Introduction: Architecting a Real-Time, Agentic Web Application

The primary objective of this project is to build and develop a sophisticated web application that integrates CrewAI agents with the Gemini API to execute complex, user-defined tasks. A core requirement is to provide users with real-time visual feedback, allowing them to observe the agentic process as it unfolds. This introduces a significant architectural challenge: how to manage long-running, asynchronous, and often CPU-intensive AI processes on the backend while maintaining a responsive, interactive, and real-time user experience on the frontend. This section outlines the chosen technology stack and architectural patterns designed to solve this problem, establishing a robust foundation for a production-grade, agentic application.

To handle the demanding nature of AI workloads, the backend architecture must be carefully considered. While FastAPI's built-in BackgroundTasks feature offers a simple way to run operations after a response is sent, it is fundamentally unsuited for this use case . BackgroundTasks run within the same process as the web server, which means CPU-intensive AI jobs would block the event loop and degrade the performance of the entire application [1](https://testdriven.io/blog/fastapi-and-celery/). Furthermore, this feature is a "fire-and-forget" system with no built-in mechanism for tracking task status or retrieving results, making it inadequate for complex workflows . The optimal solution is a more robust, distributed task queue system. We have selected a Celery and Redis stack. Celery decouples task execution from the web server process by using dedicated worker processes that can be scaled independently across multiple machines . This distributed architecture is essential for handling heavy AI computations without impacting API responsiveness [2](https://www.youtube.com/watch?v=eAHAKowv6hk). Celery also provides robust state management, allowing the application to track the status of each task and retrieve its results, a feature critical for a reliable production system .

Feature	FastAPI BackgroundTasks	Celery / Redis Stack	Justification for Choice
Scalability	Low; runs in the same server process [3](https://levelup.gitconnected.com/fastapi-background-tasks-vs-celery-which-is-right-for-your-application-dff0a7216e55).	High; distributed workers can be scaled horizontally [3](https://levelup.gitconnected.com/fastapi-background-tasks-vs-celery-which-is-right-for-your-application-dff0a7216e55).	AI tasks are resource-intensive and require a scalable solution to support multiple users.
State Management	None; "fire-and-forget" [4](https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d).	Robust; built-in

status tracking and result backend [5](https://medium.com/@wprado/streamlining-workload-distribution-in-fastapi-with-celery-b937f3018282). | Essential for monitoring complex, multi-step agentic workflows and retrieving final outputs. |
| **Reliability** | Low; tasks are lost if the server crashes [4](https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d). | High; tasks persist in the message broker (Redis) [4](https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d). | Critical for ensuring user-initiated tasks are not lost due to server-side issues. |
| **Monitoring** | None; requires custom implementation [3](https://levelup.gitconnected.com/fastapi-background-tasks-vs-celery-which-is-right-for-your-application-dff0a7216e55). | Excellent; supports tools like Flower for real-time monitoring . | Necessary for observability and troubleshooting in a production environment. |

With a scalable backend in place, the next challenge is to stream progress from the long-running Celery tasks to the user in real time. For this, WebSockets are the ideal choice. They establish a persistent, bidirectional communication channel between the FastAPI backend and the Next.js frontend, allowing the server to push incremental updates to the client without waiting for a request [6](https://unfoldai.com/fastapi-and-websockets/). The proposed architecture involves a centralized connection manager on the backend that broadcasts progress updates to all subscribed clients [6](https://unfoldai.com/fastapi-and-websockets/). This decouples the AI task's lifecycle from any single client connection, ensuring that the process continues running and its updates are available to any connected user, even if the initiating client disconnects [7](https://github.com/tiangolo/fastapi/discussions/6221).

This robust architecture is specifically designed to support complex, dynamic agentic systems. The application will leverage advanced CrewAI patterns, such as a "Coordinator and Specialist" model, where a primary agent interprets a user's intent and delegates sub-tasks to a team of specialized agents for research, data scraping, or image generation [8](https://www.firecrawl.dev/blog/crewai-multi-agent-systems-tutorial). The long-running and unpredictable nature of these hierarchical agent interactions underscores the need for the chosen architecture: a scalable Celery backend to execute the tasks reliably and WebSockets to provide the seamless, real-time feedback that is central to the application's user experience.