CS6039: Advanced Software Engineering Project Report

Ben Herdman

November 2025

1 Introduction

Ensuring that software performs exactly as intended is not always achievable when given short or rapidly approaching project deadlines. Furthermore, software testing is often neglected until last minute due to the lack of emphasis it is given. Software quality often declines because of a lack of testing and strong, coherent testing frameworks. Pre-conditions and post-conditions are a quick form of consistency checking that software engineers can use. Because pre- and post-conditions are somewhat easy to define, this paper will analyze how well market-grade LLMs perform at not only finding these conditions but also writing them and converting them to executable code.

Specifically, this paper analyzes how well ChatGPT 4.0 and Copilot (2025 Q4 model on think deeper settings) will generate pre- and post-conditions for a classic linked list implementation with a slight change as well as convert the generated post conditions to executable format in the code. ChatGPT will also rate the correctness of pre- and post-conditions that Copilot generates and vice versa.

2 Problem Addressed

Since pre-conditions and post-conditions are often neglected when writing methods/functions in software projects, writing these comments is a job that would be well suited for ChatGPT and Copilot. Having an AI assistant write comments for methods/functions as a program is developed would be a significant time saver for developers. The problem addressed is analyzing how well these LLMs generate pre- and post-conditions and then using these generated post-conditions to create executable versions of them. Furthermore, this paper uses both ChatGPT and Copilot to analyze the pre- and post-conditions that each other LLM generates. This is useful as a form of consistency checking between the human-written and AI-written conditions.

3 Methodology

ChatGPT was considered to be an exciting tool and technological marvel by 135 software engineers [1] who were surveyed on how they use ChatGPT. The overall sentiment towards LLMs for code completion tasks were split between positive and negative. Many software engineers are leveraging LLMs to speed up development time. Because of this, ChatGPT should be carefully analyzed to determine if it can develop correct pre- and post-conditions. Additionally, due to the widespread use of Microsoft in the industry, Copilot is rapidly becoming integrated with their products. Both ChatGPT and Copilot can be used as development and testing tools; this paper analyzes how well they perform.

Python is the most used programming language [2] and is therefore most common in LLM training data. In order to give the LLMs the best possible chance at success, Python was used to implement a classic linked list, see Appendix A and B (add hyperlink), with a small addition of a cursor. The cursor ADT is a pointer to an index within the linked list that adheres to the standard program invariant (see Appendix A docstring for details).

There were two approaches for testing the models. First, the model was given the prompt, herein referred to as "prompt one:" "Given this project code, find and document each method's weakest possible pre-condition as well as post-conditions following the standard Python docstring format. Next, convert the post-conditions to executable format." {code}. The other method for prompting the models is to apply some prompt-engineering techniques and ask each model the following prompt, herein referred to as "prompt two:" "Given this project code, find and document each method's weakest possible pre-condition as well as post-conditions following the standard Python docstring format. Next, convert the post-conditions to executable format. For example, the weakest possible pre-condition in code is one that, when executed, will not throw any in-built exceptions. For example, NaN is not an exception in Java, so that behavior is acceptable. When creating executable post-conditions, use asserts or ifs to make sure that control is directed away from the standard path if need be." {code}. Sections 3.1 and 3.2 will cover how ChatGPT and Copilot respectively broke down and answered each problem.

3.1 ChatGPT Approach

Without any prompt engineering techniques as well as being asked prompt one, ChatGPT was given each method within the Python code and asked to add the pre- and post-conditions to the existing docstring. ChatGPT took the approach of carefully breaking down the methods and analyzing the interplay between each interface. Overall, the model worked quickly. ChatGPT provides a self-reporting metric feature which recorded that the model thought for around 5 seconds before beginning to write out results. It took 30 seconds to write all of the data and comments out. Unlike Copilot, ChatGPT correctly wrote executable post-conditions that, in some cases, properly worked and were correct. However, in some instances, it wrote tests similar to asserting that true is equal to true. See Section 4, subsection 4.1 for more information on this issue.

For prompt two, ChatGPT was given an example of a pre-condition that did not throw any in-built exceptions in Java. Furthermore, it was specified that the flow of control needs to be directed away from the standard path. The use of this language was intentional in order to have the model think more in-depth about how best to achieve the request. The model worked exceptionally quickly and broke down each method and responded individually to each method. This approach worked slightly more rapidly than prompt one.

3.2 Copilot Approach

For prompt one, Copilot did not breakdown the given code into parts; instead it returned the entire codebase which was modified. The model's self-reported metrics thought for 15 seconds and took roughly 40 seconds to write out. Copilot's overall performance when writing executable post-conditions, without any prompting, was extremely disappointing, and the results will be covered in-depth in Section 4.

Copilot did a much better job when using prompt two. It broke down the given code by method and carefully analyzed the interface. Within each method analysis, it listed all pre-conditions but meticulously selected the weakest pre-conditions from all the conditions listed. Furthermore, it did a much better job at creating executable post-conditions when given much stricter commands.

4 Evaluation

Overall, the LLMs performed as expected when given very little prompting/guidance for writing pre- and post-conditions with the notable exception of Copilot's executable post-conditions (see subsection 4.2). It is important to note that whenever the LLMs added code to the Linked List, the Python code was run through the unit test suite to ensure no implementation errors were added prior to checking the correctness of the generated pre- and post-conditions. When using prompt one, the LLMs had a much harder time identifying the weakest pre-condition. When using prompt two, the models performed much better; this is likely due to the underlying implementation of the models being able to find more data with better prompting techniques.

In addition to being tested on writing pre- and post-conditions, the LLMs were also tested on how well they could review each other's written conditions. Surprisingly, both LLMs performed perfectly when given the other's code and told to analyze the pre- and post-conditions. Whenever ChatGPT or Copilot mistakenly included a pre-condition or had a superfluous post-condition, it was detected without fail.

4.1 ChatGPT Performance

With prompt one, ChatGPT performed slightly above average (see Figure 1 for the exact performance). It is worth noting that for all of the pre-conditions generated, none of them were incorrect; they were simply not the weakest pre-condition. However, a few post-conditions were incorrect due to unnecessary assertions added to check states that were not altered. For example, in the __add__ method ChatGPT added the assertion that both items needed to be a Linked List. This is not true; the other item simply needed to be iterable in order to not cause a crash or unwanted behavior.

When given prompt two, ChatGPT provided extremely accurate results. It correctly identified all pre- and post-conditions and properly converted the post-conditions to executable format.

ChatGPT did a good job at evaluating Copilot's code. It correctly identified any extra pre-conditions or missing post-conditions without fail for every method.

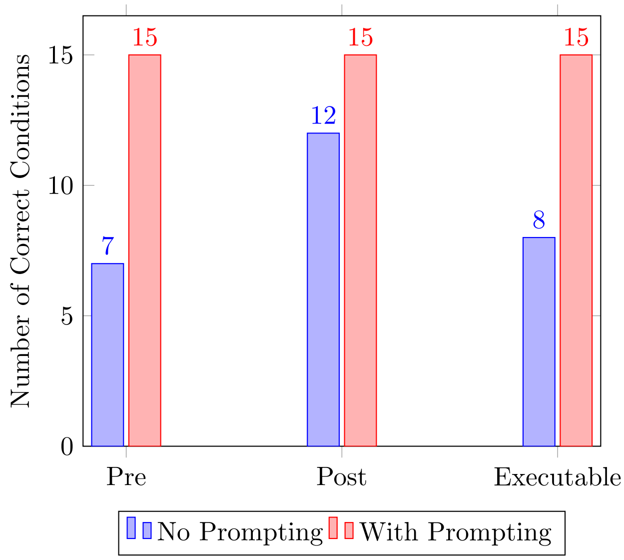

Figure 1: ChatGPT Performance

4.2 Copilot Performance

For prompt one, Copilot performed similarly to ChatGPT except when writing added executable post-conditions. Instead, Copilot added unit tests instead of simple asserts or if-conditionals. Figure 2 reports the performance of Copilot. Additionally, several of the post-conditions included conditions that would check pointers which were unmodified. For example, on the __len__ method Copilot added post-conditions that checked that the head and tail pointer were not None.

For prompt two, Copilot had an odd instance of incorrectly finding the post-conditions of methods but correctly writing the executable statement for the __iter__ and itemAtHead methods. This could partially be due to the lack of Python training data for writing iterators. Overall, it performed slightly worse than ChatGPT but still operated at acceptable levels of correctness.

Copilot correctly identified all incorrect pre- and post-conditions for ChatGPT's prompt one comments and verified that all of prompt two's comments were correct.

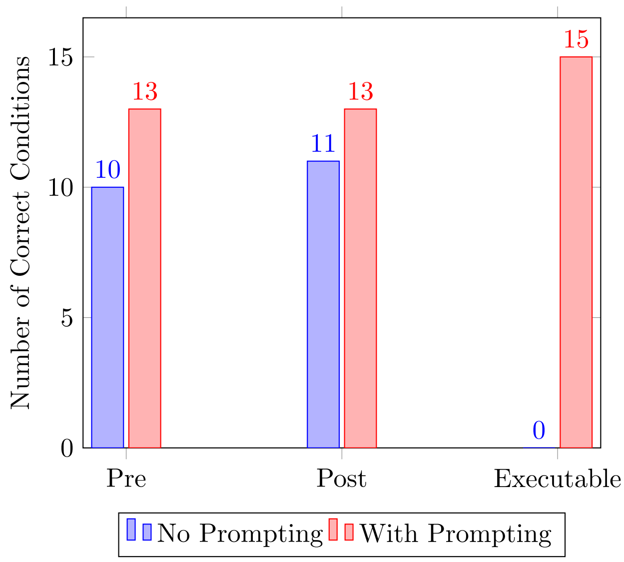

Figure 2: Copilot Performance

5 Conclusions and Future Work

Both models performed at, or above, expectations when it came to finding and documenting pre- and post-conditions for Python methods. As of December 2025, no public API exists for either LLM, but if one becomes available then further testing will be much easier to evaluate. Overall, with prompt one, both models were unable to consistently identify the weakest pre-condition without adding unnecessary tests. However, this performance substantially improved with prompt two, which is likely due to including an example of a weakest pre-condition. The most significant result of this study is that in order to guarantee the highest degree of confidence when using LLMs to write the pre- and post-conditions for code, a robust statement of exactly what is wanted and several examples for the LLMs to work with must be provided.

Some interesting future work once a public API becomes available would be to feed ChatGPT and Copilot responses into a Neural Network (NN) to evaluate those responses instead of performing a manual review. Furthermore, the data fed to a NN, along with pre-labeled data, could be leveraged to create a custom tool that specializes in finding the pre- and post-conditions in code and adding those comments as a developer works.

6 References

[1] ChatGPT Incorrectness in Software Reviews and software engineers' thoughts towards ChatGPT. `https://arxiv.org/abs/2403.16347`. Accessed: 2025-11-15.

[2] TIOBE index for programming languages. `https://www.tiobe.com/tiobe-index/`. Accessed: 2025-11-26.

[3] Microsoft documentation for adding Copilot extensions. `https://learn.microsoft.com/en-us/microsoft-365-copilot/extensibility/`. Accessed: 2025-11-26.

[4] Github repository for project, code, and readme. `https://github.com/herdbv/UCCS6030-FinalProject`

Appendix A

```python
#!/usr/bin/env python3

# ----------------------------------------------------------------------
# LListCursor.py
# Ben Herdman
# ----------------------------------------------------------------------


from __future__ import annotations
from typing import Optional, Any

from ListNode import ListNode


# ----------------------------------------------------------------------


class LListCursor:
    """LListCursor is a linked list where you can add/remove/access
    items at beginning, end, and a _cursor position in the list

    class invariant:

        1. if the list is empty, self._head, self._cursor, self._tail are
        all None

        2. if the list is not empty, self._head, self._cursor, and
        self._tail all point to an appropriate ListNode; self._head
        points to the first ListNode; self._tail points to the last
        ListNode; self._cursor points to a ListNode in the list

        3. inserting an item should not change self._cursor unless the
        list was empty, in which case self._cursor points to the one
        item in the list

        4. when deleting at the _head or _tail, self._cursor stays where
        it is unless it was at the _head or _tail; if _cursor was at the
        _head and the _head was deleted, self._cursor now refers to the
        ListNode after it. if self._cursor was at the _tail and the _tail
        is deleted, self._cursor now refers to the ListNode before it

        5. when deleting the item at the _cursor, self._cursor now
        refers to the ListNode after it, unless there is no ListNode
        after it in which case it refers to the ListNode before it.

        6. self._length indicates the number of items in the LListCursor
        """

    # ------------------------------------------------------------------

    def __init__(self, *args):
        """
        initializes empty list or list with items in args if it is not None; the _cursor
        will be the first node
        :param args: sequence of items to insert into the list
        """
        self._head: Optional[ListNode] = None
        self._cursor: Optional[ListNode] = None
        self._tail: Optional[ListNode] = None
        self._length: int = 0
        # if only one argument, see if it is iterable
        if len(args) == 1:
            try:
                # try to insert each item
                for x in args[0]:
                    self.insertAtTail(x)
            except TypeError:
                # exception raised if item not iterable so just insert it
                self.insertAtTail(args[0])
        else:
            # 0 or 2 or more arguments so iterate over them and insert them
            for x in args:
                self.insertAtTail(x)
        self.cursorToStart()
```

```python
    def __len__(self) -> int:
        """
        :return: number of items in the list
        """
        return self._length

    def __iter__(self):
        """
        iterates over items in list yielding one item at a time
        """
        # start at beginning of list
        node = self._head
        # while nodes left
        while node is not None:
            # yield item
            yield node.item
            # and move node forward
            node = node.link

    def __add__(self, other: LListCursor) -> LListCursor:
        """
        returns a new LListCursor that is the concatenation of self and other
        :param other: another LListCursor to concatenate with self
        :return: a new LListCursor that is concatenation of self and other; the
        _cursor of it should be at the beginning of the list
        """
        # make a new LList
        newList = LListCursor()

        # append the objects from the first list
        for x in self:
            newList.insertAtTail(x)

        # append the objects from the second list
        for x in other:
            newList.insertAtTail(x)

        # point the cursor to the head of the new list and return it
        newList._cursor = newList._head
        return newList

    # -----------------------------------------------------------------

    def insertAtHead(self, item: Any) -> None:
        """
        inserts item at the beginning of the list
        :param item: value to insert
        :return: None
        """
        # if length is 0 then the list is empty
        if self._length == 0:
            # insert in the item and point everything to the head
            self._head = ListNode(item)
            self._tail = self._head
            self._cursor = self._head

        else:
            # create a "temp" var to store the old head
            prevHead = self._head
            # create the new head
            self._head = ListNode(item)
            # link the old head to the new head
            self._head.link = prevHead

        self._length += 1

    def insertAfterCursor(self, item: Any) -> None:
        """
        insert item after the _cursor position
        :param item: value to insert
        :return: None
        """
        if self._cursor == self._tail:
            self.insertAtTail(item)
        else:
            self._length += 1
            # list is not empty since _cursor == _tail if it is and _cursor not at
```

```python
    _tail
            # create node
            node = ListNode(item, self._cursor.link)
            # connect _cursor to the new node
            self._cursor.link = node

    def insertAtTail(self, item: Any) -> None:
        """
        insert item at the end of the list
        :param item: value to insert
        :return: None
        """
        # if the length is zero the list is empty
        if self._length == 0:
            # create the head and point all other vars to it
            self._head = ListNode(item)
            self._tail = self._head
            self._cursor = self._head

        else:
            # list is not empty, create the tail and link it to the structure
            self._tail.link = ListNode(item)
            self._tail = self._tail.link

        self._length += 1

    def removeItemAtHead(self) -> Any:
        """
        removes first item in the list; IndexError is raised if list is empty
        :return: the item that was removed
        """
        if self._length == 0:
            # raise IndexError if list is empty
            raise IndexError('removeItemAtHead called on empty LListCursor')
        else:
            self._length -= 1
            # get item so can return it later
            item = self._head.item
            # if list is empty after the deletion
            if self._length == 0:
                # make all ListNode instance vars None
                self._head = self._cursor = self._tail = None
            else:
                # if _cursor was at _head
                if self._cursor == self._head:
                    # move _cursor forward to new first item
                    self._cursor = self._cursor.link
                # move _head forward to new first item
                self._head = self._head.link

            return item

    def removeItemAtCursor(self) -> Any:
        """
        removes item in the list that is at the _cursor; IndexError is raised if list
     is empty;
        the _cursor now points to the node after the original _cursor unless the
    _cursor was the
        last item in which case the _cursor is now the new last item
        :return: the item that was removed
        """

        # Raise exception if length is zero
        if self._length == 0:
            raise IndexError('removeItemAtCursor called on empty LListCursor')

        # if the cursor is at the tail call the tail function
        elif self._cursor == self._tail:
            item = self.removeItemAtTail()

        # if the cursor is at the head call the head function
        elif self._cursor == self._head:
            item = self.removeItemAtHead()

        else:
            # otherwise decrement the length
            self._length -= 1
```

```
            # save the item for return
            item = self._cursor.item

            # if the length is zero after decrement destroy the list
            if self._length == 0:
                self._head = self._tail = self._cursor = None

            # if deleting one of two items, destroy the list and leave the head
            elif self._length == 1:
                self._head.link = None
                self._cursor = self._tail = self._head
                return item

            else:
                # create variables to call out of conditionals
                tracker = self._head
                nodeLinkToDestroy = tracker
                # go forwards through links
                while not (tracker == self._cursor):
                    # make a copy of the previous node
                    nodeLinkToDestroy = tracker
                    # move the tracker forward
                    tracker = tracker.link

                # move the cursor forward as per the conditions of Invariant
                properCursorLocation = self._cursor.link

                # destroy the prevNode's link
                nodeLinkToDestroy.link = None
                # relink the list
                nodeLinkToDestroy.link = properCursorLocation
                # set the cursor to the proper node
                self._cursor = properCursorLocation

        return item

    def removeItemAtTail(self) -> Any:
        """
        removes last item in the list; IndexError is raised if list is empty
        :return: the item that was removed
        """

        # list is empty, throw error to crash gracefully
        if self._length == 0:
            raise IndexError('removeItemAtTail called on empty LListCursor')

        else:
            # create proper length since we are removing an item
            self._length -= 1
            # save the item for return
            item = self._tail.item

            # if deleting the last item destroy the list
            if self._length == 0:
                self._head = self._cursor = self._tail = None
                return item

            # if deleting one of two items, destroy the list and leave the head
            elif self._length == 1:
                self._head.link = None
                self._cursor = self._tail = self._head
                return item

            else:
                # create variables to call out of conditionals
                tracker = self._head
                prevNode = tracker
                count = 0
                flagOnCursor = False
                # if the cursor is at the end, send it to the start
                if self._cursor == self._tail:
                    self.cursorToStart()
                    # flag that the cursor needs moved
                    flagOnCursor = True

                # go forwards through links
                while tracker != self._tail:
```

```
                    # make a copy of the previous node
                    prevNode = tracker
                    # move the tracker forward
                    tracker = tracker.link
                    # keep track of how many links there are to move the cursor if
needed
                    count += 1

        # if the cursor was flagged for being at the end of the list
        if flagOnCursor:
            for i in range(count):
                # move the cursor forward based on count variable
                self.cursorForward()

        # destroy the prevNode's link
        prevNode.link = None
        # set the tail to the unlinked node
        self._tail = prevNode
        if flagOnCursor:
            self._cursor = self._tail
        # return the item
        return item

    def itemAtHead(self) -> Any:
        """
        returns first item; IndexError is raised if list is empty
        :return: first item in list
        """
        # Raise exception if length is zero, otherwise return item
        if self._length == 0:
            raise IndexError('itemAtHead called on empty LListCursor')

        return self._head.item

    def itemAtCursor(self) -> Any:
        """
        returns item at _cursor; IndexError is raised if list is empty
        :return: item at _cursor
        """
        # Raise exception if length is zero, otherwise return item
        if self._length == 0:
            raise IndexError('itemAtCursor called on empty LListCursor')

        return self._cursor.item

    def itemAtTail(self) -> Any:
        """
        returns list item; IndexError is raised if list is empty
        :return: first last in list
        """

        if self._length == 0:
            raise IndexError('itemAtTail called on empty LListCursor')

        return self._tail.item

    def cursorToStart(self) -> None:
        """
        move _cursor to start/_head of list
        :return:
        """
        # sends the cursor to the head to re-iterate
        self._cursor = self._head

    def cursorForward(self) -> bool:
        """
        move _cursor forward one item
        :return: True if _cursor was moved forward or False if list empty or _cursor
already at end of list
        """
        # set the cursor move check to false as it hasn't been updated
        # if the length is zero the list is empty
        if self._length == 0:
            didCursorMove = False

        # if the cursor isn't at the tail move the cursor forward
        elif self._cursor != self._tail:
```

```
            self._cursor = self._cursor.link
            didCursorMove = True

        # meant to handle if the cursor is at the tail, but acts as catch for other edge incidents
        else:
            didCursorMove = False

        return didCursorMove
# ---------------------------------------------------------------------
```

Appendix B

```python
class ListNode:
    def __init__(self, x):
        self.item = x
        self.link = None
```