

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?

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

add example simulate work and give complete code

Perfect 👍 instead of just `asyncio.sleep` , let's add **some actual simulated work** in each agent function so it looks like real processing.

For example:

- **Agent 1** → Preprocess text (normalize, lowercase, remove stopwords).
- **Agent 2** → Simulate ML inference (word count, keyword extraction).
- **Agent 3** → Postprocess (format results).

- Finally → Generate a file with results.

Here's a **complete FastAPI backend** with working code 📌

◆ `main.py`

python

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

app = FastAPI()

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

async def agent1_preprocess(text: str):
    """Simulate preprocessing step: lowercase, remove punctuation, split tokens."""
    await asyncio.sleep(1) # simulate I/O delay
    processed = re.sub(r"[^a-zA-Z0-9\s]", "", text).lower().split()
    return processed, f"Preprocessing completed → {len(processed)} tokens"

async def agent2_inference(tokens: list[str]):
    """Simulate model inference: count word frequency (mock ML model)."""
    await asyncio.sleep(2)
    freq = {}
    for t in tokens:
        freq[t] = freq.get(t, 0) + 1
    return freq, f"Inference completed → {len(freq)} unique words"

async def agent3_postprocess(freq: dict):
    """Simulate postprocessing: sort top N words."""
    await asyncio.sleep(1)
    sorted_words = sorted(freq.items(), key=lambda x: x[1], reverse=True)
    top_words = sorted_words[:5]
    return top_words, f"Postprocessing completed → top {len(top_words)} words extracted"
```

```

async def generate_file(results: list[tuple[str, int]]):
    """Generate final result file."""
    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("Top extracted keywords:\n")
        for word, count in results:
            f.write(f"{word}: {count}\n")

    return file_path

# -----
# Streaming Logic
# -----

async def run_pipeline():
    text_input = "FastAPI is an amazing framework! FastAPI makes APIs super fast, easy, and fun to build."

    # Step 1: Preprocessing
    yield f"data: Agent 1 started (Preprocessing)\n\n"
    tokens, log1 = await agent1_preprocess(text_input)
    yield f"data: {log1}\n\n"

    # Step 2: Inference
    yield f"data: Agent 2 started (Inference)\n\n"
    freq, log2 = await agent2_inference(tokens)
    yield f"data: {log2}\n\n"

    # Step 3: Postprocessing
    yield f"data: Agent 3 started (Postprocessing)\n\n"
    top_words, log3 = await agent3_postprocess(freq)
    yield f"data: {log3}\n\n"

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

    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 Now

1. **Agent 1 (Preprocess)** → cleans text, lowercases, splits tokens.

Example log:

```
nginx
```

Preprocessing completed → 13 tokens

2. **Agent 2 (Inference)** → counts word frequency.

Example log:

```
sql
```

Inference completed → 10 unique words

3. **Agent 3 (Postprocess)** → extracts top 5 words.

Example log:

```
css
```

Postprocessing completed → top 5 words extracted

4. **Final File** → writes keywords into `files/output_<timestamp>.txt`.

Example file content:

```
yaml
```

Top extracted keywords:

fastapi: 2

is: 1

an: 1

amazing: 1
framework: 1

This way you now have **actual logic** running inside each agent instead of dummy sleep.

👉 Do you want me to also give you the **Next.js 15 frontend code** updated to show a **live progress bar** along with logs?

```
async def agent1_preprocess(text: str):  
    """Simulate preprocessing step: lowercase, remove  
    punctuation, split tokens."""  
    await asyncio.sleep(1) # simulate I/O delay  
    processed = re.sub(r"^[a-zA-Z0-9\s]", "",  
text).lower().split()  
    return processed, f"Preprocessing completed →  
{len(processed)} tokens"
```

why await if use means api response won't delay?



Great question 👍

Let's clear up what's happening with `await asyncio.sleep(1)` in your `agent1_preprocess`.

♦ Why `await` is used

- In **async FastAPI**, every endpoint (or background task) can run concurrently.
- `await asyncio.sleep(1)` is a **non-blocking sleep**.
 - It **does not block** the entire FastAPI server.
 - It just says "pause this coroutine for 1s, meanwhile allow other tasks/requests to run."
- If you used a normal `time.sleep(1)`, it **would block the entire server thread**, and no other request could be served during that time.

So:

-  `await asyncio.sleep(1)` → simulates I/O without blocking.
 -  `time.sleep(1)` → blocks, delays everything.
-

♦ Will it delay API response?

Yes and No:

- **Yes** → This particular request's **response stream** will include the 1s pause before the next `yield`. (That's the point — simulating time taken by preprocessing/inference/etc.).
- **No** → It **won't delay other API requests** because FastAPI is async. While one request is "sleeping," others can proceed.

Example timeline with `await asyncio.sleep(1)` inside pipeline:

ini

T=0s → Agent 1 started sent

T=1s → Preprocessing completed sent

T=3s → Inference completed sent

T=4s → Postprocessing completed sent

T=5s → File generated sent

The client **sees steps in real-time**, not only at the end.

That's why `StreamingResponse` + `await` is powerful — the response "drips" step by step.

♦ What if we remove `await`?

If you remove `await asyncio.sleep(1)`, the pipeline will just run instantly:

- The UI will receive **all steps at once**, almost immediately.
 - That's fine if your agents are already doing real CPU/IO work.
 - But during simulation, `await asyncio.sleep()` makes it behave like a real agent (slow, incremental).
-

✓ Conclusion:

Use `await` when simulating I/O-bound or async work (DB calls, API calls, file I/O). It ensures **other requests are not blocked** and the **UI gets progressive streaming updates**.

Do you want me to show you an example where **Agent 1 actually calls an external API (like Wikipedia or Hugging Face)** instead of `asyncio.sleep`, so you see `await` in real-world usage?

add one more agent get data from postgres db, please don't use `asyncio.sleep()` if not required, if required then only use

Perfect 👍 now we'll extend the pipeline with a **fourth agent** that fetches some data from a **Postgres database**.

Key points:

- No artificial `asyncio.sleep()` unless really needed.
 - Use **async database access** (`asyncpg` or `SQLAlchemy AsyncSession`).
 - Stream logs to the UI as soon as DB data is fetched.
-

◆ Updated FastAPI Example with Postgres Agent

Here I'll use `asyncpg` for simplicity. (You can swap with `SQLAlchemy` if you already use it in your project.)

```
python

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

```

app = FastAPI()

# -----
# Database Setup
# -----
DATABASE_URL = "postgresql://postgres:password@localhost:5432/mydb"

async def get_db_connection():
    return await asyncpg.connect(DATABASE_URL)

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

async def agent1_preprocess(text: str):
    """Preprocess: lowercase, remove punctuation, split tokens."""
    processed = re.sub(r"^[^a-zA-Z0-9\s]", "", text).lower().split()
    return processed, f"Preprocessing completed → {len(processed)} tokens"

async def agent2_inference(tokens: list[str]):
    """Inference: simple word frequency count."""
    freq = {}
    for t in tokens:
        freq[t] = freq.get(t, 0) + 1
    return freq, f"Inference completed → {len(freq)} unique words"

async def agent3_postprocess(freq: dict):
    """Postprocess: extract top 5 words."""
    sorted_words = sorted(freq.items(), key=lambda x: x[1], reverse=True)
    top_words = sorted_words[:5]
    return top_words, f"Postprocessing completed → top {len(top_words)} words extracted"

async def agent4_fetch_from_db():
    """Fetch some data from Postgres."""
    conn = await get_db_connection()
    try:
        rows = await conn.fetch("SELECT id, name FROM users LIMIT 3;")
        data = [{"id": r["id"], "name": r["name"]} for r in rows]
        return data, f"Database query completed → fetched {len(data)} rows"
    finally:
        await conn.close()

```

```

async def generate_file(results: list[tuple[str, int]], db_data: list[dict]):
    """Generate final result file."""
    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("Top extracted keywords:\n")
        for word, count in results:
            f.write(f"{word}: {count}\n")

        f.write("\nSample DB rows:\n")
        for row in db_data:
            f.write(f"{row['id']}: {row['name']}\n")

    return file_path

# -----
# Streaming Logic
# -----

async def run_pipeline():
    text_input = "FastAPI is an amazing framework! FastAPI makes APIs super fast, easy, and fun to build."

    # Step 1: Preprocessing
    yield f"data: Agent 1 started (Preprocessing)\n\n"
    tokens, log1 = await agent1_preprocess(text_input)
    yield f"data: {log1}\n\n"

    # Step 2: Inference
    yield f"data: Agent 2 started (Inference)\n\n"
    freq, log2 = await agent2_inference(tokens)
    yield f"data: {log2}\n\n"

    # Step 3: Postprocessing
    yield f"data: Agent 3 started (Postprocessing)\n\n"
    top_words, log3 = await agent3_postprocess(freq)
    yield f"data: {log3}\n\n"

    # Step 4: Database Query

```

```

yield f"data: Agent 4 started (Database Fetch)\n\n"
db_data, log4 = await agent4_fetch_from_db()
yield f"data: {log4}\n\n"

# Step 5: File Generation
yield f"data: Generating final file...\n\n"
file_path = await generate_file(top_words, db_data)

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")

```

◆ Flow Now

1. **Agent 1** → preprocesses text into tokens.
2. **Agent 2** → counts word frequencies.
3. **Agent 3** → extracts top 5 words.
4. **Agent 4** → connects to Postgres and fetches some rows.
5. **Final File** → contains keywords + DB results.

Example file output:

yaml

Top extracted keywords:

```

fastapi: 2
is: 1
an: 1
amazing: 1
framework: 1

```

Sample DB rows:

```

1: Alice

```

2: Bob

3: Charlie

✓ No `asyncio.sleep()` here unless simulating I/O was necessary. The DB query itself is an async I/O operation, so `await` is needed but not `sleep`.

Do you want me to adapt this for **SQLAlchemy async ORM** instead of `asyncpg` (since many FastAPI projects already use SQLAlchemy)?