Section 2: The Agentic Core - Designing Dynamic CrewAI Systems

Building on the robust, real-time architecture established in the previous section, we now turn to the intelligent core of the application: the agentic system. This section details the design and implementation of a dynamic CrewAI framework responsible for interpreting user goals, executing complex tasks, and generating structured outputs compatible with our WebSocket streaming infrastructure. The design prioritizes flexibility, specialization, and the advanced reasoning capabilities of the Gemini API.

The Coordinator and Specialist Model

To effectively handle diverse and complex user requests, we adopt a hierarchical "Coordinator and Specialist" model, a powerful pattern for orchestrating multi-agent systems . This architecture designates a primary agent as a "Coordinator," whose main purpose is to understand a user's high-level goal, classify their intent, and delegate specific sub-tasks to a team of specialized agents [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).

The **Coordinator Agent** acts as the central point of contact for the user's query. It is configured with a broad goal to comprehend user intent and orchestrate the workflow, requiring the allow_delegation: true parameter to enable it to pass tasks to other agents in the crew [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).

The **Specialist Agents** are designed as experts in narrow domains, ensuring they perform their designated functions with high efficacy. An agent performs better with a clearly defined, non-overlapping role [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial). The initial team of specialists includes:

Agent	Role	Description
search_agent	Factual Researcher	Handles real-time information lookups and data retrieval from the web [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).
research_agent	In-depth Analyst	Performs comprehensive analysis on complex topics, synthesizing information from multiple sources [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).
scraper_agent	Data Extractor	Extracts and structures specific data from designated web pages or documents [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).
image_agent	Visual Creator	Generates images based on textual descriptions

provided by the user or other agents [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial). |

This model allows the system to dynamically assemble the right team for the job. For instance, a request to "research the latest advancements in AI and create a summary" would be routed by the Coordinator to the research_agent, which would then execute the task.

Dynamic Configuration and Gemini Integration

To maintain flexibility and separate configuration from application logic, agents and tasks are defined in agents.yaml and tasks.yaml files [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial). This approach makes the system more readable and allows for easier modification of agent personas or task descriptions without altering Python code [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial). User-specific inputs are dynamically injected into these definitions at runtime using placeholders like {user_input}, which are populated when a task is initiated [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).

The reasoning power for each agent is provided by Google's Gemini API. This integration is achieved by specifying the language model within the agent's configuration.

yaml
Example from agents.yaml
llm: google/gemini-2.0-flash

This llm parameter instructs the agent to use the specified Gemini model for all its reasoning and decision-making processes . For more advanced use cases requiring custom configurations, a pre-configured Gemini API client instance can be passed directly to the agent's llm parameter in Python, a pattern that offers maximum control over the model's behavior [2](https://medium.com/the-ai-forum/build-a-local-reliable-rag-agent-using-crewai-and-groq-013e5d557bcd).

To ensure stability and control over agent execution, several key parameters are configured:

Parameter	Description
max_iter	Sets the maximum number of iterations an agent can perform for a single task [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).

max_rpm	Limits the number of requests per minute to avoid rate-limiting issues with external tools or APIs [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).
verbose	Enables detailed logging of the agent's thought process and actions, which is crucial for debugging [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).
allow_delegation	Controls whether an agent can delegate tasks to other agents in the crew [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial).

Structuring Outputs for Real-Time Streaming

A critical aspect of this design is ensuring the agentic system's output is compatible with the real-time streaming architecture detailed in Section 1. The crew's final output must be predictable and well-structured to be reliably transmitted over WebSockets. To achieve this, we leverage CrewAI's structured output capabilities.

By defining a Pydantic BaseModel and assigning it to a task's output_pydantic attribute, we compel the agent to return its final result in a format that conforms to the model's schema [3](https://docs.crewai.com/concepts/tasks). This guarantees that the output is a validated, structured object, perfect for serialization and transmission.

Furthermore, to enhance the quality and reliability of the generated content, we will implement a "Grader Agent" pattern [2](https://medium.com/the-ai-forum/build-a-local-reliable-rag-agent-using-crewai-and-groq-013e5d557bcd). A dedicated Grader_agent can be added to the crew's workflow to validate the output of a specialist agent against criteria like relevance, factual accuracy, and adherence to the required format before the final result is produced [2](https://medium.com/the-ai-forum/build-a-local-reliable-rag-agent-using-crewai-and-groq-013e5d557bcd).

This structured and validated output is the payload that the FastAPI backend will stream to the Next.js frontend. As described in the previous section, the CrewAI task will be executed in a separate thread using run_in_threadpool to avoid blocking the server [4](https://stackoverflow.com/questions/67599119/fastapi-asynchronous-background-tasks-blocks-other-requests). Upon completion, the crew's final, structured result is published to all connected WebSocket clients. While the entire agentic process runs to completion on the backend, the frontend will receive the final, complete response and can render it with a "typewriter effect" to simulate a real-time stream, providing a polished and responsive user experience [1](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-

tutorial). This approach bridges the complex, multi-step reasoning of the agentic core with the user's expectation of immediate, continuous feedback.