

Send step completions UI

i have fast api, backed code api multiple agents to complete steps, i need to send completion steps to send UI

Got it 👍 You have a **FastAPI backend** where multiple "agents" (workers, processes, or tasks) complete steps in sequence (or parallel), and you want to **send these step completions in real-time to the UI** so the frontend can show progress.

There are a few approaches you can take depending on how "live" you want the updates to be:

◆ Option 1: Server-Sent Events (SSE)

- Simple to implement in FastAPI.
- Browser `EventSource` API can consume updates.
- Best for streaming logs / progress step updates.

FastAPI Backend

```
python

from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import asyncio

app = FastAPI()

async def step_generator():
    steps = ["Agent 1 started", "Agent 1 completed",
            "Agent 2 started", "Agent 2 completed",
            "All done ✅"]

    for step in steps:
        yield f"data: {step}\n\n"
        await asyncio.sleep(2) # simulate processing delay

@app.get("/progress")
```

```
async def progress():
    return StreamingResponse(step_generator(), media_type="text/event-stream")
```

Frontend (Next.js / React)

```
tsx

import { useEffect, useState } from "react";

export default function Progress() {
  const [logs, setLogs] = useState<string[]>([]);

  useEffect(() => {
    const eventSource = new EventSource("http://localhost:8000/progress");

    eventSource.onmessage = (event) => {
      setLogs((prev) => [...prev, event.data]);
    };

    return () => {
      eventSource.close();
    };
  }, []);

  return (
    <div>
      <h2>Progress</h2>
      <ul>
        {logs.map((log, i) => (
          <li key={i}>{log}</li>
        ))}
      </ul>
    </div>
  );
}
```

◆ Option 2: WebSockets

- More flexible (bidirectional communication).
- Useful if you want the **UI to also send messages back** (e.g., cancel job, add new step).

FastAPI Backend

```
python

from fastapi import FastAPI, WebSocket
import asyncio

app = FastAPI()

@app.websocket("/ws/progress")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    steps = ["Agent 1 started", "Agent 1 completed",
             "Agent 2 started", "Agent 2 completed",
             "All done ✅"]

    for step in steps:
        await websocket.send_text(step)
        await asyncio.sleep(2)

    await websocket.close()
```

Frontend (React / Next.js)

```
tsx

import { useEffect, useState } from "react";

export default function Progress() {
    const [logs, setLogs] = useState<string[]>([]);

    useEffect(() => {
        const ws = new WebSocket("ws://localhost:8000/ws/progress");

        ws.onmessage = (event) => {
            setLogs((prev) => [...prev, event.data]);
        };

        ws.onclose = () => {
            console.log("Connection closed");
        };

        return () => ws.close();
    }, []);
}
```

```

}, []);

return (
  <div>
    <h2>Progress</h2>
    <ul>
      {logs.map((log, i) => (
        <li key={i}>{log}</li>
      ))}
    </ul>
  </div>
);
}

```

◆ Option 3: Polling

- Easiest but not real-time.
- Backend stores progress in DB or cache (Redis).
- UI polls `/status/{task_id}` every few seconds.

✅ Recommendation:

- If you need **real-time push updates only from backend → frontend**, go with **SSE**.
- If you need **bi-directional communication (UI ↔ backend)**, go with **WebSockets**.

Do you want me to give you a **full FastAPI + Next.js 15 (src/app)** working example with **SSE** (simpler) or **WebSockets** (more flexible)?

we can use StreamingResponse, UI side on generating need to show steps, and final api response is file url

Perfect ✅ That means:

- Your **FastAPI backend** runs multiple agents (steps).

- You want to **stream step updates** to the UI as they happen.
- At the **end of streaming**, you want to send the **final file URL** (download link, result path, etc.).

This can be done nicely with `StreamingResponse` (SSE style).

◆ FastAPI Backend (`main.py`)

```
python

from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import asyncio

app = FastAPI()

async def generate_steps():
    steps = [
        "Agent 1 started",
        "Agent 1 completed",
        "Agent 2 started",
        "Agent 2 completed",
        "Finalizing..."
    ]

    for step in steps:
        yield f"data: {step}\n\n"
        await asyncio.sleep(2) # simulate work

    # ✅ Send final file URL at the end
    file_url = "http://localhost:8000/files/output.pdf"
    yield f"data: {{\"file_url\": \"{file_url}\"}}\n\n"

@app.get("/progress")
async def progress():
    return StreamingResponse(generate_steps(), media_type="text/event-stream")
```

◆ UI Side (Next.js 15, `src/app/progress/page.tsx`)

tsx

```
"use client";

import { useEffect, useState } from "react";

export default function ProgressPage() {
  const [logs, setLogs] = useState<string[]>([]);
  const [fileUrl, setFileUrl] = useState<string | null>(null);

  useEffect(() => {
    const eventSource = new EventSource("http://localhost:8000/progress");

    eventSource.onmessage = (event) => {
      try {
        const data = JSON.parse(event.data);
        if (data.file_url) {
          // ✅ Final file URL received
          setFileUrl(data.file_url);
        }
      } catch {
        // Normal step update
        setLogs((prev) => [...prev, event.data]);
      }
    };

    eventSource.onerror = () => {
      console.error("EventSource failed.");
      eventSource.close();
    };

    return () => {
      eventSource.close();
    };
  }, []);

  return (
    <div className="p-6">
      <h2 className="text-lg font-semibold mb-4">Progress</h2>
      <ul className="space-y-2">
        {logs.map((log, i) => (
          <li key={i} className="text-gray-700">
        ))}
      </ul>
    </div>
  );
}
```

```

        {log}
      </li>
    })
  </ul>

  {fileUrl && (
    <div className="mt-6">
      <a
        href={fileUrl}
        className="px-4 py-2 bg-blue-600 text-white rounded-lg shadow hover:bg-blue-700"
        target="_blank"
        rel="noopener noreferrer"
      >
        Download Result
      </a>
    </div>
  )}
</div>
);
}

```

♦ Flow

1. When UI calls `/progress` , FastAPI **streams step updates** (`Agent started` , `Agent completed` , etc.).
2. UI appends each step to the **logs list**.
3. At the end, backend sends a **JSON payload with** `file_url` .
4. UI detects it and shows a **download button**.

👉 Do you also want me to show how to **serve the generated file** in FastAPI (e.g., `/files/output.pdf`), so clicking "Download" works directly?

2/2

api side in all teps related write some realtime logic
instaed of dummy `asyncio.sleep(2)`

Got it 👍 You don't want `asyncio.sleep(2)` — instead you want to run **your real agent logic step-by-step**, and stream each step to the UI in real-time until the final file is ready.

Here's a **practical pattern** you can use with `StreamingResponse` :

◆ FastAPI Backend (`main.py`)

```
python

from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import asyncio
import os

app = FastAPI()


# Example: function that simulates your multi-agent pipeline
async def run_agents_and_generate_file():
    # Step 1: Agent 1 starts
    yield f"data: Agent 1 started\n\n"
    # --- Replace with real logic ---
    await asyncio.sleep(1) # simulate actual work
    # Example: do some computation / DB call
    yield f"data: Agent 1 completed\n\n"

    # Step 2: Agent 2 starts
    yield f"data: Agent 2 started\n\n"
    await asyncio.sleep(1)
    yield f"data: Agent 2 completed\n\n"

    # Step 3: Finalizing (file generation)
    yield f"data: Finalizing output file...\n\n"
    await asyncio.sleep(1)

    # Example: Write a dummy file (replace with real generation logic)
    output_dir = "files"
    os.makedirs(output_dir, exist_ok=True)
    file_path = os.path.join(output_dir, "output.txt")
    with open(file_path, "w") as f:
        f.write("This is the generated result.\n")
```



```
#  Send final file URL to the UI
file_url = f"http://localhost:8000/{file_path}"
yield f"data: {{{\"file_url\": \"{file_url}\"}}}\n\n"

@app.get("/progress")
async def progress():
    return StreamingResponse(run_agents_and_generate_file(), media_type="text/event-stream")
```

◆ Serving Files

So that the frontend can **download the result**:

```
python

from fastapi.staticfiles import StaticFiles

# Mount /files path for downloading results
app.mount("/files", StaticFiles(directory="files"), name="files")
```

Now, if the backend generated `files/output.txt`, it's available at:

```
bash

http://localhost:8000/files/output.txt
```

◆ Frontend (Next.js 15 – `src/app/progress/page.tsx`)

(same as before, no dummy logic on backend anymore)

```
tsx

"use client";

import { useEffect, useState } from "react";

export default function ProgressPage() {
    const [logs, setLogs] = useState<string[]>([]);
```

```

const [fileUrl, setFileUrl] = useState<string | null>(null);

useEffect(() => {
  const eventSource = new EventSource("http://localhost:8000/progress");

  eventSource.onmessage = (event) => {
    try {
      const data = JSON.parse(event.data);
      if (data.file_url) {
        setFileUrl(data.file_url); // ✅ Final file ready
      }
    } catch {
      setLogs((prev) => [...prev, event.data]); // Normal step log
    }
  };

  eventSource.onerror = () => {
    console.error("EventSource failed.");
    eventSource.close();
  };

  return () => {
    eventSource.close();
  };
}, []);

return (
  <div className="p-6">
    <h2 className="text-lg font-semibold mb-4">Progress</h2>
    <ul className="space-y-2">
      {logs.map((log, i) => (
        <li key={i} className="text-gray-700">
          {log}
        </li>
      ))}
    </ul>

    {fileUrl && (
      <div className="mt-6">
        <a
          href={fileUrl}
          className="px-4 py-2 bg-blue-600 text-white rounded-lg shadow hover:bg-blue-700"

```

```

        target="_blank"
        rel="noopener noreferrer"
    >
        Download Result
    </a>
</div>
})
</div>
);
}

```

✅ This way:

- Each **agent step** runs your actual logic (not `sleep`).
- You stream logs (`yield f"data: ... \n\n"`) in real-time.
- At the end, you generate a **real file** and send its URL.
- UI shows progress logs **live** and a **download button** at the end.

👉 Do you want me to extend this so each agent runs in **parallel (concurrent tasks)** and you stream updates as soon as each one finishes?

give actual functions code also

Perfect 👍 let's make this **real**, with actual functions for each agent step, instead of dummy `sleep` .

Here's a **complete FastAPI example** where:

- Each agent runs its own logic (like data preprocessing, ML model inference, file generation, etc.).
- Progress is streamed in **real-time** to the frontend.
- Final output file URL is sent when done.

♦ FastAPI Backend (`main.py`)

python

```

from fastapi import FastAPI
from fastapi.responses import StreamingResponse
from fastapi.staticfiles import StaticFiles
import asyncio
import os
import time

app = FastAPI()

# -----
# Example Agent Functions
# -----

async def agent1_preprocess():
    """Simulate preprocessing step (e.g., clean input data)."""
    await asyncio.sleep(1) # simulate work
    return "Preprocessing completed"

async def agent2_inference():
    """Simulate model inference (e.g., run ML model)."""
    await asyncio.sleep(2)
    return "Model inference completed"

async def agent3_postprocess():
    """Simulate postprocessing (e.g., format results)."""
    await asyncio.sleep(1)
    return "Postprocessing completed"

async def generate_file():
    """Generate final file after all agents are done."""
    output_dir = "files"
    os.makedirs(output_dir, exist_ok=True)
    file_path = os.path.join(output_dir, f"output_{int(time.time())}.txt")

    with open(file_path, "w") as f:
        f.write("This is the generated result file.\n")
        f.write("✅ Completed by all agents.\n")

    return file_path

# -----
# Streaming Logic

```

```
# -----
```

```
async def run_pipeline():
    # Step 1: Run Agent 1
    yield f"data: Agent 1 started\n\n"
    result1 = await agent1_preprocess()
    yield f"data: {result1}\n\n"

    # Step 2: Run Agent 2
    yield f"data: Agent 2 started\n\n"
    result2 = await agent2_inference()
    yield f"data: {result2}\n\n"

    # Step 3: Run Agent 3
    yield f"data: Agent 3 started\n\n"
    result3 = await agent3_postprocess()
    yield f"data: {result3}\n\n"

    # Step 4: Finalizing - File Generation
    yield f"data: Generating final file...\n\n"
    file_path = await generate_file()

    file_url = f"http://localhost:8000/{file_path}"
    yield f"data: {{\"file_url\": \"{file_url}\"}}\n\n"

@app.get("/progress")
async def progress():
    return StreamingResponse(run_pipeline(), media_type="text/event-stream")

# Serve files so UI can download
app.mount("/files", StaticFiles(directory="files"), name="files")
```

◆ What Happens Here

- `agent1_preprocess()` → simulates data cleaning.
- `agent2_inference()` → simulates running ML model.
- `agent3_postprocess()` → simulates formatting.
- `generate_file()` → creates an actual output file
(`files/output_<timestamp>.txt`).

- Each step sends **real-time logs** (`yield f"data: ...\n\n"`).
 - Final step sends JSON with `file_url` .
-

◆ Example UI Output

When you open the frontend, you'll see logs streaming like:

```
cpp
```

```
Agent 1 started  
Preprocessing completed  
Agent 2 started  
Model inference completed  
Agent 3 started  
Postprocessing completed  
Generating final file...
```

And then finally, a **download button** for the generated file.

👉 Do you want me to also show a **concurrent (parallel) version**, where all agents run in parallel and the UI receives their updates as soon as each finishes?