ChatGPT

Implementing a GraphFlow-Based Multi-Agent Chatbot

Master Orchestrator Agent with GraphFlow

Structured Orchestration: GraphFlow is designed for scenarios where you need strict control over conversation flow. Unlike free-form group chats, GraphFlow uses a directed graph to determine which agent speaks and when [1]. In practice, this means you can emulate a *top-level orchestrator* by defining one agent (or node) that routes the conversation to different next steps based on content or outcomes. Each node in a GraphFlow graph is an individual agent, and edges define how control passes between them [2]. This makes it feasible to implement a **master orchestrator agent**: for example, a lead agent node can examine the user's request and *conditionally* forward the task to an appropriate sub-agent or sub-team node.

Conditional Routing: GraphFlow supports conditional branches on edges, enabling dynamic routing of conversation. The orchestrator agent can include special tokens or flags in its message (e.g. `"NEXT: resume_parser"`) that trigger specific edges. The framework lets you attach conditions to edges (like `"APPROVE"` vs. `"REVISE"`) so that different agent paths execute based on the orchestrator's output [3]. This pattern is similar to the *Magentic-One* design where a lead agent plans tasks and delegates to others [4]. In GraphFlow, however, the graph structure itself encodes this logic. In summary, a top-level orchestrator is **feasible**: it can be one of the GraphFlow agents that decides the next node, or you can rely on the graph's conditional transitions to automatically route between subflows.

Supporting Features: Microsoft AutoGen provides patterns that inform this approach. In fact, GraphFlow can be seen as a more controlled extension of a SelectorGroupChat (where a model chooses the next speaker) [5]. The key difference is that GraphFlow's orchestrator logic is deterministic and encoded in the graph. You can design the graph such that the orchestrator agent always starts, and then branches out to different teams (e.g. onboarding vs. parsing) via edges. This approach is supported by GraphFlow's ability to handle sequential, parallel, and looping flows in one combined structure [1].

Encapsulating Multi-Agent Subflows in GraphFlow

Graph Nodes vs. Teams: In GraphFlow, each node represents a single conversable agent [2]. There isn't a built-in notion of a "team as a node" – you typically enumerate all participating agents in the graph. However, you can still encapsulate subflows conceptually. One approach is to treat a *sub-team* as a collection of agents connected by their own mini-graph (or using a simpler team pattern), and have the orchestrator interface with that collection via a single gateway agent. For example, you might have an **onboarding subflow** consisting of a UserProxyAgent (for user Q&A) and an AssistantAgent that guides the user. You could designate one agent from that subflow (perhaps the one that produces the final outcome of onboarding) as the representative node in the main GraphFlow. The orchestrator (or preceding node) would pass control to this representative agent, which internally carries out the sub-conversation with its peers.

Using GroupChat Patterns: Another tactic is leveraging group chat teams (like `SelectorGroupChat` or `RoundRobinGroupChat`) inside a graph. While you cannot directly insert a `SelectorGroupChat` object as a GraphFlow node, you can simulate it. For instance, a *GraphFlow node* could be a custom agent that, upon activation, spins up a SelectorGroupChat among a subset of agents to handle a task, then returns a result. This keeps the **modularity** of sub-teams – each sub-team's logic is contained in that custom agent's implementation. Best practices here include keeping the interface between subflow and main flow clean (e.g. the main flow passes a clearly defined task message to the subflow agent, and expects a standardized result message back). This isolation ensures changes within a sub-team don't ripple into the global graph, preserving modularity.

Modularity and Isolation: To maintain encapsulation, define each sub-task's agents and prompts separately. For example, have one mini-workflow for "Resume Parsing" (maybe an agent that extracts text, another that analyzes it) and another for "LinkedIn Profile Consolidation." In the top-level GraphFlow, treat them as black boxes: the orchestrator (or graph logic) hands off to the entry agent of the subflow and waits for a completion token or message. You might use GraphFlow's **sequential chaining** to link these components: e.g. `Onboarding -> ResumeParse -> ProfileConsolidation` connected by edges. Even though GraphFlow doesn't natively nest graphs, this linear (or conditional) chaining achieves a similar effect. Each subflow's internal conversation can be thought of as happening during the node's turn. In practice, careful design of system messages and parsing of outputs is required so that one agent's output cleanly triggers the next sub-team's input.

Incremental Development: Testing and Debugging GraphFlow

Start Small: It's wise to build the chatbot flow incrementally. Begin with a simple GraphFlow containing just a couple of agents in sequence, verify it works, then extend. For example, first implement the onboarding Q&A with a user proxy and assistant, and get that working independently. AutoGen allows you to run any team (including GraphFlow or simpler teams) and observe the conversation turn by turn. Use the `run_stream` method to execute the flow asynchronously and iterate over events (messages) as they occur [6]. Wrapping this in the `Console` utility will pretty-print each agent message with sources, making it easy to follow the dialog in real time [6].

Unit Testing Agents: Each agent's behavior can be unit tested by simulating messages to it. For instance, if you have a resume parser agent, you can call it with a fixed prompt (via its underlying model client's `create` method or by initiating a two-agent chat) to see if it produces the expected output format. Because agents are just Python objects, you can programmatically send them `UserMessage` or other agent messages and verify their response content. Ensuring each agent adheres to a contract (expected input/output) will make integration in GraphFlow smoother.

Testing Sub-graphs: If you have defined subflows (like a mini conversation for profile consolidation), test those in isolation using either a small GraphFlow or a GroupChat. AutoGen's team classes are composable – you could temporarily run a `SelectorGroupChat` with just the sub-team's agents to verify their interaction logic, before embedding that logic into the GraphFlow. This incremental approach ensures that when you wire everything together, each part is already known to work.

Debugging Tools: AutoGen provides robust debugging support. You can enable detailed logging of agent messages and model calls by adjusting the logging level [7]. Setting the `TRACE_LOGGER_NAME` and `EVENT_LOGGER_NAME` to `DEBUG` will output step-by-step traces of message passing and agent decisions. This is extremely helpful for understanding why a certain edge was taken or why a conversation looped. Additionally, if your GraphFlow isn't behaving as expected, you can instrument specific agents to print or log their state (for example, log the content of a message that triggers a condition). Since GraphFlow emits each message as an event, you can also write tests that assert “after running the flow on input X, agent Y should eventually receive message Z”, using the streamed events.

Autogen Console: The `Console` stream viewer is invaluable during development. By running `await Console(flow.run_stream(task="..."))` you get a live view of the conversation with nicely formatted outputs [8]. This helps you manually verify the flow of control – you can see if, say, after the onboarding agent finishes, the next message comes from the resume parser agent as expected. If not, perhaps your graph edges or conditions are misconfigured. In complex flows, it can also be useful to insert *checkpoints* or dummy agents whose sole job is to output a log message (“---Finished onboarding---”) to demarcate stages during testing.

Visualization and Telemetry for GraphFlow

Built-in Tracing: AutoGen 0.4+ includes **OpenTelemetry** integration for tracing agent workflows [9]. By enabling tracing, you can collect a timeline of events and agent actions as the GraphFlow executes. In practice, this means you can hook up an OpenTelemetry backend (like Jaeger or Zipkin) and see a **visual trace** of your multi-agent run – each agent call becomes a span, and you can inspect timings, outputs, and the path taken through the graph. This is extremely helpful for performance tuning and debugging complex flows, as you get a bird’s-eye view of the conversation path and parallelism. The tracing can be enabled with just a few lines (installing the `opentelemetry-sdk` and configuring an exporter) [10] [11]. Once set up, every GraphFlow run will emit structured trace events that you can visualize to ensure the agent path matches expectations.

Event Logging: Aside from full telemetry, standard logging (as mentioned) is another way to visualize runtime behavior. The event log will show each message exchange, including which agent sent it and the content. This textual “play-by-play” can be analyzed or even fed into a simple visualization script if needed (for example, generating a sequence diagram of agent interactions). AutoGen’s logging is built on Python’s logging module, so you can redirect these logs to a file or UI for review [12].

AutoGen Studio: If you prefer a graphical interface, **AutoGen Studio** provides a low-code UI for multi-agent workflows. In its Playground mode, it offers a **visual representation of message flow through a transition graph** [13]. Essentially, you can see the agents as nodes and an arrow showing which agent spoke to which. This can confirm that your GraphFlow’s transitions are occurring in the right order. Studio also allows interactive stepping through conversations, which can be great for debugging (you can pause and inspect state or even inject human messages via a UserProxyAgent). Keep in mind AutoGen Studio (as of v0.4) is primarily a prototyping tool and might not support every GraphFlow feature if very new, but it’s useful for visualization and quick experiments.

Third-Party Tools: Because GraphFlow is a directed graph at heart, you can also export or serialize the graph for external visualization. AutoGen supports serializing components (the `DiGraphBuilder` can output the graph structure). With a little scripting, you could dump this to a DOT file for Graphviz or use libraries like NetworkX to visualize the static structure of your flow. For runtime telemetry, beyond OpenTelemetry, one could integrate custom callbacks on each agent’s output to capture metrics (like tokens used, time taken per node) and plot them. The key is that AutoGen doesn’t lock you out of the process – it’s open enough that you can tap into events and craft any monitoring around it.

Planning the Graph vs. Building Iteratively

Upfront Planning Pros and Cons: Designing the full orchestration graph in advance ensures you consider how all parts will fit together. For a well-understood problem, drawing out the entire flow (all agents and transitions) can save refactoring later. For example, if you know the system must handle onboarding, resume parsing, and profile consolidation in sequence, you might lay out those GraphFlow nodes and edges from the start. Pre-planning helps identify the required interfaces between modules – e.g. the format of data handed from the resume parser to the profile consolidator. If your architecture is complex, a high-level plan (possibly informed by the earlier PDF recommendation) guides development so that each sub-team knows what to expect from others.

However, **premature over-planning** can be risky. Large multi-agent graphs are hard to get perfect on paper, and LLM agent behavior can be unpredictable. If you plan everything at once, you might lock in assumptions that don't hold during real interactions.

Incremental Build Approach: In most cases, it's best to take an **iterative approach**: implement a minimal end-to-end slice of the system, then gradually enhance it. For instance, start with a simple linear flow: User -> Assistant (onboarding) -> Assistant (does a trivial "parsing") -> Assistant (consolidates dummy data). Even if the later steps are placeholders, having a running pipeline from the get-go is valuable. It provides a framework to measure and a skeleton to fill in. As you flesh out resume parsing logic, you can slot it into that pipeline and test immediately, rather than building it in isolation and hoping it integrates.

When to Plan Fully: Upfront planning is more important when **interfaces are rigid or external**. If the resume parsing agent and LinkedIn agent are being developed by different people or must conform to external APIs, you should define those interactions early (e.g. the resume parser will output JSON of parsed fields, which the consolidator will consume). Also, if your GraphFlow will involve complex branching logic (say different user types skip certain steps), mapping that out in a flowchart helps ensure you don't paint yourself into a corner with the graph design (GraphFlow can handle conditional logic, but you need to set up those edge conditions explicitly from the start).

Guiding Principles: A good strategy is to **plan the scaffold, but build the house room by room**. Define the major stages (nodes and general data flow) early – this is analogous to high-level architecture. But implement and test them incrementally, one transition at a time, to verify each piece works and adjust as needed. Expect to iterate: the beauty of GraphFlow is that adding a new node or edge later is straightforward (especially if you used `DiGraphBuilder` to construct the flow). Keep the graph design modular (each agent does one role), which makes it easier to swap in improved agents or add new ones without reworking the whole structure.

Local LLM Constraints (Ollama) and Agent Design

Using Local Models: Relying solely on local LLMs via `OllamaChatCompletionClient` means all your agents use models served by Ollama (e.g. LLaMA variants) instead of cloud APIs. The good news is AutoGen fully supports this – an Ollama client is provided out-of-the-box for local models [14], so GraphFlow and all agent types will work with it. You avoid API costs and your data stays on-premise [15], but there are important implications for system design:

- **Performance and Parallelism:** Local models (especially on CPU) may be slower and can typically handle only one request at a time unless you run multiple instances. If your GraphFlow tries to execute agents in parallel (GraphFlow does support parallel branches), note that a single Ollama server might queue those requests. You may need to configure agents to run sequentially or spin up multiple model servers for true parallelism. Monitor the throughput and latency; it might be beneficial to keep some agents' prompts very concise or use smaller models where possible to reduce load.

- **Context Limitations:** Check the context window of your chosen local model. Many open models have 2K–4K token limits (though newer ones like Llama2 70B may go up to 4K or more). In a multi-agent setting, messages can accumulate. **Plan for shorter prompts or implement memory** (e.g., periodically summarize context or use the AutoGen memory/RAG features [16] [17]) so that the local model isn't overwhelmed with the entire conversation history. If your agents are verbose, consider using filters or truncating older messages in the agent state.

- **Model Compatibility:** Not all open-source models support advanced features like function calling or structured outputs natively. As noted in the AutoGen documentation, some local model APIs or proxies lack certain OpenAI API features (function call support, etc.) [18] . If your chatbot relies on function calling (tools) or wants JSON outputs, ensure the model supports it or adjust your prompts accordingly. AutoGen can enforce a JSON output format via output parsers or by using the `structured_output` flag if the model client supports it (the Ollama integration in AutoGen 0.4.8 does mention support for structured output [19]). You might need to be more prescriptive in prompts for weaker models to get the desired behavior.

- **Quality and Alignment:** Large local models (like 30B, 70B) can be quite capable, but smaller ones (7B, 13B) might struggle with complex reasoning or following nuanced instructions. Design your GraphFlow with this in mind: assign tasks according to model strength. For example, if using a 7B model as a "validator" agent to double-check something, that might be fine (since it's simpler language processing). But for the heavy lifting (e.g., an agent that reads a resume and extracts key insights), you might want a stronger model. AutoGen allows different agents to use different model clients, even in one flow. You could run an *orchestrator agent* on a more powerful local model and a minor utility agent on a lightweight model. In fact, the Magentic-One project experimented with using a more powerful model for the high-level planner while others used a smaller one [20] . This selective allocation can improve efficiency – use the "big brain" model only where needed, and default to smaller or faster models for routine tasks.

- **Prompt Size and Token Management:** With local models, you must be mindful of token usage since everything runs on your hardware. Autogen's `llm_config` for each agent can include parameters like `max_tokens` and temperature. It might be wise to limit `max_tokens` for agents that don't need long answers (like a Yes/No validator) to save time. Also consider enabling caching of LLM calls if you'll be iterating a lot during development – AutoGen 0.4 supports caching model outputs to disk or Redis [21] [22] , which can greatly speed up debugging interactions when using local models.

Tradeoffs: In summary, using Ollama local LLMs gives you privacy and cost benefits, but you trade off some convenience. Plan for the constraints: keep agent prompts focused, possibly incorporate retrieval augmentation if the model's knowledge is limited, and manage the conversation length actively. Ensure to test the entire GraphFlow with the actual local models early on – something that works with GPT-4 via OpenAI might need prompt tweaking to work with a 13B model from Ollama.

Reusable Agent Patterns for Modularity

Designing a complex agent system is easier if you leverage common **agent design patterns** and reuse them as building blocks:

- **Reflection (Critique Loop):** A powerful pattern is to have a *generator* agent produce an output and a *critic* agent review it, potentially in a loop until criteria are met [23]. We saw this in the GraphFlow example where a "reviewer" agent either APPROVES or asks for a REVISION [3]. You can generalize this: e.g. a content generator agent and a policy-checker agent – if the policy agent flags an issue, the content agent revises. Encapsulating this as a subflow (or a reusable graph snippet) makes it easy to bolt on quality control to any stage. The critic could be as simple as checking for completeness of required fields in a resume, or as complex as verifying factual accuracy with a tool.

- **Validator/Gatekeeper Agents:** Similar to a critic, you might have a validator agent whose sole job is to verify something and output a confirmation. For example, after profile consolidation, a validator agent could check "does the consolidated profile have both resume and LinkedIn info merged?" and if not, trigger a fallback. These agents are reusable – you can plug in different validation functions (format checking, content safety filters, etc.) without altering the main logic.

- **Tool-Using Agents:** Often you'll want certain agents to interact with external APIs or perform calculations. AutoGen makes it possible to integrate tools or code execution in agent replies (via `AssistantAgent` writing code or `ToolAgent` classes). A common pattern is a **retriever agent** that, upon need, calls a search tool or database. In a modular GraphFlow, you might include a node that is essentially "SearchAgent," which receives a query and returns results (possibly to a synthesis agent). *Magentic-One* demonstrates this pattern: the orchestrator delegates file reading to a FileSurfer agent and web queries to a WebSurfer agent [4]. By wrapping these functionalities in dedicated agents, you keep your graph nodes specialized and easily testable. If later you don't need web search, you can remove or replace that agent without disrupting others.

- **User Confirmation/Proxy:** A **UserProxyAgent** can be inserted wherever human approval or input is needed [24]. This is a reusable checkpoint pattern – for instance, after assembling the final profile, you might have a "ConfirmationAgent" (maybe just a UserProxy that asks "Does this look correct? (Y/N)"). If yes, the flow proceeds to completion; if no, perhaps it triggers an earlier node to revise. This human-in-the-loop pattern ensures extensibility; you can decide at deployment time to keep it fully automated or allow intervention at key junctures.

- **Stateful Memory Agents:** If certain information needs to persist and be consulted by multiple agents (e.g., the user's name or preferences gathered in onboarding used later in resume parsing), consider a simple memory or context agent. While not an agent in the sense of LLM, AutoGen's state management or a dedicated context object passed around can serve here. Design patterns like *blackboard* or *ledger* (Magentic-One's Task Ledger and Progress Ledger are examples) keep a shared state that agents update and read [25] [26]. In GraphFlow, you might implement this by giving each agent access to a common memory (AutoGen supports attaching memory stores to agents), or by having the orchestrator agent maintain a summary and feed it into prompts.

- **Reusable Subgraphs:** Although GraphFlow doesn't directly allow a node to be a subgraph, you can still reuse subflow logic by defining a function or class to build a snippet of graph for a given purpose. For example, you might create a helper that returns a little GraphFlow for “research a topic” consisting of a Researcher and Analyst agent loop (similar to the literature review example in docs). Your main GraphFlow can then plug in an agent that triggers this helper and collects the result. This pattern requires manually bridging the flows, but it keeps your high-level graph cleaner.

By combining these patterns, you ensure your multi-agent system is **modular and extensible**. Need to add a new tool? Just introduce a new tool agent and perhaps a branch in the graph where appropriate. Want to improve output quality? Slot in a critic/validator loop before finalizing. Each pattern (generative pairs, validators, tool-wrappers, human-in-loop) can be developed and tested independently, then composed in GraphFlow for the overall solution. This way, Farzan Bhai can confidently grow the chatbot’s capabilities piece by piece, knowing that each component follows a known pattern and the GraphFlow orchestrates them in a structured way.

Sources:

1. AutoGen GraphFlow usage and conditional flows [1] [3]
2. AutoGen GraphFlow design (nodes as agents, directed edges) [2] [5]
3. AutoGen debug and testing (Console, logging) [6] [7]
4. AutoGen tracing and visualization tools [9] [13]
5. Model client support and Ollama integration [14] [18] [20]
6. Agent patterns and design examples [23] [4]

---

[1] [2] [3] [6] [16] [17] GraphFlow (Workflows) — AutoGen
https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/graph-flow.html

[4] [20] [25] [26] Magentic-One — AutoGen
https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/magentic-one.html

[5] [8] AutoGen v0.5.6 released : r/AutoGenAI
https://www.reddit.com/r/AutoGenAI/comments/1kdu20x/autogen_v056_released/

[7] [12] Logging — AutoGen
https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/logging.html

[9] [10] [11] Tracing and Observability — AutoGen
https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/tracing.html

[13] AutoGen Studio — AutoGen
https://microsoft.github.io/autogen/dev//user-guide/autogenstudio-user-guide/index.html

[14] Model Clients — AutoGen
https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/components/model-clients.html

[15] [19] AutoGen + Ollama: Using OllamaChatCompletionClient
https://www.gettingstarted.ai/run-autogen-agents-with-ollama-locally/

[18] LiteLLM with Ollama | AutoGen 0.2
https://microsoft.github.io/autogen/0.2/docs/topics/non-openai-models/local-litellm-ollama/

[21] [22] Migration Guide for v0.2 to v0.4 — AutoGen
https://microsoft.github.io/autogen/stable//user-guide/agentchat-user-guide/migration-guide.html

[23] Reflection — AutoGen
https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/design-patterns/reflection.html

[24] Multi-agent Conversation Framework | AutoGen 0.2
https://microsoft.github.io/autogen/0.2/docs/Use-Cases/agent_chat/