Vectorial AI Take-Home Assignment: Full-Stack Multi-Agent Conversational AI

Background & Objective

Vectorial AI is an innovative company at the forefront of AI-driven product development. We build intelligent systems that can autonomously collaborate and assist across different roles and functions. In this take-home assignment, you will demonstrate your ability to integrate AI end-to-end – from backend logic to user interface – by building a multi-agent conversational application using LangGraph. LangGraph is a framework for creating stateful LLM applications and orchestrating multi-agent systems (think of it as building a graph of AI agents that can work together).

Objective: Design and implement an interactive agent system that simulates multiple persona-driven AI agents (nodes) with distinct knowledge boundaries. The user should be able to chat with these different personas to collaboratively solve a task or carry out a workflow. The goal is to showcase how separate AI agents, each with specialized knowledge or role, can be composed into a larger solution – reflecting a realistic enterprise scenario where different departments or experts must coordinate. We encourage you to be creative in your approach, architecture, and interface. Explain your design reasoning – specifically how you decided to compose the agents and how you partitioned the data for each agent's knowledge domain. This assignment is designed to be completable in roughly 3 days of focused work (around 10–15 hours).

1 Dataset & Resources

To ground your agents in realistic conversations and knowledge, we will use an open-source dataset that simulates multi-entity interactions. We suggest the **Cornell Movie-Dialogs Corpus** (or a similarly structured dataset) as a foundation. The Cornell corpus contains a large collection of fictional conversations extracted from movie scripts – 220,579 conversational exchanges between over 9,000 characters from 617 movies. Each conversation in the dataset involves dialogue between characters, along with metadata (such as which movie and character names). Why this dataset? It provides a rich variety of dialogues and multiple personas (movie characters) engaging in conversations. While it's not an "enterprise" dataset per se, you can treat different subsets of the data as analogues for different character archetypes or personality types. For example, dialogues from witty, humorous characters might ground your **Comedic Relief Friend** persona, while conversations from wise, experienced characters inform your **Wise Mentor** persona. Cautious, analytical dialogue could form the knowledge base for a **Skeptical Realist**, and optimistic, hopeful exchanges could feed an **Optimistic Dreamer**. This allows you to simulate a diverse group of character personalities with distinct conversational styles:

Example: You might designate one persona as a **"Comedic Relief Friend"** using dialogue lines from funny, supportive sidekick characters across various movies, another as a **"Wise Mentor"** using conversations from experienced, guiding characters, a **"Skeptical Realist"** drawing from cautious, analytical characters, and an **"Optimistic Dreamer"** built on hopeful, idealistic character dialogue. The user could then interact with each to get different perspectives on a challenge—each agent contributing insights from their unique character archetype.

Feel free to be creative in how you partition or select the data for each persona. You may use the entire Cornell corpus or a relevant portion (for faster prototyping, using a subset by character types or personality traits might be wise). You can download the Cornell Movie-Dialogs Corpus from its official site or obtain it via Kaggle. Provide a brief description of your chosen subset and processing steps in your README.

You are expected to use the LangGraph library (part of the LangChain ecosystem) for orchestrating agents. Each agent can have its own prompt/LLM and will only access its designated knowledge. Refer to the LangGraph documentation for guidance on defining nodes and managing stateful interactions. If you encounter difficulties, you may use an alternative orchestration method—just document your choice clearly. (That said, we strongly encourage exploring LangGraph for this assignment.)

2 Project Requirements

Your task is to build an interactive multi-agent conversational system that meets the following requirements:

1. **Multiple Persona Agents:** Define at least 3 distinct persona agents (nodes) in your system. Each agent should have a well-defined boundary of knowledge/data:
    - **Data Partitioning:** Partition the dataset so that each persona agent has access to a different subset. This could be based on character archetypes, personality traits, dialogue types, or any logical split that reflects different character personalities. The key is that no single agent has the entire dataset; each has a partial view representing its "character type" or personality.
    - **Persona Definition:** Give each agent a clear character archetype or personality (e.g., **Comedic Relief Friend**, **Wise Mentor**, **Skeptical Realist**, **Optimistic Dreamer**). Craft an initial prompt for each agent that establishes its personality, conversational style, and tone. The agents should have distinct voices or styles reflecting their character archetypes (you can achieve this via prompt engineering or system messages).
2. **LLM Backbone:** Use a Large Language Model of your choice (OpenAI GPT-4/GPT-3.5, an open-source model like Llama 2, etc.) as the backbone for generating agent responses. You do not need to train a model from scratch – leveraging a pre-trained model with prompting and the provided data for context is sufficient. Each agent could use the same underlying LLM but with different prompts and context (their data) to differentiate knowledge and behavior.
3. **Interactive Chat Interface:** Implement a way for a user to chat with the agents and solve a task or workflow by leveraging their different perspectives:
    - The simplest approach is a text-based chat interface (could be a command-line prompt, a web UI, or a notebook interface) where the user can direct questions or commands to specific personas, and receive answers. Full-stack integration means we'd love to see a front-end component — for example, a minimal web app (using Flask/FastAPI + a simple HTML/JS or React, or a Streamlit/Gradio app) that allows the user to choose

or switch between personas in a chat. This isn't strictly required if time is tight, but a usable interface will make your solution stand out.

- The user should be able to engage in a multi-turn conversation. Ideally, the agents maintain context of the conversation (at least within their own session). You can use LangGraph's state management to preserve dialogue context in each agent node. For example, if the user asks the "Comedic Relief Friend" something and then follows up, that agent should remember the prior query (this could be handled via the graph state or agent memory).

4. **Task/Workflow Simulation:** Design a scenario that requires the user to interact with multiple character personas to achieve a goal. For instance, a user might:

   - Ask the **Comedic Relief Friend** to lighten the mood and provide a fun perspective on a problem.
   - Consult the **Wise Mentor** for thoughtful guidance and experience-based advice.
   - Get reality checks from the **Skeptical Realist** about potential pitfalls or concerns.
   - Seek encouragement and positive outlook from the **Optimistic Dreamer**.

   Document this scenario in your README: explain what the simulated task or workflow is and which character persona is supposed to help with which part.

5. **Agent Collaboration & Boundaries:** Ensure that each agent only uses its own data when answering. Part of the challenge is how the user (or the system) knows which agent to consult for which question:

   - You might let the user decide which persona to address at each query (e.g., tagging the message with @Comedic, @Mentor, @Skeptical). This is simplest and keeps agents separate.
   - For a more advanced approach (optional), you could have a coordinator agent or logic that routes the question to the appropriate agent(s) automatically. For example, a top-level agent could analyze the query and delegate to the right persona node in LangGraph.
   - If the workflow requires agents to share information (e.g., the **Skeptical Realist** needs a detail that only the **Wise Mentor** knows), think about how to implement that. Perhaps the user explicitly conveys info from one to another, or your system passes the state between agents behind the scenes. Maintain clear boundaries though: an agent should not suddenly access another's data except via an interaction (state passed through the graph or user message). Part of the evaluation will be how well you enforce the persona boundaries (e.g., an agent should respond with "I don't know" or suggest contacting the other agent if asked something outside its character knowledge).

6. **LangGraph Orchestration:** Use LangGraph to configure the above multi-agent workflow. Define each persona as a node in the LangGraph. You will configure the edges (interaction flow) as needed:

   - Leverage LangGraph features such as state management (to preserve conversation context) and potentially subgraphs or human-in-the-loop pauses. For example, LangGraph supports waiting for user input as part of the graph; you can use this to allow the user to decide the next step in a workflow.

- The graph might be relatively simple (e.g., if the user manually interacts with each agent, the graph could just be a series of independent nodes each triggered by user input). Or it can be more complex (e.g., a coordinator node that calls different persona subgraph nodes based on input). Design what makes sense for your scenario.
- Note: If you have difficulty with LangGraph due to its newness, you may implement the logic using plain code or another framework (like LangChain agents without LangGraph, or a custom dispatcher). Just document why and how you did so. The primary goal is demonstrating multi-agent orchestration, not LangGraph itself – but using LangGraph will show you can adopt our tooling.

7. **Testing & Robustness:** We value reliable, well-tested code:

   - **Unit Tests:** Write unit tests for your core logic (e.g., if you have functions for routing queries, formatting prompts, or processing dataset context). You don't need to test the LLM's internal behavior, but you should test any deterministic helper functions or data processing.
   - **Integration Test / Demo:** Provide a way to run a basic scripted conversation (possibly in a notebook or script) that demonstrates a user interacting with the system to complete the chosen task. This can serve as both a proof that your system works end-to-end and as documentation for the grader. For example, in your README or separate demo script, you might show: User asks Agent A a question $\rightarrow$ Agent A responds $\rightarrow$ User asks Agent B another question $\rightarrow$ Agent B responds combining info $\rightarrow$ final outcome achieved. This can be a text log or, if you built a UI, screenshots. Ideally, also include instructions to run an interactive session (like ”run python app.py and go to localhost:5000” or ”open the notebook and follow the prompts”).
   - **Error Handling:** Anticipate and handle at least basic errors or edge cases. For instance, if the user asks an irrelevant question or addresses the wrong agent, how does your system handle it? It's fine if agents politely refuse or redirect the user. The system shouldn't crash or hang on bad input. Using LangGraph's control flow, ensure that each step in the conversation either produces a result or a graceful error message.

3 Deliverables

By the end of the assignment, you should deliver a GitHub repository (or equivalent) containing:

1. **Codebase:** All source code for your solution, including:

   - The implementation of the multi-agent system (LangGraph definition, agent prompts, etc.).
   - Any supporting scripts or modules (data preprocessing, prompt templates, etc.).
   - Front-end or UI code, if applicable (not mandatory, but encouraged).

2. **Dataset (subset) & Preprocessing:** Include the dataset (or a link/instructions to obtain it) and any code you used to preprocess or load it. If the dataset is large, you can instruct how to download it rather than including it in the repo. Ensure that any filtering or partitioning logic for creating persona-specific knowledge bases is reproducible (e.g., a script that reads the raw data and outputs files for each agent).

3. **README Documentation:** A comprehensive README that includes:
    - **Project Overview:** A summary of the problem you tackled and your solution approach. Describe the personas you created and the scenario/workflow you chose to simulate. This is your chance to explain your design decisions and reasoning – why you structured the agents and data in that particular way, and how it aligns with solving the task.
    - **Setup & Running Instructions:** Clear steps to set up the environment and run the project. Include how to install dependencies, how to launch the chat interface or run the demonstration script, and how to run tests.
    - **Usage Guide:** If applicable, explain how a user can interact with the system (e.g., "You can type @Comedic: Make me laugh about this situation!" to ask the Comedic Relief Friend). Include an example session transcript or expected behavior for the provided scenario.
    - **Notes:** Any assumptions you made, any limitations, and areas for future improvement. If you didn't use LangGraph (or used a modified approach), explain that here.
4. **Tests:** Include your unit tests and any integration test scripts. Provide instructions on how to run the test suite (e.g., pytest commands). Make sure the tests are easy to run and at least some of them pass – they will be looked at to gauge code quality and diligence.
5. **Optional Demo Artifacts:** If you built a front-end, you can include a link to a live demo (if you hosted it) or a short video/gif in the README. This is optional but can showcase your work nicely. Similarly, if using a notebook, ensure it's clean and contains outputs demonstrating the conversation flow.

Ensure the repository is organized and professional. Clean, well-commented code and a polished README are part of the evaluation.

4 Optional Extensions and Creativity

We encourage creativity! While the core requirements should be met first, consider these optional extensions if you have time or want to showcase extra skills (these can earn bonus points but are not required):

- **Additional Personas or Complexity:** Add more than three agents, or create a more complex interaction pattern (e.g., agents that can also talk to each other automatically, or a hierarchy of agents where one supervises or summarizes others).
- **Memory and Learning:** Implement more sophisticated memory for agents (beyond the basic LangGraph state). For example, an agent could build up a summary of the conversation so far and use it for long-context handling, or you could use a vector database to allow an agent to query its knowledge base (the dataset) by semantic search (this would be a chance to show skills with embeddings and vector stores – quite fitting for Vectorial).
- **Tool Use:** Integrate tools or APIs. Perhaps one agent could have a tool to fetch information (web search, calculator, etc.) and the LangGraph workflow includes calling that tool. This can demonstrate agent tool-use abilities.

- **Improved UI/UX:** A slick web interface where the user can seamlessly talk to multiple agents (for example, agents are represented as chat rooms or characters the user can message) would be impressive. Creativity in visualization (distinct avatar or icon for each persona, etc.) is welcome.
- **Domain-Specific Simulation:** Frame the entire assignment in a creative narrative. For instance, simulate a fictional scenario where you need advice from different character types (like planning a heist, organizing an event, or solving a mystery). This isn't necessary, but a cohesive story can make the demo more engaging.
- **Performance Considerations:** If using a large dataset, consider and explain how you might scale the solution. Did you index the data for fast retrieval per agent? Did you face any latency issues with the model and how did you mitigate them (caching, etc.)? We don't expect a full production-ready system, but awareness of performance is a plus.

Document any extensions clearly in your README so we don't miss them during evaluation. Remember, quality over quantity – it's better to do the core requirements well than to attempt too many extensions with poor results. Only pursue extensions if you have the core working first.

5 Evaluation Criteria

Your submission will be evaluated holistically on several criteria. We aim to understand not just what you built, but how you think as an engineer. Here's what we look for:

- **Functional Completeness:** Does your multi-agent system meet the core requirements? We will run your application or demo scenario to see if a user can indeed interact with multiple personas and accomplish the intended task. Partial solutions that don't quite run or fulfill the scenario will be noted, but try to deliver an end-to-end working experience.
- **Correctness & Robustness:** When we test various inputs, do the agents respond in a reasonable way consistent with their persona and data? We will check that each agent stays within its knowledge boundary (e.g., the Comedic Relief Friend shouldn't suddenly become serious and analytical if it wasn't given that type of dialogue). We'll also see how the system handles edge cases or unsupported queries (graceful handling is enough – e.g., an agent admitting it doesn't have that info is fine).
- **Architecture & Code Quality:** We will review your code structure. Is it logical, modular, and extensible? Using LangGraph effectively (nodes, edges, state) to structure the workflow is a big plus. Even outside LangGraph, clear separation of concerns (data loading, agent logic, interface) is expected. The code should be readable with meaningful variable/function names and comments where needed.
- **Use of LangGraph and AI Techniques:** Since this assignment emphasizes LangGraph and full-stack integration, we will look at how you used the framework. Have you taken advantage of its features like statefulness, subgraphs, or others? Did you incorporate an LLM intelligently (prompt design for personas, context injection from the dataset, etc.)? Innovative prompt engineering or clever use of the dataset (such as retrieval augmentation) will be appreciated.

- **Documentation & Reasoning:** A key part of this assignment is your reasoning. We will read your README and any comments to understand why you designed the solution the way you did. We want to see that you can justify technical choices (e.g., why X number of agents, why you partitioned data in a certain way, why you chose a particular model or interface). Clarity and thoroughness of documentation matter. If something isn't working perfectly, an honest explanation of limitations or how you'd fix it goes a long way.

- **Testing & Reliability:** We will look at your tests. Do they cover important aspects of the code? Do they pass? A strong candidate will at least have sanity-check tests for critical components. Also, if we follow your setup instructions, does everything run as expected without a lot of fuss? Providing a Dockerfile or environment file, for example, can help ensure we can run your app in a straightforward way. While not strictly required, such touches show professionalism.

- **Creativity and Ambition:** This is the "wow" factor. Did you just do the minimum, or did you bring a creative spark to the assignment? This could be in the user experience (a neat UI or a compelling scenario), in the technical implementation (like an agent that uses an external knowledge tool, or an inventive way of agent collaboration), or in how you present the project. We don't expect every candidate to implement all the optional extensions, but we will definitely give extra credit for those who go above and beyond in a meaningful way.

Each of the above areas will be considered in our overall impression. We typically grade on a rubric that balances engineering skills (code, architecture, correctness) and AI skills (problem understanding, use of models, creativity) roughly equally. A submission that is very strong in one area can offset some weakness in another (for example, an amazingly creative solution that has a few bugs might still be rated highly, as would a perfectly engineered solution that maybe didn't try many fancy extras). Make sure to at least hit the core functionality and demonstrate good coding practices, and then let your unique strengths shine.

6 Getting Started

To help you get started quickly:

1. **Setup LangGraph:** Refer to the LangChain documentation for how to install and use LangGraph. You might install via pip (if available) or use the LangChain SDK that includes it. Ensure you have API access or local models ready for your LLM calls.

2. **Acquire the Dataset:** Download the Cornell Movie-Dialogs Corpus (e.g., from the official site or Kaggle). Review its content and think about how to split it. You might write a small script to split dialogues by character types or personality traits. Pay attention to the data format (there are typically files like movie_lines.txt and movie_conversations.txt in that corpus).

3. **Plan Personas:** Sketch out your character archetypes and what data each will use. Decide on a scenario that will require input from multiple personality types to solve. Keep the scenario simple enough to implement in limited time, but engaging enough to showcase multi-agent cooperation.

4. **Prototype Iteratively:** It may help to first get a single-agent chatbot working with LangGraph on one subset of data, then expand to multiple agents. Test how you'll inject the

persona's knowledge (perhaps as part of the prompt or via retrieval). Then add the interaction logic between agents or user-agent switching.

5. **Stay Organized & Time-Bound:** Given the ∼3-day expectation, break your time into setting up the basics (Day 1), implementing core multi-agent logic (Day 2), and testing/refining and documentation (Day 3). Don't spend too long on one fancy feature at the expense of overall completeness. We value a well-rounded submission.

We're excited to see your solution! This assignment is an opportunity to showcase how you think, create, and code in the realm of modern AI. Good luck, and have fun – a bit of personal flair in your project is welcome. Vectorial AI prides itself on technical rigor and innovation, so we'll be looking for those qualities in your work. We hope this project lets you demonstrate why you'd be a great addition to our team.