Technical Architecture and Implementation Strategy for an AI-Powered Agricultural Commodity Marketplace

I. Executive Summary

This report outlines a technical blueprint for developing a sophisticated web-based system designed to function as an online remote marketplace connecting farmers of differentiated bulk commodities with overseas buyers and logistics providers. The platform aims to facilitate business contacts through AI-generated user profiles, intelligent search and matchmaking, and integrated communication tools, reminiscent of a personalized matching service. Key functionalities include mobile asset uploads, Langchain-driven AI processing for profile creation using Retrieval-Augmented Generation (RAG), indirect profile editing capabilities, granular privacy controls, dynamic profile galleries, and a curated reference library integrated via RAG. The system is specified to be Dockerized, primarily utilize Python and open-source technologies, be scalable, and accommodate future extensions like profile seeding and virtual seller collections.

The core technical recommendations favor a **Microservices Architecture** built using the **FastAPI** Python framework, leveraging its performance and asynchronous capabilities. Data persistence will rely on **PostgreSQL** for structured data and user management, and a robust open-source vector database like **Weaviate** or **Milvus** for handling the embeddings crucial for RAG functionalities. **Langchain** will serve as the central framework for orchestrating AI workflows, including text processing, embedding generation, RAG-based profile creation, and search/matchmaking, utilizing **Function/Tool Calling** for reliable structured outputs. The mobile interface for asset uploads will be developed using **Expo**. Profile content will be cached using **Redis** with event-based invalidation. Communication will be handled via a simple **WebSocket** system designed with considerations for potential network restrictions like China's Great Firewall. Deployment will utilize **Docker** containers managed via a **cloud-based orchestration platform** (e.g., AWS Fargate, Google Cloud Run, or managed Kubernetes) and automated through **CI/CD pipelines**. Monitoring will involve both standard Application Performance Monitoring (APM) and specialized **LLM observability tools** like Langfuse or Phoenix.

The successful implementation hinges on the effective integration of these AI components, particularly the RAG pipelines for profile generation and search. Careful consideration must be given to data privacy enforcement through robust access control mechanisms, the legal and ethical implications of potential profile seeding from public sources, and ensuring the scalability and maintainability of the

microservices architecture. Strategic use of caching and asynchronous processing will be vital for performance and user experience.

II. Foundational Architecture and Technology Stack

The foundation of the platform requires careful consideration of the overall system architecture, the core backend framework, data storage solutions, and the mobile application platform. These choices significantly impact scalability, maintainability, performance, and development velocity.

A. Architectural Design: Evaluating Monolith vs. Microservices

The choice between a monolithic and a microservices architecture is fundamental. A monolithic application bundles all software components together, often within a single codebase and deployment unit.[1] In contrast, a microservices architecture decomposes the application into smaller, independent services that communicate via APIs.[1]

Monolithic architectures offer simplicity in initial setup and deployment, making them potentially faster for developing prototypes or simple applications.[1] Debugging can initially seem easier as tracing occurs within a single environment.[4] However, monolithic systems become increasingly complex and difficult to modify or maintain over time, as changes in one area can impact the entire codebase.[1] They represent a single point of failure, increasing risk during updates [1], and scaling requires upgrading resources for the entire application, which is often inefficient and costly, especially when only specific components experience high load.[1] Furthermore, monolithic designs restrict technology diversity, making it harder to use the best tool for each specific job.[4]

Microservices, while requiring more upfront planning and infrastructure setup [1], offer significant long-term advantages for this platform. Key benefits include independent scalability of components, allowing resources to be allocated precisely where needed (e.g., scaling AI inference services independently from the chat service), leading to better resource utilization and cost-effectiveness.[1] Technology diversity is a major advantage; different microservices can utilize different technology stacks best suited for their function (e.g., Python with specific AI libraries for processing, another language/framework for real-time communication if needed).[4] Fault isolation improves resilience, as the failure of one service is less likely to bring down the entire platform.[3] Continuous delivery and deployment (CI/CD) become simpler and faster for individual services [4], and development can be parallelized across independent teams.[4] Microservices are also inherently well-suited for cloud-native deployment strategies.[1]

Drawbacks include increased complexity in deployment, inter-service communication management (latency concerns [2]), distributed debugging [1], and potential challenges in service discovery.[2] Expertise in distributed systems, APIs, and containerization is necessary.[1]

The nature of the proposed application, with its distinct and potentially resource-intensive components (user profile management, AI-driven content processing, RAG indexing and search, real-time chat, reference library management), strongly favors a **Microservices Architecture**. The need for independent scaling, particularly for the AI/ML components (LLM inference, embedding generation, vector search), is paramount for both performance and cost management. Scaling a monolithic application to handle peak AI processing load would unnecessarily scale less demanding components like user authentication or static content serving, leading to significant resource waste.[1] The ability to use specialized tools like Langchain, specific LLMs, and vector databases within dedicated services further reinforces this choice.[4] The long-term benefits of maintainability, scalability, fault isolation, and technological flexibility provided by microservices align better with the envisioned growth and complexity of the platform, outweighing the higher initial setup cost and complexity.[1]

Table II.A: Microservices vs. Monolith Comparison for the Commodity Marketplace Platform

Feature	Monolithic Architecture	Microservices Architecture	Recommendation for Project
Scalability	Scales entire application; inefficient & costly [1]	Independent scaling of services; efficient & cost-effective [1]	Microservices
Technology Diversity	Difficult; single stack often enforced [4]	High; best tool per service (e.g., Python/AI, other) [4]	Microservices
Development Complexity	Initially simpler, grows exponentially complex [1]	Higher initial planning, easier long-term management [1]	Microservices (Long-term)

Deployment Complexity	Simpler initially (single unit) [1]	More complex (multiple units, orchestration) [1]	Microservices (Trade-off)
Operational Overhead	Lower initially	Higher (monitoring, discovery, distributed debugging) [1]	Microservices (Trade-off)
Fault Isolation	Low; single point of failure [1]	High; failures typically isolated to specific services [3]	Microservices
Cost (Initial)	Lower [1]	Higher (planning, infrastructure) [1]	Monolith
Cost (Long-term)	Increases with complexity, scaling cost [1]	More cost-effective due to targeted scaling, maintainability [1]	Microservices
Suitability for Project	Poor fit due to diverse, scalable components (esp. AI)	Excellent fit; supports independent scaling, tech diversity, long-term maintainability	**Microservices**

B. Core Backend Framework: Selecting the Optimal Python Framework

Given the choice of Python, selecting the right web framework is crucial for implementing the microservices backend. The primary contenders are Django, Flask, and FastAPI.

Django is a mature, "batteries-included" framework offering a comprehensive suite of built-in features like an ORM, admin interface, authentication, and caching.[5] It enforces a structure suitable for large, complex applications such as content management systems or e-commerce platforms.[5] While robust and scalable, Django has a steeper learning curve [5] and can feel heavyweight for API-centric microservices.[7] Its performance is generally lower than Flask or FastAPI [7], and while Django REST Framework (DRF) adds powerful API capabilities [8], the core framework's synchronous nature is less ideal for highly I/O-bound tasks common in AI applications.

Flask is a minimalist "micro-framework," providing core routing and request handling while allowing developers maximum flexibility to choose libraries for other functionalities (ORM, auth, etc.).[5] It has a shallow learning curve and is excellent for small-to-medium applications, APIs, and prototypes.[5] However, this flexibility means more manual setup for larger projects, and the lack of an enforced structure can sometimes lead to inconsistencies.[6] While generally faster than Django, it doesn't match FastAPI's performance, especially in asynchronous scenarios.[7]

FastAPI is a modern, high-performance framework specifically designed for building APIs.[5] Built upon Starlette (for ASGI async capabilities) and Pydantic (for data validation), it offers exceptional speed, especially for asynchronous operations.[5] This asynchronous nature is critical for efficiently handling I/O-bound operations like communicating with external LLM APIs, databases, or WebSocket connections without blocking the server. FastAPI automatically generates interactive API documentation (Swagger UI/OpenAPI) based on Python type hints, which also provide robust data validation and improve code clarity.[5] While its ecosystem is newer and smaller than Django's or Flask's [5], it is rapidly growing [7] and its design philosophy aligns perfectly with building performant, type-safe microservices.[6] An initial learning curve might exist for developers unfamiliar with Python's asyncio.[6]

For this project, **FastAPI** is the strongly recommended framework. Its high performance, native asynchronous support, automatic data validation via Pydantic, and excellent API documentation features make it ideal for building the API-driven microservices required.[5] The async capabilities are particularly beneficial for interacting with potentially slow AI services and databases efficiently. Pydantic models simplify defining and validating data structures passed between services and in API requests/responses, crucial in a microservices context. The automatic OpenAPI documentation streamlines development and integration between services and with the frontend/mobile app.

Table II.B: Python Web Framework Comparison

Feature	Django	Flask	FastAPI	Recommendation for Project
Performance	Slower [7]	Medium; faster than Django [7]	High; built for speed [5]	FastAPI

Async Support	Added later; not core design	Basic support via extensions	Native ASGI support; core design [5]	FastAPI
Built-in Features	High ("batteries-included": ORM, Admin, Auth) [5]	Low (minimalist core); requires extensions [6]	Medium (Validation, Docs); relies on ecosystem for ORM, etc. [5]	FastAPI (API Focus)
Learning Curve	Steeper [5]	Low [5]	Moderate (esp. async if new) [6]	FastAPI (Manageable)
Community/Ecosystem	Very Large, Mature [7]	Large, Active [7]	Growing, Active; smaller than Django/Flask [5]	FastAPI (Sufficient)
API Development Focus	Strong via DRF [8]; framework is broader	Good via extensions; flexible	Primary focus; built-in validation & docs [5]	FastAPI
Suitability for Microservices	Possible but can be heavyweight	Good due to lightweight nature	Excellent due to performance, async, API focus [6]	FastAPI
Suitability for Project	Less ideal due to API/async focus	Viable, but requires more setup	Ideal fit for performance, async AI/IO, API needs, microservices architecture	**FastAPI**

C. Data Persistence Strategy: Relational and Vector Database Recommendations

A robust data persistence strategy requires both relational and vector databases to handle the different types of data and query patterns involved.

Relational Database: A relational database is necessary for storing structured user information (login credentials, profile metadata like name, region, contact details, verification status), relationships between entities (farmers, buyers, logistics

providers, collection memberships), application configuration, privacy settings associated with content, and potentially message history or metadata for the chat system.

PostgreSQL is recommended as the relational database. It is a mature, powerful, and open-source object-relational database system known for its reliability, feature richness, and extensibility.[9] It offers strong support for standard SQL, complex queries, indexing, transactions, and data integrity. Features like JSONB data types are useful for storing semi-structured profile data, and its support for roles and permissions, including group roles [10], facilitates managing access control. PostgreSQL schemas allow for logical organization of database objects (e.g., separating user data from reference library metadata).[11] Numerous Python libraries, including SQLAlchemy (and thus SQLModel, which integrates well with FastAPI [9]), provide excellent connectivity.

Vector Database: A vector database is essential for the RAG capabilities central to this platform.[13] It stores high-dimensional vector embeddings of unstructured data (text chunks from uploaded documents, AI-generated image descriptions, reference materials) and enables efficient semantic similarity search.[13] This is crucial for finding relevant content for profile generation, matchmaking searches, and answering questions using the reference library.[16]

Considering the requirements for open-source, self-hostable options with robust metadata filtering (for privacy and dynamic gallery/search functionality) and scalability, the following are evaluated:

- **FAISS:** Primarily a library for efficient similarity search algorithms, not a full database solution.[19] It lacks features for data management, metadata filtering, and production deployment without significant integration effort. While performant, especially with GPUs [20], it's unsuitable as the primary vector store here.
- **ChromaDB:** An open-source vector store focused on developer experience and ease of use, particularly popular within Langchain RAG workflows.[19] Its Python-native design and simple API make it excellent for rapid prototyping and smaller-scale applications.[19] However, it may lack the advanced enterprise features, proven large-scale scalability, and potentially the performance of metadata filtering required for this production system compared to Milvus or Weaviate.[15]
- **Milvus:** A highly scalable, open-source vector database designed for large-scale AI applications.[19] It supports distributed architectures, billions of vectors,

numerous index types (including IVF, HNSW, DiskANN), hybrid search via scalar filtering, and RBAC.[15] While powerful and suitable for demanding production workloads, it generally involves higher operational complexity compared to Chroma or Weaviate.[19]

- **Weaviate:** An open-source vector database that uniquely combines vector search with structured data concepts, often represented as a knowledge graph.[19] It offers robust metadata filtering, hybrid search capabilities [22], a GraphQL API [19], and a modular architecture supporting multiple vector index types.[19] It provides both self-hosted and managed cloud options [21] and integrates well with Langchain.[23]

Given the need for robust metadata filtering (essential for privacy enforcement and dynamic gallery/search functionality [15]), the potential need for future scalability, and the benefits of hybrid search, **Weaviate** or **Milvus** are the recommended vector databases. Chroma, while simpler to start with [19], might prove limiting as the application scales and filtering requirements become more complex. The choice between Weaviate and Milvus depends on factors like API preference (GraphQL vs. SDKs), the importance of Weaviate's graph features, or the need for Milvus's wider array of index types.[15] Weaviate's Langchain integration for hybrid search appears mature.[23] Selecting one of these more feature-rich options aligns better with the platform's long-term requirements, even if it entails slightly higher initial setup complexity compared to Chroma.

Table II.C: Open Source Vector Database Comparison

Feature	ChromaDB	Milvus	Weaviate	FAISS (Library)	Recommendation for Project
Ease of Use	High; Python-native, simple API [19]	Moderate; higher operational complexity [19]	Moderate-High; good documentation, API options [19]	Low (as DB); requires integration [19]	Weaviate/Milvus (Trade-off)
Scalability	Lower; single-node focused	Very High; distributed architecture, billions of	High; scalable architecture	N/A (Library scales with implementati	Weaviate/Milvus

	initially [15]	vectors [15]	[19]	on)	
Index Types	Limited compared to Milvus/Weaviate	Extensive (14+ types incl. HNSW, IVF, DiskANN, GPU) [15]	Supports multiple index types (e.g., HNSW) [19]	Multiple algorithms (IVF, HNSW, etc.) [20]	Milvus (Most options)
Metadata Filtering	Yes (scalar filtering) [15]	Yes (scalar filtering, advanced pre/post filtering) [15]	Yes (robust filtering capabilities) [22]	N/A (Handled by surrounding system)	Weaviate/Milvus
Hybrid Search	Yes (scalar filtering) [15]	Yes (scalar filtering, improving prefiltering) [15]	Yes (keyword + vector search) [23]	N/A (Search algorithms focus)	Weaviate/Milvus
Langchain Integration	Very Tight [19]	Good [20]	Good [23]	Yes (as index component)	Weaviate/Milvus/Chroma
Primary Use Case	RAG prototyping, smaller systems [19]	Large-scale, high-performance AI applications [20]	Vector search + structured data/graphs, flexible apps [19]	High-speed similarity search component [19]	Weaviate/Milvus
Operational Complexity	Low [19]	High [19]	Moderate [21]	N/A	Weaviate (Potentially lower)
Suitability for Project	Good for prototype, potentially limited for prod	Excellent fit for scale/filtering, higher ops	Excellent fit for scale/filtering, potentially easier ops than Milvus	Unsuitable as standalone DB	**Weaviate or Milvus**

D. Mobile Application Platform: Leveraging Expo

The requirement specifies using a free or cheap platform like Expo for the mobile app development, primarily for user registration and asset uploads. Expo is a framework and platform built around React Native, enabling the development of native iOS and Android apps (and web apps) using JavaScript and TypeScript.[25]

Expo provides Expo Application Services (EAS), a set of cloud services that significantly simplify the process of building, submitting, and updating mobile applications.[25] This aligns well with potentially reducing development and operational overhead. Expo boasts a large collection of pre-built modules and a vibrant community, often simplifying tasks like accessing device features or implementing push notifications.[25]

Regarding cost, Expo offers a compelling free tier that includes a reasonable number of monthly builds (30) and updates for a limited number of monthly active users (1,000 MAUs).[26] Paid plans scale based on usage, offering more resources like build credits, higher MAU limits for updates, increased build concurrency, and larger storage/bandwidth allowances.[26] This pricing structure fits the "free or cheap" requirement, allowing the project to start with minimal cost and scale financially as it grows.

For file uploads, the Expo/React Native app will not handle storage directly. Instead, it will use standard web technologies (like the fetch API or libraries such as axios) to send the selected files (text, PDF, images, video) via HTTP POST requests to a dedicated backend microservice built with FastAPI. This backend service will be responsible for receiving the files, authenticating the user, associating the files with the user account, and storing them appropriately (e.g., in cloud storage like AWS S3 or a persistent volume). Concerns about the free tier limitations of services like Firebase Storage [27] further justify the approach of handling uploads via a custom backend endpoint.

Therefore, **Expo is confirmed as a suitable and cost-effective platform** for developing the mobile application component. A dedicated FastAPI microservice must be designed and implemented to handle the reception and storage of file uploads initiated from the Expo application.

III. AI-Powered Profile Creation and Management

The core value proposition involves using AI, particularly Langchain and LLMs, to automatically generate rich user profiles from uploaded assets. This requires a

pipeline encompassing asset ingestion, processing, AI analysis, RAG, structured generation, and user-controlled refinement.

A. Ingesting and Processing User Assets (Text, PDF, Images, Video)

The first step involves receiving and processing the diverse digital assets uploaded by users via the Expo mobile app.

Receiving Service: A dedicated FastAPI microservice should handle file uploads. This service needs to authenticate requests, identify the user, validate file types/sizes, and store the raw files in a designated, user-specific location (e.g., a cloud storage bucket like S3, or a persistent volume if self-hosting). Upon successful upload, this service should trigger subsequent processing tasks.

Asynchronous Processing: Given that processing steps like OCR, video frame extraction, or complex document analysis can be time-consuming, it's crucial to perform them asynchronously. The upload endpoint should quickly save the file and enqueue a processing task using a distributed task queue system like Celery (discussed further in Section VII.C). This prevents blocking the upload request and provides a responsive user experience. The user receives immediate confirmation of the upload, while the processing happens in the background.

Text Extraction (PDF, DOCX): For extracting textual content from documents:

- **PDFs:** pypdf is a well-maintained library suitable for extracting text from standard PDFs.[28] PyMuPDF (fitz) is another strong alternative.[28]
- **DOCX:** The python-docx library is the standard choice for reading .docx files.
- **Advanced Needs:** If documents contain complex layouts, tables, or embedded images requiring OCR, libraries like Unstructured [30] or Kreuzberg [32] offer more sophisticated parsing. Unstructured can identify document elements (paragraphs, tables) and perform OCR, providing structured output (even HTML for tables) useful for semantic chunking.[30] It requires extra dependencies (like poppler-utils for local use [30]) and potentially an API key for its hosted service. Azure Document Intelligence is another option providing layout analysis and Markdown output.[33]
- *Recommendation:* Begin with pypdf and python-docx for simplicity. Evaluate Unstructured [30] if basic extraction proves insufficient for capturing necessary details from complex PDFs.

Image Processing (Metadata, Preparation): To handle images:

- Use the **Pillow** library (a fork of PIL).[34] It excels at opening various image formats,

performing basic manipulations (resizing, format conversion if needed for consistency), and extracting metadata.[34] Specifically, Image.getexif() combined with PIL.ExifTags.TAGS allows retrieval of standard EXIF metadata (camera details, capture time, potentially GPS coordinates if present and permitted) in a readable format.[36] Basic image properties like dimensions and format are also readily available.[36] While OpenCV (cv2) is more powerful for computer vision tasks [35], Pillow is sufficient and simpler for metadata extraction and preparation.[34]

Video Processing (Frame Extraction, Thumbnails): For video content:

- The goal is to extract visual information for AI analysis or thumbnails, not complex editing. **MoviePy** is recommended.[38] It provides a higher-level interface than direct FFmpeg bindings (ffmpeg-python [41]) or OpenCV for video file operations.[38] MoviePy can easily iterate through video frames (iter_frames [41]) or extract specific frames (get_frame [39]) to be saved as images. Note that MoviePy relies on FFmpeg being installed on the system.[40] OpenCV (cv2.VideoCapture, cap.read() [38]) is also capable but often geared more towards real-time processing.[41]
- The processing task should extract a set number of representative frames (e.g., one every few seconds, or keyframes if detectable) or generate a specific thumbnail image. These extracted frames/thumbnails can then be passed to the image analysis AI (Section III.B).

The output of these processing tasks (extracted text, image metadata, video frames/summaries) should be stored in a standardized format, linked to the original asset and user ID, ready for the subsequent AI analysis and RAG embedding stages.

Table III.A: Recommended Python Libraries for Asset Processing

Asset Type	Recommended Library(ies)	Key Features / Justification	Relevant Sources
PDF Text Extraction	pypdf	Actively maintained, standard library for extracting text from most PDFs. Simple API.	[28]
DOCX Text Extraction	python-docx	Standard, widely used library	(Standard Practice)

		specifically for reading .docx file content and structure.	
Advanced Doc Proc.	Unstructured	Handles complex layouts, tables, OCR within PDFs. Provides structured element output. Local/API options.	[30]
Image Metadata/Prep	Pillow	Extracts EXIF metadata, basic properties (size, format). Handles various formats, simple API for basic image ops.	[35]
Video Frames/Thumbs	MoviePy	High-level interface for video operations (frame extraction). Easier than direct FFmpeg/OpenCV for this task. Requires FFmpeg.	[38]

B. Generating Rich Image Metadata: AI vs. Traditional Approaches

The user requires LLM-generated metadata for images depicting farm scenes (buildings, fields, equipment, grains, people). This goes beyond traditional metadata.

Traditional EXIF metadata, extractable using Pillow [36], provides factual context like camera model, timestamp, and potentially GPS coordinates, but it cannot describe the *content* or *semantic meaning* of the image.

AI-powered metadata generation, typically through **image captioning** or **Visual Question Answering (VQA)** models, is necessary to meet the requirement for descriptive metadata. These are multimodal models trained to understand visual content and generate relevant text.[42] For the farm images described, such a model could generate descriptions like "A modern combine harvester working in a golden

wheat field at sunset" or "Close-up of healthy grain kernels in a storage silo."

Several open-source multimodal models are suitable candidates:

- **BLIP / BLIP-2:** Specifically designed for captioning and VQA tasks. BLIP-2 demonstrated strong performance in comparative studies.[42]
- **LLaVA:** A popular model known for its visual instruction-following capabilities, suitable for generating descriptions or answering questions about images.
- **Others:** Models like GIT, Qwen2-VL, and Cosmos 2 have also shown promise.[42] Some models might excel at specific styles, like more atmospheric descriptions (Gemini, Qwen2-VL [42]) or concise tags versus full sentences.[44] CLIP itself is primarily an encoder used *within* these models, not a generator.[43]

Implementation typically involves using the Hugging Face transformers library in Python to load a pre-trained multimodal model and its associated processor.[45] The processor prepares the image input, and the model generates the text output (caption). Running these models effectively usually requires GPU acceleration due to their size and computational demands. Self-hosting provides control but requires managing the infrastructure, while cloud-based APIs (if available for suitable models) offer an alternative.

The recommended approach is to implement **AI-based image captioning** using a state-of-the-art open-source multimodal LLM. **BLIP-2** or **LLaVA** serve as good starting points due to their focus on captioning/VQA and availability.[42] Evaluation on domain-specific images (farm photos) is advised to select the best-performing model. The AI-generated captions should be **combined with the factual EXIF metadata** extracted using Pillow.[36] This combination provides a richer dataset for the subsequent profile generation LLM: the AI caption describes *what* is in the image, while EXIF provides context about *when, where, and how* it was captured. This combined metadata should be associated with the image and prepared for RAG indexing.

C. Text Processing, Embedding, and RAG Preparation

To enable RAG functionality for profile generation and search, the extracted text content (from documents, AI image captions) needs to be processed, converted into embeddings, and stored efficiently.

Text Chunking: Large documents exceed LLM context limits and hinder precise retrieval.[16] Chunking breaks text into smaller, semantically meaningful units.

- *Fixed-Size Chunking:* Methods like CharacterTextSplitter [16] or

RecursiveCharacterTextSplitter [46] split by character count with overlap. They are simple but risk breaking sentences or paragraphs mid-thought, potentially harming semantic coherence.[33]

- *Semantic Chunking:* This approach aims to split along natural boundaries like paragraphs, sections, or based on document layout analysis. Tools like Unstructured [30] or Azure Document Intelligence [33] can identify these structures during initial processing. This method generally yields higher-quality chunks for RAG as it preserves context better.
- *Recommendation:* Prioritize **semantic chunking**. If using Unstructured for PDF/DOCX processing, leverage its output structure (paragraphs, titles, etc.) to define chunks. If only simple text extraction is available, use Langchain's RecursiveCharacterTextSplitter [46], carefully tuning chunk size and overlap to balance granularity and context preservation. For tabular data identified during extraction [30], consider embedding a textual summary of the table rather than trying to embed the raw structure directly.

Embedding Model: An embedding model transforms text chunks into dense numerical vectors that capture semantic meaning, enabling similarity comparisons.[13]

- *Options:* High-quality commercial models like OpenAI's embeddings [16] are available via API but incur costs. Numerous open-source models, particularly Sentence Transformers (SBERT) [48] like all-MiniLM-L6-v2 [17] or higher-performing models from benchmarks (e.g., MTEB leaderboard), can be run locally or via inference endpoints.[46]
- *Recommendation:* Aligning with the open-source preference, start with a strong **open-source Sentence Transformer model** (e.g., all-mpnet-base-v2, bge-large-en, or similar) accessed via Langchain's HuggingFaceEmbeddings integration. Ensure the model choice is suitable for the agricultural/business domain. Crucially, the **same embedding model must be used for indexing all content (user profiles, reference library) and for embedding user queries** during retrieval.[18] Maintain flexibility to switch to a paid API if performance necessitates it.

Vector Storage: The generated embeddings, along with the original text chunk and relevant metadata, must be stored in the chosen vector database (Weaviate or Milvus recommended in Section II.C).[14]

- *Metadata Inclusion:* It is critical to store comprehensive metadata alongside each vector embedding. This must include: user_id, source_asset_id (e.g., filename or URL), chunk_id or sequence number, the assigned privacy_level (from user input, see Section III.F), content_type (e.g., 'pdf_text', 'image_caption',

'reference_article'), and potentially keywords or topic tags. This metadata is essential for filtering retrieval results based on user identity, privacy settings, content source, or other criteria during profile generation, search, and gallery building.[15]
- *Langchain Integration:* Utilize the Langchain vector store integrations (e.g., WeaviateVectorStore [24]) for adding (add_documents [14]) and searching these embeddings.

D. Langchain for Automated Profile Drafting and Structured Output

The system needs to generate a structured sales profile using an LLM, informed by the user's processed and embedded content via RAG. Langchain provides the necessary components to orchestrate this.

Core RAG Workflow:

- A Langchain chain, likely built using LangChain Expression Language (LCEL) [46] or potentially the RetrievalQA chain [16], forms the backbone.
- This chain takes an initial instruction (e.g., "Generate a profile for user X") and potentially user-provided refinement instructions (see Section III.E).
- A **Retriever** component queries the vector store (Weaviate/Milvus) to find text chunks relevant to the user and the task, using the user ID and appropriate privacy filters based on the context (profile generation should access all approved user content). Langchain vector store integrations provide the .as_retriever() method.[16]
- A **Prompt Template** structures the input for the LLM, incorporating the retrieved context ({context}) and the specific task instructions ({instructions}, {question}).[17] Example: "Based on the following materials provided by the farmer ({context}), generate a compelling sales profile highlighting their unique offerings (products, certifications, facilities). Follow any specific user instructions: {user_instructions}. Structure the output according to the provided schema."
- An **LLM** (integrated via Langchain wrappers like ChatOpenAI, Ollama, HuggingFaceHub [16]) receives the formatted prompt and generates the profile content. The choice of LLM (e.g., GPT-4o, Llama 3, Mixtral) will impact quality and cost. Flexibility to switch models is beneficial.

Ensuring Structured Output: The generated profile needs a consistent structure (e.g., JSON) for reliable use by the frontend and caching mechanisms. Relying solely on prompt instructions for formatting is unreliable.[54] The recommended approach is to use model features that enforce schema adherence:

- **Function/Tool Calling:** Define the desired output structure using a Pydantic

model in Python [56] or a JSON Schema.[57] This schema represents the profile fields (e.g., profile_title: str, summary: str, key_products: List[str], facility_details: str, relevant_image_ids: List[str]).
- **Langchain Integration:** Use Langchain's with_structured_output() method [55] or bind_tools() [56] to bind this schema to an LLM that supports tool/function calling (e.g., OpenAI models [54], Gemini [60], many modern open-source models). This forces the LLM to generate a response that conforms strictly to the defined schema, typically as a JSON object.[54] with_structured_output conveniently handles both binding the schema and parsing the LLM's response into the specified Pydantic object or dictionary.[55]

This approach provides high reliability for the structured profile data needed downstream.[54] The resulting JSON object can then be easily rendered into HTML for web display or used by other services.

E. Implementing the Indirect Profile Editing Workflow

Users need to refine their AI-generated profiles without direct text editing, instead using content management and natural language instructions [User Query, items 7, 8].

Mechanism:

1. **Content Management:** The user interface must display the list of assets (PDFs, images, etc.) contributing to the current profile. Users should be able to select or deselect assets. This action updates a status flag associated with the asset or its chunks (e.g., in PostgreSQL or as vector store metadata). During the RAG retrieval step for profile regeneration, the retriever must be configured to *only* consider chunks from 'active' or 'selected' assets belonging to that user. Removing an asset effectively removes its content from the context used for generation.
2. **Natural Language Instructions:** A dedicated input field in the review/edit web form allows users to provide instructions like "Emphasize our commitment to sustainable practices" or "Remove mentions of specific older equipment."
3. **Instruction Integration & Regeneration:**
    - *Simple Prompt Injection:* The easiest method is to incorporate the user's natural language instructions directly into the profile generation prompt template passed to the LLM (e.g., "...Follow these specific instructions: {user_instructions}."). The LLM attempts to follow these directions during generation.
    - *Agentic Interpretation (Advanced):* For more complex instructions requiring interpretation or specific actions beyond simple stylistic changes, a Langchain Agent could be employed.[16] The agent would first receive the

instruction, use an LLM (and potentially custom tools) to understand its intent (e.g., "emphasize sustainability" might mean retrieving and highlighting specific chunks about certifications or practices), and then guide the RAG and generation process accordingly. This could involve modifying the retrieval query, filtering retrieved chunks, or performing post-generation editing based on the instruction. Function calling could parse the natural language instruction into a structured command for the agent or generation process.[58]

4. **Preview and Approval:** After any content change or new instruction, the RAG profile generation process (Section III.D) is re-executed. The resulting structured profile data is rendered and presented to the user in the web form for preview. The user can then either approve this version (which updates the live profile and potentially the cache) or provide further instructions/content modifications, continuing the iterative refinement loop.

Recommendation: Implement the **content management** feature allowing users to toggle asset inclusion, ensuring the RAG retriever filters based on this status. For **natural language instructions**, begin with **simple prompt injection**. This is easier to implement initially. If this proves insufficient for handling the nuance of user requests, transition to the more sophisticated **Agentic Interpretation** approach using Langchain Agents and potentially function calling to parse instructions into actionable steps.[58] The iterative nature of this workflow (Generate -> Review -> Modify -> Regenerate -> Approve) requires efficient regeneration and clear state management.

F. Designing Granular Privacy Controls

The platform requires fine-grained privacy settings for individual profile elements (text segments, images, videos), with levels like "public," "registered users," "verified users," "permission only," and "private" [User Query, item 9].

Implementation Strategy:

1. **Metadata Storage:** The privacy level chosen by the user for each piece of content must be stored reliably.
    - Store the privacy level as metadata associated with each text chunk in the **vector store**. This allows for efficient filtering during RAG retrieval.[15]
    - Also store the privacy level (or link to a privacy setting) in the **PostgreSQL** database, associated with the resource identifier (e.g., image ID, document ID, potentially generated text segment ID). This provides an authoritative source for access control checks, especially for complex rules. While seemingly redundant, this dual storage supports both efficient retrieval filtering and robust application-level enforcement.

2. **User Interface:** The profile editing interface must allow users to assign a privacy level to each distinct component of their profile (e.g., individual images displayed, specific paragraphs generated by the LLM). This requires that the generated profile structure (from III.D) be sufficiently granular or that the UI allows tagging sections of the rendered profile.
3. **Access Control Implementation:** Enforcement requires backend logic.
    - **Role-Based Access Control (RBAC):** Define roles corresponding to the privacy levels (e.g., anonymous, registered, verified, owner). Additionally, handle the "permission only" case, which implies specific user-to-resource grants.
    - **Casbin Integration:** Use an RBAC library like **Casbin** [62], which integrates with FastAPI via libraries like fastapi-user-auth [63] or fastapi-authz.[64] Casbin allows defining policies (e.g., allow, verified_role, resource_X, read if resource_X.privacy == 'verified') that can be evaluated at runtime. Policies can be stored in files or the database.
    - **Layered Filtering:**
        - *Retrieval Filtering:* During RAG (for profile display, search, etc.), use the vector store's metadata filtering capabilities [15] as the first line of defense. The retriever should only fetch chunks whose privacy_level metadata matches or is less restrictive than the *viewer's* role (e.g., a registered user can see public and registered content).
        - *Application-Level Filtering:* After retrieving potential content or generating profile data, apply Casbin policy checks within the FastAPI service before returning data to the user. This is essential for enforcing complex rules and ensuring correctness, especially for "permission only" grants which likely require checking a separate permissions table in PostgreSQL.
4. **Database Schema (PostgreSQL):** Design the schema to support this.[66] Include tables for Users, Roles, UserRoles (linking users to roles), Resources (representing profile items), ResourcePrivacy (linking resource ID to privacy level string or enum), and potentially ResourcePermissions (mapping user_id, resource_id, permission_type for specific grants). PostgreSQL's built-in roles [10] or row-level security [12] could also be leveraged, but an application-level RBAC system like Casbin often provides more flexibility.

Recommendation: Store privacy levels as **metadata in both the vector store and PostgreSQL**. Implement **RBAC using FastAPI with Casbin**.[62] Employ a **layered filtering approach**: use vector store metadata filtering during RAG retrieval for efficiency [15], followed by authoritative application-level checks using Casbin policies

before presenting data.

G. Caching Generated Profile Content

To reduce latency when displaying profiles in dynamic galleries, the generated profile web object should be cached [User Query, item 10].

Caching Strategy:

- **Target:** Cache the final, structured representation (e.g., the approved JSON object from III.D, or potentially pre-rendered HTML if rendering is complex) of a user's profile. Caching should occur *after* the user approves a version.
- **Storage:** Given the microservices architecture, a distributed cache is necessary for consistency across service instances. **Redis** is the recommended choice due to its speed, widespread adoption, and excellent integration with Python/FastAPI using libraries like aioredis [68] or aiocache.[69] In-memory caches (cachetools [70]) are unsuitable as they are not shared between service instances.
- **Key:** Use the user_id or a unique profile_id as the cache key.
- **Invalidation:** Stale profile data in galleries is unacceptable. Therefore, **event-based invalidation** is required. When a user successfully approves a new profile version via the corresponding API endpoint, that endpoint must explicitly trigger an update or deletion of the cached entry for that user_id in Redis. Time-To-Live (TTL) caching [69] is too imprecise for this requirement, as it could lead to outdated profiles being displayed until the TTL expires.

Implementation: Integrate **Redis** as a distributed cache.[68] Implement logic within the profile approval service/endpoint to **invalidate/update the Redis cache** for the specific user upon successful approval. Services responsible for fetching profiles for display (e.g., the gallery service) should check Redis first using the user_id key. If a cache hit occurs, return the cached data. If a miss occurs (which should ideally only happen if a profile hasn't been approved/cached yet), fetch from the primary source (PostgreSQL) or handle appropriately (e.g., indicate the profile is unavailable). FastAPI dependencies or middleware can encapsulate the cache-checking logic.[69]

IV. Dynamic Profile Gallery Implementation

Galleries provide curated views of user profiles based on specific criteria, leveraging the generated profiles and metadata.

A. Logic for Dynamic Gallery Construction

Galleries need to be built dynamically based on criteria like region or product

offerings, using user metadata [User Query, item 11].

Filtering Mechanism:

- The primary source for filtering criteria (region, product categories, verified status, collection membership) will be the structured data stored in **PostgreSQL**.
- A dedicated backend service (e.g., a Gallery Service microservice) will receive requests specifying the gallery criteria (e.g., via path parameters like /galleries/region/{region_name} or query parameters).
- This service executes a query against the PostgreSQL database to identify users matching the criteria. For example: SELECT user_id FROM UserProfiles JOIN Users ON UserProfiles.user_id = Users.id WHERE UserProfiles.region = :region AND UserProfiles.main_product = :product AND Users.status = 'active' AND Users.privacy_allows_gallery = TRUE. This query returns a list of user_ids eligible for inclusion in the specific gallery. Privacy settings (e.g., a user opting out of gallery inclusion) must be respected in this query.

B. Efficiently Embedding Profiles

Once the list of candidate user_ids is obtained, the gallery service needs to assemble the actual profile content for display.

Leveraging the Cache:

- For each user_id returned by the database query, the gallery service attempts to retrieve the corresponding cached profile object (JSON or HTML snippet) from **Redis**, using the user_id as the key (as implemented in Section III.G).
- **Cache Hits:** If the profile is found in Redis, this cached version is used directly.
- **Cache Misses:** A cache miss during gallery generation should ideally not trigger profile regeneration. The intended flow is that profiles are cached upon user approval. A miss likely means the profile isn't approved, isn't meant to be public, or there's an issue with the caching/invalidation logic. The service might choose to simply omit users with cache misses from the current gallery view or log an error. Fetching from the primary DB (PostgreSQL) might be a fallback, but the cached version is preferred for performance.
- **Rendering:** The gallery service collects the successfully retrieved (cached) profile objects. It then formats these into the final gallery structure (e.g., a JSON list of profile summaries, or server-rendered HTML containing profile cards) to be sent to the frontend client. Implement pagination (e.g., using LIMIT and OFFSET in the initial database query and subsequent cache lookups) to handle potentially large numbers of matching profiles efficiently.

FastAPI Implementation: Define FastAPI endpoints corresponding to different gallery types (e.g., /galleries/regional, /galleries/product, /galleries/recent).[73] These endpoints encapsulate the logic: query PostgreSQL for user IDs based on criteria, fetch corresponding cached profiles from Redis, handle pagination, and return the formatted gallery data.

V. Intelligent Search and Matchmaking

The platform aims to provide an intelligent search function that goes beyond simple keyword matching, using LLMs and RAG to identify high-value prospects based on user-defined criteria.

A. Candidate Selection via Metadata Filtering

The search process begins by narrowing down the pool of potential matches using structured metadata.

Implementation:

- A web form allows the searching user (e.g., a buyer) to specify criteria (e.g., commodity type: "Hard Red Winter Wheat", location: "Kansas, USA", certifications: "Organic", required volume: "> 500 tons").
- The search microservice receives these criteria.
- Similar to gallery generation (IV.A), the service first executes a query against the **PostgreSQL** database using these structured criteria. This query filters the entire user base (sellers, in this case) down to a manageable list of **candidate user_ids** whose profile metadata matches the basic requirements.
- This pre-filtering step is crucial for efficiency, as it significantly reduces the amount of data that needs to be processed by the more computationally expensive RAG and LLM steps that follow.

B. Leveraging Langchain and RAG for Matchmaking

Once a list of candidates is identified via metadata filtering, RAG and LLMs are used to perform a deeper semantic analysis and comparison.

RAG Context Construction:

- Dynamically build a temporary RAG context containing relevant, accessible content chunks from the profiles of *only* the candidate users identified in step V.A.
- This involves retrieving embeddings and text chunks from the vector store (Weaviate/Milvus) associated with the candidate user_ids.
- Crucially, **privacy filters** must be applied during this retrieval based on the

searching user's permissions. For example, a registered buyer might only be able to access 'public' and 'registered user' level content from the candidates' profiles. Use the vector store's metadata filtering capabilities.[15]

Langchain Implementation:

- **Prompt Engineering:** Design sophisticated prompt templates for the matchmaking LLM. These templates should clearly define the LLM's role (e.g., "expert commodity broker"), incorporate the specific search criteria provided by the user ({search_criteria}), and instruct the LLM to analyze the provided candidate profile context ({context}) to identify the best matches and explain the reasoning. Example: "Analyze the following profiles of potential suppliers: {context}. Identify the top 3 suppliers that best meet the following requirements: {search_criteria}. Provide a brief justification for each recommendation, referencing specific details from their profiles."
- **Retriever Configuration:** Configure a Langchain retriever to operate *only* on the temporary RAG context built from the filtered candidates. Efficient metadata filtering within the vector store query is key here.
- **Chain Execution:** Utilize a Langchain chain (LCEL preferred [49]) to orchestrate the flow: take the user's search criteria, use the configured retriever to fetch relevant chunks from the candidate context, format the prompt, and invoke the LLM.[16]
- **Handling Multiple Candidate Contexts:** Comparing multiple profiles within a single RAG call presents challenges.[74] The LLM needs to differentiate information belonging to different candidates. Potential strategies include:
    - *Standard RAG:* Retrieve the most relevant chunks across all candidates based on the search criteria and feed them into a single context window for the LLM to synthesize.[47] This is simplest but risks confusing the LLM if context attribution isn't clear.[74] Requires careful prompt design.
    - *Per-Document QA then Synthesis:* Run a RAG QA process individually for each candidate against the search criteria ("Does candidate A meet the criteria? Why/how?"). Then, feed these individual evaluations/summaries into a second LLM call tasked with comparing and ranking the candidates.[74] This adds complexity but provides clearer separation. Langchain's document comparison tools might offer relevant patterns.[75]
    - *Advanced Retrieval:* Techniques like RAG Fusion (generating multiple query variants) [76] or using ensemble retrievers [50] could potentially improve the quality of retrieved chunks across diverse candidates, indirectly aiding the comparison task.
- *Recommendation:* Begin with the **Standard RAG approach**, carefully crafting the prompt to guide the LLM in comparing candidates based on the combined

context.[47] Ensure the LLM's context window is adequate. If this yields poor results (e.g., inaccurate comparisons, hallucinations about which candidate has which attribute), then investigate the more complex **Per-Document QA then Synthesis** approach.[74]

C. Presenting Search Results and Summaries

The output of the matchmaking process needs to be presented clearly to the user.

Structured Output: Employ the reliable structured output mechanism (Function/Tool Calling with a Pydantic schema via with_structured_output [54]) for the matchmaking LLM call (V.B). The schema should enforce a format like:

JSON

```
{
  "search_summary": "Overall observations about the matches found...",
  "top_prospects":
    },
    //... other prospects
  ],
  "notes": "Consider verifying transport availability for candidate_2..."
}
```

This ensures the backend receives predictable, easily parsable data.

Gallery Generation: Extract the list of user_ids from the top_prospects field in the LLM's structured output. Use this list to dynamically construct a gallery of these specific candidates, fetching their cached profile objects from Redis (as detailed in IV.B).

Display: The frontend application receives the structured JSON response. It displays the textual search_summary and notes, iterates through the top_prospects to show the reasoning for each match, and renders the dynamically generated gallery of candidate profiles.

VI. Inter-User Communication

A simple, private communication system is required to allow registered users to initiate

contact and exchange information, designed with potential network restrictions in mind.

A. Designing a Simple Messaging System with FastAPI

Technology Choice: WebSockets
WebSockets provide a suitable technology for enabling real-time, bidirectional communication between users, fitting the requirement for a chat-like system. FastAPI offers robust, built-in support for WebSockets, leveraging the underlying capabilities of the Starlette framework.77

Implementation Details:

1. **WebSocket Endpoint:** Define a WebSocket endpoint in a dedicated FastAPI microservice, likely parameterized by user ID (e.g., @app.websocket("/ws/{client_id}")).[78]
2. **Connection Management:** Implement a ConnectionManager class or a similar pattern within the service instance.[80] This manager will maintain a dictionary or list mapping authenticated user_ids to their active WebSocket connection objects.
3. **Connection Handling:** When a client attempts to connect, the endpoint authenticates the user (see below) and, if successful, calls websocket.accept() [78] and registers the connection (connectionmanager.connect(websocket, user_id)).[80]
4. **Message Routing:** Inside a loop (while True:), use data = await websocket.receive_text() to listen for incoming messages.[78] Parse the message to determine the recipient user_id. Look up the recipient's active WebSocket connection in the ConnectionManager. If found, use await recipient_websocket.send_text(message_content) to forward the message.[80]
5. **Offline Messaging:** If the recipient is not currently connected (no active WebSocket in the manager), store the message in a dedicated Messages table in PostgreSQL, linked to sender and recipient IDs. Implement logic for delivering stored messages when the recipient next connects.
6. **Disconnection Handling:** Wrap the message receiving loop in a try...except WebSocketDisconnect: block to gracefully handle client disconnections and remove the connection from the ConnectionManager (connectionmanager.disconnect(websocket, user_id)).[77]
7. **Authentication:** Secure the WebSocket endpoint. Since standard headers might be tricky after the initial HTTP upgrade request, authentication typically involves passing a token (e.g., JWT) as a query parameter (/ws/{client_id}?token=...) or potentially via a cookie during the initial handshake.[78] Use FastAPI's dependency injection system within the WebSocket endpoint function to validate the token

and identify the user before accepting the connection.[78]

8. **Simplicity:** Adhere to the requirement for simplicity. Focus on 1-to-1 text messaging for exchanging contact details. Avoid implementing features like group chat, file sharing, read receipts, or complex presence indicators initially. Persist message history in PostgreSQL for retrieval.

Alternative (Simpler, GFW-Resilient): Asynchronous Notifications: If real-time chat proves too complex or faces significant GFW issues (see VI.B), a fallback is an asynchronous messaging system. Users compose messages via a standard HTTP POST request. The backend stores the message in PostgreSQL and triggers an email or in-app notification to the recipient. This avoids persistent WebSocket connections entirely but sacrifices real-time interaction.

B. Navigating Network Restrictions (e.g., Great Firewall of China)

The Great Firewall of China (GFW) employs various techniques, including Deep Packet Inspection (DPI), IP/DNS blocking, keyword filtering, and connection resets, to control internet access.[81] This poses a potential challenge for the WebSocket-based chat system. Standard VPNs are often unreliable as they are actively targeted.[81]

Mitigation Strategies for WebSocket Reliability:

1. **Use WSS (WebSocket Secure): Essential.** Always use WSS, which encrypts WebSocket traffic over TLS/SSL. This prevents trivial keyword filtering of the payload by DPI and makes the traffic resemble standard HTTPS traffic.[83]
2. **Standard Port (443):** Run the WSS endpoint on the standard HTTPS port (443). Traffic on common ports is less likely to be blocked outright than traffic on non-standard ports.[83] Firewalls (including the GFW) are configured to allow standard web traffic.
3. **Avoid Direct IP Addresses:** Rely on domain names for connection endpoints. IP blocking is a primary GFW technique.[82] Ensure DNS resolution works reliably, potentially using DNS over HTTPS (DoH) on the client side if possible, although server-side DNS is more critical here.
4. **Content Delivery Network (CDN) with China Presence:** Utilizing a CDN with Points of Presence (PoPs) inside or near mainland China can significantly improve performance and reliability for users within China.[84] This reduces the need for traffic to traverse the GFW multiple times. However, CDNs operating in China must comply with local regulations.[84] A multi-CDN strategy might offer redundancy.[87]
5. **Simple Payloads:** While WSS encrypts the content, keeping message payloads simple and avoiding known sensitive keywords might reduce the (already low) risk

of triggering heuristic-based blocking, although this is speculative.

6. **Keep-Alives:** Ensure appropriate keep-alive mechanisms are configured for the WebSocket connection to prevent premature closure by intermediate proxies or firewalls that might not fully understand persistent connections.[88]
7. **Avoid Obfuscation:** While techniques exist to disguise traffic (e.g., Shadowsocks, V2Ray, Obfsproxy) [83], implementing them adds significant complexity and maintenance burden. They are part of an ongoing cat-and-mouse game with censors and offer no guarantee of success.[83] It's generally not advisable for a standard application feature.
8. **Testing and Monitoring:** The most crucial step is to **test connectivity and performance from within China** using real users or specialized monitoring services.[84] Monitor connection success rates, latency, and message delivery for users in affected regions.

Recommendation: Implement the chat using **WSS on port 443**. Host the backend on robust infrastructure using **domain names**. Strongly consider employing a **CDN with China PoPs**.[84] Keep the chat protocol and features simple. **Monitor performance from China**. Be aware that perfect reliability cannot be guaranteed. If WSS proves consistently problematic, the asynchronous notification system (VI.A) offers a more resilient, albeit non-real-time, alternative as it relies solely on standard HTTPS requests.

Table VI.B: Great Firewall Compatibility Strategies for WebSocket Chat

Strategy	Description / Rationale	Pros	Cons / Risks	Recommendation	Relevant Sources
Use WSS / Port 443	Encrypts traffic (like HTTPS), uses standard port less likely to be blocked.	Essential for security, harder to inspect.	None (standard best practice).	**Essential**	83
CDN with China PoPs	Caches content/proxies	Improved speed & reliability in	Cost, CDN must comply with CN	**Recommended**	84

	connections closer to users, reduces GFW traversal.	China.	regulations.		
Use Domain Names	Avoids direct IP blocking, a common GFW tactic.	More resilient than static IPs.	DNS can still be manipulated (less likely for WSS).	**Essential**	[82]
Simple Payload/Protocol	Reduces potential attack surface or triggers for heuristic blocking (speculative).	Lower complexity.	May not significantly impact GFW detection.	**Recommended**	(Best Practice)
Keep-Alives	Prevents intermediate proxies from closing idle connections prematurely.	Improves connection stability.	Requires correct configuration.	**Recommended**	[88]
Monitoring from China	Provides actual data on performance and reliability within the target region.	Realistic assessment of user experience.	Requires specific tools or testers.	**Essential**	[84]
Fallback (Long Polling)	Uses standard HTTPS requests if WebSockets	Increased reliability if WS blocked.	Higher latency, less efficient, added	**Optional**	[88]

	fail, potentially more resilient.		complexity.		
Obfuscation Protocols	Attempts to disguise traffic (e.g., as generic HTTPS) to evade DPI.	Potential to bypass some blocking.	Complex, unreliable, constantly targeted by GFW.	**Avoid**	[83]

VII. Integrating a Curated Reference Library

The platform requires the ability to build and maintain a repository of curated reference materials (PDFs, documents, images, scraped web content) accessible via RAG for answering user questions in combination with user-specific data.

A. Building and Maintaining the Reference RAG Repository

Content Ingestion:

- **File Uploads:** Implement an administrative interface (potentially part of a dedicated admin microservice) to allow uploading reference files (PDF, DOCX, potentially images if relevant visual information needs indexing). Use the same file processing libraries as for user assets (Section III.A: pypdf, python-docx, Pillow, potentially Unstructured [30]) to extract text and relevant metadata. Store these files in a separate "reference library" area in the chosen storage backend.
- **Web Scraping:**
  - *Mechanism:* Develop scrapers for designated URLs. For static HTML sites, requests combined with BeautifulSoup is often sufficient for fetching and parsing content.[89] For more complex sites requiring crawling multiple pages or handling forms, the Scrapy framework provides more structure and features, including asynchronous requests.[89] If target sites rely heavily on JavaScript for rendering content, browser automation tools like Selenium or Playwright [91] may be necessary, although they are slower and more resource-intensive. Langchain's WebBaseLoader [18] offers a convenient way to load content from URLs directly into Langchain documents.
  - *Extraction:* Focus on extracting the main article/content text, stripping away navigation, ads, footers, and other boilerplate to ensure clean data for embedding.

- *Legal/Ethical Considerations:* This is critical. **Always check and respect the robots.txt file and the Terms of Service (ToS) of the target websites**.[92] Many sites prohibit scraping. Only scrape publicly accessible data (no logins required). Avoid excessive request rates that could overload the target server (implement delays/throttling). If an official API is available, use it instead of scraping.[94] Be mindful of copyright; scraped content should generally only be used for internal RAG context, not republished.[92] Consult legal counsel regarding the specific URLs and intended use.

Processing and Storage:

- **Chunking and Embedding:** Apply the same text chunking strategy (semantic chunking preferred [33]) and the **identical embedding model** used for user profile data (Section III.C) to the extracted reference content.[17] Consistency in embedding model is vital for meaningful similarity search across user and reference data.
- **Vector Store Organization:** Store the reference material embeddings in the same vector database instance (Weaviate/Milvus) but logically separated from user profile data. This can be achieved using:
  - **Separate Collections/Indexes:** Most vector databases support multiple collections (e.g., a user_profiles collection and a reference_library collection). This provides clear separation.
  - **Metadata Flag:** Alternatively, store all embeddings in a single collection but include a metadata field like source_type: 'reference' or source_type: 'user_profile'. Queries can then filter by this field. Weaviate and Milvus support namespaces or partitions which can also serve this purpose.[15]
- **Metadata:** Store relevant metadata with each reference chunk embedding: original source URL or filename, scrape/upload timestamp, potentially manually assigned tags or categories for topic filtering (e.g., 'crop_disease', 'market_report', 'logistics_regulation').

B. Combining User and Reference Context in Queries

Users should be able to ask questions that require drawing information from both their specific profile context and the general reference library. Example: "Based on my listed wheat variety, what are the relevant government quality specifications from the reference library?"

Langchain Implementation:

1. **Multiple Retrievers:** Define at least two distinct Langchain retriever objects:
   - user_profile_retriever: Configured to search the vector store

collection/namespace/metadata corresponding to user profile data, filtered by the specific user_id of the user asking the question and respecting privacy settings relevant to that user accessing their own data.
    - reference_library_retriever: Configured to search the vector store collection/namespace/metadata corresponding to the reference library. This retriever might also support filtering by topic/category metadata if implemented.
2. **Combining Retrieved Context:** When a user asks a question requiring combined context:
    - *Option 1: Parallel Retrieval & Concatenation:* Use Langchain's RunnableParallel [95] or similar asynchronous execution to query both retrievers simultaneously with the user's question (or a derived query). Collect the lists of retrieved Document objects from both sources. Concatenate these lists into a single list to form the combined context. Implement de-duplication if necessary. Pass this combined list to the LLM prompt. This is the simplest approach but requires careful management of the total context size to fit within the LLM's limits.
    - *Option 2: Sequential Retrieval:* Retrieve relevant information from one source first (e.g., user profile), then use elements from that retrieved context to formulate a more targeted query for the second source (e.g., reference library). This can lead to more relevant retrieval from the second source but adds latency and complexity.
    - *Option 3: Agentic Decision Making:* Employ a Langchain Agent.[61] The agent analyzes the user's question and decides whether it needs information from the user profile, the reference library, or both. It then invokes the appropriate retriever(s) and synthesizes the information from the retrieved contexts to formulate the final answer. This offers the most flexibility but is also the most complex to implement correctly.
3. **Prompting:** The prompt template for the final LLM call must be designed to handle potentially combined context. It should instruct the LLM on how to use information from both sources if provided. Example: "Answer the user's question: '{question}'. Use the following user-specific context if relevant: '{user_context}'. Also consider the following reference material: '{reference_context}'. Synthesize this information to provide a comprehensive answer."

Recommendation: Implement the **Multiple Retriever** approach. Start with **Parallel Retrieval & Concatenation (Option 1)** due to its relative simplicity.[95] Monitor the quality of results and context window usage. If simple concatenation proves inadequate (e.g., irrelevant information dominates, context window overflows, LLM

struggles to synthesize), then explore the more sophisticated **Agentic Decision Making (Option 3)** approach.[61]

C. Automating Content Updates (Web Scraping and Scheduling)

The reference library content obtained from web scraping needs periodic updates to remain current.

Scheduling Mechanism:

- To automate the re-scraping process, a scheduling mechanism is required.
- **Celery with Celery Beat:** If Celery is already being used for asynchronous task processing (e.g., for user asset uploads as suggested in III.A), using Celery Beat for scheduling is a natural fit.[96] It leverages the existing Celery infrastructure (workers, message broker like Redis).
- **APScheduler:** A lightweight, pure-Python library specifically for scheduling tasks. It can be run within a dedicated service or even integrated into a FastAPI application using its asyncio support. It's a simpler alternative if Celery is not otherwise needed.[96]
- **System Cron:** A basic option using the operating system's cron daemon to trigger Python scraping scripts. Less integrated with the application's state and environment.
- *Recommendation:* Use **Celery Beat** if Celery is adopted for other background tasks for consistency.[96] Otherwise, **APScheduler** provides a capable and Python-native solution without the overhead of a full Celery setup.

Update Workflow:

1. The scheduler (Celery Beat or APScheduler) triggers a scraping task for a configured URL or set of URLs at the defined interval (e.g., daily, weekly).
2. The web scraping logic (from VII.A) executes, fetching and extracting the current content from the target URL.
3. Compare the newly scraped content with the version currently stored (e.g., by hashing the content or using a text diffing library).
4. If a significant change is detected:
    - Re-chunk the new content using the standard chunking strategy.
    - Generate embeddings for the new chunks using the standard embedding model.
    - Update the vector store: This might involve deleting the old embeddings/chunks associated with that URL and adding the new ones, or updating existing entries if the vector store supports updates based on IDs. Ensure metadata (like the 'last_updated' timestamp) is refreshed.

5. Log the update status (e.g., success, failure, no change detected).

Manual Trigger: Additionally, provide an administrative interface where authorized users can manually trigger an immediate update for specific URLs or the entire scraped reference library collection. This is useful for forcing refreshes outside the regular schedule.

VIII. Platform Extensions and Considerations

The user query outlines potential future extensions: seeding profiles from public data and supporting virtual seller collections.

A. Seeding Profiles from Public Sources: Technical Methods and Implications

The idea is to pre-populate the platform with profiles derived from publicly available sources (websites, directories), creating "unclaimed" accounts that real users can later verify and take over.

Technical Implementation:

1. **Scraping:** Identify target public directories or websites containing information about potential users (farmers, buyers). Develop specific scrapers for each source using tools like requests and BeautifulSoup for simpler sites, or Scrapy for more extensive crawling.[89]
2. **Data Extraction:** Parse the scraped HTML to extract key business information: name, location, contact details (email/phone if public), product types, descriptions, etc.
3. **Structuring:** Clean and structure the extracted data into a consistent format suitable for profile creation.
4. **Account & Profile Creation:** For each unique entity identified:
    - Create a user record in the PostgreSQL Users table with a status like 'seeded' or 'unclaimed'. Generate a unique user ID.
    - Create a corresponding basic profile record in a UserProfiles table, populating it with the scraped structured data.
5. **Content Generation (Optional):** The scraped data might be minimal. Consider using an LLM (similar to III.D but without RAG on user assets) to generate a brief introductory profile description based solely on the publicly scraped information.
6. **Embedding:** Embed the scraped/generated textual content associated with the seeded profile into the vector store (Weaviate/Milvus), linking it to the synthetic user_id and marking it with appropriate metadata (e.g., status: 'seeded').
7. **Claiming Process:**
    - *Discovery:* Allow users to search for potentially existing seeded profiles (e.g.,

by name and location).
    - *Verification:* Implement a secure verification mechanism for a user to claim a profile. This could involve:
        - Sending a verification link/code to a publicly listed email address associated with the profile.
        - Requiring documentation upload for manual review by administrators.
        - Domain name verification if a website is listed.
    - *Security:* This process must prevent fraudulent claims. Use standard account security practices. Store claim attempts and status securely in PostgreSQL.[67]
    - *Transition:* Upon successful verification, update the user account status to 'active', grant the claiming user full control over the profile, and allow them to initiate the standard profile editing workflow (uploading their own assets, providing instructions), which would typically replace the seeded content.[98] Ensure claimed profiles are treated like regular profiles henceforth. Provide an option for users to hide or delete a seeded profile they own if they don't want it visible [User Query]. Hidden profiles must be excluded from galleries and searches.

Legal and Ethical Implications: This approach carries **significant risks** and requires careful legal review.

- **Terms of Service (ToS) Violation:** Most websites explicitly prohibit automated scraping in their ToS.[92] Systematically scraping data to build a competing database/directory is a clear violation and could lead to legal action or IP blocking.
- **Data Privacy (GDPR, CCPA, etc.):** Even publicly available business information can contain personal data (e.g., contact names, emails of sole proprietors). Scraping, storing, and processing this data, especially to create profiles without explicit consent, is highly likely to violate data protection regulations like GDPR and CCPA.[92] Fines and reputational damage can be substantial.
- **Copyright:** Textual descriptions or images scraped from public sources may be protected by copyright. Reproducing this content in the platform's profiles without permission constitutes infringement.[92]
- **Accuracy and Representation:** Scraped data can be inaccurate or outdated. Displaying potentially incorrect information associated with a business without their consent is ethically problematic and could lead to disputes.
- **Computer Fraud and Abuse Act (CFAA):** While the hiQ Labs vs. LinkedIn case suggested scraping publicly accessible data might not violate the CFAA *per se* [92], the act of creating profiles and potentially implying affiliation could still face challenges under other legal doctrines (e.g., unfair competition, breach of

contract via ToS).

Recommendation: Proceed with extreme caution and prioritize legal consultation. The technical feasibility of scraping and seeding profiles is overshadowed by the substantial legal and ethical risks involved.[92] Before implementing this feature:

1. **Consult legal counsel** specializing in data privacy (GDPR/CCPA), web scraping, and intellectual property law in all relevant jurisdictions (including those of the data sources and the platform's users).
2. If proceeding, strictly limit scraping to **non-personal, purely business directory information** (e.g., company name, address, general industry category) from sources that explicitly permit reuse or where the risk is deemed acceptable after legal review. Avoid scraping descriptive text, images, or personal contact details.
3. Ensure the **claiming process is highly secure** and requires robust verification.[98]
4. Be transparent about the source of seeded data.
5. Consider **alternative growth strategies** that do not involve scraping, such as direct user outreach, partnerships with agricultural organizations, or providing tools that incentivize organic sign-ups.

B. Implementing Support for Virtual Seller Collections

The platform should support "synthetic sellers" or "virtual collections" representing groups of real sellers (e.g., cooperatives, regional marketing bodies) [User Query, Virtual Collections].

Data Modeling (PostgreSQL):

- **Collections Table:** Store information about the collection entity itself.
  - Columns: collection_id (Primary Key, e.g., UUID), name (VARCHAR), description (TEXT), profile_data (JSONB to store structured profile info like address, contact, generated text), manager_user_id (Foreign Key to Users table, representing the admin), created_at, updated_at.
- **CollectionMemberships Table:** Manage the many-to-many relationship between individual sellers (Users) and Collections.
  - Columns: membership_id (Primary Key), collection_id (Foreign Key to Collections), user_id (Foreign Key to Users, the seller member), member_status (e.g., ENUM 'pending', 'active', 'inactive'), joined_at. Use unique constraints on (collection_id, user_id).[10]
- **Roles/Permissions:** Define roles (e.g., collection_manager) and use Casbin or PostgreSQL roles [10] to control who can create collections, add/remove members, and edit collection profiles.

- **Schema Organization:** Place these tables within a dedicated PostgreSQL schema (e.g., marketplace) for better organization.[11]

Profile Generation and Representation:

- **Collection Profile:** Develop an interface for the designated manager_user_id to provide information (text, potentially documents/images) about the collection. Use a similar RAG process (Section III.D) to generate a structured profile for the collection itself, stored in the profile_data JSONB field of the Collections table.
- **Member Profiles:** Individual seller profiles are generated and managed as described previously (Section III).

Display Logic and Functionality:

- **Identification:** Add a type indicator (e.g., entity_type: 'individual' or entity_type: 'collection') to profile data fetched for galleries and search results to allow the UI to distinguish between them.
- **Gallery Display:** Modify gallery generation logic (Section IV) to optionally include Collection entities based on their profile data matching gallery criteria. The UI should render collection profiles differently (e.g., distinct card style).
- **Search:** Collections should be indexed and searchable based on their own name, description, and profile_data. Search results must clearly label collection entities.
- **Drill-Down:** On a collection's profile page, include a section listing its members. This requires a query joining Collections, CollectionMemberships, and Users tables, filtered by collection_id and member_status = 'active'. Each listed member should link to their individual profile page (subject to standard privacy controls).
- **Badging:** When displaying an individual seller's profile (in search results, galleries, or their dedicated page), query the CollectionMemberships table for that user_id where member_status = 'active'. If memberships exist, retrieve the corresponding collection names from the Collections table and display them as badges on the seller's profile.
- **Member Visibility Control:** Add a boolean flag to the Collections table (e.g., hide_members_in_general_results). Modify gallery and search queries to exclude individual members of such collections unless explicitly drilling down or performing a search that specifically targets members.

Messaging:

- Allow users to initiate messages directed to a Collection. The backend messaging service needs to identify the target as a collection (using collection_id).
- Implement routing logic: Messages sent to a collection should be delivered to the designated manager_user_id associated with that collection in the Collections

table.

FastAPI Implementation: Create new API endpoints for CRUD operations on collections and memberships (e.g., /collections, /collections/{collection_id}/members). Ensure proper authorization checks using Casbin based on user roles (e.g., only a collection_manager or platform admin can modify a collection). Update existing endpoints for galleries, search, and profile display to incorporate collection data, drill-down links, and badging information.[9]

IX. Deployment, Operations, and Monitoring

Deploying and operating a complex, multi-component system involving microservices, databases, and AI models requires careful planning around containerization, CI/CD, infrastructure, and monitoring.

A. Best Practices for Dockerizing the Application

Docker provides containerization, ensuring consistency across development, testing, and production environments.

Dockerfile Best Practices:

- **Base Image:** Use official, slim Python images (e.g., python:3.11-slim-bullseye) for smaller image sizes and reduced attack surface. Pin the version tag for reproducibility.[99]
- **Dependency Management:** Copy the requirements.txt file first, then install dependencies using pip install --no-cache-dir --require-hashes -r requirements.txt. Using --no-cache-dir reduces image size, and --require-hashes enhances security. Installing dependencies in an earlier layer leverages Docker's layer caching.[99]
- **Application Code:** Copy application code after dependencies are installed.[99]
- **Non-Root User:** Create and switch to a non-root user within the Dockerfile (RUN groupadd... && useradd..., USER appuser) for improved security. Ensure file permissions are set correctly.
- **Working Directory:** Set a WORKDIR for clarity and consistency.[99]
- **Expose Port:** Use EXPOSE to document the port the application listens on (e.g., EXPOSE 8000).[99]
- **Entrypoint/CMD:** Use the **exec form** of CMD (e.g., CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]) to ensure Uvicorn runs as PID 1 and receives signals correctly (e.g., for graceful shutdown), which is important for lifespan events and orchestration.[99]
- **.dockerignore:** Use a .dockerignore file to exclude unnecessary files and

directories (e.g., .git, venv, __pycache__, test files, local configuration) from the build context, speeding up builds and reducing image size.

Multi-Stage Builds: While less critical for interpreted languages like Python compared to compiled languages [100], multi-stage builds can be beneficial if:

- There are build-time dependencies (e.g., C compilers for certain packages) not needed at runtime.
- Frontend assets are compiled/built within the Dockerfile.
- Using tools like Poetry or PDM, where a virtual environment can be created in a build stage and only the environment copied to the final, slimmer runtime stage.[101]
- *Recommendation:* Start with **single-stage builds** for simplicity.[99] Adopt multi-stage builds later if image size becomes a significant concern or complex build dependencies arise.[101]

Docker Compose: Utilize docker-compose.yml extensively for defining and running the multi-container application stack (FastAPI services, PostgreSQL, Redis, Weaviate/Milvus, Celery workers/beat, monitoring tools like Langfuse) during local development and integration testing.[99] This simplifies environment setup and ensures service interactions can be tested locally.

AI Model Handling: Large AI models (embedding models, LLMs, image captioning models) can bloat container images. Consider strategies like:

- Downloading models during container startup (adds startup time).
- Mounting models from external volumes or shared storage.
- Using dedicated model serving solutions (e.g., Triton Inference Server, TorchServe, or cloud provider AI endpoints) separate from the application containers.

B. Establishing CI/CD Pipelines

Continuous Integration and Continuous Deployment (CI/CD) pipelines automate the process of testing, building, and deploying application updates, improving speed and reliability.

Choice of Platform:

- **GitLab CI/CD:** Tightly integrated if using GitLab for source control. Configuration is via .gitlab-ci.yml.[103] Offers built-in container registry, secret management, and shared or self-hosted runners.[104] Supports Docker-in-Docker for building images within jobs.[104] Numerous templates and examples exist.[103]
- **GitHub Actions:** Integrated with GitHub repositories. Uses YAML workflow files in

.github/workflows/.[105] Large marketplace of reusable actions simplifies common tasks (e.g., checkout, setup Python, build Docker, SSH deploy [105]). Similar core concepts (jobs, steps, runners, secrets).

Typical Pipeline Stages:

1. **Lint & Format:** Run static analysis tools (e.g., flake8, mypy) and code formatters (black, isort) to ensure code quality and consistency.
2. **Test:** Execute automated tests (pytest). This stage often requires setting up dependent services (DB, cache) using Docker Compose within the CI environment or connecting to dedicated test instances.[106] Ensure services are ready before tests run.[106]
3. **Build:** Build Docker images for each modified microservice using the Dockerfiles defined in IX.A.[99] Tag images with unique identifiers (e.g., Git commit SHA, semantic version).
4. **Push:** Push the built and tagged Docker images to a container registry (e.g., GitLab Container Registry [104], Docker Hub, AWS ECR, Google Artifact Registry [101]).
5. **Deploy (Staging/Production):** Trigger deployment of the new container images to the target environment(s). The specific commands depend on the chosen orchestration platform (e.g., kubectl apply -f deployment.yaml, docker stack deploy -c compose.yml --with-registry-auth, gcloud run deploy, AWS Copilot CLI commands, or SSH scripts to update running containers [104]). Implement strategies for zero-downtime deployments (e.g., rolling updates, blue-green deployments).

Recommendation: Select the CI/CD platform integrated with the chosen version control system (**GitLab CI/CD** or **GitHub Actions**). Implement a robust pipeline covering automated testing, Docker image building/pushing, and deployment. Securely manage credentials (API keys, deployment keys) using the CI/CD platform's built-in secrets management features.[104]

C. Evaluating Scalable Deployment Options (Cloud vs. Self-Hosted)

Deploying containerized microservices requires an orchestration platform and decisions about hosting infrastructure.

Container Orchestration:

- **Kubernetes (K8s):** The industry standard, offering immense power, flexibility, and a vast ecosystem for managing complex containerized applications at scale.[107] It handles deployment, scaling, load balancing, service discovery, and self-healing. However, self-managing K8s has a steep learning curve and significant operational overhead.[107] Managed K8s services (AWS EKS, Google GKE, Azure

AKS) abstract away much of the infrastructure management but incur costs. Kubernetes is well-suited for deploying Langchain applications.[108]
- **Docker Swarm:** Docker's native orchestration tool. Simpler to set up and manage than K8s, especially for teams already familiar with Docker.[107] Suitable for less complex deployments or smaller teams.[107] It has fewer features and a smaller community compared to K8s.[107]
- **Serverless Containers (AWS Fargate / Google Cloud Run):** These platforms run containers without requiring management of the underlying virtual machines.[109] They offer automatic scaling and a pay-per-use model, which can be cost-effective, especially for applications with variable workloads.[109] They significantly simplify operations. AWS Fargate integrates with ECS, while Google Cloud Run can run containers directly. Spot instances (Fargate Spot) can offer further cost savings for fault-tolerant workloads.[109]

Cloud vs. Self-Hosted:

- **Cloud (AWS, GCP, Azure):** Offers managed services (databases, caches, orchestrators, load balancers), pay-as-you-go pricing, scalability, and global infrastructure.[109] Reduces operational burden significantly. Ideal for getting started quickly and scaling dynamically. Potential downsides include costs escalating with usage and potential vendor lock-in.[109]
- **Self-Hosted (On-Premises / VPS / Bare Metal):** Provides maximum control over hardware and software. Can be more cost-effective for compute-intensive workloads (like AI model serving) with high, stable utilization, especially using budget providers.[109] However, it incurs substantial operational overhead for managing hardware, networking, security, databases, orchestration, backups, and requires significant in-house expertise. Scalability is typically manual and requires upfront investment.

Recommendation for Scalability and Cost-Effectiveness: For a new, complex application like this, starting with a **managed cloud environment** is generally recommended to reduce operational burden and leverage scalable infrastructure.

1. **Preferred Option (Simpler Ops): AWS ECS with Fargate or Google Cloud Run**.[109] These serverless container platforms abstract away server management, allowing focus on the application. Use managed databases (e.g., AWS RDS/Aurora for PostgreSQL, Google Cloud SQL) and caches (AWS ElastiCache, Google Memorystore). Deploy the vector database (Weaviate/Milvus) as containers on the platform or use a managed cloud offering if available. This approach balances scalability, cost (especially with Spot/savings plans), and operational simplicity.
2. **Alternative (More Control): Managed Kubernetes (EKS, GKE, AKS)**.[107] Offers

more control and flexibility than serverless containers but requires more K8s expertise. Deploy all components (application services, databases, cache, vector DB) as containerized workloads within the K8s cluster, potentially using Helm charts or operators for managing stateful services like PostgreSQL and Weaviate/Milvus.

While self-hosting specific compute-heavy components like AI inference might become cost-effective later at high scale [109], the initial operational complexity makes a managed cloud approach more pragmatic for launching the platform. Focus on cloud cost optimization techniques like right-sizing instances, using ARM-based processors (AWS Graviton [109]), leveraging spot instances, and implementing auto-scaling.

Table IX.C: Deployment Orchestration/Hosting Comparison

Option	Scalability	Flexibility	Ease of Management	Cost (Infrastructure)	Cost (Operational)	Ecosystem	Learning Curve	Suitability for Project
Kubernetes (Managed - EKS/GKE)	Very High	Very High	Moderate	Moderate-High	Moderate	Very Large	High	**Strong Candidate**
Kubernetes (Self-Hosted)	High	Very High	Low	Potentially Lower	Very High	Very Large	Very High	Less Suitable (Ops)
Docker Swarm (Self-Hosted)	Moderate	Moderate	Moderate-High	Potentially Lower	High	Smaller	Moderate	Less Suitable (Scale)
Serverless	Very	High	Very	Variable	Low	Large	Low-Moderat	**Strong Candi**

Containers (Fargate/Run)	High		High	(Usage-based)			e	**date**
Self-Hosted (No Orchestration)	Low	Low	Low	Variable	Very High	N/A	Low	Unsuitable

---

Recommendation: Start with Serverless Containers or Managed Kubernetes in the Cloud.

D. Monitoring Strategy for LLM-Based Components

Monitoring is critical, especially for applications relying heavily on LLMs, which exhibit unique behaviors and failure modes. A comprehensive strategy includes both standard APM and LLM-specific observability.

LLM-Specific Monitoring Needs: Beyond typical metrics (CPU, memory, latency, error rates), monitoring LLM interactions requires tracking [110]:

- **Cost:** Token consumption per API call, aggregated by user, feature, or time period.
- **Latency:** Time-to-first-token and total generation time for LLM responses.
- **Quality & Accuracy:** Assessing relevance, coherence, factuality, and adherence to instructions. Detecting hallucinations is crucial. This often requires a combination of automated metrics (e.g., embedding similarity to ground truth, LLM-as-judge evaluations) and human feedback mechanisms.
- **Drift:** Monitoring changes in model performance or output style over time as underlying models evolve or input data patterns shift.[112]
- **Safety & Bias:** Tracking instances of harmful, biased, or inappropriate outputs, and monitoring refusals from safety filters.[54]
- **RAG Performance:** Evaluating the relevance of retrieved documents (precision/recall [17]) and their impact on final answer quality.

Observability Tools:

- **LangSmith:** Langchain's commercial platform, offering tight integration for

tracing, debugging, evaluation, and monitoring of Langchain applications.[110] Has a free tier for getting started.[110]

- **Open Source Alternatives:** Several powerful open-source tools have emerged:
  - **Langfuse:** MIT licensed, provides comprehensive tracing, debugging, evaluation datasets, prompt management, and collaboration features. Supports major frameworks (Langchain, OpenAI, LlamaIndex) and can be self-hosted.[111] Docker setup is feasible.[102] A strong contender.
  - **Phoenix (by Arize):** ELv2 licensed, focuses on robust tracing and evaluation, including built-in hallucination detection. Compatible with Langchain, LlamaIndex, OpenAI, and OpenTelemetry.[110]
  - **Traceloop OpenLLMetry:** Apache 2.0 licensed, instruments LLM interactions and exports traces in the vendor-neutral OpenTelemetry (OTel) format, allowing integration with various backends like Jaeger, Grafana, or Datadog.[110]
  - Others like **Helicone** (MIT, proxy-based logging [110]), **Lunary** (Apache 2.0, tracking tool [110]), and **TruLens** (Apache 2.0, focus on feedback functions for evals [110]) offer different focuses and features.
- **General APM Tools:** Platforms like Datadog [110], Coralogix [111], Dynatrace, New Relic are adding LLM-specific features. If an APM tool is already in use, leveraging its LLM capabilities can provide unified monitoring. OpenTelemetry provides a standard way to instrument applications for tracing and metrics, compatible with many backends.[110]

Recommendation: Implement a **two-pronged monitoring strategy**:

1. **LLM-Specific Observability:** Deploy a dedicated, open-source LLM observability platform like **Langfuse** [111] or **Phoenix**.[110] Langfuse's focus on the end-to-end LLM engineering lifecycle seems particularly well-suited. Integrate application services (especially those making LLM calls or using Langchain) with the chosen platform's SDKs. Utilize its features for tracing requests, logging prompts/responses, tracking costs, managing evaluation datasets, and facilitating human feedback.
2. **Standard APM & Tracing:** Instrument all microservices using **OpenTelemetry**.[110] This provides vendor-neutral distributed tracing and metrics collection. Send OTel data to a suitable backend (e.g., self-hosted Jaeger/Prometheus/Grafana stack, or a commercial APM tool that supports OTel). Tools like OpenLLMetry can help bridge LLM-specific traces into the OTel ecosystem.[110]
3. **Logging:** Implement structured logging within all FastAPI services to capture critical events, errors, and application state information.

This layered approach provides visibility into both standard application health and the

unique aspects of LLM behavior, which is essential for debugging, optimizing performance and cost, and iteratively improving the AI components of the platform.[110]

Table IX.D: LLM Observability Tools Comparison (Focus on Open Source)

Tool	Key Features	Self-Hosted	License	Langchain Integration	Primary Focus	Relevant Sources
LangSmith	Tracing, Evals, Debugging, Prompt Hub, Monitoring	No (Cloud)	Commercial	Native	End-to-end Langchain Dev & Ops	[110]
Langfuse	Tracing, Evals, Prompt Mgmt, Debugging, Cost Tracking, Collaboration, SDKs	Yes	MIT	Yes	Collaborative LLM App Dev & Ops	[102]
Phoenix (Arize)	Tracing, Evals (incl. Hallucination), OTel Compatible	Yes	ELv2	Yes	LLM Tracing & Evaluation	[110]
OpenLLMetry	Instruments LLM calls, Exports OTel traces	Yes (N/A)*	Apache 2.0	Yes	Bridging LLM Traces to OpenTelemetry	[110]

Helicone	Proxy-based Logging, Monitoring, Debugging, Cost Tracking	Yes	MIT	Yes	Logging & Monitoring via Proxy	[110]
Lunary	Tracking, Radar (Categorization), Chatbot focus	Yes	Apache 2.0	Yes	Chatbot Management & Enhancement	[110]
TruLens	Evals via Feedback Functions, Python focus	Yes	Apache 2.0	Yes	Qualitative Analysis & Evaluation	[110]

**OpenLLMetry is primarily an instrumentation library; the backend (Jaeger, Grafana) is self-hosted.*

Recommendation: Start with Langfuse or Phoenix for dedicated LLM observability, complemented by OpenTelemetry for general APM.

X. Conclusion and Recommendations

This report has detailed a comprehensive technical architecture and implementation strategy for the proposed AI-powered agricultural commodity marketplace. The analysis indicates that the project is feasible using modern, predominantly open-source technologies centered around Python.

The recommended architecture is based on **Microservices** leveraging the **FastAPI** framework, offering the necessary scalability, flexibility, and performance required for the diverse components, especially the AI-driven features. Data persistence should utilize **PostgreSQL** for structured data and user management, complemented by a robust vector database like **Weaviate** or **Milvus** to power the essential Retrieval-Augmented Generation (RAG) capabilities. **Langchain** serves as the central nervous system for AI workflows, orchestrating data loading, processing, embedding,

RAG retrieval, and interaction with LLMs, with **Function/Tool Calling** recommended for generating reliable structured outputs for profiles and search results. The **Expo** platform provides a cost-effective solution for the mobile asset upload application. **Redis** should be employed for caching generated profile content to enhance gallery performance, using event-based invalidation. A simple **WebSocket** system built with FastAPI can handle real-time communication, with careful consideration (WSS, Port 443, CDNs) for potential **Great Firewall** challenges.

Key recommendations include:

- **Prioritize Asynchronous Processing:** Use task queues (e.g., Celery) for time-consuming asset processing (OCR, video analysis) initiated by uploads.
- **Embrace Semantic Chunking:** Where possible, use semantic chunking for RAG to improve retrieval relevance.
- **Invest in Robust RAG:** Carefully design RAG pipelines for profile generation and search, potentially using multi-step approaches for complex comparison tasks. Use consistent embedding models.
- **Enforce Structured Outputs:** Utilize LLM function/tool calling via Langchain (with_structured_output) for reliable data generation.
- **Implement Layered Security:** Combine vector store metadata filtering with application-level RBAC (using Casbin) for granular privacy control.
- **Use Distributed Caching:** Employ Redis with event-based invalidation for profile caching.
- **Address GFW Proactively:** Design the chat system using WSS/443 and consider CDNs; monitor from China.
- **Manage Reference Library Carefully:** Respect robots.txt/ToS for web scraping and use scheduling (Celery Beat/APScheduler) for updates.
- **Exercise Extreme Caution with Seeding:** Profile seeding via scraping carries significant legal and ethical risks (ToS, GDPR/CCPA, Copyright); **seek legal counsel before proceeding**.
- **Model Virtual Collections:** Use appropriate PostgreSQL schema design (separate tables, memberships) and modify display/search logic.
- **Standardize Deployment:** Use Docker best practices and automate builds/deployments via CI/CD (GitLab CI / GitHub Actions).
- **Choose Cloud Orchestration:** Start with managed cloud container platforms (Fargate, Cloud Run, or Managed K8s) for operational efficiency.
- **Monitor Comprehensively:** Implement both standard APM (via OpenTelemetry) and LLM-specific observability (using tools like Langfuse or Phoenix).

A phased implementation is advisable. Begin with the core user registration, asset

upload, basic RAG profile generation, and gallery display. Subsequently, refine the search/matchmaking RAG, implement the chat system, build out the reference library, and finally tackle extensions like virtual collections and (if legally cleared) profile seeding. Continuous monitoring and evaluation, particularly of the AI components, will be crucial for iterative improvement and ensuring the platform delivers on its promise of intelligent matchmaking in the agricultural commodities space.

Works cited

1. Monolithic vs Microservices - Difference Between Software Development Architectures, accessed April 30, 2025, https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/
2. Monolith Versus Microservices: Weigh the Pros and Cons of Both Configs | Akamai, accessed April 30, 2025, https://www.akamai.com/blog/cloud/monolith-versus-microservices-weigh-the-difference
3. Microservices vs SOA Architecture: What's the Difference? - ClickIT, accessed April 30, 2025, https://www.clickittech.com/devops/microservices-vs-soa-architecture/amp/
4. Monolithic vs Microservices [Pros, Cons & Best Use Cases] - Clockwise Software, accessed April 30, 2025, https://clockwise.software/blog/monolithic-architecture-vs-microservices-comparison/
5. Top 10 Python Web App Frameworks for 2025: Choose the Best - Creole Studios, accessed April 30, 2025, https://www.creolestudios.com/top-python-web-frameworks/
6. Which Python Web Framework Should You Choose Flask Django or FastAPI?, accessed April 30, 2025, https://embarkingonvoyage.com/blog/technologies/which-python-web-framework-should-you-choose-flask-django-or-fastapi/
7. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? | The PyCharm Blog, accessed April 30, 2025, https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/
8. 2025's Top 10 Python Web Frameworks Compared - Leapcell, accessed April 30, 2025, https://leapcell.io/blog/top-10-python-web-frameworks-compared
9. SQL (Relational) Databases - FastAPI, accessed April 30, 2025, https://fastapi.tiangolo.com/tutorial/sql-databases/
10. PostgreSQL Role Membership - Neon, accessed April 30, 2025, https://neon.tech/postgresql/postgresql-administration/postgresql-role-membership
11. How to Create a Database Schema in PostgreSQL: An Ultimate Guide - Airbyte, accessed April 30, 2025, https://airbyte.com/data-engineering-resources/create-database-schema-in-pos

tgresql
12. PostGraphile | PostgreSQL Schema Design, accessed April 30, 2025, https://www.graphile.org/postgraphile/postgresql-schema-design/
13. Vector Databases Compared: Pinecone, Milvus, Chroma, Weaviate, FAISS, and more, accessed April 30, 2025, https://zackproser.com/blog/vector-databases-compared
14. Vector stores - LangChain, accessed April 30, 2025, https://python.langchain.com/docs/concepts/vectorstores/
15. Comparing Vector Databases: Milvus vs. Chroma DB - Zilliz blog, accessed April 30, 2025, https://zilliz.com/blog/milvus-vs-chroma
16. Building and deploying Web App using LangChain - Packt, accessed April 30, 2025, https://www.packtpub.com/en-us/learning/how-to-tutorials/building-and-deploying-web-app-using-langchain
17. Building a Production-Ready RAG System with LangChain and ChromaDB: A Practical Guide for Intermediate Developers, accessed April 30, 2025, https://www.tenxdeveloper.com/blog/building-a-production-ready-rag-system-with-langchain-and-chromadb
18. Create LLM apps using RAG - LanceDB Blog, accessed April 30, 2025, https://blog.lancedb.com/create-llm-apps-using-rag/
19. Vector Database Comparison: Pinecone vs Weaviate vs Qdrant vs FAISS vs Milvus vs Chroma (2025) | LiquidMetal AI, accessed April 30, 2025, https://liquidmetal.ai/casesAndBlogs/vector-comparison/
20. Top 5 Open Source Vector Databases in 2024 - GPU Mart, accessed April 30, 2025, https://www.gpu-mart.com/blog/top-5-open-source-vector-databases-2024/
21. An Honest Comparison of Open Source Vector Databases - KDnuggets, accessed April 30, 2025, https://www.kdnuggets.com/an-honest-comparison-of-open-source-vector-databases
22. Top 5 Vector Databases to Use for RAG (Retrieval-Augmented Generation) in 2025, accessed April 30, 2025, https://apxml.com/posts/top-vector-databases-for-rag
23. Weaviate Hybrid Search - LangChain, accessed April 30, 2025, https://python.langchain.com/v0.2/docs/integrations/retrievers/weaviate-hybrid/
24. Weaviate - LangChain, accessed April 30, 2025, https://python.langchain.com/docs/integrations/vectorstores/weaviate/
25. Expo, accessed April 30, 2025, https://expo.dev/
26. Expo Application Services (EAS) Pricing, accessed April 30, 2025, https://expo.dev/pricing
27. Can Firebase Storage still be used under the free Spark plan for development?, accessed April 30, 2025, https://stackoverflow.com/questions/79598937/is-firebase-storage-no-longer-free-any-alternatives-for-small-projects
28. How to extract text from a PDF file via python? - Stack Overflow, accessed April

30, 2025, https://stackoverflow.com/questions/34837707/how-to-extract-text-from-a-pdf-file-via-python
29. Extract text from PDF File using Python - GeeksforGeeks, accessed April 30, 2025, https://www.geeksforgeeks.org/extract-text-from-pdf-file-using-python/
30. How to load PDFs - LangChain, accessed April 30, 2025, https://python.langchain.com/docs/how_to/document_loader_pdf/
31. How to load PDFs | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/v0.2/docs/how_to/document_loader_pdf/
32. Introducing Kreuzberg: A Simple, Modern Library for PDF and Document Text Extraction in Python - Reddit, accessed April 30, 2025, https://www.reddit.com/r/Python/comments/1if3axy/introducing_kreuzberg_a_simple_modern_library_for/
33. Retrieval-Augmented Generation (RAG) with Azure AI Document Intelligence, accessed April 30, 2025, https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/concept/retrieval-augmented-generation?view=doc-intel-4.0.0
34. Image Processing in Python with Pillow - Auth0, accessed April 30, 2025, https://auth0.com/blog/image-processing-in-python-with-pillow/
35. Image processing with OpenCV-Pillow for beginners - Kaggle, accessed April 30, 2025, https://www.kaggle.com/code/inancolak/image-processing-with-opencv-pillow-for-beginners
36. Extracting Image Metadata in Python Pillow - Tutorialspoint, accessed April 30, 2025, https://www.tutorialspoint.com/python_pillow/python_pillow_extracting_image_metadata.htm
37. How to Extract Image Metadata in Python, accessed April 30, 2025, https://thepythoncode.com/article/extracting-image-metadata-in-python
38. How to Extract Frames from Video in Python - Cloudinary, accessed April 30, 2025, https://cloudinary.com/guides/video-effects/how-to-extract-frames-from-video-in-python
39. Creating Thumbnails for a Video in Python with FFMPEG - DEV Community, accessed April 30, 2025, https://dev.to/shannonlal/creating-thumbnails-for-a-video-in-python-with-ffmpeg-4e
40. Steps to create thumbnails from video in python using moviepy - Hardweb, accessed April 30, 2025, https://hardweb.dev/blogs/posts/2020/steps-to-create-thumbnails-from-video-in-python-using-moviepy.html
41. How to Split Videos into Frame in Python - Cloudinary, accessed April 30, 2025, https://cloudinary.com/guides/video-effects/how-to-split-videos-into-frame-in-python
42. Image Captioning Using Multimodal LLMs - Lund University Publications,

accessed April 30, 2025, https://lup.lub.lu.se/student-papers/record/9185108/file/9185109.pdf
43. Comparing local large language models for alt-text generation - Dries Buytaert, accessed April 30, 2025, https://dri.es/comparing-local-llms-for-alt-text-generation
44. Automated Image Captioning with LLMs - Recognize Anything, BLIP-2, and Kosmos-2, accessed April 30, 2025, https://www.youtube.com/watch?v=lBdfAB1SoKc
45. Python Image Captioning Tutorial | Image To Text Blip Python Guide - YouTube, accessed April 30, 2025, https://www.youtube.com/watch?v=CzO8VuaHfKM
46. LangChain Playbook for NeMo Retriever Text Embedding NIM - NVIDIA Docs Hub, accessed April 30, 2025, https://docs.nvidia.com/nim/nemo-retriever/text-embedding/latest/playbook.html
47. Build a Retrieval Augmented Generation (RAG) App: Part 1 | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/docs/tutorials/rag/
48. Embedding models | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/docs/concepts/embedding_models/
49. Build an LLM RAG Chatbot With LangChain - Real Python, accessed April 30, 2025, https://realpython.com/build-llm-rag-chatbot-with-langchain/
50. Multi-Doc RAG: Leverage LangChain to Query and Compare 10K Reports - datascience.fm, accessed April 30, 2025, https://datascience.fm/multi-doc-rag-on-10k-reports/
51. How to retrieve using multiple vectors per document | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/docs/how_to/multi_vector/
52. LangChain: A Complete Guide & Tutorial - Nanonets, accessed April 30, 2025, https://nanonets.com/blog/langchain/
53. LangChain: DuckDuckGo Quick Guide (Web Search) - Kaggle, accessed April 30, 2025, https://www.kaggle.com/code/ksmooi/langchain-duckduckgo-quick-guide-web-search
54. Structured Outputs - OpenAI API, accessed April 30, 2025, https://platform.openai.com/docs/guides/structured-outputs
55. A Beginner's Guide to Returning Structured Outputs in LangChain - DEV Community, accessed April 30, 2025, https://dev.to/aiengineering/a-beginners-guide-to-returning-structured-outputs-in-langchain-36ac
56. Structured outputs - LangChain, accessed April 30, 2025, https://python.langchain.com/docs/concepts/structured_outputs/
57. Structured outputs - LangChain.js, accessed April 30, 2025, https://js.langchain.com/docs/concepts/structured_outputs/
58. Tool calling | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/docs/concepts/tool_calling/
59. How to do tool/function calling | 🦜 LangChain, accessed April 30, 2025, https://python.langchain.com/docs/how_to/function_calling/
60. Introduction to function calling | Generative AI on Vertex AI - Google Cloud,

accessed April 30, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/function-calling
61. AI Agent Workflows: A Complete Guide on Whether to Build With LangGraph or LangChain, accessed April 30, 2025, https://towardsdatascience.com/ai-agent-workflows-a-complete-guide-on-whether-to-build-with-langgraph-or-langchain-117025509fa0/
62. Casbin · An authorization library that supports access control models like ACL, RBAC, ABAC for Golang, Java, C/C++, Node.js, Javascript, PHP, Laravel, Python, .NET (C#), Delphi, Rust, Ruby, Swift (Objective-C), Lua (OpenResty), Dart, accessed April 30, 2025, https://casbin.org/
63. fastapi_user_auth - PyPI, accessed April 30, 2025, https://pypi.org/project/fastapi_user_auth/
64. PyCasbin - GitHub, accessed April 30, 2025, https://github.com/pycasbin
65. pycasbin/fastapi-authz - GitHub, accessed April 30, 2025, https://github.com/pycasbin/fastapi-authz
66. Database Design - Users and their privacy - Stack Overflow, accessed April 30, 2025, https://stackoverflow.com/questions/5211799/database-design-users-and-their-privacy
67. Best Practices for Designing a User Authentication Module | Vertabelo Database Modeler, accessed April 30, 2025, https://vertabelo.com/blog/user-authentication-module/
68. Using Redis with FastAPI, accessed April 30, 2025, https://redis.io/learn/develop/python/fastapi
69. Caching in FastAPI: Unlocking High-Performance Development: - DEV Community, accessed April 30, 2025, https://dev.to/sivakumarmanoharan/caching-in-fastapi-unlocking-high-performance-development-20ej
70. Fastapi Caching Techniques Explained | Restackio, accessed April 30, 2025, https://www.restack.io/p/fastapi-answer-caching-techniques
71. Caching strategies - Building High-Performance Web APIs with FastAPI - StudyRaid, accessed April 30, 2025, https://app.studyraid.com/en/read/8388/231229/caching-strategies
72. Accelerate Machine Learning Model Serving with FastAPI and Redis Caching - KDnuggets, accessed April 30, 2025, https://www.kdnuggets.com/accelerate-machine-learning-model-serving-with-fastapi-and-redis-caching
73. First Steps - FastAPI, accessed April 30, 2025, https://fastapi.tiangolo.com/tutorial/first-steps/
74. Document comparison RAG, the struggle is real. : r/LocalLLaMA - Reddit, accessed April 30, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1cn659i/document_comparison_rag_the_struggle_is_real/
75. Document Comparison | 🦜 LangChain, accessed April 30, 2025,

https://python.langchain.com/v0.1/docs/integrations/toolkits/document_comparison_toolkit/
76. Optimal way to do implement multiple RAG contexts · langchain-ai langchainjs · Discussion #6707 - GitHub, accessed April 30, 2025, https://github.com/langchain-ai/langchainjs/discussions/6707
77. Fastapi Mount Socketio Integration | Restackio, accessed April 30, 2025, https://www.restack.io/p/fastapi-answer-mount-socketio
78. WebSockets - FastAPI, accessed April 30, 2025, https://fastapi.tiangolo.com/advanced/websockets/
79. WebSockets - FastAPI, accessed April 30, 2025, https://fastapi.tiangolo.com/reference/websockets/
80. Simple chat Application using Websockets with FastAPI - GeeksforGeeks, accessed April 30, 2025, https://www.geeksforgeeks.org/simple-chat-application-using-websockets-with-fastapi/
81. End User Computing Nerd vs the Great Firewall of China - WWT, accessed April 30, 2025, https://www.wwt.com/blog/end-user-computing-nerd-vs-the-great-firewall-of-china
82. Deconstructing the Great Firewall of China - Cisco ThousandEyes, accessed April 30, 2025, https://www.thousandeyes.com/blog/deconstructing-great-firewall-china
83. Advancing Obfuscation Strategies to Counter China's Great Firewall: A Technical and Policy Perspective - arXiv, accessed April 30, 2025, https://arxiv.org/html/2503.02018v1
84. How the Great Firewall of China Affects Performance of Websites Outside of China, accessed April 30, 2025, https://www.dotcom-monitor.com/blog/how-the-great-firewall-of-china-affects-performance-of-websites-outside-of-china/
85. Wireguard is banned in China, anything I could do about it? - Reddit, accessed April 30, 2025, https://www.reddit.com/r/WireGuard/comments/10zt05u/wireguard_is_banned_in_china_anything_i_could_do/
86. The Great Firewall of China: What It Is and How to Get Around It | EXPERTE.com, accessed April 30, 2025, https://www.experte.com/internet-censorship/great-firewall-china
87. Six solutions to keep your website from going down in China | GCC, accessed April 30, 2025, https://www.goclickchina.com/blog/six-solutions-to-keep-your-website-from-going-down-in-china/
88. Do you still need a WebSocket fallback in 2024? - Ably Realtime, accessed April 30, 2025, https://ably.com/blog/websocket-compatibility
89. Python Scrapy vs Requests with Beautiful Soup Compared - ScrapeOps, accessed April 30, 2025, https://scrapeops.io/python-web-scraping-playbook/python-scrapy-vs-requests

-beautiful-soup/
90. Scrapy vs. Requests: Which One Is Better For Web Scraping? - Bright Data, accessed April 30, 2025, https://brightdata.com/blog/web-data/scrapy-vs-requests
91. Best Python Web Scraping Libraries - GitHub, accessed April 30, 2025, https://github.com/luminati-io/Python-scraping-libraries
92. Is Web Scraping Legal? Yes, If You Do It Right - HasData, accessed April 30, 2025, https://hasdata.com/blog/is-web-scraping-legal
93. All you need to know about Ethical Web Scraping - PureVPN Blog, accessed April 30, 2025, https://www.purevpn.com/blog/what-is-ethical-web-scraping/
94. Is Web Scraping Legal? - Oxylabs, accessed April 30, 2025, https://oxylabs.io/blog/is-web-scraping-legal
95. multiple prompt in a single RAG chain · Issue #16063 - GitHub, accessed April 30, 2025, https://github.com/langchain-ai/langchain/issues/16063
96. Python, NoSQL & FastAPI Tutorial: Web Scraping on a Schedule - DEV Community, accessed April 30, 2025, https://dev.to/datastax/python-nosql-fastapi-tutorial-web-scraping-on-a-schedule-4d3l
97. Boost Your FastAPI App's Performance with Celery - Nikhil Akki's blog, accessed April 30, 2025, https://nikhilakki.in/boost-your-fastapi-apps-performance-with-celery
98. Secure Data Connect with authorization and attestation - Firebase - Google, accessed April 30, 2025, https://firebase.google.com/docs/data-connect/authorization-and-security
99. FastAPI in Containers - Docker, accessed April 30, 2025, https://fastapi.tiangolo.com/deployment/docker/
100. Is multistage docker possible into FastApi application - Stack Overflow, accessed April 30, 2025, https://stackoverflow.com/questions/78230406/is-multistage-docker-possible-into-fastapi-application
101. Build and Deploy a LangChain-Powered Chat App with Docker and Streamlit, accessed April 30, 2025, https://www.docker.com/blog/build-and-deploy-a-langchain-powered-chat-app-with-docker-and-streamlit/
102. amine-akrout/parenting_assistant_rag: AI-Powered Parenting Assistant - GitHub, accessed April 30, 2025, https://github.com/amine-akrout/parenting_assistant_rag
103. GitLab CI/CD examples, accessed April 30, 2025, https://docs.gitlab.com/ci/examples/
104. How To Set Up a Continuous Deployment Pipeline with GitLab CI/CD on Ubuntu, accessed April 30, 2025, https://www.digitalocean.com/community/tutorials/how-to-set-up-a-continuous-deployment-pipeline-with-gitlab-on-ubuntu
105. Deploying a FastAPI Application with CI/CD Pipeline: HNG Task 2 - DEV Community, accessed April 30, 2025,

https://dev.to/dipe_/deploying-a-fastapi-application-with-cicd-pipeline-hng-task-3-5598
106. GitLab CI Docker Compose: ConnectionError When Accessing FastAPI Service, accessed April 30, 2025, https://stackoverflow.com/questions/79534820/gitlab-ci-docker-compose-connectionerror-when-accessing-fastapi-service
107. Kubernetes vs Docker Swarm: Practical Comparison For 2024 - Bacancy Technology, accessed April 30, 2025, https://www.bacancytechnology.com/blog/kubernetes-vs-docker-swarm
108. How do I deploy a LangChain application on Kubernetes? - Milvus, accessed April 30, 2025, https://milvus.io/ai-quick-reference/how-do-i-deploy-a-langchain-application-on-kubernetes
109. What would be the most cost effective cloud deployment scheme for me? : r/aws - Reddit, accessed April 30, 2025, https://www.reddit.com/r/aws/comments/1jtdgio/what_would_be_the_most_cost_effective_cloud/
110. LLM Observability Tools: 2025 Comparison - lakeFS, accessed April 30, 2025, https://lakefs.io/blog/llm-observability-tools/
111. 10 LLM Observability Tools to Know in 2025 - Coralogix, accessed April 30, 2025, https://coralogix.com/guides/aiops/llm-observability-tools/
112. Top 6 LangSmith Alternatives in 2025: A Complete Guide | Generative AI Collaboration Platform, accessed April 30, 2025, https://orq.ai/blog/langsmith-alternatives