ChatGPT

System Architecture and Build Plan

We propose a modular MVP architecture with a **Streamlit** frontend and a **FastAPI** backend, integrating OpenAI's GPT-5 Nano model for LLM capabilities, a React Flow–based diagram builder, and PDFKit for report export. Streamlit's new chat elements (`st.chat_input`, `st.chat_message`) will serve as the user interface for conversation, leveraging Streamlit's session state to maintain chat history [1]. The frontend will also embed a React Flow component (via a Streamlit custom component) to allow interactive flow-chart design. The FastAPI backend will expose REST endpoints for chat processing, flow data handling, and PDF generation. This architecture – Streamlit for UI, FastAPI for logic – is recognized as a "powerful combo" that lets developers build AI apps rapidly and efficiently [2].

- **Frontend (Streamlit Chat + Flow UI):** Users interact through a single-page Streamlit app. A chat window (using `st.chat_input` and `st.chat_message`) collects user messages and displays assistant responses. A React Flow–based canvas component is embedded (using, e.g., the streamlit-react-flow wrapper) to let the user drag/drop nodes and edges in a flow diagram [3] [4]. The flow designer component can capture interactions (e.g. node clicks) and the current graph state can be saved as JSON (via React Flow's `toObject()` API) [5]. For example, when the user clicks "save" on the diagram, the app can call `flow_data = react_flow_instance.toObject()` to serialize nodes/edges into JSON and send it to the backend.

- **Backend (FastAPI Service):** The FastAPI server provides asynchronous endpoints for each function. A `/chat` endpoint accepts POST requests with the user's message (and optional context); it calls the OpenAI API and returns the LLM's response. A `/flow` endpoint could accept or return JSON representations of the diagram (e.g. to save or restore flows). A `/generate_pdf` endpoint takes HTML (or data) and runs PDFKit to return a PDF file. Using Python's async support ensures efficient I/O when calling external services (OpenAI). FastAPI can be launched with Uvicorn using multiple worker processes to handle concurrent users if needed [6].

Streamlit Chat Interface

In Streamlit, we will use the built-in chat elements to build the user's conversation UI. For example, each user message is captured by `st.chat_input()` and displayed with `st.chat_message("user")`, and the assistant response with `st.chat_message("assistant")` [1]. Streamlit's session state (e.g. `st.session_state["history"]`) will store the running chat history and any flow state. The Streamlit app code might include an event loop such as:

```
if user_input := st.chat_input("Type your message"):
    st.session_state.history.append({"role": "user", "content": user_input})
    assistant_response =
call_fastapi_chat_endpoint(st.session_state.history)
    st.session_state.history.append({"role": "assistant", "content":
assistant_response})
```

```
    for msg in st.session_state.history:
        st.chat_message(msg["role"]).write(msg["content"])
```

This approach directly connects the Streamlit UI to the backend API. In practice, we can use `requests` or `httpx` in Streamlit to send the user message JSON to FastAPI and await a JSON reply. This keeps the frontend lean (no heavy model in browser) and uses Streamlit's strength in rapidly building UIs [2].

Flow Designer Integration (React Flow)

For the flow-designing component, we integrate React Flow via a Streamlit custom component (e.g. streamlit-flow-component) [3]. React Flow enables interactive node-and-edge diagrams; features like drag-and-drop, customizable nodes, and automatic layouts greatly simplify flow creation [4]. The component runs in the browser as part of the Streamlit app. When the user edits the flow, the JavaScript sends the new graph state to Python. The component captures node/edge events – for example, clicking a node can trigger a Python callback. The entire flow can be saved to a JSON object using React Flow's `toObject()` (or similar) API [5]. For instance, on a "Save Flow" button press, the frontend could execute code equivalent to:

```
const flow = reactFlowInstance.toObject();
fetch("/flow/save", {method: "POST", body: JSON.stringify(flow)});
```

The FastAPI `/flow/save` endpoint would then store this JSON (or process it further). This design lets the user visually design a workflow or logic graph, and seamlessly share it with the backend.

FastAPI Backend Service

The backend will be built with **Python/FastAPI**, which excels at high-performance, async web APIs. We structure the code into routes: e.g. `/chat` for LLM chat, `/flow` for flow data, and `/pdf` for PDF generation. For each request, FastAPI returns JSON (or PDF bytes). The FastAPI app might look like:

```
from fastapi import FastAPI, Request
import openai, pdfkit

app = FastAPI()

@app.post("/chat")
async def chat(request: Request):
    data = await request.json()
    prompt = data["prompt"]
    # Call OpenAI GPT-5 Nano:
    resp = openai.ChatCompletion.create(model="gpt-5-nano",
messages=data["context"])
    answer = resp.choices[0].message.content
    return {"reply": answer}

@app.post("/generate_pdf")
async def gen_pdf(request: Request):
    data = await request.json()
```

```
    html = data["html"]
    pdfkit.from_string(html, 'output.pdf')
    return FileResponse("output.pdf", media_type='application/pdf')
```

This example shows using the OpenAI Python API inside FastAPI endpoints. Since GPT calls are I/O-bound, we use async endpoints (and an async HTTP client) to avoid blocking the server thread. With no user authentication needed (single-user MVP), no user database or login flows are required; the session state is entirely in-memory per Streamlit session.

For scalability, FastAPI can be run with multiple workers (using Uvicorn's `--workers` flag or behind a production server). Each worker can handle requests in parallel [6]. In a larger deployment, we would containerize the app and use Kubernetes or cloud services to scale horizontally [7].

OpenAI GPT-5 Nano Integration

We will use **OpenAI's GPT-5-nano** (the lightweight variant of GPT-5) as the LLM backend for chat and generation tasks. According to OpenAI, GPT-5 is released in three sizes – `gpt-5`, `gpt-5-mini`, and `gpt-5-nano` – allowing trade-offs between performance and latency [8]. GPT-5-nano provides a fast, lower-cost model ideal for an interactive chatbot MVP. In the FastAPI `/chat` endpoint, we pass the conversation (as a list of messages) to `openai.ChatCompletion.create(model="gpt-5-nano", ...)`. We can adjust parameters like `max_tokens`, `temperature`, and even the new `verbosity` / `reasoning_effort` options introduced in GPT-5 if needed. The assistant's response is then returned to Streamlit and displayed in the chat.

Because GPT-5 excels at following instructions and coding tasks [9], it can also assist in interpreting the flow diagrams or generating text based on them. For instance, the backend could encode the flow JSON as a prompt context (or convert it to natural language) to let GPT-5 summarize or explain the designed process.

PDF Generation with PDFKit

To produce a PDF report (e.g. of the chat transcript or the designed flow), we'll use **Python-PDFKit**. PDFKit wraps the `wkhtmltopdf` tool to convert HTML content into PDF [10]. For example, the backend could generate an HTML string that includes the chat log and flow graphics (exported as SVG or image). Then:

```
import pdfkit
html_string = "<html><body><h1>Conversation</h1>...</body></html>"
pdfkit.from_string(html_string, 'report.pdf')
```

This simple call converts the HTML into `report.pdf` [11]. PDFKit supports complex layouts, headers/footers, and multi-page documents [12], which is ideal if the flow or chat is lengthy. The FastAPI `/generate_pdf` endpoint can return the PDF file so the Streamlit UI can offer it for download. Because this uses an external binary (wkhtmltopdf), we ensure it is installed on the server. PDFKit allows passing options like page size, margins, or delays to let dynamic content render correctly [13].

Deployment and Scalability

For a scalable deployment, we will containerize both the Streamlit app and FastAPI service (e.g. with Docker). Each container can be deployed on a cloud provider or Kubernetes cluster. A reverse proxy (like Nginx) can route port 80 to the Streamlit UI (port 8501) and API requests to FastAPI (e.g. port 8000). FastAPI will run under Uvicorn with multiple workers (`uvicorn main:app --workers 4`) so it uses all CPU cores [6]. Alternatively, we can deploy on platforms like AWS ECS/EKS or Heroku which manage scaling automatically.

Because we deliberately omitted user authentication (single-user MVP), horizontal scaling is simpler: any new instance of FastAPI just works statelessly. If we later add multi-user support, we would need to synchronize session state (e.g. storing context in Redis or a database).

In summary, the build process is: 1. **Initialize the project** with Python virtual environments and install dependencies (`streamlit`, `fastapi`, `uvicorn`, `openai`, `pdfkit`, etc.).
2. **Develop the Streamlit UI**, using `st.chat_input` / `st.chat_message` for chat and a React-Flow component for the flow designer. Keep UI logic minimal; route user inputs to backend API calls.
3. **Develop FastAPI endpoints**: `/chat` calls GPT-5-nano, `/flow` handles JSON graph data, `/generate_pdf` calls PDFKit. Use `async` functions and a suitable HTTP client.
4. **Integrate the pieces**: Connect Streamlit to FastAPI via HTTP calls. Test the full loop: user types -> FastAPI/gpt -> response -> Streamlit. Save and restore flows via JSON exchange.
5. **Add PDF output**: Once chat and flow are working, add a button that sends HTML to `/generate_pdf` and returns a PDF for the user.
6. **Containerize and Deploy**: Write Dockerfiles for frontend and backend, define a simple Kubernetes or Docker Compose setup for horizontal scaling. Use Uvicorn workers to handle load [6].

By following this blueprint, we ensure a clear separation of concerns: Streamlit handles interactive UI elements, React Flow handles visual diagrams, FastAPI handles logic and I/O, GPT-5 Nano provides smart responses, and PDFKit handles reports. This setup leverages the strengths of each component and is well-documented by both the Streamlit and FastAPI communities [2] [1].

Sources: Streamlit and FastAPI documentation and community posts [1] [2]; React Flow examples [3] [5]; PDFKit tutorial [11]; OpenAI GPT-5 announcement [8]; FastAPI deployment guide [6] [7].

---

[1] Build a basic LLM chat app - Streamlit Docs
https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/build-conversational-apps

[2] Introduction to Building AI-Powered Apps with Streamlit and FastAPI - DEV Community
https://dev.to/cypriantinasheaarons/introduction-to-building-ai-powered-apps-with-streamlit-and-fastapi-73d

[3] [4] New Component: Streamlit Flow - Beautiful, Interactive and Flexible Flow Diagrams in Streamlit - Custom Components - Streamlit
https://discuss.streamlit.io/t/new-component-streamlit-flow-beautiful-interactive-and-flexible-flow-diagrams-in-streamlit/67505

[5] Save and Restore - React Flow
https://reactflow.dev/examples/interaction/save-and-restore

[6] [7] Deployments Concepts - FastAPI
https://fastapi.tiangolo.com/deployment/concepts/

8 9 Introducing GPT-5 for developers | OpenAI
https://openai.com/index/introducing-gpt-5-for-developers/

10 11 12 13 How to Generate PDFs from HTML with Python-PDFKit (HTML string, HTML File and URL) - APITemplate.io
https://apitemplate.io/blog/how-to-generate-pdfs-from-html-with-python-pdfkit/