into analyzable units, and processed by specialized analysis engines for risk detection, rights extraction, and timing extraction. Optional LLM refinement and clause rewriting improve the user-facing quality of results when external model services are available. In parallel, the system ingests document chunks and embeddings into the RAG subsystem so that users can ask natural-language questions over stored agreements.

Persistent data is stored primarily in PostgreSQL 16 with pgvector support [14], [15]. The database stores users, documents, folders, analysis results, document chunks, and vector embeddings, while uploaded files and generated artifacts are kept in local upload storage for the implemented system. The complete local development and demonstration environment is orchestrated through Docker Compose, which includes the database, backend, frontend, and pgAdmin services. This layered design makes the system easier to maintain, test, and extend while allowing the same backend capabilities to be used consistently by the web app, mobile app, and browser extension.

3.2. Client Layer Design

The final client layer consists of three user-facing surfaces: the web application, the native mobile application, and the browser extension. All clients communicate with the same FastAPI backend through authenticated HTTP requests and use JWT tokens for protected operations. This keeps authentication, document analysis, vault management, folder operations, version comparison, and chat behavior consistent across platforms.

The web application is implemented with React Native and Expo Router and runs in the browser through react-native-web. It supports the main document-management workflow, including login, dashboard access, document upload, analysis result viewing, vault browsing, alerts, comparison, and document chat. File-based navigation is preserved from the detailed design stage.

The web application also includes a local-mode configuration context that stores whether local mode is enabled and which local backend URL should be used. This configuration is persisted in localStorage and synchronized with the backend settings API, allowing users to switch between cloud-backed operation and a fully local backend from the interface.

The mobile application is implemented with React Native CLI for Android and iOS and uses tab-based navigation with Home, Documents, Vault, Alerts, and More sections. In the final system, the mobile app supports camera-based document scanning, OCR submission, background status polling, local completion notifications, deadline alerts, calendar export, folder management, document type override, clause rewrite display, and global vault chat. It also includes a full settings screen for appearance, notifications, privacy/security, analysis defaults, and calendar defaults; a detector preferences screen for configuring the background agreement detector; an original document viewer; inline extracted-text highlighting; PDF annotation overlays; risk glance bars; skeleton loading states; and PIN-lock with auto-lock timers.

The browser extension is the third client surface added after the detailed design report. It is implemented with Vite, React, and TypeScript for Chrome/Chromium browsers. The extension detects likely legal agreement pages using keyword heuristics, extracts readable webpage text with Mozilla Readability.js, submits the text to /api/v1/documents/analyze-text, and displays summary, risk, and rights results in a tabbed popup.

Across all clients, long-running backend operations are handled asynchronously. After submission, clients poll analysis_status and ingestion_status fields instead of blocking until processing is complete. This design allows document analysis, OCR, model inference, clause rewriting, and RAG ingestion to run in the background while keeping the user interface responsive.

3.3. Backend and API Layer Design

The backend is implemented as a FastAPI application and serves as the central coordination layer between the clients, processing services, configuration layer, and database. All API routes are versioned under /api/v1, and the final system includes route modules for authentication, documents, folders, chat, and settings. The settings API exposes runtime configuration for local

mode, allowing the frontend to read and update whether the backend should use cloud providers or local Ollama-backed execution.

The authentication API manages user registration, login, token issuance, and user profile access. Passwords are hashed before storage, and protected endpoints require JWT bearer authentication. Each protected request resolves the current user before executing business logic, allowing the backend to enforce ownership checks consistently across documents, folders, analyses, chunks, chat queries, and comparisons.

The settings API provides GET and PATCH operations for runtime configuration. The main settings introduced for local operation are LOCAL_MODE, OLLAMA_BASE_URL, and OLLAMA_CHAT_MODEL. When LOCAL_MODE is enabled, backend services branch away from external OpenAI or Google calls and use Ollama-compatible local model endpoints where applicable.

The document API handles the main document lifecycle. It supports PDF upload, text-based analysis, document status retrieval, analysis result retrieval, highlighted PDF access, version history, and document comparison. It also performs input validation, including rejection of empty text submissions and non-PDF uploads for the document-upload endpoint. Long-running document processing is scheduled as a background task, and the document record stores separate analysis_status, analysis_error, ingestion_status, and ingestion_error fields.

The folder API supports vault organization by allowing users to create folders, browse folder structures, and move documents between folders. The chat API exposes the RAG-based question-answering functionality over stored agreements. In both cases, database queries are scoped by the authenticated user so that one user cannot access another user’s vault contents, folders, document chunks, or analysis results.

Overall, the backend layer is designed to keep client logic simple while centralizing security, validation, ownership enforcement, and workflow coordination. The clients submit requests and display structured results, while the backend manages the complete document lifecycle from ingestion and analysis to retrieval, comparison, and chat.

3.4. Core Analysis Pipeline Design

The core analysis pipeline is the main intelligence layer of Agreemind. After a document is submitted, the background analysis service extracts text from the input, runs summarization and contract analysis concurrently where possible, routes the document to the appropriate contract engine, performs clause-level analysis, computes the final risk score, and stores the structured result. The pipeline supports both digital PDFs and image-based scans. For PDFs, the implementation first attempts direct PyMuPDF text extraction and falls back to OCR only for scanned or low-text pages.

The pipeline follows a two-stage design. In the first stage, deterministic NLP and machine learning components always run. The ContractSegmenter divides the document into semantic atoms with absolute character offsets. The selected risk engine detects risky clauses, the RightsExtractor identifies actionable user rights, and the TimingExtractor detects deadlines, notice periods, renewal windows, payment obligations, and effective dates. These components produce structured candidates with evidence text, confidence values, and source spans.

The second stage is optional and depends on the active execution mode. In cloud mode, risk, rights, and timing candidates may be passed to LLM refiners to reduce false positives, improve rationales, normalize deadlines, and make explanations clearer. In local mode, the pipeline disables Stage 2 LLM refinement and runs only the deterministic Stage 1 engines. As a result, the system still produces risk, rights, timing, scoring, and span-grounded outputs without relying on external LLM APIs, but does not apply LLM-enhanced refinement to those findings.

The LLM refinement layer now contains two forms of refinement. The ToS Risk Refiner operates over raw or lightly processed ToS risk candidates, while the Specialized Risk Refiner is used for pre-labelled candidates produced by the NDA, Banking, and General engines. The specialized refiner verifies or discards local model findings, re-rates severity in context, writes user-facing rationales, and may surface a candidate as a risk, a right, or both.

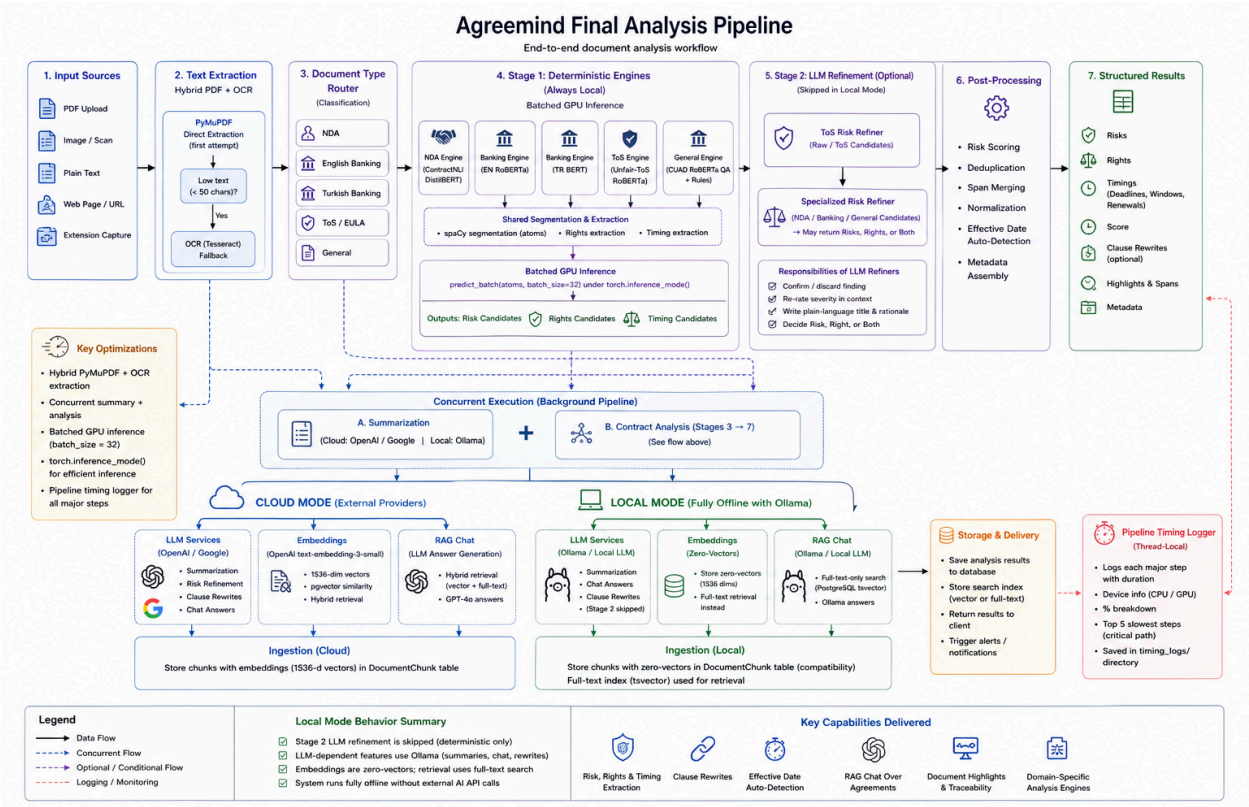

Figure 2: Agreemind Final Analysis Pipeline

A key refinement in the final design is contract-type-specific routing. The router classifies documents into NDA, English banking, Turkish banking, Terms of Service, or general contracts, and then selects the corresponding engine. The NDA engine uses an NLI-style ContractNLI model, the Turkish banking engine uses a fine-tuned Turkish BERT classifier, the English banking engine uses a fine-tuned RoBERTa classifier, the ToS engine uses a LexGLUE Unfair-ToS RoBERTa classifier, and the general engine uses a CUAD-based question-answering model with supplementary rules.

The final pipeline also includes risk scoring and optional clause rewriting. Risk scores are computed from severity and confidence values and aggregated into an overall score with a per-category breakdown. If clause rewrites are enabled by the user, risky clauses are passed to a rewrite service that generates risk-reducing alternative language and estimates the projected severity after replacement. All final outputs are stored in a structured analysis schema containing risks, rights, reminders, score metadata, and processing metadata.

Risk classification is optimized through batched GPU inference. Instead of processing atoms one by one, the pipeline groups atoms into padded batches, controlled by the gpu_batch_size configuration parameter, and performs transformer inference under torch.inference_mode().

The timing extraction stage also attempts to auto-detect the agreement's effective date. When a valid effective date is found and the document does not already have one, the backend persists it in the Document.effective_date field.

3.5. RAG Chat System Design

The retrieval-augmented generation chat system allows users to ask natural-language questions across their stored agreements. It is implemented as a separate backend service connected to the document ingestion workflow and exposed through the /api/v1/chat route. The purpose of this subsystem is not to generate general legal advice, but to answer questions using the user's own stored document content.

During ingestion, the extracted document text is split into sentence-aware chunks of approximately 1000 characters with 200-character overlap. Each chunk preserves metadata such as document_id, user_id, document_name, chunk_index, and page_number. In cloud mode, chunks are embedded using OpenAI text-embedding-3-small, producing 1536-dimensional vectors stored in the DocumentChunk table. In local mode, the system stores zero-vectors for compatibility with the schema and relies on PostgreSQL full-text indexing rather than external embedding generation.

At query time, the retrieval strategy depends on the execution mode. In cloud mode, the system uses hybrid retrieval by combining pgvector cosine similarity with PostgreSQL full-text search and merging results through Reciprocal Rank Fusion. In local mode, the system uses a full-text-only retrieval function, avoiding the need for external embedding APIs while preserving document-grounded search behavior.

The final answer is generated by GPT-4o in cloud mode or by the configured Ollama chat model in local mode. In both cases, the model is prompted to answer only from retrieved agreement excerpts, avoid unsupported speculation, and state when the available context is insufficient. The

response returned to the client includes both the generated answer and structured source metadata.

3.6. Data Model and Storage Design

The final data model is centered on three main persistent entities: Document, DocumentAnalysis, and DocumentChunk. The Document model stores ownership and document-level metadata, including filename, display name, description, privacy flag, folder assignment, file size, page count, MIME type, version information, detected document type, effective date, and processing status fields. The separate analysis_status, analysis_error, ingestion_status, and ingestion_error fields allow the frontend to distinguish between analysis failures and RAG ingestion failures.

The DocumentAnalysis model stores the structured analysis output as JSON. This output includes detected risks, rights, reminders, aggregate risk score, risk level, per-category risk breakdown, and metadata about the engine and processing run. Storing the result as JSON provides flexibility because different contract engines may produce different metadata while still conforming to a shared top-level schema.

The DocumentChunk model supports the RAG subsystem. Each chunk stores its document reference, user reference, document name, chunk index, page number, text content, and 1536-dimensional embedding vector. These embeddings are stored directly in PostgreSQL using pgvector, allowing semantic retrieval and relational ownership filtering to be handled in the same database. In local mode, the embedding field is populated with a zero-vector placeholder so that the same database schema can be reused while retrieval is handled through PostgreSQL full-text search.

The storage layer uses PostgreSQL 16 with pgvector as the primary database. This avoids introducing a separate vector database and keeps user records, document metadata, folder structures, analysis results, and embeddings under one persistence layer. For local development and demonstration, the system is orchestrated through Docker Compose with containers for PostgreSQL/pgvector, the FastAPI backend, the Expo web frontend, and pgAdmin.

4. Development/Implementation Details

4.1. Development Environment and Repository Structure

Agreemind was developed as a multi-component software system consisting of client applications, a backend service, analysis modules, and a separate machine learning training repository. The main application repository contains the FastAPI backend, the Expo-based web frontend, the React Native CLI mobile application, and the browser extension. The separate agreemind-ml repository contains the training, evaluation, inference, and HuggingFace upload scripts for the specialized legal-domain models.

The backend implementation is organized around the backend/app/ directory. API routes are placed under backend/app/api/v1/, including auth.py, documents.py, folders.py, and chat.py. Core processing logic is separated into services such as the background analysis service, document analysis service, clause rewrite service, RAG service, version service, and comparison service. This separation keeps request handling, business logic, model inference, and persistence concerns modular.

The browser extension is located in the extension/ directory and is implemented with Vite, React, and TypeScript. It includes content-script logic for webpage detection and extraction, popup UI components for displaying analysis results, and communication logic for submitting extracted webpage text to the backend. The mobile application is implemented separately from the Expo web application using React Native CLI, which allowed native Android/iOS functionality such as camera scanning, local notifications, calendar export, and AsyncStorage-based alert persistence.

For local deployment and demonstration, the system uses Docker Compose. The implemented compose environment contains four services: db, using pgvector/pgvector:pg16; backend, running the FastAPI application with uvicorn and hot reload; frontend, running the Expo web build; and pgadmin, used for database inspection and management. This environment made it possible to run the main application stack consistently during development and testing.

The Docker Compose configuration also supports GPU-enabled backend execution. The backend container declares NVIDIA GPU device reservations so that transformer inference can use GPU

acceleration when the host provides the NVIDIA Container Toolkit. The compose configuration also exposes LOCAL_MODE, OLLAMA_BASE_URL, and OLLAMA_CHAT_MODEL environment variables, allowing the same deployment setup to run either cloud-backed or locally backed LLM workflows.

4.2. Backend Implementation

The backend was implemented with FastAPI and serves as the central service layer for authentication, document processing, vault management, folder operations, comparison, and chat [13]. The API is organized under the /api/v1 prefix with four route modules: auth.py, documents.py, folders.py, and chat.py. This structure keeps each major workflow separated while still exposing a unified API to the web application, mobile application, and browser extension.

A settings route was added under /api/v1/settings to support runtime configuration of local mode. The backend configuration now includes LOCAL_MODE, OLLAMA_BASE_URL, and OLLAMA_CHAT_MODEL. These settings determine whether LLM-dependent services call external providers or an Ollama-compatible local model endpoint. This branching is applied across summarization, RAG chat, and other LLM-backed services.

Authentication is implemented using password hashing and JWT bearer tokens. During login, the backend verifies the user credentials and issues a signed token. Protected endpoints use a shared current-user dependency to decode the token, retrieve the user record, and enforce user-specific access. This same ownership check is applied to document retrieval, folder operations, analysis access, version comparison, and RAG chat queries.

The document route implements the main lifecycle of an agreement. When a valid PDF, image scan, or text submission is received, the backend creates a document record and schedules processing through a FastAPI background task. The document stores separate analysis_status and ingestion_status fields so that the frontend can poll analysis progress independently from RAG ingestion progress. If either stage fails, the corresponding error field is updated instead of silently failing.

PDF extraction uses a hybrid PyMuPDF-based strategy. The backend first attempts direct text extraction for each page and only renders a page for Tesseract OCR if the extracted text is below a minimum threshold. This improves speed and text quality for digital PDFs while preserving support for scanned documents.

The background pipeline also parallelizes independent work where possible. Since summarization and contract analysis both depend only on the extracted full text, they are submitted concurrently using a ThreadPoolExecutor with two workers. This reduces total processing time compared with running summarization and analysis sequentially.

Input validation was added to prevent invalid processing requests. The /analyze-text endpoint rejects empty or whitespace-only text with HTTP 400. The document upload endpoint rejects unsupported non-PDF files by checking both MIME type and file extension. Corrupted or unreadable PDFs are handled during processing by transitioning the document into a failed state rather than returning misleading analysis results.

The backend also exposes document-management operations used by the vault and comparison features. Folder operations allow users to organize documents into hierarchies, while versioning fields allow uploaded agreements to be linked as document versions. The comparison endpoint validates ownership of both selected documents before generating a diff report. This backend-centered design keeps the clients lightweight and ensures that security, validation, and document lifecycle logic are enforced consistently.

4.3. Analysis Pipeline Implementation

The analysis pipeline was implemented as the core backend processing workflow that transforms raw agreement input into structured legal insights. The pipeline is executed by the background analysis service after a document record is created. It first extracts text from the submitted input, using PDF text extraction for digital PDFs and OCR for scanned images. The extracted text is then summarized, classified by document type, segmented, analyzed, optionally refined, scored, and persisted as a structured DocumentAnalysis record.

The first analysis stage consists of deterministic NLP and machine learning components. The ContractSegmenter splits the agreement into semantic atoms such as sentences, list items, and clause fragments while preserving absolute character offsets. These atoms are then processed by three analysis lanes: risk detection, rights extraction, and timing extraction. The risk lane uses the engine selected by the document router, while the rights and timing lanes identify actionable entitlements, deadlines, renewal windows, notice periods, payment obligations, and effective dates.

Risk model inference was refactored from per-atom prediction to batched GPU inference. The new predict_batch(atoms, batch_size=32) method tokenizes atoms as padded batches, runs a single transformer forward pass for each batch, and returns predictions for all atoms. The gpu_batch_size pipeline configuration controls this behavior separately from LLM prompt batching, and inference is executed under torch.inference_mode() to reduce overhead.

Segmentation and auxiliary extraction were also refactored through a shared _segment_and_extract() method, centralizing spaCy-based segmentation together with rights and timing extraction across ToS, NDA, banking, and general analysis methods.

The second stage uses optional LLM refinement when provider credentials are available and local mode is not active. The implementation includes both general refinement and a specialized refiner for domain-engine candidates. The specialized refiner receives already-labelled candidates from the NDA, Banking, and General engines, verifies or discards them, re-rates severity, writes plain-language rationale text, and may return both RiskFinding and RightFinding objects from a single candidate. When local mode is enabled, this refinement stage is skipped and the deterministic Stage 1 findings are returned directly.

After refinement, the scoring module computes the final risk score. Each finding contributes according to its severity and confidence, and category scores are aggregated into an overall bounded score between 0 and 1. If clause rewriting is enabled, risky clauses are processed by the clause rewrite service, which generates risk-reducing alternative wording and estimates the projected severity after replacement.

A major implementation detail is high-precision offset tracking. All engines attempt to preserve absolute character spans so that findings can be highlighted in the original document. When an engine provides evidence text but invalid offsets, the backend attempts to recover the span through string search. This ensures that analysis results remain traceable to the original agreement text and supports the highlighted PDF workflow.

4.4. Machine Learning Model Implementation

The machine learning implementation was separated into domain-specific pipelines rather than a single universal classifier. This decision was made because different contract categories contain different clause structures, terminology, and risk patterns. The final system therefore uses specialized engines for NDAs, Turkish banking contracts, English banking contracts, Terms of Service/EULA documents, and general contracts.

For Terms of Service and EULA-style agreements, the system uses Agreemind/lexglue-roberta-unfair-tos, a RoBERTa-base model fine-tuned on the LexGLUE Unfair-ToS dataset. This engine detects unfair clause categories such as limitation of liability, unilateral termination, unilateral change, content removal, contract-by-using, choice of law, jurisdiction, and arbitration.

For NDA analysis, the system uses Agreemind/contractnli-distilbert-nda, a DistilBERT model fine-tuned on the ContractNLI task. The engine evaluates confidentiality-related hypotheses against each clause and detects whether important protections are present, absent, or contradicted. These outcomes are then mapped into risk findings using predefined severity rules.

For Turkish banking agreements, the system uses Agreemind/turkish-banking-bert, a Turkish BERT model fine-tuned on manually labeled Turkish banking contracts. The training pipeline includes dataset preparation, model training, cross-validation, inference scripts, and optional synthetic augmentation. The model supports fourteen banking-specific risk labels, including hidden fees, unilateral terms changes, broad collateral, default escalation, currency risk, and bundled insurance.

For English banking agreements, the system uses Agreemind/en-banking-roberta, a RoBERTa model fine-tuned on annotated English banking contracts. In addition to model inference, this engine includes extensive post-processing such as document noise filtering, clause splitting tuned for banking agreements, definition-section handling, per-label thresholds, overlap deduplication, severity adjustment, and suppression rules. These refinements improved the precision of English banking risk detection significantly during development.

For general contracts, the system uses a CUAD-based question-answering approach with supplementary rule-based analysis. The model checks for risk-relevant contract topics such as renewal terms, exclusivity, non-compete, termination rights, assignment restrictions, liability limits, liquidated damages, warranty terms, insurance, IP ownership, and license grants. Batch inference and metadata-question skipping were implemented to improve runtime performance.

All specialized models and training scripts are maintained in the separate agreemind-ml repository. The trained models are hosted under the Agreemind organization on HuggingFace Hub and are loaded by the backend engines during inference. This separation between model training and application integration makes the production backend easier to maintain while keeping the ML development workflow reproducible.

4.5. Client Implementation

The web client was implemented with React Native and Expo Router, running in the browser through react-native-web. It provides the main browser-based workflow for authentication, dashboard access, document upload, analysis result viewing, vault browsing, alerts, comparison, and document chat. The file-based navigation structure makes the web application easier to extend with new pages while keeping route organization clear.

The web client also includes an AppConfigContext that manages local-mode configuration. It stores isLocalMode, localUrl, and related setters, persists them in localStorage, synchronizes the state with the backend settings API, and routes frontend API calls to the configured local backend when local mode is active.

The mobile client was implemented separately using React Native CLI for Android and iOS. This allowed the project to support native device features such as camera-based document scanning, image upload for OCR, local push notifications, native calendar export, AsyncStorage persistence, and app-level security controls. The mobile UI uses tab-based navigation with Home, Documents, Vault, Alerts, and More sections.

The mobile implementation was expanded with a full settings screen, detector preferences screen, original document viewer, extracted document text view, PDF document view, risk glance bars, and additional reusable UI components. The settings screen manages appearance, notification preferences, privacy and security options, analysis defaults, and calendar defaults. The detector preferences screen allows users to configure background agreement detection sensitivity, alert style, detected document categories, danger types, and ignored apps.

For document inspection, the mobile app can display the original uploaded PDF, image, or raw text directly in the app. The extracted text view supports inline highlights for risks and rights, while the PDF view overlays risk and right annotations on top of the original document. The mobile app also includes risk-at-a-glance bars, skeleton loading states, typing indicators, reusable gradient headers, press animations, and a PIN-lock system with auto-lock timers.

The browser extension was implemented with Vite, React, and TypeScript. Its content script detects legal agreement pages through keyword heuristics, extracts readable text using Mozilla Readability.js, and submits the extracted text to the backend text-analysis endpoint [23]. The popup interface displays the returned analysis in Summary, Risks, and Rights tabs, allowing users to inspect webpage agreements without manually copying text into the web application.

Across all clients, API communication is handled through authenticated HTTP requests to the FastAPI backend. Clients store and attach JWT tokens for protected operations and rely on backend status fields for asynchronous workflows. This keeps client-side logic focused on presentation, navigation, polling, and user interaction, while the backend remains responsible for validation, processing, and data ownership enforcement.

4.6. RAG and Chat Implementation

The RAG chat system was implemented to let users ask questions over their stored agreements. During document ingestion, the backend splits extracted agreement text into sentence-aware chunks of approximately 1000 characters with 200 characters of overlap. Each chunk is stored with metadata such as document ID, user ID, document name, page number, and chunk index.

For semantic retrieval, each chunk is embedded using OpenAI text-embedding-3-small, which produces 1536-dimensional vectors [16]. These vectors are stored in PostgreSQL through the pgvector extension in the DocumentChunk table. Storing embeddings in PostgreSQL keeps vector search, full-text search, ownership filtering, and relational document metadata in the same persistence layer.

When the user submits a chat question, the backend retrieves relevant context through hybrid search. It performs vector similarity search using pgvector and keyword-based full-text search using PostgreSQL tsvector. The results from both retrieval methods are merged using Reciprocal Rank Fusion, and the strongest retrieved chunks are expanded with neighboring chunks to preserve local context.

The final answer is generated using GPT-4o with a system prompt that constrains the model to answer from the retrieved agreement excerpts. The response includes a direct answer and structured source metadata, including document name, document ID, chunk index, page number, and text preview. This allows the frontend to show which stored agreement excerpts supported the answer.

4.7. Version Comparison and Highlighting Implementation

The document comparison system was implemented through comparison_service.py and the /api/v1/documents/compare endpoint. Although the data model still supports version chains through parent_document_id, version_number, and is_latest_version, the final comparison interface was redesigned to allow users to select any two documents from their vault, not only two versions of the same document.

The comparison endpoint accepts two document IDs and validates that both documents exist and belong to the authenticated user. If the request contains missing parameters, non-existent documents, or documents owned by another user, the backend returns an appropriate error instead of generating a comparison. This prevents comparison results from leaking private document content across users. The compare screen provides a searchable document list and can also accept pre-selected document IDs from other screens through route parameters.

The comparison service generates a diff-based report that identifies added, removed, and modified content between two documents. It also applies heuristic checks for changes in dates, money amounts, and document structure, since these changes may be important even when the textual difference is small. The result is used to help users understand how a contract has changed across versions.

PDF highlighting was implemented using span-based analysis results. Risk findings store absolute character offsets that identify where the risky text appears in the original document. When valid offsets are unavailable but evidence text is present, the backend attempts to recover the span through string matching. These offsets allow the document viewer to connect analysis findings to the corresponding source text, making the analysis more inspectable and traceable.

4.8. Testing Implementation Support

The backend test suite was implemented with pytest in backend/tests/test_report_cases.py. The suite contains 20 report-level test cases implemented as 39 individual test functions. These tests cover the main workflows of the final system, including authentication, PDF upload, text analysis, highlighted PDF access, folder operations, version comparison, browser extension submission, validation failures, corrupted PDF handling, logout behavior, performance behavior, and cross-user data isolation.

To keep the tests deterministic and practical to run in a development environment, the test setup uses an in-memory SQLite database instead of the production PostgreSQL database. Heavy external dependencies such as ML models, OCR libraries, operating-system-specific components, and pgvector are mocked where necessary. The pgvector Vector type is replaced

with a LargeBinary mock so that schema-related tests can run without requiring the full production vector database stack.

The test suite verifies both successful workflows and negative cases. For example, it checks that users can register and log in, upload and retrieve analyzed documents, move documents between folders, compare versions, and submit pasted text. It also verifies that invalid credentials are rejected, empty text returns HTTP 400, unsupported file uploads return HTTP 400, non-existent comparison documents return errors, corrupted PDFs fail gracefully, and one user cannot access another user's documents.

A performance-oriented test also verifies that the upload response returns quickly when background processing is used. This supports the architectural decision to offload long-running analysis work instead of blocking the client until the full pipeline completes.

5. Test Cases and Results

Test ID	TC-01	Category	Functional	Severity	
Objective	Verify that a registered user can log in successfully and access the main dashboard.				
Steps	1. Open the Agreemind web application. 2. Navigate to the login page. 3. Enter a valid registered email address and password. 4. Click the Log In button.				
Expected	1. The web application loads without errors and displays the landing/login interface. 2. The login form is displayed correctly with fields for email and password. 3. The system accepts the entered credentials without validation errors.				

	4. The user is authenticated successfully and redirected to the main dashboard or vault page.
Date & Result	01.05.2026 Pass

Test ID	TC-02	Category	Functional	Severity	
Objective	Verify that a user can upload a PDF agreement and receive a summary and risk analysis.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the agreement upload page. 3. Select a valid PDF agreement file from local storage. 4. Submit the file for analysis. 5. Wait for the system to complete processing.				
Expected	1. The user session is active and the system allows access to upload functionality. 2. The upload interface is visible and ready to accept a document. 3. The system accepts the PDF file and begins processing without file format errors. 4. The uploaded agreement is sent to the backend successfully and an analysis task is created. 5. The system displays the generated plain-language summary together with detected risky clauses/categories.				

Date & Result	01.05.2026 Pass

Test ID	TC-03	Category	Functional	Severity	
Objective	Verify that risky clauses are highlighted correctly in the document viewer after analysis.				
Steps	1. Log in and upload or open an already analyzed agreement. 2. Wait until the analysis report page is displayed. 3. Click on one of the detected risk categories or flagged clauses. 4. Observe the corresponding location in the document viewer.				
Expected	1. The system opens the selected agreement and displays its associated analysis results. 2. The report page loads with a summary, risk list, and document viewer. 3. The selected risk item becomes active and triggers navigation to the related part of the agreement. 4. The relevant clause or text span is highlighted clearly in the document viewer, matching the selected risk item.				
Date & Result	02.05.2026 Pass				

Test ID	TC-04	Category	Data Integrity	Severity	
Objective	Verify that an analyzed agreement can be saved to the vault and retrieved				

	later through search.
Steps	1. Log in to the Agreemind system. 2. Upload and analyze an agreement, or open an existing analyzed agreement. 3. Save the agreement to the personal vault. 4. Navigate to the vault page. 5. Search for the agreement by title or keyword.
Expected	1. The user is authenticated and has access to vault operations. 2. The selected agreement is available in analyzed form and ready to be saved. 3. The system stores the agreement and confirms that it has been added to the personal vault. 4. The vault page displays saved agreements associated with the current user account. 5. The search returns the saved agreement correctly, and the user can reopen it without data loss.
Date & Result	02.05.2026 Pass

Test ID	TC-05	Category	Functional	Severity	
Objective	Verify that two versions of the same agreement can be compared and differences are displayed correctly.				

Steps	1. Log in to the Agreemind system. 2. Open the comparison feature from the dashboard or vault. 3. Select Version A and Version B of the same agreement. 4. Start the comparison process. 5. Review the comparison output.
Expected	1. The user is authenticated and can access the comparison functionality. 2. The comparison interface allows the user to choose two saved agreement versions. 3. Both versions are accepted by the system without compatibility or retrieval errors. 4. The system processes both versions and generates a comparison report. 5. The report clearly displays added, removed, or modified clauses, allowing the user to identify meaningful changes between the two versions.
Date & Result	02.05.2026 Pass

Test ID	TC-06	Category	Functional	Severity	
Objective	Verify that the system rejects invalid login credentials and prevents unauthorized access.				
Steps	1. Open the Agreemind web application. 2. Navigate to the login page. 3. Enter an invalid email address or incorrect password.				

	4. Click the Log In button.
Expected	1. The application loads successfully and shows the login interface. 2. The login form is displayed correctly. 3. The system accepts the input format and submits the login request. 4. The system denies authentication, displays an appropriate error message, and does not grant access to the dashboard or vault.
Date & Result	02.05.2026 Pass

Test ID	TC-07	Category	Functional	Severity	
Objective	Verify that the system supports agreement analysis through pasted text input.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the agreement input page. 3. Paste a valid Terms of Service or contract text into the text input field. 4. Submit the text for analysis. 5. Wait for the result page to load.				
Expected	1. The user is authenticated and has access to the agreement input page. 2. The input area is visible and accepts pasted text. 3. The system stores the pasted text temporarily and validates that it is non-empty. 4. The system processes the pasted agreement text without requiring a file upload.				

	5. The result page displays a summary and detected risky clauses/categories for the pasted text.
Date & Result	02.05.2026 Pass

Test ID	TC-08	Category	Functional	Severity	
Objective	Verify that the system handles empty or invalid pasted input gracefully.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the agreement input page. 3. Leave the text input field empty, or paste only whitespace. 4. Click the **Analyze** button.				
Expected	1. The user is logged in and the input page is accessible. 2. The text input field is displayed and editable. 3. The system accepts the button click and checks the provided input. 4. The system rejects the request, displays a validation message indicating that the agreement text cannot be empty, and does not start analysis.				
Date & Result	02.05.2026 Pass				

Test ID	TC-09	Category	Functional	Severity	
Objective	Verify that unsupported or invalid file types are rejected during upload.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the agreement upload page. 3. Select a file in an unsupported format (for example, .exe or another invalid type). 4. Attempt to submit the file for analysis.				
Expected	1. The user is authenticated and the upload page is accessible. 2. The upload interface is displayed correctly. 3. The system receives the selected file and checks its type before processing. 4. The system rejects the file, informs the user that the file format is unsupported, and does not create an analysis task.				
Date & Result	03.05.2026 Pass				

Test ID	TC-10	Category	Functional	Severity	
Objective	Verify that the browser extension can send a webpage for analysis and open the result in Agreemind.				
Steps	1. Open a webpage containing Terms of Service or other agreement text				

	in the browser. 2. Click the Agreemind browser extension icon. 3. Trigger the analysis action from the extension interface. 4. Wait for the extension to send the page content or URL to the backend. 5. Open the generated result in the Agreemind web application.
Expected	1. The webpage loads successfully and the extension is available in the browser. 2. The extension interface opens without errors and recognizes the current page context. 3. The extension successfully starts the analysis request. 4. The system receives the page information and processes it without requiring manual copy-paste. 5. The analysis result opens in Agreemind and displays the expected summary and risk findings for the selected webpage.
Date & Result	03.05.2026 Pass

Test ID	TC-11	Category	Functional	Severity	
Objective	Verify that a new user can create an account successfully through the sign-up flow.				
Steps	1. Open the Agreemind web application. 2. Navigate to the sign-up page.				

	3. Enter valid registration information, including email and password. 4. Submit the sign-up form. 5. Wait for account creation to complete.
Expected	1. The application loads successfully and displays the authentication interface. 2. The sign-up form is accessible and fields are displayed correctly. 3. The system accepts the entered registration information without validation errors. 4. The system creates a new user account and confirms successful registration. 5. The user is redirected to the dashboard or prompted to log in with the new account.
Date & Result	03.05.2026 Pass

Test ID	TC-12	Category	Functional	Severity	
Objective	Verify that a logged-in user can log out successfully and lose access to protected pages.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the main dashboard or vault page. 3. Click the Log Out button. 4. Attempt to revisit a protected page such as the vault or dashboard.				

Expected	1. The user is authenticated successfully and the dashboard is accessible. 2. Protected pages are visible only to the authenticated user. 3. The system ends the user session and redirects the user to the login or landing page. 4. Access to protected pages is denied until the user logs in again.
Date & Result	03.05.2026 Pass

Test ID	TC-13	Category	Functional	Severity	
Objective	Verify that the vault preserves the correct association between agreements and their analysis results after multiple uploads.				
Steps	1. Log in to the Agreemind system. 2. Upload and analyze two different agreements. 3. Save both agreements to the personal vault. 4. Open each saved agreement from the vault one by one. 5. Compare the displayed summaries and risk findings with the originally generated results.				
Expected	1. Log in to the Agreemind system. 2. Upload and analyze two different agreements. 3. Save both agreements to the personal vault. 4. Open each saved agreement from the vault one by one. 5. Compare the displayed summaries and risk findings with the originally generated results.				

Date & Result	03.05.2026 Pass

Test ID	TC-14	Category	Functional	Severity	
Objective	Verify that searching the vault with a non-matching keyword is handled clearly and without errors.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the personal vault. 3. Enter a keyword that does not match any saved agreement title or content. 4. Submit the search query. 5. Observe the search results area.				
Expected	1. The user is authenticated and the vault page loads correctly. 2. The vault interface displays the search bar and previously saved agreements. 3. The system accepts the keyword and processes the search request. 4. The system completes the search without crashing or freezing. 5. The user is shown a clear “no results found” state instead of an empty or broken interface.				
Date & Result	03.05.2026 Pass				

Test ID	TC-15	Category	Functional	Severity	
Objective	Verify that the system responds within an acceptable time when analyzing a typical agreement document.				
Steps	1. Log in to the Agreemind system. 2. Upload a valid agreement document of typical size. 3. Submit the document for analysis. 4. Measure the elapsed time until the summary and risk results are displayed.				
Expected	1. The user session is active and upload functionality is available. 2. The system accepts the document without errors. 3. The backend begins processing the uploaded document and provides visible feedback that analysis is in progress. 4. The system completes analysis within an acceptable response time for a normal document and displays the results without timeout or failure.				
Date & Result	03.05.2026 Pass				

Test ID	TC-16	Category	Functional	Severity	
Objective	Verify that the sign-up process rejects registration when an already registered email address is used.				
Steps	1. Open the Agreemind web application.				

	2. Navigate to the sign-up page. 3. Enter an email address that is already associated with an existing account. 4. Fill in the remaining required fields with valid values. 5. Submit the sign-up form.
Expected	1. The application loads correctly and the sign-up page is accessible. 2. The sign-up form is displayed properly. 3. The system accepts the entered email format and other inputs. 4. The system checks whether the email is already registered. 5. The system rejects the registration request, informs the user that the email is already in use, and does not create a duplicate account.
Date & Result	03.05.2026 Pass

Test ID	TC-17	Category	Functional	Severity	
Objective	Verify that the system handles corrupted or unreadable PDF files gracefully during upload.				
Steps	1. Log in to the Agreemind system. 2. Navigate to the agreement upload page. 3. Select a corrupted or unreadable PDF file. 4. Submit the file for analysis. 5. Observe the system response.				

Expected	1. The user is authenticated and can access the upload page. 2. The upload interface is available and accepts the selected file. 3. The system receives the file and attempts to parse it. 4. The system detects that the PDF cannot be processed correctly. 5. The system reports the failure clearly to the user and does not generate a misleading summary or analysis result.
Date & Result	03.05.2026 Pass

Test ID	TC-18	Category	Functional	Severity	
Objective	Verify that the comparison feature handles invalid version selections safely.				
Steps	1. Log in to the Agreemind system. 2. Open the comparison page. 3. Select only one agreement version, or select an invalid/missing version combination. 4. Attempt to start the comparison process.				
Expected	1. The user is authenticated and the comparison page loads correctly. 2. The comparison interface is displayed and allows version selection. 3. The system accepts the user interaction and validates the selected versions before comparison starts. 4. The system prevents the comparison request from proceeding, informs the user that two valid versions are required, and avoids generating an invalid comparison report.				

Date & Result	04.05.2026 Pass

Test ID	TC-19	Category	Functional	Severity	
Objective	Verify that a user cannot access another user's saved agreements or analysis results.				
Steps	1. Log in as User A and save at least one analyzed agreement in the vault. 2. Log out from User A's account. 3. Log in as User B with a different account. 4. Navigate to the vault and attempt to access User A's saved agreement directly or indirectly.				
Expected	1. User A can authenticate and store agreements in their own vault. 2. User A's session is terminated successfully after logout. 3. User B is authenticated successfully and sees only their own account context. 4. The system prevents User B from viewing, opening, or retrieving any agreement or analysis result that belongs to User A.				
Date & Result	04.05.2026 Pass				

Test ID	TC-20	Category	Functional	Severity	
Objective	Verify that the browser extension handles unsupported or non-analyzable pages without crashing.				
Steps	1. Open a webpage where agreement text cannot be extracted properly (for example, a highly dynamic or unsupported page). 2. Click the Agreemind browser extension icon. 3. Attempt to start the analysis from the extension. 4. Observe the response shown by the extension or connected web app.				
Expected	1. The webpage loads and the extension remains available in the browser. 2. The extension interface opens correctly. 3. The system attempts to process the page content or URL and detects that extraction is not possible or insufficient. 4. The extension or web app informs the user clearly that analysis could not be completed for the current page and does not crash, freeze, or display invalid results.				
Date & Result	04.05.2026 Pass				

6. Maintenance Plan and Details

The maintenance plan for Agreemind focuses on keeping the system reliable, secure, extensible, and usable after the final implementation. Since the project contains multiple client applications,

a FastAPI backend, domain-specific ML engines, RAG infrastructure, and external AI integrations, maintenance must address both traditional software updates and model/pipeline updates.

6.1. Corrective Maintenance

Corrective maintenance refers to fixing defects discovered after deployment or demonstration. The automated test suite already provides a foundation for this by covering authentication, document upload, text analysis, folder operations, version comparison, validation failures, corrupted PDFs, browser extension submission, and cross-user data isolation. When a bug is found, the expected maintenance process is to reproduce the issue, add or update a regression test, fix the implementation, and rerun the full pytest suite before merging the change.

For runtime failures, the backend already stores separate analysis_error and ingestion_error fields. These fields should be used to identify whether a failure occurred during document analysis, OCR, LLM refinement, clause rewriting, or RAG ingestion. This separation makes debugging easier because a failed analysis and a failed embedding-ingestion process do not appear as the same error state to the developers or the frontend.

6.2. Adaptive Maintenance

Adaptive maintenance will be required when external dependencies, platforms, APIs, or model providers change. Agreemind depends on frameworks and services such as FastAPI, React Native, Expo, browser-extension APIs, PostgreSQL, pgvector, OpenAI, Google Gemini, HuggingFace Transformers, and mobile OS features such as notifications, camera access, and calendar export. These dependencies should be updated carefully, with compatibility testing after each major upgrade.

The browser extension may require additional adaptation as Chrome/Chromium extension APIs evolve. Similarly, the mobile application may require updates when Android or iOS changes permission rules for camera access, notifications, background behavior, or calendar integration. Because these features are isolated inside the client layer, platform-specific changes can be handled without redesigning the backend analysis pipeline.

The local-mode architecture also improves adaptive maintainability. If external AI providers change pricing, rate limits, model availability, or API behavior, Agreemind can continue operating through Ollama-backed local inference for summarization and chat, while deterministic transformer engines remain local. This reduces dependency on a single external provider and gives future deployments more flexibility.

6.3. Perfective Maintenance

Perfective maintenance aims to improve existing functionality without changing the system's core purpose. For Agreemind, this includes improving the user interface, making analysis explanations clearer, reducing false positives in risk detection, improving clause rewrite quality, and making alerts and comparison reports more understandable. User feedback should be used to identify confusing outputs, overly technical explanations, or workflows that require too many steps.

The analysis engines should also be improved over time using error analysis. Misclassified clauses, noisy findings, missed deadlines, and weak rewrite suggestions should be collected during testing and reviewed. The specialized-engine architecture makes this practical because improvements can be applied to one domain, such as English banking or NDA analysis, without affecting all other document types.

6.4. Preventive Maintenance

Preventive maintenance is necessary to reduce the probability of future failures. The backend should continue to enforce strict input validation, ownership checks, and clear error handling. New endpoints should follow the same pattern used in the final implementation: authenticate the user, validate input, check ownership, execute the requested operation, and return structured errors when necessary.

The project should also maintain clear separation between route handlers, services, models, and analysis engines. If new document types or risk categories are added, they should be

implemented as separate modules rather than being inserted directly into existing engines. This prevents the analysis code from becoming difficult to modify as the system grows.

6.5. Model and Pipeline Maintenance

The ML models require a separate maintenance process from the application code. Domain-specific models should be periodically evaluated on new examples, especially when new contract formats or legal language patterns are encountered. For the Turkish banking, English banking, NDA, and general contract pipelines, future updates should preserve the training scripts, evaluation scripts, labeled datasets, and HuggingFace upload workflow in the agreemind-ml repository.

Model updates should not be deployed only because they improve training metrics. They should also be checked against end-to-end system behavior, including precision of risk findings, severity calibration, span correctness, and usefulness of explanations. If a new model version changes label behavior, the backend thresholds, suppression rules, severity mappings, and frontend display assumptions may also need to be reviewed.

6.6. Data and Storage Maintenance

The database should be maintained through regular backups, schema migration tracking, and monitoring of storage growth. Since the system stores document metadata, analysis JSON, document chunks, and vector embeddings, storage requirements can increase quickly as users upload more agreements. Old failed records, temporary files, and debug outputs should be cleaned periodically to avoid unnecessary disk usage.

The pgvector indexes should also be monitored as the number of document chunks grows. If retrieval performance decreases, index parameters, chunk size, overlap size, or retrieval limits may need to be adjusted. Because embeddings are stored in PostgreSQL together with relational metadata, database maintenance directly affects both vault operations and RAG chat quality.

6.7. Security and Privacy Maintenance

Security maintenance is especially important because Agreemind processes private legal and financial documents. Password hashing, JWT handling, ownership checks, and protected route behavior should be reviewed whenever authentication or authorization code is modified. Any new feature that accesses documents, analyses, chunks, folders, or comparison results must include user-scoped database queries.

Privacy maintenance also requires careful handling of external AI calls. When document text is sent to OpenAI, Google Gemini, or other external services for summarization, embeddings, refinement, chat, or clause rewriting, the system should keep this behavior transparent to users. Future versions should maintain the principle that user documents are used only for requested functionality and not for unrelated training or sharing.

6.8. Deployment and Operational Maintenance

The Docker Compose setup provides a reproducible local deployment environment with the database, backend, frontend, and pgAdmin services. For future deployment, environment variables, API keys, database credentials, and model configuration should be documented and kept separate from the source code. Logs should be monitored for repeated analysis failures, API timeout patterns, failed external service calls, and database errors. GPU-enabled deployment should also be maintained where available. The backend container is configured with NVIDIA GPU device reservations so that local transformer inference can use GPU acceleration in compatible environments. Operational documentation should include how to configure LOCAL_MODE, OLLAMA_BASE_URL, and OLLAMA_CHAT_MODEL, including different host addresses required for Docker on Mac/Windows and Linux.

Before each release or demonstration, the team should run the automated backend test suite, verify that Docker Compose starts all required services, test at least one full upload-analysis-chat workflow, and confirm that the mobile and extension workflows still communicate with the backend correctly. This release checklist reduces the risk of integration failures between independently maintained components.

7. Other Project Elements

7.1. Consideration of Various Factors in Engineering Design

7.1.1. Constraints

- **Public Health**

Public health considerations had no direct impact on the design of Agreemind since the system is primarily a digital legal analysis platform rather than a healthcare or safety-critical system.

Impact Level: **0 / 10**

- **Safety**

Safety considerations were evaluated primarily in the context of system reliability and preventing harmful misuse of information. While Agreemind does not directly affect physical safety, incorrect or misleading analysis of legal documents could potentially lead to poor user decisions. To mitigate this risk, the system presents analysis results as informational guidance rather than legal advice. The interface clearly links generated insights to the original contract text so users can verify interpretations.

Impact Level: **3 / 10**

- **Security**

Security was one of the most influential factors in the design process. Agreemind processes potentially sensitive documents such as rental agreements, employment contracts, and terms of service. Protecting user data and maintaining confidentiality were therefore major priorities. The system design includes encrypted communication between system components, secure storage of documents in protected storage services, and logical separation of user data. Additionally, access control mechanisms ensure that only authorized users can access their stored agreements. On mobile devices, the implemented PIN-lock and auto-lock mechanism provides an additional layer of protection against unauthorized access when a device is shared or left unattended.

Impact Level: **9 / 10**

- **Welfare**

User welfare was an important design consideration. Many individuals accept agreements without fully understanding their consequences due to complex legal language. Agreemind aims to empower users by providing simplified explanations, highlighting risky clauses, and identifying obligations and deadlines. By increasing transparency and awareness, the system helps users make more informed decisions about agreements that affect their daily lives.

Impact Level: **7 / 10**

- **Global Factors**

Global factors were considered mainly in terms of deployment and accessibility. The system is designed to operate on widely used platforms such as web browsers and mobile devices, allowing users from different regions to access the service. Additionally, the modular architecture allows the integration of jurisdiction-specific rule sets and regulatory knowledge bases in the future, enabling the system to adapt to different legal environments.

Impact Level: **5 / 10**

- **Cultural Factors**

Cultural factors were considered during the design of the system because legal language, contract structures, and expectations about agreements can vary across different cultures and regions. People from different cultural backgrounds may interpret legal terms differently or may have different levels of familiarity with legal documents. To address this, the system focuses on providing clear explanations and maintaining traceability between generated insights and the original document text. This helps users verify the analysis and reduces the risk of misinterpretation. However, the current system primarily assumes agreements written in English and common consumer contract formats. Supporting multiple languages and adapting explanations to different legal cultures may require further improvements in future versions of the system.

Impact Level: **2 / 10**

- **Social Factors**

Social factors played a meaningful role in the design of Agreemind. Many individuals lack access to professional legal assistance due to cost or availability. By providing automated analysis of agreements, the system aims to reduce this accessibility gap and make legal information easier to understand for everyday users. The platform is therefore designed to prioritize usability and clarity for non-expert users.

Impact Level: **6 / 10**

- **Environmental Factors**

Environmental considerations had limited influence on the system design. As a cloud-based software platform, Agreemind does not involve physical manufacturing or hardware components. However, efficient cloud resource usage and scalable architecture were considered to minimize unnecessary computational overhead and energy consumption during large-scale document analysis.

Impact Level: **2 / 10**

- **Economic Factors**

Economic factors influenced several design decisions. The system is designed to provide an affordable alternative to traditional legal consultation for understanding agreements. The architecture emphasizes scalable cloud infrastructure and modular services so that operational costs can remain manageable while supporting a growing number of users. These considerations also support the long-term sustainability of the platform.

Impact Level: **5 / 10**

Constraint	Impact level	Effect
Public health	None	The project is not related to public health.
Public safety	Medium	The system must safely handle user data and prevent misuse.
Security	High	Strong data protection and secure storage are required for user documents.
Public welfare	High	Helps users better understand agreements and avoid harmful terms.
Global factors	Medium	Legal regulations vary across countries and may affect system expansion.
Cultural factors	Low	Legal interpretation and expectations vary across cultures.
Social factors	High	Improves access to legal understanding for non-experts.
Environmental factors	Low	The system has minimal environmental impact as a software platform.
Economic factors	Medium	Automated analysis can reduce time and cost for users.

Table 1: Table of Constraints

7.1.2. Standards

- **IEEE 830 - Software Requirements Specifications [26]**

The IEEE 830 standard is utilized to rigorously define the functional and non-functional requirements of the system, ensuring clarity and completeness in the specification document.

- **UML 2.5.1 - Unified Modeling Language [27]**

UML 2.5.1 is employed as the standard for visualizing the system's architecture, ensuring that the interactions between the diverse client applications and the backend are clearly mapped. Specifically, Sequence and Class diagrams will be used to model the modular architecture.

- **REST API Guidelines [28]**

The system adheres to REST architectural principles to ensure stateless and scalable communication between the backend and its various clients, including the web dashboard, mobile app, and browser extension. This standardization allows for efficient resource management.

7.2. Ethics and Professional Responsibilities

Agreemind processes documents that may contain sensitive legal, financial, personal, and contractual information. Therefore, ethical and professional responsibility was an important consideration throughout the project. The team recognized that even though the system is not a substitute for a lawyer, its outputs may influence how users understand agreements and make decisions. For this reason, the system was designed as an informational assistant rather than a legal-advice provider. Summaries, risk labels, rights, deadlines, and rewrite suggestions are intended to help users inspect agreements more carefully, not to make legally binding decisions on their behalf.

A major ethical responsibility was protecting user privacy. Uploaded contracts may include personally identifiable information, banking details, employment terms, business information, or confidential clauses. To address this, the system uses authenticated access, user-scoped database queries, and ownership checks so that users can only access their own documents, analyses, folders, chunks, and comparison results. The backend test suite also verifies cross-user data isolation, ensuring that one user cannot retrieve another user’s stored agreements or analysis outputs.

The team also considered the risk of misleading or overconfident AI-generated outputs. Legal documents are complex, and automated systems may misclassify clauses, miss context, or

generate incomplete explanations. To reduce this risk, Agreemind preserves traceability between system outputs and the original document text. Risk findings include evidence spans and rationale fields, highlighted document views connect findings back to the source text, and RAG chat responses include source metadata. This allows users to verify the basis of the output instead of relying only on a generated summary.

Professional responsibility also required careful handling of external AI services. The system uses external providers for summarization, embeddings, optional LLM refinement, chat, and clause rewriting. Because these services may receive portions of user documents, the system design treats external AI calls as part of the requested functionality rather than unrelated data use. In future deployment, this responsibility should be communicated transparently through user-facing privacy notices and configuration options, especially for users who submit sensitive or confidential agreements.

The introduction of local mode further supports the ethical responsibility of minimizing unnecessary external data exposure. When local mode is enabled, LLM-dependent services can run through a locally hosted Ollama model instead of sending document content to external AI providers. This gives users and deployers a stronger privacy-preserving option for sensitive agreements, while preserving the same deterministic local transformer engines used by the normal analysis pipeline.

The team also aimed to reduce harm caused by inaccurate risk classification. Instead of using a single generic model for all contracts, the final system uses separate engines for different document domains, including NDAs, Turkish banking contracts, English banking contracts, Terms of Service, and general contracts. This design was chosen to improve domain relevance and reduce false positives. The optional LLM refinement stage further filters noisy findings and improves explanation quality, while the deterministic first stage helps keep the pipeline reproducible and easier to inspect.

Another ethical consideration was accessibility and public welfare. Many individuals accept agreements without fully understanding them because legal language is difficult and professional

legal help may be expensive. Agreemind attempts to support these users by presenting risks, rights, deadlines, and summaries in clearer language. However, the system must avoid creating a false sense of legal certainty. Therefore, the appropriate professional framing is that Agreemind improves awareness and document readability, while users should still consult qualified professionals for legal decisions with serious consequences.

The project also reflects professional responsibility in testing and validation. The final backend includes automated tests for successful workflows as well as validation errors, corrupted PDFs, invalid comparisons, duplicate registrations, and unauthorized access. These tests help ensure that the system fails safely instead of silently producing incorrect or insecure results. Maintaining and expanding this testing discipline is necessary for any future development, especially because the system handles private documents and AI-generated legal interpretations.

Overall, the ethical responsibility of the project is to empower users without misleading them, protect sensitive data, make automated outputs inspectable, and maintain clear boundaries between informational assistance and legal advice. The final design addresses these responsibilities through privacy-aware access control, traceable analysis results, domain-specific engines, controlled error handling, and automated validation tests.

7.3. Teamwork Details

We created an issue list and listed all the features we required. Each team member chose a set of tasks from the list that aligned with their previous experiences, technical skills.

7.3.1. Contributing and functioning effectively on the team

- We held biweekly sprints and meetings along with our supervisor in which we decided on the tasks for the upcoming sprint.
- Project reports were prepared with an equal and fair distribution of tasks.
- Frontend and the UI of the project was designed collaboratively, each member designed the UI of their respective functionality. Ata Soykal refined the overall user interface.
- Edip Emre Dönger and Ata Oğuz worked on the Android and IOS applications.

- Backend functionalities were implemented by Ata Oğuz, Emir Görgülü, Edip Emre Dönger.
- Machine Learning models for the various legal fields were designed by Can Polat Bülbül and integrated into the app by Edip Emre Dönger.

7.3.2. Helping creating a collaborative and inclusive environment

- By regularly meeting and discussing the many matters involving the project we made all decisions unanimously. During these discussions each group member shared their vision of the app and their ideas.
- We used JIRA's issue trackers and Github to create a seamless collaboration environment where each member was able to actively monitor what the other members were working on.
- To ensure smooth communication outside of scheduled meetings, we used WhatsApp and Discord as our primary messaging channels, allowing team members to stay informed, share updates, and address blockers quickly regardless of time or location.

7.3.3. Taking lead role and sharing leadership on the team

Each member of the team held a distinct responsibility, and the lead role was shared flexibly across the project's phases, allowing every member to develop ownership over their area while supporting others.

- Ata Oğuz led the implementation of the core backend services, including the API architecture and database design. He coordinated the backend development effort and ensured that services were structured in a way that made integration straightforward for the rest of the team. He also implemented the RAG search functionalities. He was also responsible for several mobile functionalities such as OCR, background analysis.
- Ata Soykal led the UI design and implementation for both web and mobile clients, focusing on improving the user experience and the aesthetics of the application. He also maintained the project website and created promotional materials for the project.
- Can Polat Bülbül led the research and development of the machine learning models used for legal document analysis across different domains. He evaluated and selected the appropriate models for each document type and designed the analysis pipeline architecture.

- Emir Görgülü led the development of backend processing pipelines and the integration of external services. He was responsible for the document ingestion workflow. He also designed and implemented the Web-Extension for the web app.
- Edip Emre Dönger led the integration of the machine learning models into the application. He connected the ML layer and the backend, ensuring that model outputs were correctly consumed by the processing pipeline and surfaced accurately to users. He also implemented the analysis and refinement pipelines.

7.3.4. Meeting objectives

The main objectives defined in the project plan were largely met in the final implementation. The team completed the core agreement analysis workflow, including document upload, text extraction, risk detection, rights extraction, deadline extraction, summaries, personal vault storage, folder management, version comparison, and RAG-based document chat. The final system also exceeded the initial scope in several areas by adding a browser extension, mobile camera scanning with OCR, background analysis with notifications, clause rewriting, specialized engines for multiple contract domains, and a comprehensive automated backend test suite.

Some objectives were met with adjusted implementation details. For example, deadline alerts are not implemented through a separate alerts backend API; instead, they are derived from the analysis result's reminders field and managed on the frontend with local persistence and native calendar export. Similarly, the system's analysis architecture evolved from a more general legal-document pipeline into a domain-routed design with specialized NDA, Turkish banking, English banking, and general contract engines. Overall, the final project satisfies the planned goals of helping users understand agreements, track obligations, manage stored documents, and compare versions, while also improving responsiveness, traceability, and test coverage beyond the original plan.

7.4. New Knowledge Acquired and Applied

During the project, the team acquired and applied new knowledge across backend development, mobile development, browser-extension development, machine learning, legal-document analysis, and retrieval-augmented generation. On the backend side, the team gained practical experience with FastAPI, background task execution, input validation, user-scoped authorization, structured error handling, and PostgreSQL with pgvector. These concepts were applied directly in the document upload workflow, asynchronous analysis lifecycle, vault operations, version comparison, and RAG chat implementation.

The project also required substantial learning in natural language processing and legal-domain machine learning. The team studied how different contract types require different analysis strategies and applied this knowledge by implementing specialized engines for NDAs, Turkish banking contracts, English banking contracts, and general contracts. This included learning about multi-label classification, natural language inference, question-answering models, domain-specific post-processing, confidence thresholds, severity mapping, and false-positive reduction. The separate agreemind-ml repository reflects this learning through training, evaluation, cross-validation, inference, and HuggingFace deployment pipelines.

Another important area of acquired knowledge was retrieval-augmented generation. The team learned how to chunk legal documents, generate embeddings, store vectors in pgvector, combine semantic retrieval with PostgreSQL full-text search, apply Reciprocal Rank Fusion, expand context with neighboring chunks, and generate grounded answers with source metadata. This knowledge was applied in the global vault chat feature, allowing users to ask questions across stored agreements while keeping answers tied to retrieved document excerpts.

On the client side, the team expanded its knowledge of cross-platform development. The web application used React Native with Expo Router and react-native-web, while the mobile application required React Native CLI features such as camera access, OCR submission workflows, local notifications, AsyncStorage persistence, and native calendar export. The browser extension introduced additional learning in Vite, React, TypeScript, content scripts, browser messaging, webpage detection heuristics, and text extraction with Readability.js.

The team used several learning strategies throughout the project. Members learned through documentation reading, experimentation, incremental prototyping, debugging, code review, model evaluation, and integration testing. When unfamiliar technologies were introduced, such as pgvector retrieval, mobile notifications, browser-extension content scripts, or legal-domain model training, the team first built smaller prototypes and then integrated the working parts into the main system. This iterative learning approach helped reduce risk and allowed the team to convert newly acquired knowledge into working project features.

8. Conclusion and Future Work

Agreemind was completed as a multi-platform legal agreement assistant that helps users understand, manage, compare, and query their personal agreements. The final system supports three client surfaces: a web application, a native mobile application, and a browser extension. These clients connect to a shared FastAPI backend that manages authentication, document submission, background analysis, vault organization, version comparison, and RAG-based chat. Compared with the earlier detailed design, the final implementation is more complete and more specialized, particularly through the addition of the browser extension, OCR-based mobile scanning, background processing, clause rewriting, local/offline operation through Ollama, and domain-specific analysis engines.

The core contribution of the project is the implemented agreement analysis pipeline. Submitted documents are extracted into text, routed by contract type, segmented into semantic units, and processed for risky clauses, user rights, and timing obligations. The final system uses separate engines for NDAs, Turkish banking contracts, English banking contracts, and general contracts, which makes the analysis more maintainable and better adapted to different legal domains. Optional LLM refinement and clause rewriting improve the clarity and usefulness of the output, while risk scoring and source-span tracking make results structured and traceable.

The project also achieved the broader lifecycle-management goals of Agreemind. Users can store analyzed documents in a personal vault, organize them with folders, compare versions, view extracted alerts, export deadlines to the native calendar, and ask questions across stored agreements through the RAG chat system. The backend test suite provides additional confidence

in the implementation by covering authentication, document workflows, validation failures, folder operations, comparison, browser-extension submission, corrupted files, performance behavior, and cross-user data isolation. Overall, the final implementation satisfies the main goal of providing an accessible, consumer-oriented tool for understanding and managing agreements.

Future work should first focus on improving legal reliability and user trust. Although the current system provides traceable informational analysis, it should continue to make clear that it does not replace professional legal advice. Future versions can improve explanation calibration, show uncertainty more explicitly, and provide clearer disclaimers when the model is unsure or when the retrieved evidence is insufficient. Additional evaluation with real users and legal-domain reviewers would also help identify confusing explanations, missed risks, and overly aggressive false positives.

Another important direction is expanding document and language coverage. The final system supports several major contract types, but future versions could add more specialized engines for rental agreements, employment contracts, insurance policies, loan agreements, freelance contracts, and subscription agreements. Multilingual support can also be expanded beyond the current Turkish and English banking focus, especially because contract structure and legal expectations vary across countries and legal systems.

The RAG and comparison systems can also be improved. Future work may include stronger citation visualization, better clause-level retrieval, more robust PDF-to-text alignment, semantic comparison between versions, and automatic detection of materially important changes. The chat system could support document-specific chat, cross-document summaries, and timeline-style answers for obligations and renewals, while still remaining grounded in retrieved source text.

Finally, future deployment work should strengthen operational security, privacy controls, and scalability. This includes production-grade logging, monitoring, database backups, encryption configuration, user-facing privacy controls for external AI calls, and account/data deletion workflows.

9. Glossary

Agreemind: The proposed consumer-facing legal-document assistant that summarizes agreements, highlights risks, and helps users track obligations without providing legal advice.

Agreement: A legal text the user uploads or shares (e.g., Terms of Service, Privacy Policy, rental/subscription contract).

Clause: A meaningful segment of an agreement (sentence/paragraph/section) that expresses a rule, right, limitation, or obligation.

Risk Flag: A detected clause category that may be unfavorable to the user (e.g., data sharing, auto-renewal, unilateral change, arbitration).

Plain-Language Summary: A simplified explanation of an agreement or clause written for non-expert users.

Obligation: An action the user must do (or avoid) according to the agreement (e.g., payment, notice submission, compliance requirement).

Deadline / Notice Period: A time constraint extracted from the agreement (e.g., cancellation window, renewal date, "within 30 days").

Personal Vault: A secure personal repository where a user's processed agreements, reports, and metadata are stored for later search and comparison.

Version Comparison: A feature that identifies and presents changes between two versions of the same agreement ("what changed?").

Analysis Pipeline: The backend processing steps applied to an agreement (ingestion → text extraction → chunking → retrieval/classification → summarization → report).

AnalysisJob: A backend job that represents one analysis request from a user and its processing state (queued/running/completed/failed).

ConsentRecord: A stored record that the user explicitly permitted an agreement to be processed (especially important if external APIs are used).

RAG (Retrieval-Augmented Generation): A method where the system retrieves relevant text passages and constrains the LLM to answer using that context.

Embedding: A numeric vector representation of text used to support semantic search and retrieval in the vault.

Vector Store: A database/index optimized for similarity search over embeddings (used for vault querying and context retrieval).

HNSW: A graph-based approximate nearest neighbor indexing method commonly used for fast vector similarity search.

LLM (Large Language Model): A model used to generate summaries/explanations; in your system it must be constrained to informational output (not legal advice).

Custom Model: A smaller model you train/fine-tune for a specific subtask (e.g., risk classification or date extraction) to reduce cost and dependency on external APIs.

NER (Named Entity Recognition): A technique to detect structured entities in text (e.g., dates, organizations, money amounts).

Share Sheet / Share Intent: Mobile OS functionality that lets the user share a webpage/text into Agreemind for on-demand analysis (instead of background monitoring).

Privacy by Design: Designing the system to minimize data collection, enforce access control, and prevent model training on user data without explicit opt-in.

GDPR / Right to be Forgotten: Data protection requirements that include user deletion/export rights and limits on data retention/processing.

10. References

[1] Ironclad, “AI-powered Contract Lifecycle Management Software,” 2025. [Online]. Available: https://ironcladapp.com/product/ai-based-contract-management

[2] Kira Systems, “AI-powered Contract Analysis Software,” 2025. [Online]. Available: https://kira.ai/solutions/legal-workflow

[3] Luminance, “Legal-Grade AI Contract & Document Review Software,” 2025. [Online]. Available: https://luminance.com/solutions/legal/

[4] ToS;DR, “Terms of Service; Didn’t Read,” 2025. [Online]. Available: https://tosdr.org

[5] Open Terms Archive, “Public Archive of Online Terms and Conditions,” 2025. [Online]. Available: https://opentermsarchive.org

[6] Termzy AI, “Real-time ToS Detection Software,” 2025. [Online]. Available: https://www.termzyai.com/#features

[7] D. Hendrycks et al., “CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review,” Advances in Neural Information Processing Systems, 2021.

[8] P. Koreeda and C. D. Manning, “ContractNLI: A Dataset for Document-level Natural Language Inference for Contracts,” Findings of the Association for Computational Linguistics, 2021.

[9] Hugging Face, “Transformers: State-of-the-art Machine Learning for PyTorch, TensorFlow, and JAX,” 2025. [Online]. Available: https://huggingface.co/docs/transformers

[10] Agreemind, “Agreemind/contractnli-distilbert-nda,” Hugging Face Model Repository, 2026.

[11] Agreemind, “Agreemind/turkish-banking-bert,” Hugging Face Model Repository, 2026.

[12] Agreemind, “Agreemind/en-banking-roberta,” Hugging Face Model Repository, 2026.

[13] FastAPI, “FastAPI Documentation,” 2025. [Online]. Available: https://fastapi.tiangolo.com

[14] PostgreSQL Global Development Group, “PostgreSQL 16 Documentation,” 2025. [Online]. Available: https://www.postgresql.org/docs/16/

[15] pgvector, “pgvector: Open-source Vector Similarity Search for Postgres,” 2025. [Online]. Available: https://github.com/pgvector/pgvector

[16] OpenAI, “Text Embeddings Documentation,” 2025. [Online]. Available: https://platform.openai.com/docs/guides/embeddings

[17] OpenAI, “OpenAI API Documentation,” 2025. [Online]. Available: https://platform.openai.com/docs

[18] Google, “Gemini API Documentation,” 2025. [Online]. Available: https://ai.google.dev

[19] Meta, “React Native Documentation,” 2025. [Online]. Available: https://reactnative.dev

[20] Expo, “Expo Documentation,” 2025. [Online]. Available: https://docs.expo.dev

[21] React Navigation, “React Navigation Documentation,” 2025. [Online]. Available: https://reactnavigation.org

[22] Google Chrome Developers, “Chrome Extensions Documentation,” 2025. [Online]. Available: https://developer.chrome.com/docs/extensions

[23] Mozilla, “Readability.js,” 2025. [Online]. Available: https://github.com/mozilla/readability

[24] Artifex Software, “PyMuPDF Documentation,” 2025. [Online]. Available: https://pymupdf.readthedocs.io

[25] Tesseract OCR, “Tesseract OCR Engine,” 2025. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[26] IEEE, “IEEE Recommended Practice for Software Requirements Specifications,” IEEE Std 830-1998, 1998.

[27] Object Management Group, “Unified Modeling Language Specification, Version 2.5.1,” 2017.

[28] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000.