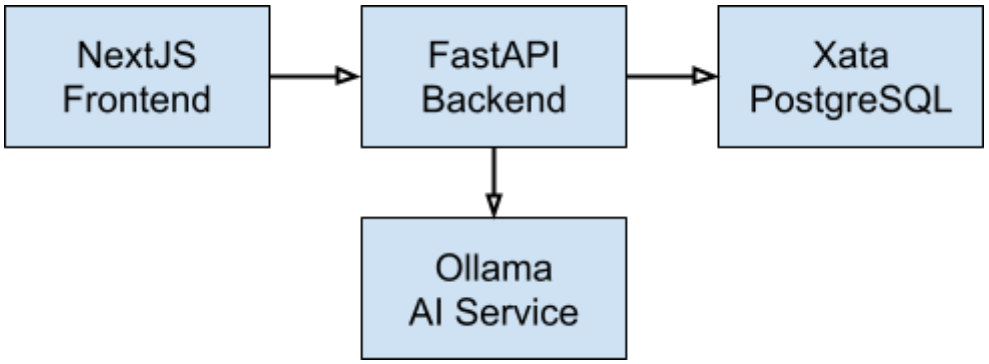
Architecture Overview

System Overview

Gyst is an AI-native document management system designed to act as an intelligent knowledge base for teams. The revised system adopts a two-tier architecture comprising a Next.js application (handling frontend, backend API, user management, and database interaction) and a dedicated Python FastAPI service leveraging CrewAI and RAG tools for AI-driven tasks. Files are stored locally, and metadata, tags, and user information are managed in an SQLite database.



Component Details

Next.js Application

The Next.js application serves as the user-facing interface and the primary backend coordination layer. It handles user authentication, manages the database, serves files, and orchestrates calls to the Python AI service.

Frontend Components:

- **AppComponent**: Main application container, potentially incorporating role-based UI elements.
- **FileExplorerComponent**: Displays the organization's folder/file structure accessible to the user.
- **DocumentViewerComponent**: Renders different document types based on content retrieved via the backend.
- **ChatInterfaceComponent**: Handles user interaction with the AI agent.
- **SearchComponent**: Manages document and tag search based on database queries.
- **AuthComponents**: Handles user login, registration, and role display.

Backend Components (Next.js API Routes):

- **Auth API**: Handles user authentication, registration, and session management (NextAuth V5). Manages user roles.
- **Document API**: Handles document upload (receiving file, saving locally), metadata extraction, and retrieval requests. Triggers AI tagging via the Python service upon upload. Handles serving document content for viewing.
- **Tag API**: Manages tag information, primarily interacting with the database.
- **Search API**: Processes search requests against the SQLite database (metadata, tags).
- **Chat API**: Handles user chat interactions, retrieving necessary document context (paths, metadata) from the database and forwarding the query and context to the Python service.
- **Python Service Connector**: Module responsible for making HTTP requests to the Python FastAPI service endpoints and handling responses.
- **Database Adapter**: Module specifically for interacting with the SQLite database (gyst.sqlite). Crucially, this is the only component/service that directly interacts with the SQLite database file.

Key Libraries:

- Next.js 15+
- NextAuth V5 (Beta) for authentication and role management.
- React
- react-pdf, react-markdown, etc., for document rendering.
- Tailwind CSS for styling.
- A library for interacting with SQLite (e.g., better-sqlite3 or an ORM like Prisma/Drizzle configured for SQLite).

Python Ollama Service

A dedicated Python service hosting CrewAI and its RAG tools. It performs computationally intensive AI tasks related to document analysis, tagging, chat responses, and content correlation by reading files from the shared local storage path provided by the Next.js backend.

Key Endpoints:

- **/analyze_document**: Receives a document file path (or identifier allowing path retrieval) from Next.js, triggers a CrewAI task for tagging, and returns suggested tags.
- **/chat**: Receives a user query and relevant document context (e.g., file paths, summaries, metadata) from Next.js, triggers a CrewAI chat task, and returns a generated response.
- **/correlate**: Receives specific document identifiers/paths from Next.js (e.g., results from a basic search), triggers a CrewAI task to find correlations or summarize content, and returns the result.

Core Functions:

- **Crew Orchestrator**: Sets up and runs CrewAI tasks based on incoming API requests and potentially user role context (though AI logic is universal for core tasks).
- **RAG Tool Handler**: Configures and utilizes CrewAI's RAG tools (FileRAGTool, DocxRagTool, PdfRagTool, TextRagTool) to read and process document content directly from the provided file paths.
- **Tag Generation Task**: Defines the CrewAI agents and tasks required to analyze document content via RAG and extract relevant tags.
- **Chat Response Task**: Defines the CrewAI agents and tasks required to generate conversational responses based on a query and provided document context from RAG.
- **Correlation Task**: Defines the CrewAI agents and tasks to analyze content from multiple documents via RAG and identify relationships or provide summaries.

Key Libraries:

- FastAPI for the web framework.
- Pydantic for data validation.
- crewai, crewai_tools (specifically RAG tools like FileRAGTool, DocxRagTool, PdfRagTool, TextRagTool).
- An LLM provider library compatible with CrewAI (e.g., langchain-community for Ollama, openai, etc. - assuming a local Ollama instance might still be used as the underlying LLM).
- httpx or requests if needed for external API calls (unlikely for core logic in this iteration).

Database (SQLite)

A single gyst.sqlite file is used for storing all structured data.

- **Access:** Only the Next.js application interacts directly with the gyst.sqlite file. The Python service receives necessary data from Next.js via API parameters or fetches relevant document content using file paths provided by Next.js.
- **Key Tables:**
    - **users**: Stores user information, including role (Admin, User) and organization affiliation.
        - id (string, primary key)
        - username (string, unique)
        - role (string, e.g., 'Admin', 'User')
        - organization_id (foreign key to organizations table, implicit or explicit)
        - ... other auth-related fields managed by NextAuth
        - created_at (timestamp)
    - **organizations**: (Optional but recommended for multi-tenancy implied by roles) Stores organization information.
        - id (string, primary key)
        - name (string)
    - **documents**: Stores document metadata. Linked to organizations.
        - id (string, primary key)
        - organization_id (foreign key)
        - filename (string, hashed name in ./uploads)
        - original_filename (string)
        - file_path (string, path relative to ./uploads)
        - file_type (string)
        - uploaded_by_user_id (foreign key)
        - created_at (timestamp)
    - **tags**: Stores unique tag names.
        - id (string, primary key)
        - name (string, unique)
    - **document_tags**: Maps documents to tags.
        - id (string, primary key)
        - document_id (foreign key)
        - tag_id (foreign key)
        - confidence (float, optional, from AI)

AI Service (CrewAI & RAG Tools)

CrewAI orchestrates agents and tasks to provide AI capabilities, primarily by using RAG tools to read and process document content stored locally.

- **Core Functionality**:
    - Document Analysis/Tagging: CrewAI agents read uploaded document content (via RAG tools on the file path) and generate suggested tags.
    - Conversational Responses: CrewAI agents answer user questions in the chat interface, retrieving relevant document snippets using RAG tools based on context provided by Next.js.
    - Document Correlation/Summarization: CrewAI agents can analyze content from multiple documents (via RAG tools) to find relationships, summarize information, or identify patterns upon request (e.g., in the chat interface).

- **LLM Provider**: CrewAI requires an underlying LLM. For this MVP, it could interface with a local Ollama instance or another compatible provider, focusing on smaller, lightweight models.

File Storage Approach

Files are stored locally within the project directory managed by the Next.js application:

- Base upload directory: ./uploads
- Files are stored with hashed filenames to prevent collisions.
- Original filenames, hashed filenames (filename), and relative file paths (file_path) are stored in the SQLite database.
- Simple subfolder organization within ./uploads based on organization or upload date could be added later.
- Access: The Next.js backend serves these files to the frontend. The Python FastAPI service reads these files directly using the paths retrieved by Next.js from the database and provided in API calls.

Inter-Service Communication

Communication between the Next.js backend and the Python FastAPI service is handled via HTTP:

- **Method**: RESTful API endpoints are used for structured requests.
- **Data Format**: JSON is used for data exchange.
- **Flow**: Next.js acts as the client, initiating requests to the Python service endpoints (/analyze_document, /chat, /correlate). Next.js includes necessary data (like document IDs, file paths retrieved from the DB, user queries) in the request body.
- **Authentication**: Basic authentication or API key mechanism between Next.js and the Python service could be added for service-to-service security, separate from user authentication handled by Next.js.

Security Considerations

Basic security measures for the MVP include:

- Input validation on all Next.js API endpoints.
- File type and size restrictions on uploads.
- Authentication system with role-based access control (Admins can upload, Users cannot; all authenticated users can chat/view).
- Direct database access restricted to the Next.js backend.
- File paths provided to the Python service should be validated by Next.js to prevent directory traversal attacks.

Scalability Limitations (MVP)

This architecture has intentional limitations for the MVP phase:

- **Local File Storage**: Not suitable for production scale or multi-instance deployments without a shared file system.
- **Single SQLite Database File**: Performance may degrade with very large numbers of documents or concurrent users. Not ideal for high availability.

- **Single Python Service Instance**: The CrewAI service is a potential bottleneck for processing concurrent requests.
- **Limited AI Capabilitie**s: Constrained by the choice of lightweight local LLM and the complexity of CrewAI tasks defined for the MVP.
- **Organizational Multi-Tenancy**: While user roles and organization IDs are included in the schema, robust multi-tenancy (data isolation, scaling) is not fully implemented or tested beyond ensuring users only see documents from their organization.