Appendix C

Troubleshooting Code with Chatbots

ADD PROMPTING TO THIS SECTION OF THE INTRO

Chatbots backed by large language models (LLMs), like ChatGPT, are now commonly used to troubleshoot and debug code. Sometimes chatbots are used to supplement traditional approaches to scanning and interpreting Software documentation. Other times, Chatbots might be used to replace traditional approaches to troubleshooting, such as in cases where a programming language or library has a wide internet presence—such as R's tidyverse or base Python. Large amounts of code, documentation, and common usage patterns are included in LLM training data. As a result, the chatbot can often replace traditional troubleshooting methods by providing an interactive, conversational style of debugging that adapts to the user's questions and code in real time.

We do this by providing what is called a "prompt" to the chatbot, or a written out set of instructions that tells the model what task to perform. The prompt establishes the initial context the chatbot uses to generate a response, and guides how it will respond to the problem.

Important to note, while chatbots can generate responses to patterns, they do not understand our intentions. If the problem we present a chatbot looks similar to another issue, is not common and poorly represented in the chatbot's training data, or arises from a new use case, the chatbot may misinterpret our problem or offer a solution meant for a different situation.

The following sections introduces a repeatable, human-guided troubleshooting framework designed specifically for troubleshooting code with a chatbot, including practices for writing prompts. The following sections also provide a basic explanation for how chatbots process information, with the goal of providing insight to help anticipate where conversational misalignments might lead to incorrect responses—and how to prevent or correct those mistakes. Our aim is to present a framework that makes interactions with a chatbot more predictable and productive by structuring the conversation: clearly stating the problem, providing minimal reproducible code and screenshots of the data, showing the exact error, asking a focused question, testing the suggestions, and iterating until the issue resolves. This cyclical framework helps minimize the gap between what we might intend and what the chatbot thinks we are asking, turning the debugging process into a human-guided loop.

A Troubleshooting Cycle for Working with ChatGPT

1. *Describe the problem.* During this stage we can ask ourselves: what are we hoping to achieve? We can then provide a description of the problem in plain language. For example, we can write a prompt to the chatbot saying:

> "I want to count words in the 1850 Hansard debates . . ."

However, saying what we want to do alone is not enough becuase we can get our output in many ways. For example, we might want the response to be a data frame because we are working using `tidyverse` principles but chatgpt does not understand our intentions and we could get our response in different data structure like in a nested list. Therefore, we can add to our prompt the output we expect:

> " . . . and return a dataframe with the tokens and their count per speaker."

This helps to not just perform the task, extracting bigrams, which could result in many different paths. While most things in R are in dataframes, in another language bigrams could be given back in a different format.

2. Provide the Minimal Code

If possible, aim for a small example—a few lines only:

Include the function or chunk generating the error.

Exclude unrelated steps.

This helps ChatGPT reason accurately and prevents "hallucinated" fixes.

Sometimes provide it with a script or dataset, however, this can exceed the chatbot's memory, causing _____

3. Provide a Screenshot or Snippet of the Dataset
4. Provide the Exact Error Message

Copy-paste error output verbatim, including:

The first line (most important)

Any traceback shown

This helps ChatGPT see what the interpreter saw.

5. Ask a Specific Question

Examples:

"Why is this object not being found?"

"Is this a many-to-many join?"

"Why is spacyr failing to initialize?"

"How do I get this to run in PDF LaTeX?"

Precise questions lead to precise help.

5. Receive the Explanation, Try the Fix

ChatGPT will:

Propose a diagnosis

Provide corrected code

Explain why the issue happened

Run the suggestion in your environment (RStudio, Jupyter, Terminal).

Don't skip this. Your environment is the source of truth.

6. Iterate

This cycle repeats:

New error → new snippet → new explanation → new test

Iteration is normal, especially when:

Working with large humanities corpora

Using complex libraries (tidytext, quanteda, spaCy, ggplot, plotly)

Moving across environments (Windows/Mac/Linux, HPC clusters)

Working with multimodal pipelines (e.g., OCR → parsing → tokenization → modeling)

Troubleshooting becomes a collaborative conversation, not a one-off question.

8. Consolidate the Working Solution

Once it works:

Ask ChatGPT to rewrite the code cleanly

Add comments

Make it reusable (turn into a function)

This step turns messy debugging into a stable tool you can use later or teach to others.

9. Reflect on the Pattern

Ask:

"What was the underlying cause of this error?"

"How can I avoid this in future?"

"What's the general principle here?"

This helps build technical literacy rather than dependency.

Summary Cycle Diagram (Text Version)

Describe task → Provide minimal code → Provide error → Ask a focused question → Receive explanation → Test locally → Report result → Iterate → Consolidate solution →