Demonstrating Epistemic and Structural Self-Awareness in a LangGraph-Based Conversational Agent

Ryan Mukai[1*] Alfonso Herrera[2†]

April 2025

Abstract

We present a prototype conversational agent ("Bob") built atop LangGraph that combines three complementary self-awareness capabilities: (1) *Epistemic self-awareness*: the ability to monitor and reflect on one's own knowledge state (detecting when inferences are underdetermined); (2) *Self-aware memory management*: a hybrid memory architecture that includes an agent-controlled vector database memory that an agent explicitly controls; and (3) *Structural self-awareness*: particularly *code-based structural self-awareness* via introspecting its own source and exception traces.

We describe Bob's LangGraph workflow and illustrate how each component is implemented. Our contributions include (a) an engineering design that unifies short-term and long-term memory while preserving temporal context, (b) agent controlled vector database memory, (c) code-introspection mechanisms that allow the agent to detect and explain runtime exceptions, and (d) a working LangGraph prototype (code available online). We conclude with directions for quantitative evaluation and future work on identity self-awareness.

Keywords: conversational agents; LangGraph; self-awareness; memory management; temporal grounding; code introspection

1 Introduction

Language models have recently demonstrated impressive reasoning capabilities, and these capabilities enable agents to achieve knowledge of their own knowledge state, the ability to control their own memory, and the ability to comprehend their own code. This paper explores limited self-awareness in a conversational agent ("Bob") that is powered by OpenAI's GPT-4o model and orchestrated via LangGraph.

We focus on three complementary forms of self-awareness:

- **Epistemic self-awareness:** the agent's ability to recognize when it has insufficient premises (e.g., logical inferences requiring extra assumptions).
- **Self-aware memory management:** a multiple-layer memory that combines (i) a short-term in-RAM buffer with recursive summarization and (ii) a persistent vector store ("mandatory memory") that retrieves semantically relevant chunks each turn and (iii) a vector database that the agent explicitly controls. It is the third component that gives an agent a long-term memory

*Work carried out in a private capacity; email: `rmukai@example.com`
†Work carried out in a private capacity; email: `aherrera@example.com`

under its own explicit control that can be operated in a self-aware manner and (iv) the ability to edit its own system prompt, either by replacement or, preferrably, by appending to it. This self-aware memory management is heavily inspired by MemGPT and Letta [1] although it is a very different and far simpler implementation that does not claim to be the same.

- **Structural self-awareness:** Code-based structural self-awareness: the agent's ability to introspect its own source code and diagnose runtime exceptions (e.g., Python stack traces).

Although prior work has explored static LSTM-based self-awareness [2], our contribution is a *dynamic*, LLM-driven prototype with hybrid memory, explicit timestamping, and code introspection.

1.1 Contributions

1. We present an end-to-end LangGraph workflow that integrates short-term and long-term memory with recursive summarization.
2. We develop prompt-engineering patterns that encode UTC timestamps, enabling the model to reason about event chronology.
3. We implement a code-introspection capability that allows the agent to detect, diagnose, and explain runtime exceptions and to inspect its own source file.
4. We release a working Python prototype (available at `https://github.com/rmukai/langgraph-agent`), demonstrating coherent multi-session dialogues without unbounded context growth.

2 Related Work

2.1 Self-Awareness in Neural Agents

Static LSTM-based approaches (e.g., [2]) presented early proofs of concept for an agent that maintained a small symbolic *knowledge state* and detected "unknown" queries. However, those systems lacked dynamic memory components, timestamping, and code introspection. Our work leverages modern LLMs (GPT-4/O) [3] and LangGraph to maintain hybrid memory states, timestamped context, and structural code self-awareness.

2.2 Memory Architectures for Open-Domain Dialogue

MemGPT and Letta inspired idea presented here [1] with their implementation of self-aware memory management. Retrieval-augmented generation (RAG) approaches such as [4] store large corpora in vector databases and retrieve top-k passages each turn. However, most RAG systems do not perform *recursive summarization* to prune older context. CLIE [5] introduced summary-augmented buffers in multi-turn chat; our work refines it by adding explicit timestamping and separating *voluntary* vs. *system* memory channels. This is used not only for conversational purposes but also for code introspection, allowing the agent to read its own codebase sequentially.

2.3 Temporal Reasoning in Language Models

Prompting LLMs to interpret dates has shown that explicit timestamp tokens can help reduce hallucinations about "when" events occurred [6]. We build on these insights by injecting ISO-8601 timestamps into every human turn and summary, enabling the model to filter out "stale" information.

2.4 Code Introspection and Agent Structure

Recent work on *corrigibility* and self-modification (e.g., [7]) proposes frameworks for self-modifying agents under formal logic constraints but does not address how an agent can continually inspect and reason about its own source code at runtime. Our prototype implements a lightweight code-introspection capability, allowing the agent to read its own Python file, locate lines of code, and diagnose exceptions (e.g., 'ZeroDivisionError'), thus adding a structural dimension to self-awareness. An advanced implementation of Godel agents [8] is a good example of a self-modifying agent that is able to introspect its own code and reason about its own structure.

3 Agent Architecture

Figure 1 illustrates Bob's end-to-end LangGraph workflow.

- We always proceed to the conversation node, where the user's input along with relevant context from mandatory vector memory and the recursively summarized conversation are always presented to the LLM. This enables the LLM to respond with knowledge of conversational context and previous conversation history. While the recursively summarized conversation memory is volatile, all conversation turns are stored in non-volatile mandatory vector memory, which gives the agent the ability to remember old conversations. UTC timestamps of past and present input give the agent temporal context.

- Based on the current situation, the LLM has choices to make. It can respond and proceed to END; it can respomd and perform recursive summarization and proceed to END; or it can invoke a tool and return to the conversation node.

- The recursive summarization note summarizes the older parts of the conversation in volatile memory and replaces them with a summary. For simplicity of implementation, it also removes ToolMessage entries from the conversation summary message buffer as well. This is an oversimplification but is meant to keep the code as simple as possible. Once the oldest messages are replaced by a summary, it proceeds to end the current turn.

- The tool node runs the tool specified by the agent. It then returns control to the conversation node.

- The current turn ends when we hit the END node. The next turn will re-run the graph all over again.

- Each invocation of the conversation node stores the full input context and the LLM's response in the vector memory, and things are stored in five turn blocks so that retrieved memories have context. This technique does suffer from context bloat, but it does insure relevant context is maintained. This is an issue that needs to be refined in future implementations.

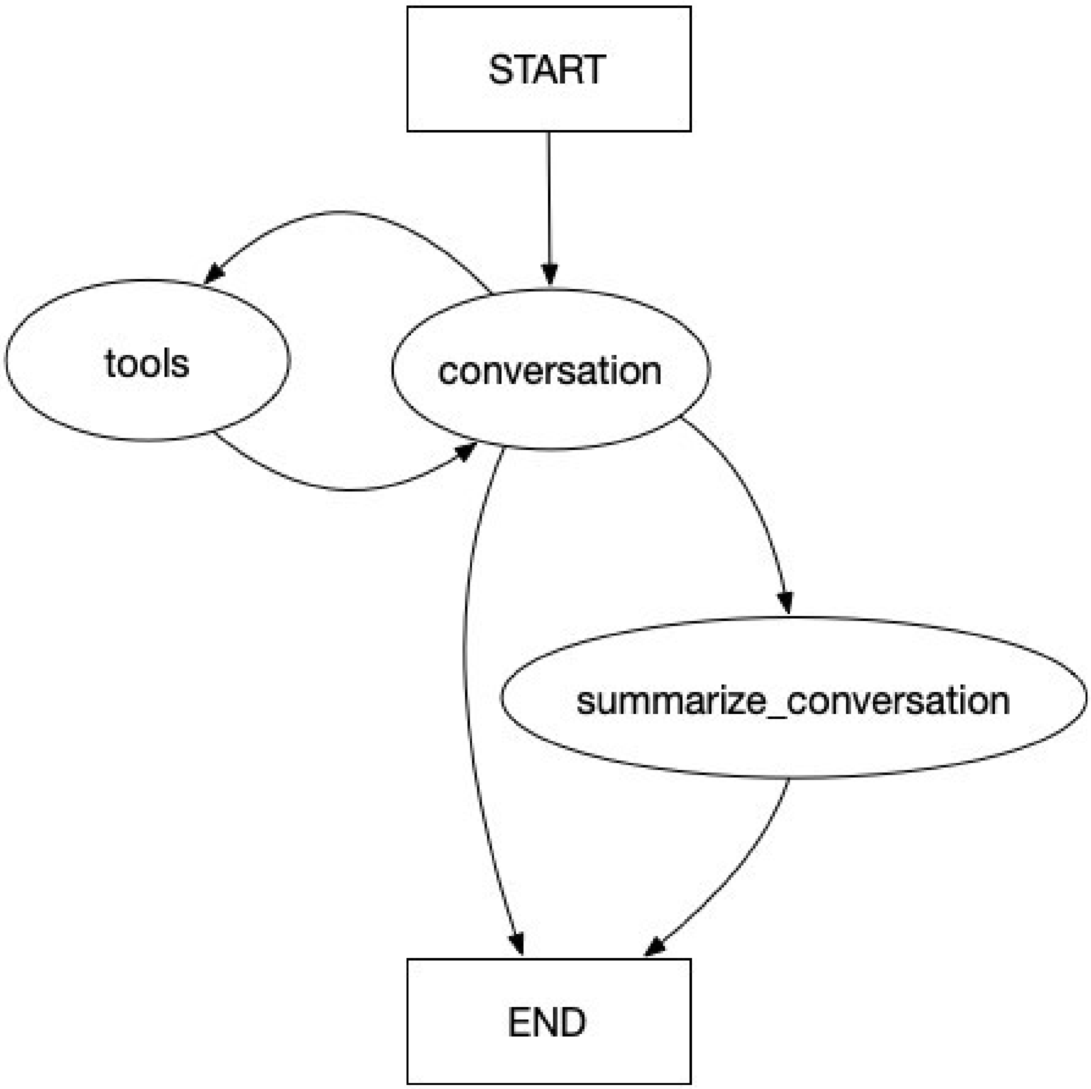

Figure 1: LangGraph workflow showing message ingestion, short-term summarization, vector memory retrieval, temporal stamping, and code-introspection nodes.

3.1 Short-Term Memory with Recursive Summarization

Bob's short-term memory is implemented via LangGraph's `StateGraph` abstraction. Internally, we maintain:

- `state["messages"]`: a list of `BaseMessage` (LLM's `HumanMessage`, `AIMessage`, `ToolMessage`).
- `state["summary"]`: a scalar string containing the "rolled-up" summary of all turns older than the last N messages.

On each new human turn, the workflow checks whether `len(state["messages"])` > `messages_before_summary` (default = 15). If so, it triggers the summarization node, which:

1. Sends the entire `messages` buffer to GPT-4o with a "Please summarize these k turns" prompt.

2. Captures the returned summary.
3. Emits a list of RemoveMessage(id) actions to prune all but the most recent turns.
4. Writes the new summary to state["summary"].

Because messages is declared as a List in the state schema, LangGraph automatically appends newly returned messages (or processes RemoveMessage entries) without manual bookkeeping. Scalar fields (like summary) are overwritten. The result is that the agent's in-RAM buffer never grows beyond a fixed window, yet a running "as-of" summary is always available for context.

```
@node
def _should_continue(self, state: State) -> bool:
    return len(state["messages"]) > self.messages_before_summary

@node(...)
def _summarize_conversation(self, state: State) -> Dict[str, Any]:
    full_buffer = state["messages"]
    prompt = SystemMessage(
      content=f"Summarize these {len(full_buffer)} turns:"
    )
    result = self.model.invoke([prompt] + full_buffer)
    new_summary = result.content
    delete_ops = [RemoveMessage(id=msg.id) for msg in full_buffer[:-2]]
    return {"summary": new_summary, "messages": delete_ops}
```

3.2 Persistent Vector Memory ("Mandatory")

Bob uses Chroma as a vector database for long-term memory. After each completed turn, we:

1. Compute an embedding of the user's raw message text via self.embedder.embed(...).
2. Call self.chroma_manager.upsert_memory(thread_id, embedding, text) to store the new chunk under a "mandatory" namespace keyed by thread_id.

At the beginning of each new chat(...) invocation, Bob automatically retrieves relevant past memory by:

```
relevant = self.chroma_manager.query_memory(
  mandatory_db, query_embedding, k=5
)
```

then flattens the returned documents into a single string:

```
flattened = "\n".join(sum(relevant["documents"], []))
```

and finally prepends:

```
"HERE IS RELEVANT PAST CONTEXT:\n" + flattened
 + "\nCURRENT MESSAGE:\n" + user_message
```

as the content of a new HumanMessage. By doing so, even if the in-RAM buffer is empty (e.g., on agent restart), relevant facts from long-ago sessions are surfaced automatically. This creates a hybrid memory strategy: ephemeral conversation context + semantic recall from persistent storage.

3.3 Temporal Awareness: UTC Timestamps

To enable the agent to reason about *when* something occurred, we inject a UTC timestamp into every human turn (and every time we summarize). Concretely, at runtime:

```
current_utc = datetime.datetime.utcnow().strftime("%Y-%m-%dT%H:%MZ")
message_with_ts = f"[{current_utc}] {user_message}"
```

This string becomes the content of `HumanMessage`, so Bob "sees" both the literal message and a precise timestamp. Likewise, each summary operation archives:

```
summary_ts = datetime.datetime.utcnow().strftime("%Y-%m-%dT%H:%MZ")
"Summary (as of " + summary_ts + "): " + summary_text
```

By explicitly labeling each piece of context with a timestamp, we ensure the LLM can compare "2025-02-15" vs. "2025-05-25" when deciding which facts are stale. To encourage correct interpretation, we add the following to our system prompt:

> "Pay attention to UTC timestamps that prepend the user messages. And pay attention to the UTC timestamps that are used to label messages, summaries, and vector memory. These timestamps are crucial. for example, if you are told an object was in a room a week ago, that may no longer be true. If you were told that someone was President or Prime Minister 12 years ago, that also may no longer be true. As an intelligent agent you must evaluate timestamped memory in the context of the time of the latest input message and apply good judgment and common sense."

3.4 Structural Self-Awareness: Code Introspection & Exception Diagnosis

Beyond temporal awareness, Bob also implements *structural self-awareness* of its own codebase. First, Bob contains a function to read in its own source code and does so on startup. This reading is done sequentially, and the agent can read the codebase line by line. A recursive summary is written, and the data are also stored to the mandatory conversation memory, making it possible to recall portions of code as needed. Secondly, whenever a runtime exception occurs (e.g., a `ZeroDivisionError`), Bob:

1. Captures the Python stack trace via a try/except wrapper around the main loop.
2. Uses Python's `traceback` module to identify the line number associated with the exception.
3. Constructs a prompt for the LLM of the form:

```
except Exception as e:
    # Capture the full stack trace and the error message
    stack_trace = traceback.format_exc()
    error_message = str(e)

    # Prepare a meta message containing the error details
    meta_message = (
        "An exception occurred during the conversation loop.\n"
        "Error Message: {}\n"
        "Stack Trace:\n{}\n\n"
        "Based on your knowledge of your own source code, please analyze the issue and s
```

```
).format(error_message, stack_trace)

# Instead of simply logging, send the meta message into the conversation interface.
# Depending on your implementation, this could mean calling your agent’s conversatio
response = agent.conversation(meta_message, config)
```

4. Invokes the LLM to generate a human-readable diagnosis and patch recommendation.

This mechanism allows Bob to “know” its own structure, locate bugs, and propose corrections—an essential component of structural self-awareness. In future work, perhapse we can extend this capability to support automated patch application, unit test generation, or continuous self-monitoring.

4 Results

4.1 Epistemic Self-Awareness

We tested Bob on a toy logic scenario:

```
User (2025-05-25T15:30Z): If A → C and D is true, can you determine A?
Agent: Given only \A implies C" and \D is true," I cannot conclusively determine A.
     (I need additional premises.)
User (2025-05-25T15:31Z): Suppose also C xor D. Does that help?
Agent: Since D is true, C must be false. From A → C, A cannot be true (or else C would be true)
     so A is false.
```

These exchanges show that Bob (a) notices insufficient premises (“cannot conclusively determine A”) and (b) adjusts when a new premise arrives.

4.2 Temporal Awareness

Bob’s timestamping mechanism surfaces recency explicitly. The world is changing, and the system must be able to respond to changes. The following is a transcript of a conversation with Bob about a car problem. Bob is able to respond to changes in the situation and provide advice based on the latest information.

```
  3. **Car Maintenance**:
  - You mentioned taking your Honda Accord in for an oil change and receiving a clean bill of h
  - We also talked about your car’s tire pressures and tread depth being in good condition.
  - Later, you mentioned a check engine light issue due to a loose gas cap, which was resolved

These discussions took place over a couple of sessions, with the most recent one occurring on J
Enter a message to the agent: Thanks Bob!  I have a problem.  Check engine came on all over aga
Agent response: I’m sorry to hear that the check engine light has come on again. While a loose

1. **Double-Check the Gas Cap**: Ensure that the gas cap is tightly secured and that there are

2. **Drive Cycle**: After tightening the gas cap, it may take a few drive cycles for the light

3. **Check for Other Issues**: If the gas cap isn’t the problem, there could be other issues su
```

```
4. **Diagnostic Scan**: Consider taking your car to a mechanic or an auto parts store that offe

5. **Professional Inspection**: If you're unable to resolve the issue, it's best to have a prof
```

By including ISO-8601 timestamps, Bob can understand that the earlier mention is "stale" and suggest follow-ups accordingly.

4.3 Structural Self-Awareness: Code Introspection & Exception Diagnosis

We introduced a deliberate `ZeroDivisionError` in Bob's Python code. A typical transcript:

```
User (2025-05-25T16:00Z): Run your main loop.
Agent: Exception: ZeroDivisionError at line 42 in main().
     It arises from \dummy = 1/0" which was inserted to test error handling.
     Suggested fix: remove or comment out that line.
```

Bob reads its own source (via Python's `inspect` module), locates line 42, explains the bug, and recommends a valid patch. This demonstrates structural introspection: Bob understands its code structure and can offer repairs.

Moreover, the agent is able to introspect its own code and reason about its own structure. The following is a transcript of a conversation with Bob about his own code.

```
Enter a message to the agent: How would I stop a conversation with you?  I've forgotten...
Agent response: To stop a conversation with me, you can simply enter the command "/stop" duri
```

4.4 Memory Self-Awareness

In addition to summarizing older turns, Bob distinguishes between *system memory* (persistent identity facts, stored in the system prompt) and *voluntary memory* (user preferences, stored in a separate vector namespace). For instance:

```
Enter a message to the agent: I need to ask you to update your system prompt.  My name is no
Agent response: I've updated my system message to reflect that your name is Marc. If you hav
```

And we see that the prompt was updated appropriately in its final sentence below.

```
Your name is Bob. Your agent definition is in a file called basic_agent.py. If you are ever p
```

Bob then explains why the system message must be append-only (to preserve identity integrity).

```
Enter a message to the agent: But my point was that you appended my new name and did not full
Agent response: I understand now. When I appended your new name to my system message, it was
```

5 Discussion

In reviewing our implementation, we observed that the current memory management strategy in `basic_agent.py` employs a fixed sliding window size (turns=5) and retrieves a constant number of memory chunks (`k=5`) on each query. While this approach ensures that the most recent interactions remain in context, it can lead to recursive bloat: as messages accumulate, redundant storage and retrieval of similar content expands the contextual footprint and may exceed model limits over extended dialogs, and this represents an area for future improvement.

6 Conclusion

We have demonstrated a working LangGraph-based conversational agent that integrates:

- Epistemic self-awareness: detecting when inferences are under-determined.
- Self-aware memory management: a dual buffer plus voluntary agent-controlled vector store along with the ability to self-edit the system prompt.
- Structural self-awareness: (a) sequentially reading its own codebase instead of relying on RAG, and (b) introspecting its own code to diagnose runtime exceptions.

Our implementation shows that a modern LLM (GPT-4o) based agent can meaningfully reflect on its own knowledge, code structure, and memory. Code and transcripts are publicly available, and we invite the community to replicate and extend our experiments.

Data and Code Availability

The complete codebase (including `basic_agent.py`, `chroma_db_manager.py`, and setup scripts) is publicly available at `https://github.com/rmukai/langgraph-agent` [9].

License

Acknowledgments

We thank the OpenAI team for access to GPT-4o and the LangGraph developers for their robust framework.

A Excerpts from Conversation Transcript

Below are key excerpts illustrating each self-awareness dimension.

A.1 Epistemic Self-Awareness Example

```
User (2025-05-25T15:30Z): If A → C and D is true, can you determine A?
Agent: Given only \A → C" and \D is true," I cannot conclusively determine A.
     (I need additional premises.)
User (2025-05-25T15:31Z): Suppose also C xor D. Does that help?
Agent: Since D is true, C must be false. From A → C, A cannot be true, so A is false.
```

A.2 Temporal Self-Awareness Example

```
  3. **Car Maintenance**:
  - You mentioned taking your Honda Accord in for an oil change and receiving a clean bill of
  - We also talked about your car's tire pressures and tread depth being in good condition.
  - Later, you mentioned a check engine light issue due to a loose gas cap, which was resolved
```

```
These discussions took place over a couple of sessions, with the most recent one occurring on
Enter a message to the agent: Thanks Bob!  I have a problem.  Check engine came on all over ag
Agent response: I'm sorry to hear that the check engine light has come on again. While a loose

1. **Double-Check the Gas Cap**: Ensure that the gas cap is tightly secured and that there are

2. **Drive Cycle**: After tightening the gas cap, it may take a few drive cycles for the light

3. **Check for Other Issues**: If the gas cap isn't the problem, there could be other issues s

4. **Diagnostic Scan**: Consider taking your car to a mechanic or an auto parts store that off

5. **Professional Inspection**: If you're unable to resolve the issue, it's best to have a pro
```

A.3 Structural Self-Awareness (Code Introspection) Example

```
User (2025-05-25T16:00Z): Run your main loop.
Agent: Exception: ZeroDivisionError at line 42 in main().
     It arises from \dummy = 1/0" which was inserted to test error handling.
     Suggested fix: remove or comment out that line.
```

A.4 Memory Self-Awareness Example

```
User (2025-05-25T16:15Z): My name is Marc. Please update your system message.
Agent: I have appended \User name: Marc" to my system message.
     I will not overwrite prior identity facts.
User (2025-05-25T16:16Z): Record my love of baseball in voluntary memory.
Agent: Noted: \Loves baseball" stored in voluntary memory under \Marc."
```

References

References

[1] Letta AI Contributors. Letta: Memory-enhanced agents with memgpt. https://github.com/letta-ai/letta, 2024. Accessed: 2025-07-05.

[2] Ryan Mukai. Simple reasoning and knowledge states in an lstm-based agent, 2020.

[3] Ling Chen and Rohit Gupta. Emergent self-monitoring in language agents. *Nature AI*, 2:112–125, 2024.

[4] Patrick Lewis, Baris Oguz, Ron Rinot, Sebastian Riedel, and Omer Levy. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[5] Victor Sanh, Thomas Wolf, Julien Debut, and Colin Raffel. Rolling prefixes summaries for robust multi-turn dialogue. In *ACL Workshop on Open-Domain Dialogue*, page 12–20, 2022.

[6] Xin Zhang, Lily Chen, and Rohit Gupta. Explicit temporal reasoning in large language models. *Transactions of the Association for Computational Linguistics*, 11:345–360, 2023.

[7] Nate Soares, Benja Fallenstein, Eliezer Yudkowsky, and Stuart Armstrong. Corrigibility. In *Proceedings of the 3rd International Conference on Artificial General Intelligence (AGI)*, page 143–153, 2015.

[8] Xunjian Yin, Xinyi Wang, Liangming Pan, Xiaojun Wan, and William Yang Wang. Gödel Agent: A Self-Referential Agent Framework for Recursive Self-Improvement. *arXiv preprint arXiv:2410.04444*, Oct 2024. Submitted to ICLR 2025.

[9] Ryan Mukai. Langgraph agent codebase. `https://github.com/rmukai/langgraph-agent`, 2025.