SocraticGraph: A Multi-Agent Framework for Self-learning and Critique

Manuel Osorio
March 1, 2026

This paper presents **SocraticGraph**, a multi-agent orchestration framework designed to facilitate self-learning through a stateful, directed agent graph. Conventional Large Language Model (LLM) prompting often suffers from "direct-answer bias," which hinders user learning rather than improving it. SocraticGraph addresses this by implementing a state-managed ensemble of specialized agents—including an *Arbiter* for intent routing, *Elenchus* for logical cross-examination, *Aporia* for paradox generation, and *Maieutics* for asking the user leading questions on the subject—all coordinated via the **LangGraph** library. By utilizing the **LangChain** integration for the **Ollama** local backend and a custom **Dialectic** auditor for real-time mastery evaluation, the framework provides a quantifiable approach to Socratic dialogue, ensuring learners reach a mastery threshold of ≥ 0.9 before concluding an instructional topic, all without worries of privacy issues.

1 Program Architecture

SocraticGraph uses modern Python libraries for multi-agent agentic workflows, using LangGraph and LangChain as the foundation to build its 5-agent graph. From a high level of abstraction, the user inputs a message into the command line, it gets passed to model, who then passes the users message to the best suited model out of three to respond. The users response to the second model is then judged by a third model to decide if the user is learning or not.

SocraticGraph follows the standard principles of Object Oriented Programming, separating parts of the workflow to different classes, which are housed in different files. The project

consists of several files as of the writing of this document and a command line interface to run the file:

- `agent_graph.py`
- `agent_state.py`
- `agents.py`
- `history.py`
- `main.py`

While each file holds key information to the success of the workflow, the most pertinent parts of the codebase are identified and explored in-depth in the following sections.

1.1 State Management

The smallest file of the bunch, `agent_state.py`, is the primary way that information from an active loop of the agent graph gets stored for future use. While the class `SocraticState` is used to store data in a key-value arrangement similar to the default Python dict object, the biggest difference is in the properties of the 'messages' key, where values are added due to the `operator.add()` included at the end of the key. This adds new information to the dict instead of following the default python behavior of overriding information. This specialized dict holds all information required to hold the state of the agents and can be passed on to the LangGraph graph class as the schema to hold the information in, which itself is then passed to and from each node in the graph.

1.2 Agent Orchestration

The graph itself contains several nodes, which use a judge agent (named *Arbiter*) to decide which one of the other three nodes should serve the user's input. This input is then evaluated by an auditor agent (named *Dialectic* after that part of the Socratic process) and assigned a score. For a sufficiently high enough score ≥ 0.9, *Dialectic* concludes the conversation.

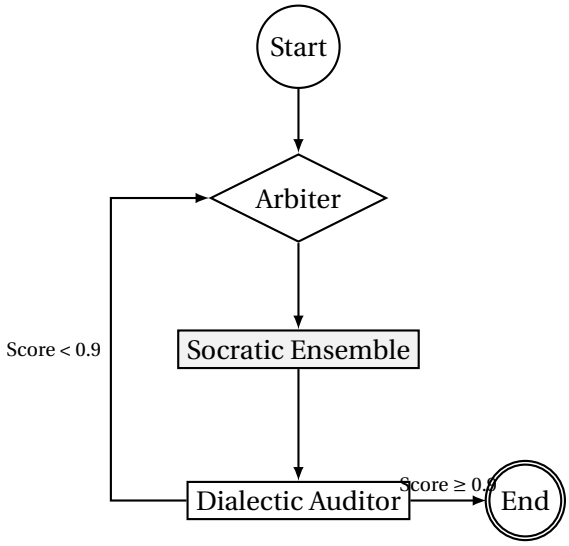

Figure 1.1: SocraticGraph State Transition Diagram

While Figure 1.1 shows the general structure of the graph where *Arbiter* has to pick another model, the way in which it does is one of the most unique parts of this multi-agentic framework. Instead of simply having one "teacher" agent, there are three agents specialized to cover different parts of the Socratic method.

1.2.1 System Prompts

The system prompts for each model were crafted around having an identity and a specific goal in the process of the Socratic method (hence their designations). While a simple prompt could have been chosen such as "question the user" for a single agent with the purpose of questioning, the diversity of each system prompt allows for specific, unique approaches that are tailored to the Socratic method.

1.2.2 Model Choice

Other than having tailored prompts, having multiple agents gives the opportunity for picking specialized Large Language Models (LLM) for each purpose. When running the command line interface with `context_switch` set to `False`, all models run on Llama 3.1 with 8 billion parameters with `q2_K` quantization. While this is sufficient for a usable experience, context switching on purely logical-based agents like *Arbiter* or *Dialectic* to a Large Language Model like Phi-4 Mini with 3.8 billion parameters and `q4_K_M` quantization is better because of the saved memory footprint, high inference speed and strong reasoning. Like *Arbiter* and *Dialectic*, *Elenchus* also benefits from context switching; Mistral is great at following instructions and its `q3_K_S` quantization still allows for deep understanding and the "cross-examination" that *Elenchus* is tasked with doing.

Table 1.1: Agent Model Choice and Quantization Strategy

Node Role	Model Architecture	Quantization	Engineering Justification
Arbiter	Phi-4 Mini (3.8B)	`q4_K_M`	High reliability for deterministic routing.
Elenchus	Mistral (7B)	`q3_K_S`	Balanced for instruction following/debate.
Aporia	Llama 3.1 (8B)	`q2_K`	Optimized for creative breadth over logic.
Maieutics	Llama 3.1 (8B)	`q2_K`	Leverages high parameter count for analogies.
Dialectic	Phi-4 Mini (3.8B)	`q4_K_M`	Strong reasoning for objective grading.

1.3 Quantitative Mastery Evaluation

To bridge the gap between qualitative LLM dialogue and deterministic program flow, SocraticGraph implements a regex-based extraction layer within the `Dialectic` node. The auditor is prompted (via its system prompt) to return a single decimal mastery score (for example, '0.85') with no preamble or punctuation. The system then extracts that numeric value using a simple regular expression and uses it to drive conditional edge traversal in the agent graph. This keeps the evaluator's output constrained and the graph's termination logic deterministic: the run exits when the mastery score meets or exceeds the configured threshold.

1.4 Hardware Optimizations

Closed source and high parameter, cutting edge models are meant to be run on expensive hardware and for a project targeting students, there is no reasonable expectation to have a thousand dollar Graphics Processing Unit for Large Language Models. The optimizations in the quantization and parameter sizes of the model (mentioned in Table 1.1) is a strong start for optimization, but the functions contained in `history.py` help to optimize for consumer hardware outside of the Ollama backend. Predefined in `history.py` but editable through the command line interface, a `context_token_budget` variable is defined. This variable bounds the total budget for context tokens and when the context tokens start to grow beyond the budget, they are removed in a **first-in-first-out (FIFO)** manner. While this number can be changed, the default 4096 value is enough to capture most of the history while staying within the constraints of 8GB VRAM on modern consumer hardware.

2 Conclusion

SocraticGraph demonstrates the viability of executing complex, multi-agent pedagogical workflows on consumer-grade hardware. By implementing a stateful, directed agent graph and multiple models with specialized tasks, the framework successfully mitigates the "direct-answer bias" inherent in monolithic LLM architectures.

The integration of a deterministic evaluation layer ensures that instructional progress is measured quantitatively, while the use of local inference via Ollama preserves user data sovereignty. As AI continues to shift toward edge-computing, architectures like SocraticGraph provide a blueprint for private, specialized, and hardware-efficient learning tools.

REFERENCES

[1] LangChain (2023). *LangChain*. LangChain Inc. https://github.com/langchain-ai/langchain

[2] LangChain (2024). *LangGraph*. LangChain Inc. https://github.com/langchain-ai/langgraph

[3] Meta AI. (2024). *The Llama 3 Herd of Models*. arXiv preprint arXiv:2407.21783.

[4] Microsoft Research. (2024). *Phi-4 Technical Report*.

[5] Mistral AI Research Team. (2023). *Mistral 7B*. arXiv preprint arXiv:2310.06825.

[6] Ollama Team. (2023). *Ollama*. https://ollama.com