# Streaming Real-Time Updates from CrewAI Tasks with FastAPI and Next.js

This report outlines the best practices for streaming real-time updates from a long-running background task, such as a CrewAI process, in a FastAPI backend to a Next.js frontend using WebSockets. It covers the implementation of the backend endpoint, handling of the long-running task, secure client-side connection, and overall connection lifecycle management.

## 1. FastAPI Backend Implementation

The backend is responsible for managing WebSocket connections, running the long-running CrewAI task without blocking, and pushing incremental progress to the connected clients.

### 1.1. WebSocket Endpoint and Connection Management

A robust architecture requires a centralized manager to handle WebSocket connections. This manager tracks all active clients, allowing for broadcasting messages.

- **Basic Endpoint:** A WebSocket endpoint is created in FastAPI using the `@app.websocket("/ws")` decorator. The connection is established with `await websocket.accept()` [ref: 2-1].
- **Connection Manager:** A `ConnectionManager` class can be implemented to maintain a list of active connections. It should have methods to `connect`, `disconnect`, and `broadcast` messages to all connected clients [ref: 2-1].

A simple implementation of a connection manager:
```python
# From [ref: 2-1]
from typing import List
from fastapi import WebSocket

class ConnectionManager:
    def __init__(self):
        self.active_connections: List[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)
    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def broadcast(self, message: str):
        for connection in self.active_connections:
            await connection.send_text(message)
```

### 1.2. Handling Long-Running and CPU-Bound Tasks

A critical challenge with `asyncio` is that a synchronous, CPU-bound function (like a complex computation in a CrewAI agent) will block the entire server's event loop [ref: 0-3]. The solution is to run such tasks in a separate thread or process.

Method	Description	Use Case	Reference
`run_in_threadpool`	A utility from `fastapi.concurrency` that runs a function in a separate thread, preventing the event loop from blocking.	For synchronous, I/O-bound or moderately CPU-bound functions within an `async` context.	[ref: 0-3]
`loop.run_in_executor`	An `asyncio` function to run a task in a thread or process pool executor. Can be configured with a `ProcessPoolExecutor` for true parallelism.	For heavy CPU-bound tasks that need to run in a separate process to leverage multiple CPU cores.	[ref: 0-3]
Celery	A distributed task queue for handling heavy, long-running, or periodic background jobs.	For complex, production-grade systems requiring distributed task processing, retries, and monitoring.	[ref: 0-4]

For a CrewAI task, which can be computationally intensive, using `run_in_threadpool` or `run_in_executor` is essential to keep the API responsive [ref: 0-3].

### 1.3. Decoupling the Task from the WebSocket Lifecycle

A long-running task should not be tied to a single client's connection. If the client who initiated the task disconnects, the task should continue running and other clients should still be able to receive updates [ref: 2-0].

A robust architectural pattern involves a singleton `Listener` or manager class:

1. **Singleton Manager:** Create a global manager instance responsible for running the background task (e.g., the CrewAI process) [ref: 2-0].
2. **Application Lifecycle:** Start the core task logic on application startup using FastAPI's `@app.on_event("startup")` event handler. This ensures the process

runs independently of any client connections [ref: 2-0].
3. **Pub/Sub Mechanism:** The manager should implement a publish-subscribe model.
    -   When a client connects via WebSocket, it subscribes to the manager, typically by being given an `asyncio.Queue` [ref: 2-0].
    -   The manager maintains a list of all subscriber queues [ref: 2-0].
    -   As the CrewAI task generates incremental updates, the manager `put`s these updates into every subscriber's queue [ref: 2-0].
4. **WebSocket Handler:** The WebSocket endpoint handler's role is to:
    -   Create a queue for the new client and subscribe it to the manager [ref: 2-0].
    -   Enter a loop, waiting for messages from its queue (`await queue.get()`) and forwarding them to the client (`await websocket.send_text(...)`) [ref: 2-0].
    -   Upon client disconnection (which raises a `WebSocketDisconnect` exception), it must unsubscribe from the manager to clean up resources and prevent memory leaks [ref: 2-1].

This architecture ensures the CrewAI task runs once and its progress is broadcast to all currently connected and interested clients [ref: 2-0].

### 1.4. Alternative: Server-Sent Events (SSE)

For scenarios where communication is strictly one-way (server-to-client), Server-Sent Events (SSE) offer a simpler alternative to WebSockets. SSE is built on top of standard HTTP and can be implemented in FastAPI using `StreamingResponse` [ref: 0-1]. This is well-suited for streaming progress updates, live metrics, or status monitors [ref: 0-1, ref: 1-4].

## 2. Next.js Frontend Implementation

The Next.js client needs to manage the WebSocket connection's lifecycle, handle incoming messages, and display the real-time updates.

### 2.1. Connection Management with React Hooks

Managing a persistent WebSocket connection within React's re-rendering lifecycle can be tricky.

-   **`useRef` for the WebSocket Instance:** The recommended practice is to store the WebSocket client instance in a `useRef` hook. This ensures the connection object persists across component re-renders without causing new connections to be established [ref: 1-0].
-   **`useEffect` for Lifecycle Management:**
    -   A `useEffect` hook with an empty dependency array (`[]`) should be used to create the WebSocket connection when the component mounts [ref: 1-0].

    -   The cleanup function returned by this `useEffect` is the ideal place to close the connection (`ws.current.close()`) when the component unmounts, preventing memory leaks [ref: 1-0].
-   **Event Handlers:** Inside `useEffect`, assign functions to the WebSocket's `onopen`, `onmessage`, `onclose`, and `onerror` event handlers [ref: 1-2].

### 2.2. Receiving and Displaying Data

-   **State Management:** Use a `useState` hook to maintain an array of messages received from the backend [ref: 1-2].
-   **Updating State:** The `onmessage` handler will parse the incoming event data and update the state, triggering a re-render to display the new information [ref: 1-2].

Example of a React component for handling WebSocket connection:
```javascript
// Based on principles from [ref: 1-0] and [ref: 1-2]
import React, { useState, useEffect, useRef } from 'react';

const CrewAIStatus = () => {
    const [messages, setMessages] = useState([]);
    const ws = useRef(null);

    useEffect(() => {
        // Connect only once
        ws.current = new WebSocket("ws://localhost:8000/ws/crew-updates");
        ws.current.onopen = () => console.log("WebSocket connected");
        ws.current.onclose = () => console.log("WebSocket disconnected");

        const wsCurrent = ws.current;

        // Cleanup on unmount
        return () => {
            wsCurrent.close();
        };
    }, []);

    useEffect(() => {
        if (!ws.current) return;

        // Message handler
        ws.current.onmessage = (e) => {
            const message = e.data;
            setMessages((prevMessages) => [...prevMessages, message]);
        };
    }, []); // Note: In a real app, dependencies might be needed if the handler logic changes

    return (
        <div>
            <h1>CrewAI Task Updates</h1>
            <ul>
                {messages.map((msg, index) => (
                    <li key={index}>{msg}</li>
                ))}
            </ul>
        </div>
    );
};
```

For cleaner code, this logic can be encapsulated within a custom hook, such as `useSocket` [ref: 1-1].

## 3. Security, Scalability, and Stability

### 3.1. Authentication

Authenticating WebSocket connections is challenging because browsers do not allow setting the `Authorization` header on the initial WebSocket request [ref: 2-4].

-   **Recommended Method:** Pass the JWT via the `Sec-WebSocket-Protocol` header.
    1. On the client, the token is included in the subprotocol array during WebSocket initialization: `new WebSocket(url, ["yourprotocol", "base64.websocket.bearer." + b64_encoded_token])` [ref: 2-4].
    2. On the server, an ASGI middleware intercepts the request, extracts the token from the `sec-websocket-protocol` header, and rewrites it into the standard `authorization` header [ref: 2-4].
    3. This allows FastAPI's standard dependency injection system (`Depends`) to handle JWT verification on the WebSocket endpoint as it would for a regular HTTP route [ref: 2-4].
-   **Alternative Methods:** Passing the token as a query parameter is another option, but it is generally considered less secure as the token can be exposed in server logs and browser history [ref: 2-1].

Once the authentication mechanism is in place, the WebSocket endpoint can be protected using a dependency that verifies the user's credentials [ref: 2-1, ref: 2-3].

### 3.2. Scalability

When deploying multiple instances of the FastAPI application, an in-memory connection manager will not work, as a client connected to one instance will not receive messages broadcast from another.

-   **Solution:** Use an external message broker with a publish/subscribe system, such as Redis [ref: 0-1, ref: 2-1].
-   **Architecture:**
    -   The long-running task manager publishes updates to a Redis channel [ref: 2-1].
    -   Each FastAPI server instance subscribes its connected WebSocket clients to that Redis channel [ref: 2-1].
    -   Redis handles broadcasting the messages to all server instances, which then forward them to their respective clients [ref: 2-1].

### 3.3. Connection Stability

-   **Heartbeats:** To detect and clean up stale or dead connections, implement a heartbeat (ping/pong) mechanism. This can be achieved by running a concurrent `asyncio.create_task` that periodically sends a "ping" message to the client [ref: 0-0, ref: 2-1]. If a response is not received or a send fails, the connection can be considered dead and closed.
-   **Graceful Disconnects:** Always wrap the main WebSocket loop in a `try...except WebSocketDisconnect` block to ensure resources are cleaned up properly when a client disconnects, such as unsubscribing from any pub/sub systems [ref: 0-1, ref: 2-1].