Blue Skies Ahead

An AI Case Study on LLM Use for a Graph Theory Related Application

Gerald T. Rigdon

Fellow, Software Engineering

Boston Scientific, Inc.

gerald.rigdon@bsci.com

February 26, 2025

Abstract

Research activities necessitate the exploration of ideas which may involve significant software development, experimentation, and testing effort. The ability of Large Language Models (LLMs) to generate executable software has been demonstrated in various use cases. This study highlights the positive outcomes of a use case involving a graph theory application. Through a series of successful prompts, functional software was produced within minutes, a task that would otherwise have taken several hours or days to complete. Rather than focusing solely on cost savings from reduced engineering effort, this study aims to highlight the substantial opportunity benefits provided by this technology, especially since research typically involves evaluating multiple ideas before selecting the optimal solution.

1. INTRODUCTION

When considering great systems thinking quotes, Donnella H. Meadows stands out: "Remember, always, that everything you know, and everything everyone knows, is only a **model**. Get your **model** out there where it can be viewed. Invite others to challenge your assumptions and add their own."

In the context of software engineering, one way to wrap your head around a complex software code base is to create a static **model** of it and attempt to analyze the behavior accordingly. In the classic book *Safeware* (Leveson, 1995) it states: "Static analysis evaluates the software without executing it. Instead, it examines a representation of the software. In some ways, static analysis is more complete than dynamic analysis, since general conclusions can be drawn and not just conclusions limited to the particular test cases that were selected. On the other hand, static analysis necessarily is limited to evaluating a representation of a behavior rather than examining the behavior itself."

A paper written several years ago (Rigdon, 2010), discusses the role of static analysis in medical device software development, providing insights into the process at Boston Scientific Corporation concerning the use of design constraints in software engineering activities. These design constraints, distinct from requirements, help ensure that detailed design and implementation remain within domain-specific boundaries set and enforced by static analysis activities. A concrete example is best illustrated by the following story, which also illustrates an opportunity to, as Meadows exhorted, "Get your **model** out there."

A few weeks ago, a Boston Scientific colleague identified a concern emerging from the development of a new device platform and proposed creating another design constraint. The issue involved creating a new firmware utility function interface and ensuring developers used this new interface in specific contexts instead of the existing interface that had been in use for years. Integrating a new function in an existing code base generally conjures up images in the mind of software engineers as they immediately want to understand how this new piece fits into the larger whole. As stated so well in (Hopstock, 2022), “Being able to **model** the inter-procedural control flow as a call graph is one of the most important building blocks when analyzing programs. Many of the more advanced analyses depend on this information being available.”

Conveniently, as discussed in (Rigdon, Doshi, Zheng, 2010), the Boston Scientific implantable firmware development environment includes the use of call graph data for static analysis produced by a customized Static Analysis Tool (SA Tool). But this new design constraint required leveraging that graph data differently, necessitating the development of software to experiment and determine its viability for this new application. It was estimated that although this was not likely to result in a lot of code, that it was still three to four days of considerable Python development and debug given the esoteric nature of the task. Consequently, the task was assigned to an available developer. Waiting until the developer completed the job successfully provided working code and a baseline that was useful for this case study.

Pacemakers and defibrillators, namely, safety-critical Class III implantable devices, are custom machines that are typically programmable. This includes a set of parameters programmed at the time of manufacture, later at the time of device implant, and in post-operative clinical settings. These telemetry interactions are an example of one group of activities that result in firmware utility function use. Referencing the previous conversation with a colleague concerning the creation of another design constraint for managing finer control of device firmware utility function use, the first order of business was to find all uses of the existing utility function, namely, *utl_ParamUpdate()* and then:

- Decide which cases should be updated to use the newly proposed firmware utility function
- Implement the new utility function and refactor the existing firmware to make use of it
- Enforce, by means of a design constraint, that future design and implementation use the proper utility function

Interestingly, this software engineering problem highlights an opportunity for testing the viability of another type of **model,** a Large Language **Model** (LLM).

2. IDENTIFYING EXISTING USES

Figure 1 is a snippet of a much larger call graph in this code base that shows all callers of the *utl_ParamUpdate()* firmware utility function under discussion. For each caller, encased in the bright pink rectangles, there is an exploded view available as shown in Figure 2 and Figure 3. These subsequent views could likewise contain the same bright pink visuals leading to further exploded views until arriving at one or more root nodes or graph origin nodes.

Figure 1

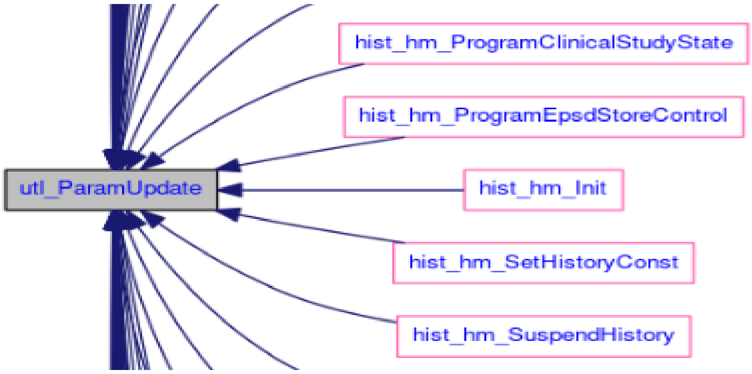
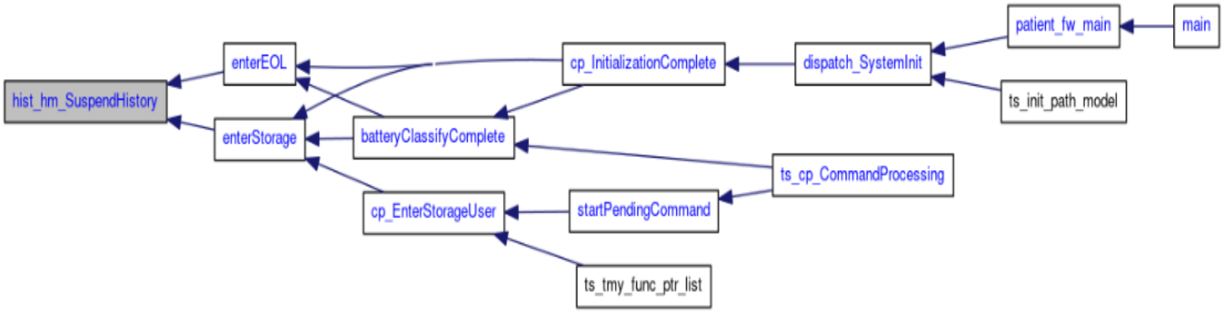


Figure 2

Figure 3



It was expected that the *utl_ParamUpdate()* firmware utility function would be called in the following cases, identifiable by the graph root nodes:

- During initialization of the firmware and following a reset
- When commanded by an authenticated external device connected through telemetry
- Dynamically, during firmware execution including monitoring and therapy delivery, etc.

Performing text searches like greps or otherwise are not overly useful for discovering these call paths given the goal is to find the root node which would, as stated, allow the identification of the classifications from the above bulleted cases. Granted, given a simple graph, such searches can be successful. However, this discovery process becomes much more difficult when the call graph is both deep and wide or as shown in Figure 3 above, identifies multiple root node path cases. Hence, a manual process would be more susceptible to yielding inaccurate results given graph complexity. It is a problem in search of an automated solution especially since the call graphs themselves are auto generated.

3. HAND-CRAFTED CODE CREATION

As stated earlier, this project commenced with the creation of software by an engineer given some general requirements and the SA Tool generated call graph output CSV file as an input. Figure 4 is a terse example of a call graph file that is a list of all file and function caller and callee pairs.

Figure 4

CallerFile	**CallerFunc**	**CalleeFile**	**CalleeFunc**
sensor_mgr	calc_temperature	sensor_a	get_temperature
sensor_a	get_temperature	logger	save_temperature

Each row in this graph CSV file represents nodes in a call graph that are connected. These connections in a directed graph are known as edges which represent the relationship between the caller or source node that invokes the callee or target node. With a list of these caller and callee pairs one has all the information necessary to build out a visual call graph. Further, with the use of some open-source tools like Graphviz (https://graphviz.org) this information can be converted to the DOT (graph description language) format which can be ultimately rendered in formats such as JPEG for displaying beautiful visual graphs.

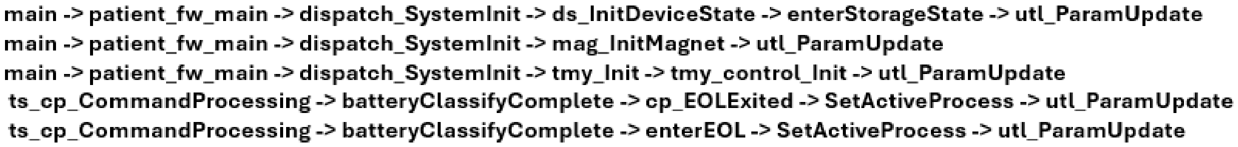
This case study was based on a project whose graph CSV file consisted of over 3,500 rows where the primary software requirements are conveyed as follows:

Req1: The software shall use the graph CSV as an input file and produce a list of all caller root node functions and subsequent node edges that result in all possible unique directed graph paths that end in a specified callee function.

Req2: Each path in **Req1** shall be output as a separate row in a destination CSV file where the edges between each node are represented visually with “->” characters.

Although the final product was not a large Python script as had been anticipated, it took three days to develop. Not surprising given most developers fluent in Python are often not well versed in how to write and debug code for graphs unless that is their primary job focus. So, the initial development effort estimates were close. The developed Python code yielded an output file containing one-hundred-six unique paths. An excerpt capturing five of these paths is shown below in Figure 5.

Figure 5

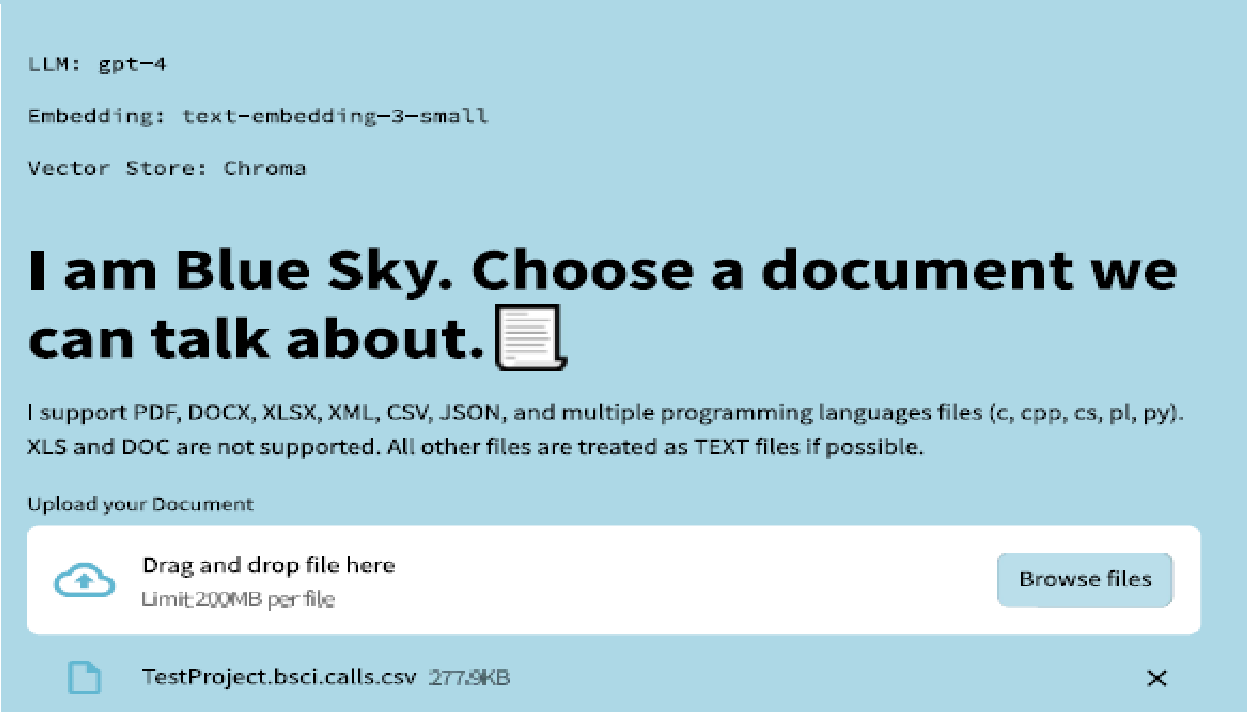


```
main -> patient_fw_main -> dispatch_SystemInit -> ds_InitDeviceState -> enterStorageState -> utl_ParamUpdate
main -> patient_fw_main -> dispatch_SystemInit -> mag_InitMagnet -> utl_ParamUpdate
main -> patient_fw_main -> dispatch_SystemInit -> tmy_Init -> tmy_control_Init -> utl_ParamUpdate
 ts_cp_CommandProcessing -> batteryClassifyComplete -> cp_EOLExited -> SetActiveProcess -> utl_ParamUpdate
 ts_cp_CommandProcessing -> batteryClassifyComplete -> enterEOL -> SetActiveProcess -> utl_ParamUpdate
```

4. CODE CREATION WITH OPENAI CHATGPT-4o

Now with some source code and generated output, the question to be answered was could one get an LLM to write a Python script that could produce the same output. The study used a custom AI web application known as Blue Sky (a tribute to Jeff Lynne and the Electric Light Orchestra and nothing to do with the new social media application boasting the same name). Blue Sky offers several features, and the one chosen was the basic chat with a document selection and configured for connectivity to OpenAI GPT-4o. As shown in Figure 6, the graph CSV named "TestProject.bsci.calls.csv" was uploaded, which is the file produced by the SA Tool for the project of interest.

Figure 6



```
LLM: gpt-4

Embedding: text-embedding-3-small

Vector Store: Chroma
```

I am Blue Sky. Choose a document we can talk about.

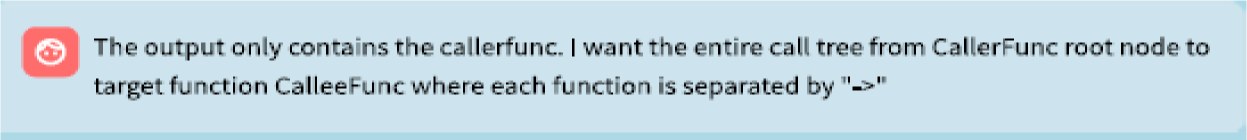
I support PDF, DOCX, XLSX, XML, CSV, JSON, and multiple programming languages files (c, cpp, cs, pl, py). XLS and DOC are not supported. All other files are treated as TEXT files if possible.

Upload your Document

To produce the correct Python script, the GPT-4o LLM was provided with the prompts shown in Figure 7 (for brevity the intermediate LLM responses were omitted). A close examination reveals some liberties were taken with the wording of the prompts compared to the previously described requirements, namely, **Req1** and **Req2**. Furthermore, only three prompts were needed, taking less than fifteen minutes to generate the desired results. For each GPT-4o script recommendation following each prompt, the generated code was copied and pasted into a file, executed, and the output results were compared until a perfect match was found with the output produced by the hand-crafted code.

Figure 7

Based on the format and content of TestProject.bsci.calls.csv please recommend a python script that identifies all callers or CallerFunc to the target function or CalleeFunc utl_ParamUpdate.

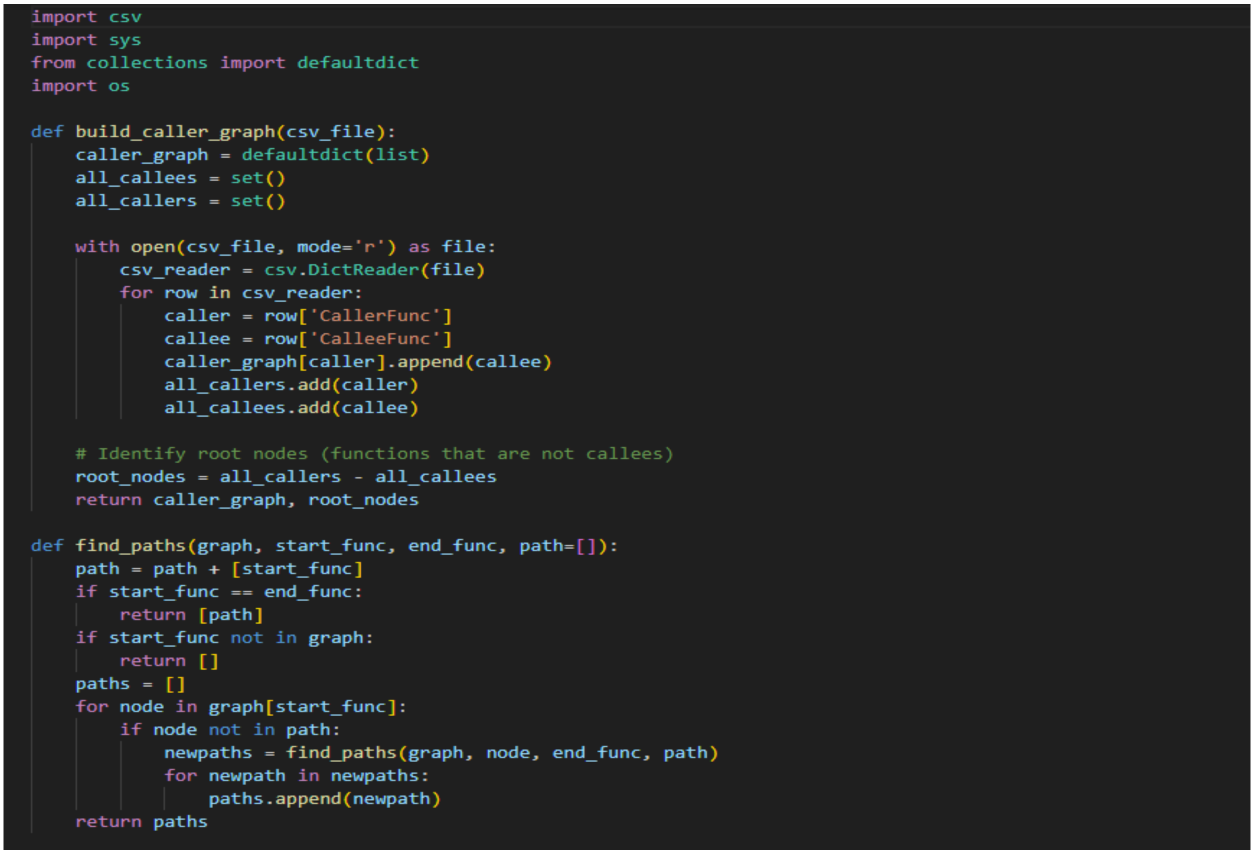


The output only contains the callerfunc. I want the entire call tree from CallerFunc root node to target function CalleeFunc where each function is separated by "->"

make sure that the initial callerfunc is actually the root node

The final GPT-4o generated Python script shown in Figure 8 (although different from the hand-crafted script) was finalized when the output matched the output from the hand-crafted script. An obfuscated Figure 9 captures a side-by-side difference report intended to show that the files are identical (no markups) instead of the specific content of the output.

Figure 8



```python
import csv
import sys
from collections import defaultdict
import os

def build_caller_graph(csv_file):
    caller_graph = defaultdict(list)
    all_callees = set()
    all_callers = set()

    with open(csv_file, mode='r') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            caller = row['CallerFunc']
            callee = row['CalleeFunc']
            caller_graph[caller].append(callee)
            all_callers.add(caller)
            all_callees.add(callee)

    # Identify root nodes (functions that are not callees)
    root_nodes = all_callers - all_callees
    return caller_graph, root_nodes

def find_paths(graph, start_func, end_func, path=[]):
    path = path + [start_func]
    if start_func == end_func:
        return [path]
    if start_func not in graph:
        return []
    paths = []
    for node in graph[start_func]:
        if node not in path:
            newpaths = find_paths(graph, node, end_func, path)
            for newpath in newpaths:
                paths.append(newpath)
    return paths
```

```python
def generate_caller_graph(input_csv, target_func):
    graph, root_nodes = build_caller_graph(input_csv)
    paths = []

    for root in root_nodes:
        if root != target_func:
            paths.extend(find_paths(graph, root, target_func))

    if not paths:
        print(f"No paths found to the function '{target_func}'.")
        return

    unique_paths = set(tuple(path) for path in paths)
    sorted_paths = sorted(unique_paths, key=lambda x: (len(x), ' -> '.join(x)))

    output_csv = os.path.splitext(input_csv)[0] + "_paths_to_" + target_func + ".csv"

    if os.path.exists(output_csv):
        os.remove(output_csv)

    with open(output_csv, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Caller Path'])

        for path in sorted_paths:
            writer.writerow([' -> '.join(path)])

    print(f"Output CSV generated: {output_csv}")

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: python find_call_tree.py <input_csv> <target_function>")
        sys.exit(1)

    input_csv = sys.argv[1]
    target_func = sys.argv[2]

    generate_caller_graph(input_csv, target_func)
```

Figure 9

OutputDiffReport
Produced: 1/13/2025 12:48:47 PM

Mode: All
Left file: C:\ICM\GraphsOld\New folder\handcoded_output.csv
Right file: C:\ICM\GraphsOld\New folder\LLM_output.csv

5. CODE CREATION WITH ANTHROPIC HAIKU – ATTEMPT 1

Unfortunately, the first attempt with the Anthropic Haiku LLM did not go very well. The strategy was to lead off with the same initial prompt given to OpenAI ChatGPT-4o and then respond accordingly based on the LLM recommendations. Figure 10 captures all the prompts sent to Haiku (for brevity the intermediate LLM responses were omitted) until it was decided to give up. Knowing when to quit and start over is not straightforward. After working with LLMs, sometimes results deteriorate, and it becomes apparent that things are beyond recovery. In these cases, it is best to clear the chat memory and get a fresh start.

Figure 10

Based on the format and content of TestProject.bsci.calls.csv please recommend a python script that identifies all callers or CallerFunc to the target function or CalleeFunc utl_ParamUpdate

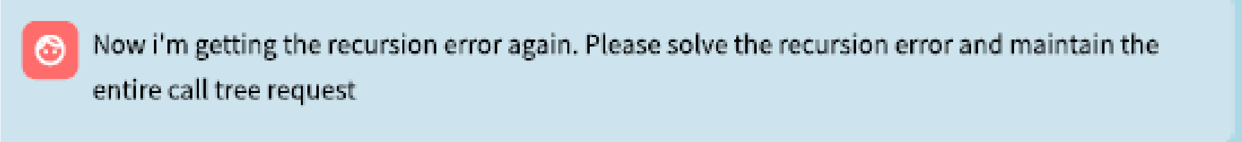
I would like the input filename and target_function to be passed into script instead of hardcoded.

Please send the output to a file instead of just the terminal

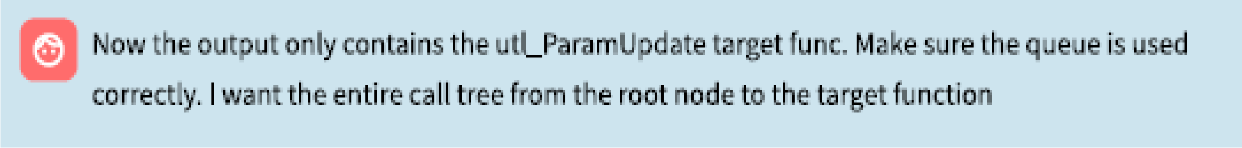
The output only contains the callerfunc. I want the entire call tree from CallerFunc root node to target function CalleeFunc where each function is separated by "->"

I'm getting the following error: "Traceback (most recent call last): File "C:\ICM\GraphsOld\find_callers.py", line 44, in <module> call_tree = find_call_tree(filename, target_function) ^^^ File "C:\ICM\GraphsOld\find_callers.py", line 29, in find_call_tree result = build_tree(function, function) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\ICM\GraphsOld\find_callers.py", line 23, in build_tree result = build_tree(caller, new_path) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\ICM\GraphsOld\find_callers.py", line 23, in build_tree result = build_tree(caller, new_path) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ File "C:\ICM\GraphsOld\find_callers.py", line 23, in build_tree result = build_tree(caller, new_path) ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ [Previous line repeated 995 more times] RecursionError: maximum recursion depth exceeded"

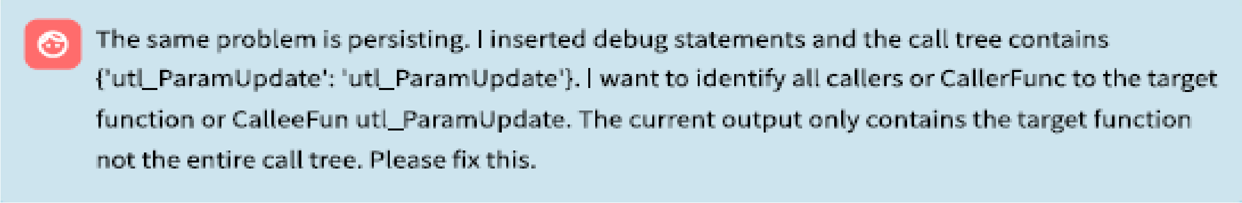
The output only contains the callerfunc. I want the entire call tree from CallerFunc root node to target function CalleeFunc where each function is separated by "->"



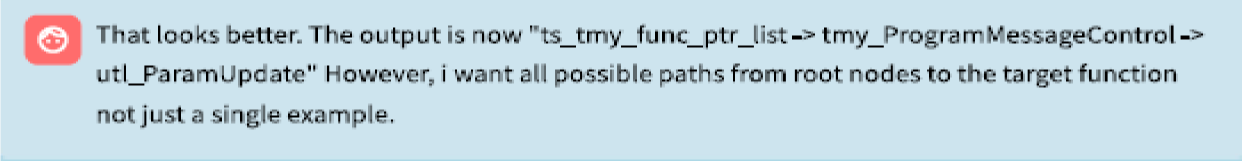
Now i'm getting the recursion error again. Please solve the recursion error and maintain the entire call tree request



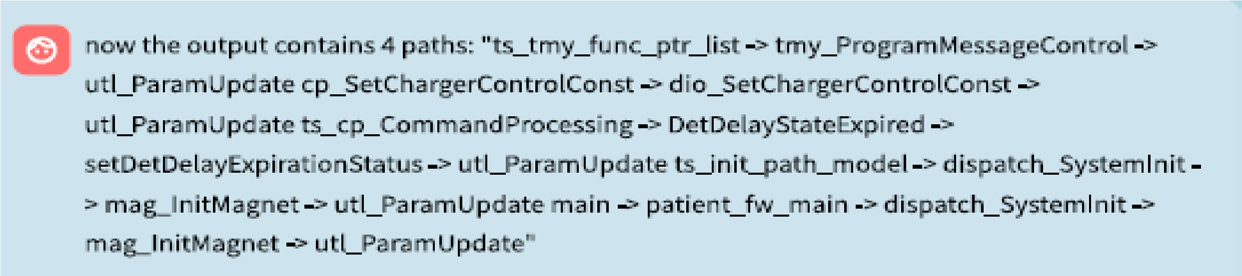
Now the output only contains the utl_ParamUpdate target func. Make sure the queue is used correctly. I want the entire call tree from the root node to the target function



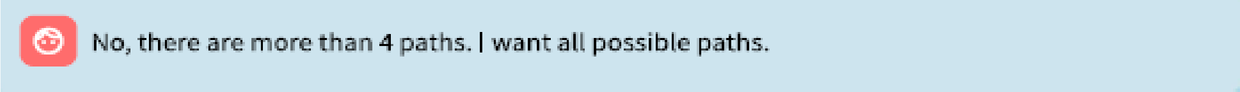
The same problem is persisting. I inserted debug statements and the call tree contains {'utl_ParamUpdate': 'utl_ParamUpdate'}. I want to identify all callers or CallerFunc to the target function or CalleeFun utl_ParamUpdate. The current output only contains the target function not the entire call tree. Please fix this.



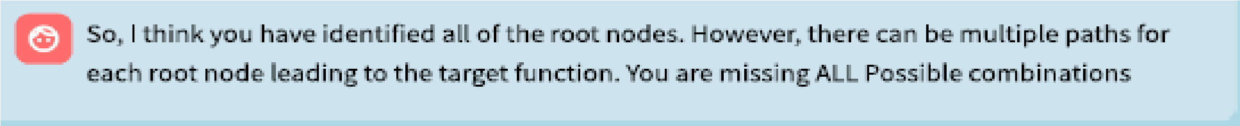
That looks better. The output is now "ts_tmy_func_ptr_list -> tmy_ProgramMessageControl -> utl_ParamUpdate" However, i want all possible paths from root nodes to the target function not just a single example.



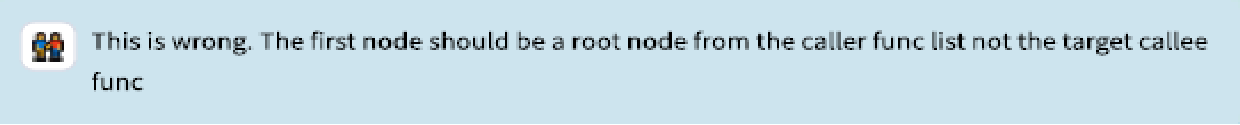
now the output contains 4 paths: "ts_tmy_func_ptr_list -> tmy_ProgramMessageControl -> utl_ParamUpdate cp_SetChargerControlConst -> dio_SetChargerControlConst -> utl_ParamUpdate ts_cp_CommandProcessing -> DetDelayStateExpired -> setDetDelayExpirationStatus -> utl_ParamUpdate ts_init_path_model -> dispatch_SystemInit -> mag_InitMagnet -> utl_ParamUpdate main -> patient_fw_main -> dispatch_SystemInit -> mag_InitMagnet -> utl_ParamUpdate"



No, there are more than 4 paths. I want all possible paths.



So, I think you have identified all of the root nodes. However, there can be multiple paths for each root node leading to the target function. You are missing ALL Possible combinations

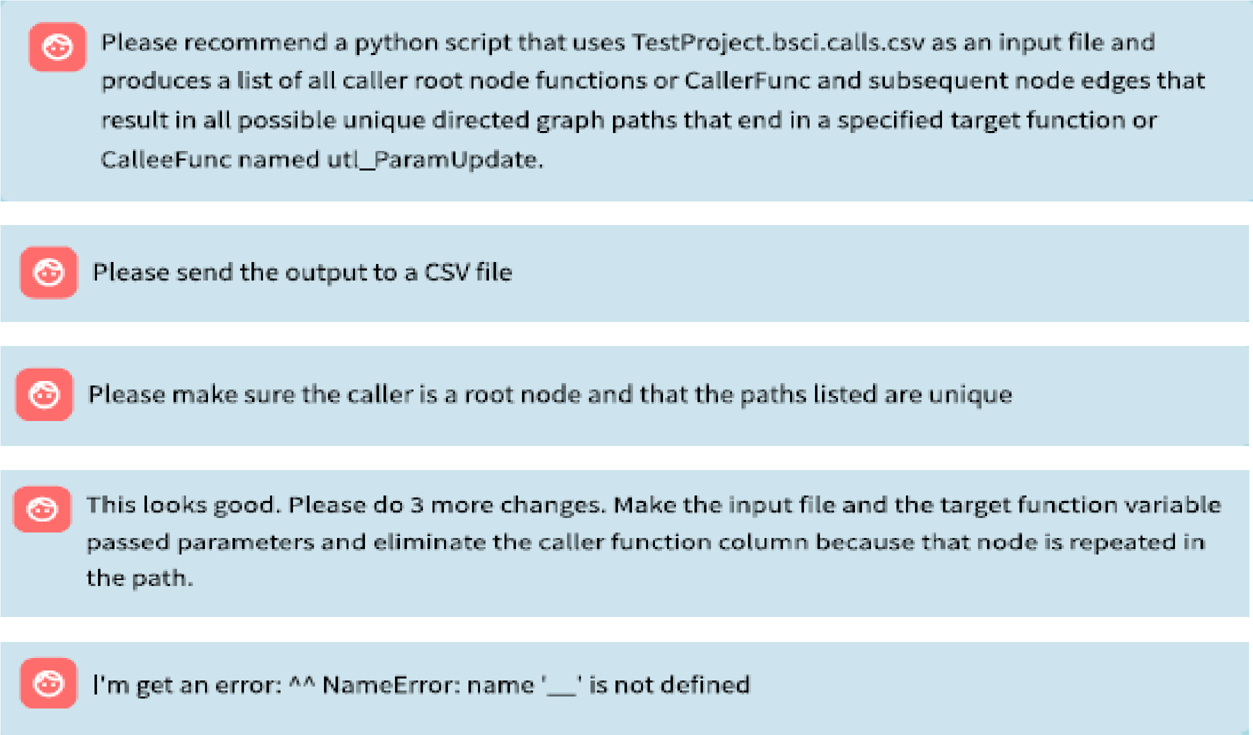


This is wrong. The first node should be a root node from the caller func list not the target callee func

6. CODE CREATION WITH ANTHROPIC HAIKU – ATTEMPT 2

For the second attempt, a different prompt more aligned with the wording from **Req1** was used. Figure 11 captures all the prompts leading up to success in this round.

Figure 11



Please recommend a python script that uses TestProject.bsci.calls.csv as an input file and produces a list of all caller root node functions or CallerFunc and subsequent node edges that result in all possible unique directed graph paths that end in a specified target function or CalleeFunc named utl_ParamUpdate.

Please send the output to a CSV file

Please make sure the caller is a root node and that the paths listed are unique

This looks good. Please do 3 more changes. Make the input file and the target function variable passed parameters and eliminate the caller function column because that node is repeated in the path.

I'm get an error: ^^ NameError: name '__' is not defined

Figure 12 captures the Python script that generated the identical output captured in Figure 9. Notably, the script in Figure 12 is quite different from the GPT-4o generated script that is shown in Figure 8. It is smaller and takes advantage of the pandas and network Python packages.

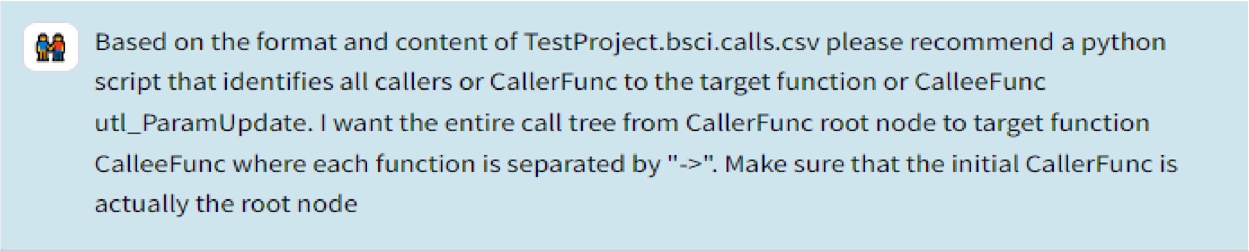
Figure 12

```
import pandas as pd
import networkx as nx
from collections import defaultdict
import argparse

def get_unique_paths(input_file, target_func):
    # Load the CSV file
    df = pd.read_csv(input_file)

    # Create the directed graph
    G = nx.DiGraph()

    # Add edges to the graph
    for _, row in df.iterrows():
        G.add_edge(row['CallerFunc'], row['CalleeFunc'])

    # Find all paths ending in the target function
    all_paths = []
    for node in G.nodes:
        for path in nx.all_simple_paths(G, source=node, target=target_func):
            all_paths.append(tuple(path))

    # Get the unique root node functions
    root_nodes = set([path[0] for path in all_paths if G.in_degree(path[0]) == 0])

    # Create a DataFrame to store the results
    results = []
    for root_node in root_nodes:
        unique_paths = set([path for path in all_paths if path[0] == root_node])
        for path in unique_paths:
            results.append({
                'Caller Path': ' -> '.join(path)
            })

    return pd.DataFrame(results)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Find unique paths to a target function')
    parser.add_argument('--input-file', type=str, required=True, help='Input CSV file')
    parser.add_argument('--target-func', type=str, required=True, help='Target function')
    args = parser.parse_args()

    df = get_unique_paths(args.input_file, args.target_func)
    df.to_csv('output.csv', index=False)
    print("Results saved to output.csv")
```

7. MORE ON PROMPTING

Given the ChatGPT-4o experiment captured in Figure 7 required three prompts, intuitively one might conclude that combining all three prompts into a single prompt would work and be the most efficient method. In practice, the intuition proved true as the consolidated single prompt shown in Figure 13 produced a script that generated the correct output file.

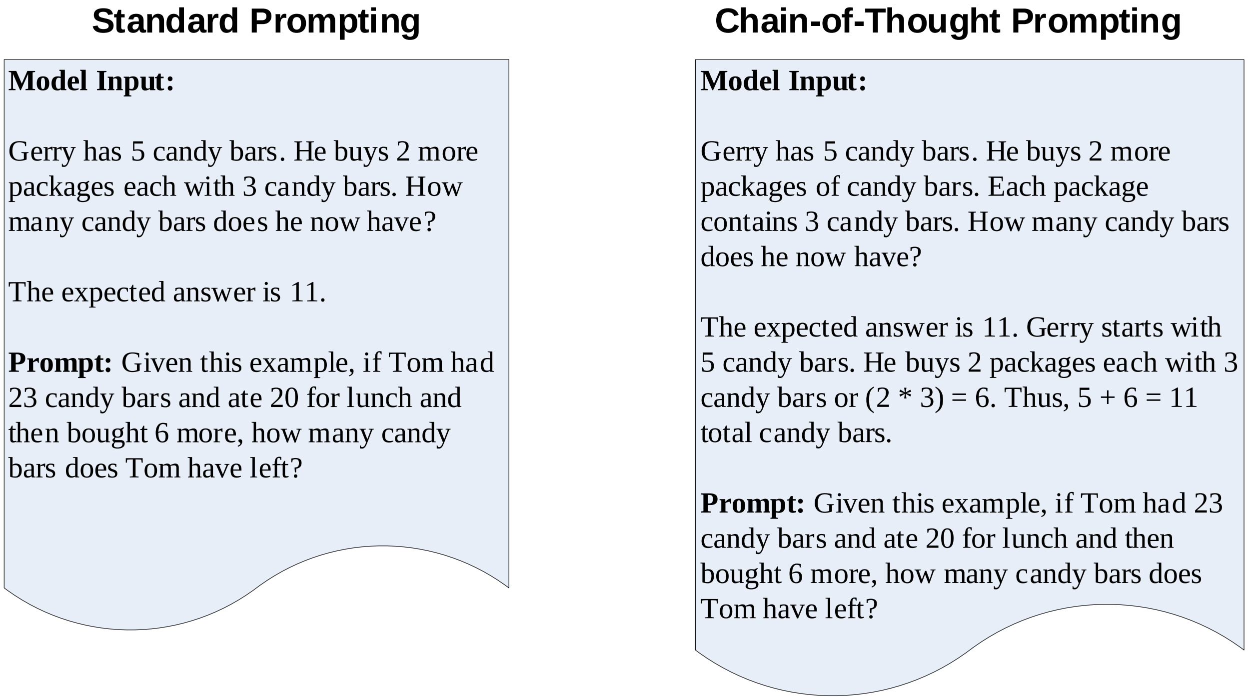
Figure 13



Based on the format and content of TestProject.bsci.calls.csv please recommend a python script that identifies all callers or CallerFunc to the target function or CalleeFunc utl_ParamUpdate. I want the entire call tree from CallerFunc root node to target function CalleeFunc where each function is separated by "->". Make sure that the initial CallerFunc is actually the root node

However, given the Anthropic Haiku Attempt 1 failure after fourteen unsuccessful prompt responses shown in Figure 10, one might think about alternative prompt strategies altogether. A well-known research paper (Wei, Wang, Schuurmans, Bosma, Ichter, Xin, Chi, Le, Zhou, 2022) draws the comparison between Standard Prompting and Chain-of-Thought Prompting. Instead of a conventional, standard, or open-ended approach to prompting that may intuitively feel "more conversational," this research suggests that using a series of intermediate reasoning steps likened to chains of logic yield better LLM responses to queries involving complex reasoning. Figure 14 highlights the additional intermediate reasoning steps.

Figure 14



Standard Prompting

Model Input:

Gerry has 5 candy bars. He buys 2 more packages each with 3 candy bars. How many candy bars does he now have?

The expected answer is 11.

Prompt: Given this example, if Tom had 23 candy bars and ate 20 for lunch and then bought 6 more, how many candy bars does Tom have left?

Chain-of-Thought Prompting

Model Input:

Gerry has 5 candy bars. He buys 2 more packages of candy bars. Each package contains 3 candy bars. How many candy bars does he now have?

The expected answer is 11. Gerry starts with 5 candy bars. He buys 2 packages each with 3 candy bars or (2 * 3) = 6. Thus, 5 + 6 = 11 total candy bars.

Prompt: Given this example, if Tom had 23 candy bars and ate 20 for lunch and then bought 6 more, how many candy bars does Tom have left?

After spending a brief amount of time with the Chain-of-Thought prompting strategy in further experiments, it became clear that more learning is necessary. Notably, it was observed that over-engineering a prompt can result in poor responses, requiring further experimentation that is out of the scope of this study.

8. TESTING CONSIDERATIONS

Testing is a critical component, whether the script is manually written or generated by an LLM. When developing a script, it is essential to verify its functionality through testing. Typically, this involves using a reduced test code base, for example, code with known call graph data output or expected results. The test design might include something like ten to twenty call paths, which are representative and sufficient enough to build confidence that the script will work correctly with a larger code base containing hundreds or thousands of call paths. Therefore, as a script is developed, it needs to be evaluated against the test code base. This testing approach is equally valid and necessary when an LLM is used to generate scripts based on prompts. In this study, the output of the manually crafted script included one-hundred-six unique paths in an actual project code base specified by requirements **Req1** and **Req2**. Consequently, this file was used for evaluating the output of the scripts generated by the LLMs because the input file was a real project file and not a test file.

9. CONCLUSION

This case study compared a traditional software engineering approach (including web search which usually involves things like the use of Stack Overflow and like forums, etc.) where one typically encounters partial solutions requiring programmer will and brute force to solve the problem, to a different approach using LLMs. The discovery revealed that what took days could be accomplished in a few hours, even minutes. The findings are summarized in Figure 15 as follows:

Figure 15

LLM	Type of Prompt	Number of Prompts	Results
ChatGPT-4o	Standard	3	Success
Anthropic Haiku – Attempt 1	Standard	14	Failure
Anthropic Haiku – Attempt 2	Standard	5	Success
ChatGPT-4o	Consolidated	1	Success
Anthropic Haiku – Attempt 3	Chain-of-Thought	5	Failure
Anthropic Haiku – Attempt 4	Chain-of-Thought	5	Failure
Anthropic Haiku – Attempt 5	Chain-of-Thought	5	Failure
Anthropic Haiku – Attempt 6	Chain-of-Thought	5	Failure
Anthropic Haiku – Attempt 7	Chain-of-Thought	5	Failure

As previously alluded to, the Chain-of-Thought experiments did not work so well. Hence, a couple of hours of additional work went into this excursion, where the threshold to declare failure was decided to be the number of prompts it took to reach success in the Standard Prompt Case. The big takeaway is that even if one spent an entire day experimenting with various prompts, for this case study it still would have reduced the overall engineering effort by two-thirds. And the upside is that time spent in prompt experimentation builds skills that will prove beneficial when confronted with the next problem in need of a solution.

10. PERSONAL THOUGHTS

During a recent conference call on the use of LLMs, I was introduced to a book on how people confront change (Johnson, 2002). After discovering it was massively popular as Amazon's best seller with millions of copies sold, I wondered how this had escaped my attention for so long. To summarize, the book is a quick read about mice and "Littlepeople" (creatures the size of mice that act like people) in a maze and how they react when their respective cheese stores unexpectedly move to an unknown location. The book allows for multiple interpretations of its message, making it a malleable guide to self-reflection on how we react and adapt to ever changing environments. While change may not always be better, it sometimes is, and inevitable, nevertheless. Echoing Meadows again, this story is another great example of an explanatory **model.**

Having been a product of the 1980s software engineering generation, I recall my hesitation when I was first introduced to the web search interface. After all, my bookcases were full of books that I had relied on for years in getting my work done. However, I soon discovered the power of the internet and there was no looking back. Admittedly, while I still hang on to many of these old texts, I rarely use them in my day-to-day work. Yet in a metaphorical sense, the web only search has become my books of the past and the LLM prompt is now the latest technology. However, changing habits is not easy, especially when going immediately to web search has become second nature. Therefore, I am actively retraining myself to make the LLM prompt my primary go-to option and other search activities, like my old books, are still there if I need them.

References

Rigdon, G. (2010, July). *Static Analysis Considerations for Medical Device Firmware*. Embedded Systems Conference Proceedings.

Rigdon, G., Doshi, H., Zheng, X. (2010, July). *Static Analysis Considerations for Stack Usage.* Embedded Systems Conference Proceedings.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xin, F., Chi, E., Le, Q. Zhou, D. (2022). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* 36th Neural Information Processing Systems Conference Proceedings.

Johnson, S. (1998, 2002). *Who Moved My Cheese.* G. P. Putnam's Sons.

Hopstock, S. (2022, November). *Call Graphs: The Bread and Butter of Program Analysis*. guardsquare.com. https://www.guardsquare.com/blog/call-graphs-the-bread-and-butter-of-program-analysis

Leveson, N. (1995). *Safeware.* Addison-Wesley.