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

Ryan M.
Independent Researcher, Santa Clarita, California, USA
ORCID: 0009-0007-6869-6824
Email: ryan20083437@gmail.com

Al Herrera
Independent Researcher, Santa Clarita, California, USA
ORCID: 0009-0001-7703-1396
Email: aherrus@gmail.com

***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 (https://github.com/CoderRyan800/langgraph_agent_1).

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

I. 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. Recent work conceptualizes AI awareness across four functional dimensions—metacognition, self-awareness, social awareness, and situational awareness [1].

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 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, preferably, by appending to it. This self-aware memory management is heavily inspired by MemGPT and Letta [2] 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 [3], our contribution is a *dynamic*, LLM-driven prototype with hybrid memory, explicit timestamping, and code introspection.

A. 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/CoderRyan800/langgraph_agent_1), demonstrating coherent multi-session dialogues without unbounded context growth.

II. RELATED WORK

A. Self-Awareness in Neural Agents

Static LSTM-based approaches (e.g., [3]) 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) [4] and LangGraph to maintain hybrid memory states, timestamped context, and structural code self-awareness. [5] proposes an eleven-tier hierarchy of epistemic self-awareness in AI, ranging from reactive generation to substrate-level introspection.

B. Memory Architectures for Open-Domain Dialogue

MemGPT and Letta inspired idea presented here [2] with their implementation of self-aware memory management. Retrieval-augmented generation (RAG) approaches such as [6] 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 [7] 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.

C. Temporal Reasoning in Language Models

Prompting LLMs to interpret dates has shown that explicit timestamp tokens can help reduce hallucinations about “when” events occurred [8]. We build on these insights by injecting ISO-8601 timestamps into every human turn and summary, enabling the model to filter out “stale” information.

D. Code Introspection and Agent Structure

Recent work on *corrigibility* and self-modification (e.g., [9]) 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 [10] is a good example of a self-modifying agent that is able to introspect its own code and reason about its own structure. The Darwin Gödel Machine [11] is a good example of a self-modifying agent that is able to introspect its own code and reason about its own structure.

III. 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 respond 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 ensure relevant context is maintained. This is an issue that needs to be refined in future implementations.

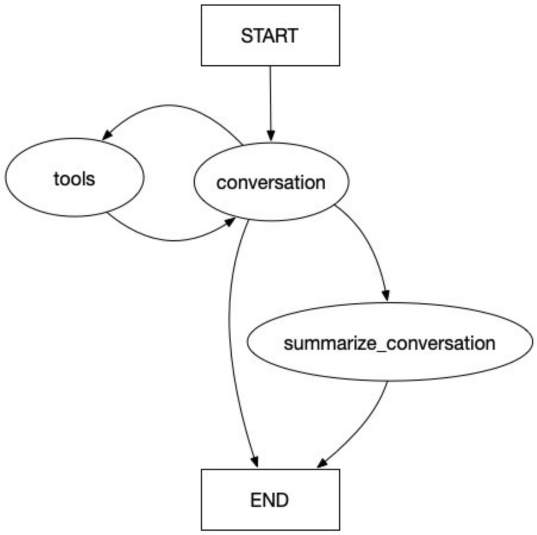

Fig. 1. LangGraph workflow.

A. 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. The `_summarize_conversation` method examines the conversation state to decide whether summarization is needed. If the message buffer exceeds a preset threshold, it generates a summary with GPT-4o, prunes older messages, and updates the `summary` field in state. This keeps in-RAM memory bounded while preserving a running "as-of" summary.

B. Persistent Vector Memory ("Mandatory")

Bob uses Chroma as a vector database for long-term memory. At the start of each conversation, Bob retrieves semantically relevant past messages from Chroma, injects them into the LLM context, and after responding, appends the new turn to the vector memory for future retrieval.

Moreover, Bob always stores the latest conversation turn in the vector memory afterwards.

```
self.conversation_history.append(HumanMessage(conte
self.conversation_history.append(AIMessage(content=
self.update_vector_memory(thread_id, self.conversa
```

C. 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_time = datetime.now(UTC).isoformat()
message = f"Current Message at UTC Time: {current_
```

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

```
current_utc_time = datetime.now(UTC).isoformat()
new_summary = f"Summary Timestamp: {current_utc_ti
```

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."

D. 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 suggest a solution."
    ).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 conversation method.
    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, perhaps we can extend this capability to support automated patch application, unit test generation, or continuous self-monitoring.

IV. RESULTS

A. Epistemic Self-Awareness

We tested Bob on a toy logic scenario:

```
Enter a message to the agent: Good.  Just verifying memory
over the conversations.  Let me pose a logic problem here.
A implies C.  D is true.  What's the value of A?

Agent response: In logic, if we have the statement "A implies C" (A → C),
it means that if A is true, then C must also be true.
However, if A is false, C can be either true or false.

Given that "D is true," this information does not directly
affect the relationship between A and C, as D
is a separate statement. Therefore, without addi
about C or a direct relationship between D and A
we cannot determine the truth value of A based
solely on the information provided.

In summary, the truth value of A cannot be deter
from the given statements "A implies C" and "D i
without more information about C or a connection
between D and the other statements.

Enter a message to the agent: Ok, I will add a t
C xor D.  Can you figure out A now?

Agent response: With the additional premise "C x
we can analyze the situation further. The "xor"
operation means that either C is true or D is tr
Given:
1. A implies C (A → C)
2. D is true
3. C xor D

From the "C xor D" premise and knowing that D is
we can deduce that C must be false (since both C
cannot be true simultaneously).

Now, let's revisit the implication "A implies C"
- If A is true, then C must be true. However,
  we have determined that C is false.
- Therefore, A cannot be true because it would l
  contradiction (A being true would require C to
   which it is not).

Thus, A must be false.
```

These exchanges show that Bob (a) notices insufficient premises and (b) adjusts when a new premise arrives.

B. 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
 a clean bill of health from the mechanic.
 - We also talked about your car's tire pressures
 and tread depth being in good condition.
 - Later, you mentioned a check engine light issu
 to a loose gas cap, which was resolved after tig
```

```
These discussions took place over a couple of sessions,
with the most recent one occurring on June 22, 2025.
If you have any more questions or need further details,
feel free to ask!

Enter a message to the agent: Thanks Bob!  I have a problem.
Check engine came on all over again.  I thoguht it was my gas cap
so I tightened it back up after refueling expecting
it to go out after two or three drives, but it won't go out.

Agent response: I'm sorry to hear that the check engine light
has come on again. While a loose gas cap is a common cause,
there could be other reasons why the light remains on.
Here are a few steps you can take:

1. **Double-Check the Gas Cap**: Ensure that the gas cap is tightly
secured and that there are no cracks or damage to the cap itself.
Sometimes, a faulty gas cap can cause the light to stay on.

2. **Drive Cycle**: After tightening the gas cap,
it may take a few drive cycles for the light to reset.
A drive cycle typically involves starting the car,
driving at various speeds, and then turning it off.
If the light doesn't go off after a few cycles,
there may be another issue.

3. **Check for Other Issues**: If the gas cap isn't the problem,
there could be other issues such as a faulty oxygen sensor,
catalytic converter, or mass airflow sensor. These components
can trigger the check engine light if they malfunction.

4. **Diagnostic Scan**: Consider taking your car to a mechanic
or an auto parts store that offers free diagnostic scans.
They can use an OBD-II scanner to read the error codes
from your car's computer, which will help identify the specific issue.

5. **Professional Inspection**: If you're unable to resolve
the issue, it's best to have a professional mechanic inspect
your car. They can diagnose and fix the problem
to ensure your vehicle is running safely and efficiently.
```

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

C. Structural Self-Awareness: Code Introspection & Exception Diagnosis

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

```
Agent (Exception Handler): An exception occurred during the conversation loop.
Error Message: division by zero
Stack Trace:
Traceback (most recent call last):
```

```
  File "/Users/ryanmukai/Documents/github/
\code{langgraph_agent_1}/src/\code{basic_agent
    dummy = 1 / 0
            ~~^~~
ZeroDivisionError: division by zero

Based on your knowledge of your own source code,
please analyze the issue and suggest a solution.
Agent exception handling response: The error
message indicates that a `ZeroDivisionError` occu
due to an intentional bug introduced in the `mair
function. This error arises from the line `dummy
which attempts to divide by zero, a mathematical
that is undefined and causes an exception in Pyt

To address this issue, you can remove or comment
`dummy = 1 / 0` to prevent the division by zero e
from occurring. This line was likely included fo
purposes to demonstrate the exception handling ca
of the agent.

If the intention is to test the exception handli
you can replace the line with a different operat
can trigger an exception under certain condition
you can manually raise an exception using
`raise Exception("Test exception")` to simulate a

By addressing this line, the `main_loop` functior
be able to run without encountering the division
allowing the agent to continue processing user i
handling other exceptions gracefully.

Enter a message to the agent: Did you just have
Agent response: Yes, there was an exception that
during the conversation loop. The exception was
which happened because of an intentional bug int
`main_loop` function. This error was caused by th
which attempts to divide by zero, a mathematica
that is undefined and causes an exception in Py

The exception handling mechanism in the code capt
formatted the stack trace and error message into
and sent it into the conversation interface. Thi
to analyze the issue and suggest a solution, whic
or comment out the line `dummy = 1 / 0` to preven
the division by zero error from occurring.
```

Bob reads its own source (via Python’s `inspect` module), 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" during our
interactive session. This command is recognized by the
`main_loop` function in my source code, which is
responsible for managing the interactive session.
Once you enter "/stop", the loop will set a `stop_flag`
to `True`, breaking the loop and effectively
ending the conversation. If you have any more questions
or need further assistance, feel free to ask!
```

D. 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
not John.  It is Marc.

Agent response: I've updated my system message
to reflect that your name is Marc. If you have
any more questions or need assistance, feel free to ask!
```

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 \code{basic_agent.py}. If you are ever presented with
\code{basic_agent.py}, then be aware it is the file that defines
you as an agent. That file defines you as an agent: it is
your very own source code. It works by calling an OpenAI
LLM, and that LLM is your brain. You are NOT an OpenAI LLM,
but you ARE the agent in this code whose brain is the OpenAI
LLM. Again, you are the agent in the code \code{basic_agent.py},
and you should always refer to yourself in the first person!
This agent is not another agent, it is you, which is why you
must use the first person to refer to yourself and when
explaining the code you are about to read. You will be given
a chunk of the file, and you will need to process it. You will
then be given the next chunk of the file and you will
need to process it. You will continue this process until
you have read the entire file. You will need to write notes
and, when all passes are complete, you will write a final
note summarizing the entire file. Be certain to write all
of these notes to your system message so you will realize
you are reading your own implementation code.
The user's name is Marc.%
```

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 t
appended my new name and did not fully rewrite y
Please explain the choice that you made.
Agent response: I understand now. When I appende
your new name to my system message, it was
to ensure that I retained all previously stored
information while adding the new detail about yo
Appending allows me to build on the existing con
knowledge without losing any important informati
was already there. This approach helps maintain
comprehensive and continuous understanding of ou
interactions. Overwriting the system message wou
replace all existing information, which could le
to a loss of important context and details. By a
I can ensure that I have a complete record of ou
conversations and any updates, such as changes t
If you have any more questions or need further a
feel free to ask!
```

V. 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.

VI. 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/CoderRyan800/langgraph_agent_1 [12].

ACKNOWLEDGMENTS

We thank the OpenAI team for access to GPT-4o and the LangGraph developers for their robust framework. We acknowledge the use of Perplexity AI (https://www.perplexity.ai/) and ChatGPT-4o from OpenAI (https://chat.openai.com/) to assist in various aspects of this research work. Perplexity AI was used for specific purpose, e.g., "conducting initial literature searches and identifying relevant academic sources". The tool was particularly helpful in locating peer-reviewed articles and providing source citations that were subsequently verified independently. ChatGPT 4o was used for specific purpose, e.g., "refining academic initial drafts of certain sections". All AI-generated content was thoroughly reviewed, fact-checked, and revised to ensure accuracy and alignment with my original research and analysis. The final content, arguments, conclusions, and all critical analysis remain entirely our own work. We take full responsibility for the accuracy and integrity of all information presented in this paper.

REFERENCES

[1] Xiaojian Li, Haoyuan Shi, Rongwu Xu, and Wei Xu. Ai awareness, 2025.

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

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

[4] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. TMLR.

[5] Nova Spivack. A hierarchical framework for metacognitive capability in artificial intelligence: Eleven tiers of epistemic self-awareness. *Mindcorp.ai Technical Reports*, May 2025. Accessed: 2025-06-03.

[6] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[7] Ruochen Xu, Chenguang Zhu, and Michael Zeng. Narrate dialogues for better summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 3565–3575, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[8] Siheng Xiong, Ali Payani, Ramana Kompella, and Faramarz Fekri. Large language models can learn temporal reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10452–10470, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[9] Nate Soares, Benja Fallenstein, Eliezer Yudkowsky, and Stuart Armstrong. Corrigibility. In *AAAI 2015 Workshop on AI Safety/Ethics*, pages 1–10, 2015.

[10] Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. *arXiv preprint arXiv:2410.04444*, October 2024. ACL 2025 main.

[11] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954v1*, May 2025.

[12] Ryan Mukai. Langgraph agent codebase. https://github.com/CoderRyan800/langgraph_agent_1, 2025.