ChatGPT

System Overview and Requirements

- **Multi-Agent Architecture:** We will build a modular pipeline with specialized agents coordinated by a central *Orchestrator Agent*. Each agent has a clear responsibility (e.g. downloading files, crawling web pages, analyzing files, answering queries) and communicates via well-defined data schemas. This approach improves scalability, clarity, and maintainability by separating concerns [1] [2]. For example, as in multi-agent AI design, one agent ("Orchestrator") manages workflow and delegates tasks to other specialized agents [1] [3].
- **LLM-Driven Agents:** An LLM ("AI brain") will underlie each agent, interpreting instructions and generating responses. For instance, the Orchestrator agent uses the LLM to parse the user's query and decide which agents to invoke next. Each specialized agent (file downloader, web crawler, file analyzer, etc.) also uses the LLM as needed for tasks like data extraction or summarization. This leverages the LLM's reasoning and language abilities in each step.
- **Technology Stack:** We use **FastAPI** for the HTTP API framework (providing high-performance async I/O, automatic request/response validation via Pydantic, and built-in background-task support) [4]. **Pydantic** models define all request inputs, inter-agent messages, and outputs, ensuring structured communication and catching errors early [5] [4]. For web crawling, we use **Crawl4AI** (an async, LLM-friendly crawler) to fetch and clean content from URLs [6]. Any long-running or heavy tasks (like large downloads or analysis) are offloaded to background processing (e.g. FastAPI BackgroundTasks or Celery) to keep the system responsive [7] [8].
- **User Requirements:** The system must accept natural-language questions (possibly including file URLs or website URLs) and produce answers in the exact format requested by the user. Answers must be comprehensive and include source citations for any factual content. The pipeline must handle *any file format* (text, PDF, image, etc.) up to ~200 MB, processing data asynchronously (without blocking) and using streaming/chunking to avoid high memory usage [8]. All communication is asynchronous so the API can handle concurrent requests efficiently [4] [8].

Data Modeling with Pydantic

- **Structured Schemas:** We define **Pydantic models** for every piece of data passed between agents. For example, inputs like download URLs, web requests, and user queries are Pydantic classes. Intermediate data (e.g. file metadata, scraped content, analysis results) also have explicit schemas. This ensures *data integrity* and consistency – Pydantic automatically checks types and required fields, preventing mismatches that can be hard to debug in a multi-agent system [5] [9]. For instance, a `FileMetadata` model might include fields like `filename: str`, `size_bytes: int`, `format: str`, etc., with validation rules.
- **Inter-Agent Contracts:** Each agent's input and output are defined by these models. This means agents communicate with *well-defined messages* rather than ad-hoc data structures. As noted by Analytics Vidhya, clear data models between agents "ensure each agent's input and output are predictable and validated, significantly reducing runtime errors and enhancing system robustness" [2]. If an agent fails to produce valid output, Pydantic will flag it immediately, allowing the orchestrator to handle errors or retry.

- **FastAPI Integration:** FastAPI uses these Pydantic models to validate incoming API requests and shape responses. For example, the orchestrator's HTTP endpoint will declare a Pydantic model for the request body, ensuring required fields are present (URL lists, user question, etc.) [4]. FastAPI then automatically parses JSON into Python objects and returns data in the correct schema format. This minimizes boilerplate code and guards against invalid inputs. We avoid manual JSON parsing and double checks by relying on FastAPI's built-in validation (powered by Pydantic) [4].

Orchestrator Agent

- **Role & Logic:** The Orchestrator Agent is the entry point and coordinator. When the user submits a query or request, FastAPI routes it to the Orchestrator handler. The Orchestrator **parses the input schema** (using Pydantic) and examines it for keys like file URLs, uploaded files, or website URLs. Based on this, it decides which sub-agents need to run. For example:
- If the input contains file download URLs, it triggers the *File Download Agent*.
- If it contains website URLs or HTML-related questions, it triggers the *HTML/Crawler Agent*.
- If uploaded files are present, it includes them for analysis.
- **Task Delegation:** The Orchestrator creates tasks or messages for other agents. This might involve calling their async functions directly or pushing jobs to a task queue. In either case, the orchestrator passes the relevant Pydantic-modeled data (e.g. a list of URLs to fetch, or HTML URLs to crawl) to the respective agent. It uses the LLM to refine queries or interpret results as needed – for instance, it may refine a natural-language question into specific file/URL lists.
- **Flow Control:** It ensures the agents run in the correct sequence. For example, it waits for the File Download Agent to finish and collect all downloaded files before invoking the File Analysis Agent on those files. Similarly, it runs the HTML agent concurrently on multiple URLs if needed. It uses FastAPI's async capabilities so multiple fetches/crawls can happen in parallel [4] [6].
- **Orchestration and Error Handling:** After delegating tasks, the Orchestrator collects the results (reports, extracted data, etc.) from each agent. It may perform validation on these results. If something is missing or invalid (e.g. a file failed to download, or parsed output is malformed), the orchestrator can retry the agent or return an error. Finally, it invokes the Answer Agent with all gathered information. The orchestrator continues to enforce the output format: if the final answer's structure does not match what the user requested, it can re-run the Answer Agent with adjusted prompts or data until the correct format is achieved. This kind of feedback loop ensures completeness and adherence to requirements.

File Download Agent

- **Purpose:** Triggered when the input includes one or more file download URLs. Its job is to fetch these files to a local or temporary storage for further analysis.
- **Asynchronous Fetching:** This agent uses FastAPI's async I/O or background tasks to download files **concurrently** (for example, using `httpx` or `aiohttp`). Handling multiple downloads at once speeds up the process. Each download is streamed to disk (not loaded fully in memory) to accommodate large files up to 200MB. This prevents memory exhaustion and blocked event loops as noted in FastAPI best practices [8].
- **Validation:** After downloading, the agent verifies file integrity (e.g. checking file size or checksum if provided) and confirms the download is complete. It then returns a list of file paths (or metadata) in a Pydantic schema. If any download fails, the agent retries a configurable number of times before reporting an error back to the orchestrator.

- **Storage:** Downloaded files are stored in a secure temporary directory or object storage with randomized file names. The agent returns references (paths or URIs) to these files. Uploaded files (if any) are also included in this list. The orchestrator then treats all these files uniformly in subsequent steps.

HTML/Crawler Agent

- **Purpose:** Activated when the input contains website URLs or when the user's question relates to web content. This agent fetches and extracts content from web pages to answer queries about them.
- **Using Crawl4AI:** We use **Crawl4AI's AsyncWebCrawler** to crawl each URL. Crawl4AI is designed for LLM-friendly crawling: it produces clean Markdown or JSON with extracted text and metadata, and it runs asynchronously [6] [10]. The agent constructs a Crawl4AI configuration (e.g. whether to run Javascript, whether to chunk text) based on the user query. It then calls `await crawler.arun(url=...)` for each URL. Crawl4AI handles images, scripts, etc., by default outputting cleaned HTML and Markdown, which the agent can feed to the LLM.
- **Parallel Crawling:** Multiple URLs are crawled in parallel to improve throughput. Crawl4AI's async architecture allows many pages to be fetched simultaneously without blocking [6]. The agent might use Crawl4AI's chunking strategies to limit content size, stopping once enough relevant information is gathered (its adaptive crawling can auto-stop once the page is sufficiently covered).
- **Error Handling and Retries:** If a crawl fails (timeout, 404, etc.), the agent retries a few times. It also respects polite crawling practices: obeying `robots.txt`, applying rate limits, and using delays to avoid hammering servers. Crawl4AI supports rate-limiting and retry out-of-the-box, which we configure according to best practices to avoid overload [11].
- **Output:** For each URL, the agent compiles a "report" of extracted content, usually as Markdown text and plain text. It does **not** send raw binary (like base64 images) to the LLM; instead, it returns text snippets and links. The report schema (Pydantic) includes fields like `url: str`, `content: str`, `links: List[str]`, etc. It also may include answers if the user's question asked specifically about the content: e.g. if the user asks "What is X on this page?", the agent can pre-run an LLM against the crawled text to extract an answer. However, often it just returns raw content for the final Answer Agent to use.

File Analysis Agent

- **Purpose:** Once files are downloaded or uploaded, this agent inspects and processes them according to type and user's query. For example, files might be PDFs, images, CSVs, code, etc., and may contain data or text the user wants analyzed or summarized.
- **Type-Specific Handling:** The agent first detects each file's format (by extension or magic bytes). For text/PDFs it extracts text (using an OCR or PDF parser). For images it might run OCR or extract metadata. For data files (CSV/JSON) it loads them into memory or streams them for analysis.
- **Data Processing:** The agent then runs the appropriate analysis. This could involve executing Python code on the data (e.g. data summarization), or passing content to an LLM with a specialized prompt to extract information. It ensures each step runs to completion and checks outputs against expected schema. The plan indicates the agent "executes code and retries until no error occurs/validation is successful," so if an analysis step fails, the agent corrects the issue or re-invokes the tool until it returns valid results.
- **Intermediate Results:** For each file, the agent outputs a structured report (dict) containing relevant analysis results. For example, a PDF might yield `{ text: "...", summary: "...",`

`extracted_data: {...} }`. These results follow Pydantic models to ensure consistency. All file reports are collected into one payload and returned to the orchestrator. If no file analysis was needed (no files provided), this agent is simply skipped.

Answer Agent

- **Purpose:** This agent assembles the final answer to the user's query. It receives all information collected so far: (a) the original question text, (b) any answers or content from the HTML/Crawler Agent, and (c) analysis reports from the File Analysis Agent. It integrates these and uses the LLM to generate the final response.
- **Contextual Integration:** The agent constructs a prompt that incorporates the user's question and the data from other agents. If only files were involved, the prompt might say "Use the analysis results below to answer the question." If only web content was involved, it might say "Use the crawled webpage content below." In cases where both sources exist, the prompt systematically includes both. The agent can use chain-of-thought or other reasoning to combine insights.
- **Cases & Prompts:** The plan suggests two scenarios: *only_file* and *only_html*. In **only_file** cases, the prompt focuses on the user's question and the file-based data. In **only_html** cases, it focuses on web content. If both exist, it merges them. These custom prompts ensure irrelevant data is excluded (keeping LLM context concise).
- **Producing the Answer:** The agent's LLM writes a comprehensive answer in natural language. Crucially, it *includes citations* or references to sources for any factual claims. For example, if content was retrieved from a webpage, it cites that page (using a bracket style or hyperlink). If results came from file content, it cites the filename or any reference given. By including sources, we meet the requirement that answers have sources embedded. The answer's final format (JSON, text, list, etc.) is exactly as the user requested in the input schema. The orchestrator may enforce this by specifying the expected format in the LLM prompt or by re-running the agent if the output format is wrong.
- **Validation Loop:** After generation, the orchestrator checks the answer's structure and completeness. If it does not match the requested format or seems incomplete, the orchestrator can invoke the Answer Agent again, possibly with additional instructions (e.g. "Your last response missed referencing the sources as required. Please include them"). This loop continues until the output is fully satisfactory. Using the LLM's evaluation and re-prompting ensures robustness in the final output.

FastAPI & Asynchronous Implementation

- **Async Endpoints:** We implement the Orchestrator and agent invocations as FastAPI endpoints or background routes. FastAPI natively supports `async def` handlers, letting us perform non-blocking I/O. For example, an API call to start processing spawns async tasks for downloading or crawling without blocking the server. FastAPI's async nature (powered by Starlette) is ideal for I/O-bound workloads [4].
- **Background Tasks & Workers:** For truly long-running jobs (like large file analysis or lengthy crawls), we use FastAPI's `BackgroundTasks` or an external task queue (e.g. Celery). This way, the HTTP request can return immediately with a task ID or "processing started" message, while the heavy work happens in the background [7]. Celery workers (with Redis/RabbitMQ) ensure tasks survive server restarts and can be monitored independently [7]. For example, the File Download Agent might schedule its downloads as background tasks so the API stays responsive.

- **Concurrency and Limits:** We leverage Python's `asyncio` to run multiple I/O-bound tasks in parallel (e.g. multiple web crawls or file downloads). Crawl4AI itself uses an `AsyncWebCrawler` that we use inside `async with` blocks [12]. We also configure maximum concurrency (number of simultaneous connections) and streaming chunk sizes to handle large files efficiently. This meets the 200MB requirement without overloading memory [8].
- **Data Streaming:** When receiving uploads (up to 200MB), we use streaming endpoints. FastAPI's `UploadFile` (or manual request stream) reads data in chunks, saving to disk on-the-fly [8]. This prevents memory exhaustion for large files [8]. Similarly, when downloading from the internet, we stream content. All endpoints declare size limits or use middleware to enforce a maximum request body (e.g. 200MB) for security, rejecting anything larger to avoid abuse.
- **Dependencies and DI:** We use FastAPI's dependency injection to manage shared resources (like database connections, HTTP clients, or LLM clients). For instance, a shared Crawl4AI or HTTP session can be provided as a dependency to each agent. This makes the code cleaner and supports reuse of connections (e.g. one Playwright browser instance for all crawls).
- **Documentation & Consistency:** FastAPI automatically generates OpenAPI docs. By defining Pydantic schemas and annotations, our API's contract is self-documenting. This helps verify that the orchestrator's input schema (which includes fields like `file_urls`, `web_urls`, `question`, etc.) matches exactly what the front-end or user query provides.

Scalability, Performance, and Optimization

- **Parallel Processing:** The architecture inherently allows parallelism. Multiple file downloads, crawls, and analyses can run simultaneously. For example, if a query has 5 URLs, the HTML agent can fetch them in parallel using Crawl4AI. If there are multiple files, the File Analysis Agent can process them concurrently (depending on CPU vs I/O bound). This keeps total latency down. Crawl4AI's asynchronous design explicitly supports multi-URL crawling [6].
- **Efficient Content Handling:** To optimize LLM usage, we chunk or summarize large content before passing it to the LLM. For example, lengthy HTML pages can be broken into smaller pieces or pre-summarized. Crawl4AI offers chunking strategies (regex, sentence-based, etc.) to reduce payload size [13]. This ensures the LLM sees only relevant data, improving speed and quality.
- **Caching:** We can optionally cache intermediate results (e.g. recently downloaded pages or file analyses) so repeated requests to the same data do not require re-fetching. Crawl4AI has built-in caching support, and we could use a Redis/memory cache for file contents or analysis outputs keyed by URL or file hash. This speeds up duplicate work.
- **Error Resilience:** Each agent includes robust error handling. For network calls (downloads, crawls), we use timeouts, retries, and backoff. If an external service is unavailable, we catch the error and either retry or log it and continue. Any LLM call (inside an agent) is also wrapped to catch parsing errors or API limits. Because each agent's output is Pydantic-validated, an LLM hiccup causing malformed output can be detected and retried with clearer prompts. This makes the overall system more reliable.

Answer Formatting and Sources

- **Source Citations:** A key requirement is that answers include sources. The Answer Agent ensures that any factual statement is backed by a source from the data. It embeds citations (e.g. `【source†Lx-Ly】`) pointing to the documents or pages used. This mirrors how RAG (Retrieval-Augmented Generation) systems include references. For example, if the user's question was

answered using content from a website, the agent quotes relevant text and cites the URL snippet from the Crawl4AI output. Similarly, if a file was analyzed, the answer notes "(based on *file X*, page Y)". This practice increases trust and traceability of the answer's claims.

- **Adhering to User's Format:** The user may request the answer in a specific format (plain text, Markdown, JSON fields, etc.). The system prompts the Answer Agent to produce exactly that format. For instance, if `data_analysis_input` specifies JSON output, the LLM is instructed to format its reply as JSON. FastAPI can enforce return types via Pydantic models if needed. After generation, the orchestrator checks format (e.g. no stray text outside JSON). If it's incorrect, it guides the LLM to correct it. This ensures the final output "is as requested in the questions" (per requirement).
- **Completeness and Sources:** The orchestrator also checks that all parts of the user's query are addressed. If any sub-question was missed, or if there is ambiguity, the orchestrator can have the Answer Agent clarify or expand. Finally, the answer is returned through FastAPI to the user, including the requested format and embedded source citations.

This comprehensive plan lays out a robust, asynchronous multi-agent pipeline using FastAPI and Pydantic for validation, Crawl4AI for web content, and LLM-powered agents for intelligence. By structuring data, handling large inputs carefully, and explicitly coordinating agents through an orchestrator, we ensure efficiency, scalability, and correctness [5] [1]. The outlined design follows best practices (as cited) and addresses all requirements: modular agents, async execution, large-file handling, and answers with sources.

Sources: We referenced best practices for multi-agent orchestration [1], structured data modeling with Pydantic [5], and the capabilities of Crawl4AI and FastAPI's async architecture [6] [4] to inform this design. Each element of the plan is grounded in these authoritative examples.

---

[1] Multi-Agent AI Systems: When to Expand From a Single Agent
https://www.willowtreeapps.com/craft/multi-agent-ai-systems-when-to-expand

[2] [5] Building a Structured Research Automation System Using Pydantic
https://www.analyticsvidhya.com/blog/2025/03/multi-agent-research-assistant-system-using-pydantic/

[3] Building End-to-End Multi-Agent AI Workflows with OpenAI Agents SDK, FastAPI, Perplexity, and Notion
https://www.aiproductivityinsights.com/p/building-ai-workflows-with-openai-agents-sdk-fastapi-perplexity-and-notion

[4] Fast API Custom Types: A Comprehensive Guide | Orchestra
https://www.getorchestra.io/guides/fast-api-custom-types-a-comprehensive-guide

[6] Crawl4AI · PyPI
https://pypi.org/project/Crawl4AI/0.3.7/

[7] Celery and Background Tasks. Using FastAPI with long running tasks | by Hitoruna | Medium
https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d

[8] Async File Uploads in FastAPI: Handling Gigabyte-Scale Data Smoothly | by Hash Block | Jul, 2025 | Medium
https://medium.com/@connect.hashblock/async-file-uploads-in-fastapi-handling-gigabyte-scale-data-smoothly-aec421335680

[9] Best Practices for Using Pydantic in Python - DEV Community
https://dev.to/devasservice/best-practices-for-using-pydantic-in-python-2021

10 11 13 Crawl4AI: Unleashing Efficient Web Scraping | by Gautam Chutani | Medium
https://gautam75.medium.com/crawl4ai-unleashing-efficient-web-scraping-1825560300c3

12 Home - Crawl4AI Documentation (v0.7.x)
https://docs.crawl4ai.com/