

TESTING REPORT

Twitter-Like API w/ SQLite

Ayesha Aamer, Ruchali Aery, Francis Garcia, Ayra Qutub, Cejiro Roque

Document tabs

Tab 1

SOFTWARE TESTING AND MAINTENANCE

Introduction

Program

Testing Strategies

Testing, Analysis, and Findings by System File/Functionality

Main

Login + Register

Display Tweets

Search Tweets

Search Users

Compose Tweet

List Followers

References

Appendix

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

Appendix F

Appendix G

Introduction

Program

The program under test is a Command Line Interface implementation of a social media platform similar to Twitter. The program is implemented with Python and SQLite. The program interfaces with a database with the following structure, with keys indicated by an asterisk.

Table 1 Users	Table 2 Tweets	Table 3 Retweets	Table 4 Mentions	Table 5 Hashtags	Table 6 Includes	Table 7 Lists	Table 8 Follows
User ID*	Tweet ID*	User ID*	Tweet ID*	Term*	List name*	List name*	Follower*
Password	Writer	Tweet ID*	Term*		Member*	Owner	Followee*
Name	Tweet date	Retweet date					Start date
Email	Text						
City	Reply To						
Timezone							

Upon start of the program, the user is prompted with the choice of registering for an account or logging in to an existing account. They are then greeted with tweets and retweets of users that they follow, along with a menu where the user can pick one from the following options for the associated choices; option 1: View tweets/retweets, option 2: Search for tweets by keywords (sorted by date) - View detailed info on selected tweets, option 3: Search for users by name or city. - View user details (tweet count, followers, following, latest tweets). Option to follow users or view more tweets, option 4: Compose and post tweets. - Followers can view and search these tweets, and option 5: View list of followers. - Select a follower to view their details or tweets.

Testing Strategies

We used both black box and white box testing methods to test this program. Black box testing is a testing method in which the tester has no access to the code, this is a validation testing method that focuses on functional and behavioral testing. White box testing is a testing method in which testers focus on the internal structure of the code with access to source code, a verification process where when given knowledge of the internal workings, you thoroughly test what is happening inside. We employed the following testing techniques from these methods:

Black Box

- Error Guessing: A software testing technique that uses a tester's experience and intuition to anticipate and uncover bugs in a program.
- Failure “Dirty” Testing: Where testers think diabolically and test every possible thing a user could do with the system to demolish the software.
- Equivalence Classes : A technique built to minimize the number of required test cases by forming partitions or equivalence classes of the input space.

White Box

Coverage Testing employed by Python Unit Tests and Mocking

- Branch Coverage: A testing technique where every branch in the code has to be executed at least once, for each decision point both true and false values need to be tested.
- Statement Coverage: A testing technique where every statement in the code is executed once.
- Unit Tests & Mocking: Unit Tests in Python is a process of testing individual units or components of a program to ensure they work as expected. Mocking Is a technique used in unit testing to simulate external dependencies allowing for items to be tested in isolation.

ChatGPT Usage

Generating Black-Box Test Cases with ChatGPT

ChatGPT was used to design black box test cases for all 7 (main, compose tweets, display tweets, list followers, search users, search tweets, login/register) main functionalities of the tested program. When prompting it was made sure that only the functionality of the specific part to be tested was described, without including any information on code and the database schema. Based just on the description of functionality, ChatGPT was able to generate test cases that included;

- Failure Testing: ChatGPT was tasked with generating test cases designed to cause the system to fail or expose vulnerabilities. Examples included; Extremely long keywords to test system limits, invalid characters or malformed inputs, empty or null keyword inputs to test how the system handles missing data.
- Error Guessing: ChatGPT was asked to create test cases by predicting common errors users might make. These included; misspelled or partially entered keywords, keywords with extra spaces, tabs, or control characters, and keywords combining valid and invalid elements.
- Equivalence Partitioning: ChatGPT was instructed to partition the input space into equivalent classes to ensure

comprehensive coverage with little redundancy. Test cases generated included; valid/invalid inputs, and boundary cases.

Generating White Box Test Cases with ChatGPT

ChatGPT was used to design white box test cases for all 7 (main, compose tweets, display tweets, list followers, search users, search tweets, login/register) main functionalities of the tested program. Additionally, it was also used to generate python code for unit testing. When prompted it was made sure that the functionality of the specific part to be tested was described, including information on code and the database schema. Based just on the description of functionality, ChatGPT was able to generate unit test that included:

- Tests for branch and statement coverage, ensuring all paths in the function were exercised.
- Error handling scenarios, such as database connection failures or invalid input types.
- Security-focused tests to validate input sanitization and resilience against SQL injection.

The generated code was implemented with minimal to heavy modifications and executed within the project's testing environment.

Documentation and Traceability

All messages and prompts were logged to ensure all outputs are traceable and could be used to assess the quality of tests produced by ChatGPT, such as: the functional descriptions provided for black-box testing, the high-level logic and behavior outlined for white-box testing, the generated test cases and unit testing code and notes on any refinements or modifications made before implementation.

Testing, Analysis, and Findings by System File/Functionality

A. Main

The main.py file served as the entry point for this program prompting users for inputs and calling the correct functions to carry out user requests. That being said the functionality and set-up for this file differed greatly from the others leading to difficulties with testing through ChatGPT.

Black Box Testing:

In order to perform the Error Guessing, Failure and Equivalence class testing techniques on this portion of the program, ChatGPT was given context of the problem and asked to generate test cases in a tabular format. The Prompt passed to ChatGPT can be found as Prompt 1A in Appendix A. The key parts of this prompt are the expansion of the context of the program, what it is meant to do, the various menu options, definitions of the testing methods and the explanation of how the results should be presented. ChatGPT responded with a short description of the testing methods and test case table format. After manually running the tests that were described we added the Actual result column to the table below - failures/faults/errors are highlighted.

Table 1A: Portion of the black box testing test cases created by ChatGPT and manually tested for main.py, full table in Appendix A.

Test ID	Category	Input	Description	Expected Result	Actual Result
F2	Failure	User enters an unsupported database file format (e.g., database.txt).	Tests behavior when a non-SQLite database file is entered.	Displays an error message: "Unsupported database file format. Please provide a valid SQLite database."	System allows txt file to be treated at database
E2	Error Guessing	User enters a valid menu option but inserts leading/trailing spaces (e.g., " 1 " or " 6 ").	Tests handling of inputs with extraneous spaces, which might cause parsing issues.	The program trims spaces and correctly interprets the input, triggering the menu option	Program does not trim response but asks for a new input
EC1	Equivalence	User enters a valid SQLite database file path (e.g., socialmedia.db).	Tests valid equivalence class for initializing the program with an accessible database file.	Program initializes successfully and displays the main menu.	Program continues

The black box testing methods worked well for testing the functionality of this piece of the program, they tested different scenarios and were on par with what we would have come up with ourselves in terms of test cases meaning ChatGPT was successful in this sense. Errors in the program were also found using the failure and error guessing methods, such that files were not correctly checked to be correct database file before being passed to the program (F2), and the other is input with extra white spaces were not trimmed (E2), this is not necessarily an error but uncovered a poor design choice in the system. As seen in the "Expected Results: columns ChatGPT often made assumptions on what the exact error message would be however even if the message differed but the effect was the same those cases fall under successes. The equivalence class method also worked successfully in creating valid and invalid scenarios for the main function. Overall the employed black box methods worked well for this system and ChatGPT was successful in creating suitable test cases.

White Box Testing:

Since main was not a function of itself but rather served as the entry point for the entire program virtually just displaying the menu options and then calling the correct function to handle the specific user inputs for that portion of the programs functionality, employing coverage testing was not the best approach as seen by the results of this portion of testing. ChatGPT was used to create the Python unit tests to test the main file to do Prompt 2A in Appendix A was passed. The key components of this prompt are the attached source code, the definition of coverage, explicitly defining that mocks should be used (especially for database connections), as well as how to format the input. The result was a python unit test file with various test cases Figure 1A and Table 2A in Appendix A. Table 2A itself outlines decent coverage of the functionalities in the main.py file however the actual test code took far too long to instantiate such that they could not be run, even though reading the test cases they seemed sufficient they called for user input and still proceeded to be unable to run. As such a follow up prompt was sent to ChatGPT to clarify instructions.

The new prompt was as follows: *“These tests must be able to run without user input, the tests you have currently passed take far too long to instantiate and nothing happens, try bypassing file input and db connection with mocks. Rewrite the test case file to fix this and be sure to test the menu options”* The result of this prompt was the unit test code in Figure 2A in Appendix A and Table 3A where we have added the Actual Results column after running the test cases.

Table 3A: Portion of ChatGPT generated test case table for coverage testing unit tests employed for main.py, full table in Appendix A.

Test ID	Test Description	Inputs	Expected Results	Actual Results
TC001	Test valid database file	test.db	mock_exists should return True; sqlite3.connect should be called with test.db as argument.	Test case passed
TC003	Test login choice (successful login)	1 (Login choice), 1 (Display Tweets), 6 (Logout)	mock_login should return 1 (user ID); display_tweets_and_retweets should be called with conn and 1 as arguments.	Test case passed

In the scenario outlined by Table 3A, all of the tests ran and passed but only amounted to ~45% coverage of main.py. Once again the tests asked for user input and the path that looked at the main menu required user login which again took far too long to instantiate and thus could not be effectively tested by this method. A better way to complete testing for this portion of the program would be to look more at functionality and mock more aspects of the program - the issue was without mocking, instantiation would take too long but with mocking there was no way to get real coverage of main.py. Further prompts were also passed to the ongoing sessions with ChatGPT to explicitly fix the issues; prompts with details; such as: *“Do not allow the tests to call for user input”, “The goal is to complete statement coverage on main.py”, “Mock a logged in user and DB connection, just test that the main menu calls the correct functions”*. However even with those details passed ChatGPT failed to provide better results than the ones showcased here. All in all we would not recommend coverage testing be used for this specific portion of the program even though it was successful in other aspects.

Taking both methods and their respective techniques into consideration, the black box methods worked well for this portion of the program and tested the program thoroughly even finding faults while the white box method was not as successful and has room for improvement.

B. Login + Register

This part of the program is responsible for the creation and access of user entries in the database. A user can register to use the program by providing their personal information, such as their name and email address, and by setting a password. If they already have an account, they can log in to the program using their username and password.

Black Box Testing

To prompt ChatGPT to write test cases for this part of the program, we specified the testing methods, the format of the test table, and the functionality the program should perform. We also provided the information of some existing users in the database so that it could test the login functionality. The prompt is presented as Prompt 1B in Appendix B. ChatGPT produced the tests as presented in Table 1B, which were run to produce the output as seen in the following table. Of the 12 produced tests, six of them pass. The reason for this is that the agent provided test cases which test for validation logic not present in the program. The prompt does not request for such tests but the agent produces them anyways. Disregarding the input validation type tests, the agent produces correct tests. These correctly test for the cases where a user logs in with valid credentials, when a user attempts to log in with invalid credentials and when a user registers for a new account.

White Box Testing

To prompt ChatGPT to write test cases for this part of the program, we specified the testing methods, the format of the test script, and the type of coverage the tests should cover. We also provided the code itself for this part. The prompt we provided to ChatGPT is presented as Prompt 2B in Appendix B. The output code generated by ChatGPT is presented in

Figure 1B. Of the 6 tests produced by ChatGPT, 4 passed. Again, the failing tests did so because they attempted to test input validation. In this case, ChatGPT was provided with the source code but it still produced tests for a nonexistent part of the program. The remaining tests correctly test for the cases where a user logs in with valid credentials, when a user attempts to log in with invalid credentials and when a user registers for a new account. The tests provide 82% line coverage.

C. Display Tweets

The `display_tweets_and_retweets` function enables users to view tweets or retweets from followed accounts in a simple Twitter-like program. Users select tweets or retweets via a menu, and redirect them to the main menu when an invalid input is entered. Tweets and retweets are displayed in five, with options to view more. Users can reply/retweet the selected tweet.

Black Box Testing:

Black box testing was applied using equivalence class partitioning, error guessing, and failure testing. We gave ChatGPT a general/high-level context about the system being tested and asked to generate test cases for each of the black box techniques we have chosen. A single prompt was passed to ChatGPT which can be found as Prompt 1D in Appendix D. ChatGPT produced a table of these test cases and were then tested by the group to check the actual output.

Table 1C: A portion of the complete test cases generated by ChatGPT. The full table can be found in Appendix C.

Category	Test Case Numbers	Examples
Equivalence Class Testing	1, 2, 3, 8, 9, 10, 18, 19	Invalid inputs, Valid inputs for tweets/retweets selection, long tweets (Test Case 8), and boundary cases (Test Case 18).
Error Guessing	4, 6, 7, 11, 14, 15, 16	SQL injection (Test Cases 6 and 7), rapid inputs (Test Case 16).
Failure Testing	5, 12, 13, 14, 17, 20	Invalid tweet ID (Test Case 5), empty input (Test Case 17), replying to non-existent tweets (Test Case 20).

Equivalence class partitioning ensured representative inputs were tested without redundancy. For example, invalid menu options (such as '3', 'a') were tested in Test Case 1, while valid and malformed tweet IDs were covered in Test Case 5. Boundary conditions, such as maximum-length tweets (Test Case 8) and exactly five tweets (Test Case 18), verified edge case handling. Error guessing test cases covered unexpected input. SQL injection attempts (Test Cases 6 and 7) confirmed the system prevents code injections that might change the database. Tests for rapid inputs (Test Case 16) and non-English responses (Test Case 11) covered how the program handles the type of inputs a user may enter. Failure testing also covered invalid scenarios like incorrect input sequence responses (Test Case 4) and replying to non-existent tweets (Test Case 20). These tests verified the system handled errors with clear feedback (error messages) and proper redirection. The black box test cases generated by ChatGPT for this module (*see Table 1C*) effectively validated this module's functionality, error handling, and security. Limitations included the inability to fully test large datasets (Test Case 19), though the tests were successful in a smaller database. Overall, the tests successfully ensured the reliability and robustness of the system. Further testing for large-scale and multi-user scenarios would strengthen performance validation.

White Box Testing:

White box testing was used to ensure comprehensive branch coverage (every branch in the code is executed at least once) and statement coverage (every statement is executed at least once). We provided ChatGPT with detailed information about how the code works for `display_tweets` module, including its logic, branches, and function calls, and asked it to generate unit test code. Mocking was explicitly required to simulate user inputs, database operations, and external function calls like `compose_tweet`. A single prompt was passed to ChatGPT; *"I have a Python-based social media application with the following functionality: display_tweets_and_retweets: presents a menu for users to view tweets or retweets. Based on the user's input ('1' for tweets and '2' for retweets), it calls either display_tweets or display_retweets. Invalid inputs return the user to the main menu. display_tweets: Fetches tweets from accounts followed by the user, ordered by date, and displays them in batches of five. If no tweets are available, it informs the user. Users can navigate through pages of tweets, select a specific tweet by ID for more options, and reply or retweet. display_retweets: Similar to display_tweets, but handles retweets instead. Displays retweets in batches of five with navigation options. I need unittest code that achieves branch coverage and statement coverage. Use mocking to simulate database connections, user inputs, and external function calls (compose_tweet, retweet, etc.). Ensure that all edge cases (e.g., no tweets available, invalid inputs, SQL injection attempts, empty tweets) are covered."*

ChatGPT produced a table of these test cases and the code was then run by the group to check the actual output. (Please see Table 2C and Figure 1C in Appendix C for the complete test cases and unittest code given by ChatGPT)

Table 2C: Summary of Table 2C. Categorized test cases. Full table is shown on Appendix C

Category	Test Case Numbers	Examples
Branch Coverage	1, 2, 3, 4, 5, 6, 7, 9	Pagination logic (Test Case 4), menu option handling (Test Cases 1-3), tweet interactions (Test Cases 5-6), SQL handling (Test Case 7).

Statement Coverage	8, 10, 11	Special cases (e.g., special characters in tweets), database updates (Test Case 10), and information retrieval (Test Case 9).
Edge Case Validation	2, 3, 8, 5, 7, 11	No tweets/retweets available (Test Cases 2-3), invalid inputs (Test Case 5), special characters (Test Case 11).

White box testing used branch coverage to validate decision points (e.g., invalid inputs in Test Case 1, pagination in Test Case 4), statement coverage to ensure all lines executed (e.g., no tweets in Test Case 2, special characters in Test Case 8), and edge case validation for scenarios like invalid IDs in Test Case 5. ChatGPT performed well for the module but lacked full system context, leading to some unrealistic output.

D. Search Tweets

The `search_tweets.py` function allows users to search for tweets based on keywords and retrieves tweets containing the given keywords in their text, hashtags or mentions; it also orders the results by date in descending order. The user can input one or more words and then view a list of the tweets that match the inputted keyword(s). It can (should) handle various inputs such as empty or invalid, non-existent keywords, special characters, or sql injection attempts.

Black box testing:

In order to perform black box testing, the Prompt 1D from Appendix D was fed into chatGPT for it to produce an adequate list of test cases without having seen the inside code.

I expected ChatGPT to provide a structured set of test cases exploring normal operations (e.g., single or multiple keywords), invalid and empty inputs, complex keywords (long strings, special characters), and attempts at SQL injection. The test cases should reflect distinct equivalence classes (valid/invalid), anticipate errors users might make, and attempt to cause failures.

Table 1D: A portion of the resulting table, showing failing tests for the black box testing method; the full table can be found in Appendix D

Test Case #	Inputs	Description	Expected Results	Actual Results
F-ST06	keywords = "" OR '1'='1"	SQL injection attempt using boolean logic to return all tweets.	Input treated as literal, no SQL injection. Possibly returns normal results or "No matching tweets found."	Lists two tweets (FAILED)
F-ST13	keywords = "%%%%%%%%%"	Only '%' characters (SQL wildcard attempt).	Interpreted as literal. Likely no results, safe handling. "No matching tweets found."	Lists tweets that do not contain '%' (FAILED)

F-ST01 tests security, ensuring no malicious code runs. F-ST02 tests performance and stability under extreme input length. These confirm the system's robustness against attempts to "break" it, however the system is not free from SQL injections, as seen in test cases F-ST06 and F-ST13.

Table 2D: A portion of the resulting table, showing error guessing tests for the black box testing method; the full table can be found in Appendix D

Test #	Inputs	Description	Expected Results	Actual Results
EC-ST08	keywords = "python #AI"	Multiple keywords mixing text and hashtag.	Returns tweets matching "python" or "#AI".	Returns tweets matching "python" or "#AI" as expected. (Pass)
EC-ST16	keywords = "#Python"	Keyword starting with '#', testing hashtag scenario again.	Returns tweets tagged "#Python" or no matches.	Returns tweets tagged "#Python" if available, else "No matching tweets found." (Pass)
EC-ST19	keywords = "unicodeÜ"	Keyword with accented/unicode character in last position.	If no tweet matches "unicodeÜ", no results. No crash.	"No matching tweets found." unless tweets contain "unicodeÜ" exactly. (Pass)

E-ST08 checks handling of multiple keywords mixing text and hashtag., E-ST16 tests a keyword starting with a hashtag, and E-ST19 tests unusual character inputs. These confirm the system's resilience in common or slightly odd real-world inputs.

Table 3D: A portion of the resulting table, showing equivalence class tests for the black box testing method; the full table can be found in Appendix D

Test Case #	Inputs	Description	Expected Result	Actual Results
EC-ST01	keywords = "Python"	Valid single keyword, normal case.	Returns tweets containing "Python" or "No matching tweets found."	Returns tweets containing "Python" if available, else "No matching tweets found." (Pass)
EC-ST03	keywords = "python ai"	Multiple valid keywords (both common words).	Returns a union of tweets containing "python" or "ai".	Returns tweets containing either "python" or "ai" as expected. (Pass)

EC-ST15	keywords = "python ai user"	Multiple keywords more than two.	Returns tweets containing any of "python", "ai", or "user".	Returns tweets containing "python", "ai", or "user" as expected. (Pass)
---------	--------------------------------	----------------------------------	---	---

EC-ST01, EC-ST03, and EC-ST15 cover valid classes (single or multiple keywords)

The test cases from the failure testing method aimed to challenge the system with extreme, malicious, and very abnormal inputs to expose potential vulnerabilities. For example, Test Case F-ST01 involved an SQL injection attempt using malicious keywords ("'; DROP TABLE tweets; --") to verify if the program could effectively prevent such attacks. The expected result was that the input would be sanitized and result in a safe error message like "No matching tweets found.". However, not all failure tests were successful, for example, test Case F-ST06 ("'; OR '1'='1") attempted to change the SQL logic to return all tweets, expecting the input to be treated as a literal string.

Error guessing methods focused on typical user mistakes or unusual inputs that could lead to unexpected behavior. For example, Test Case EC-ST08 tested multiple keywords mixing text and hashtags ("python #AI"), , Test Case EC-ST16 examined a keyword starting with a hashtag ("#Python"), and Test Case EC-ST19 checks keywords with an accented Unicode character ("unicodeÜ") to make sure it is handled by the system properly. Equivalence partitioning also categorized inputs into distinct classes to ensure comprehensive coverage. Test Case EC-ST01 involved a valid single keyword ("Python"), Test Case EC-ST03 used multiple valid keywords ("python ai"), and Test Case EC-ST15, with multiple keywords beyond two ("python ai user").

Errors Found in Program: F-ST06 and F-ST13.

White box testing:

White box testing was used to achieve comprehensive branch and statement coverage for the search_tweets.py function. The goal was to make sure that every possible execution path within the function was tested, and verify the internal logic and flow of the code. To help this, I provided ChatGPT with detailed information about the module's functionality and the underlying database schema, and asked it to generate unit tests that use mocking to simulate database connections, user inputs. Prompt 2D in Appendix D was passed in. The full code produced can be found in Figure 1D in the Appendix.

Resulting test case table:

ChatGPT generated a comprehensive suite of unit tests for search_tweets.py, each designed to target specific branches and statements within the function, and the tests were organized into a test case table detailing the Test ID, Inputs, Description, Expected Results, and Actual Results.

Table 4D: A portion of the resulting table, showing failing tests for the white box testing method; full table in Appendix D

Test ID	Inputs	Description	Expected Results	Actual Results
TC-01	Keywords: " "	Test behavior when no valid keywords are provided (empty/whitespace).	Message: "No valid keywords provided. Please enter keywords to search."	Pass
TC-02	Keywords: "python", Mocked Tweets: []	Test behavior when a valid keyword is provided but no matching tweets are found.	Message: "No matching tweets found."	Pass
TC-03	Keywords: "python", Mocked Tweets: ≤5 tweets, Input Sequence: 'back'	Test displaying up to five tweets and user opting to go back immediately.	Tweets are listed, followed by prompt: "Enter the number of a tweet to see more information, '0' to view more, or 'back' to go back: "	Fail

On executing the test suite using coverage.py, the coverage report showed that search_tweets.py achieved 52% coverage. Out of the 16 tests, 10 failed, primarily due to assertion mismatches and incorrect mocking configurations, this was despite constant reruns of prompts. As an example, Test Case TC-01 focused on handling scenarios where no valid keywords were provided; tested and passed. However, Test Case TC-03, failed; the test did not find the expected prompt in the print calls. Another important test was TC-05, it acted like a user selecting a valid tweet to retweet and expected the retweet function to have the correct parameters. This test also failed, as the retweet function was not called. Test Case TC-09 successfully tested the system's resilience against SQL injection attempts by providing malicious keywords because the function correctly handled the input without executing unintended database operations, and passed. Overall, out of the 16 tests executed, 10 failed, highlighting areas within the search_tweets.py module that require attention, but more likely highlighting the fact that chatgpt at many times was simply unable to provide working tests; especially due to the fact that majority of the black box test cases– which were manually tested – were passed. The coverage report further showed these gaps, revealing that the module achieved 52% coverage with substantial portions of the code, particularly lines 67-79, 86-87, and 94-129, remaining untested.

In general, chatGPT was very helpful with black box testing as the tester would not have to manually brainstorm all the test cases themselves, but in the case of white box testing, chatGPT did quite poorly and failed a huge amount of test

cases – which makes it hard to tell if the issues in the program remain or if the test cases themselves are written incorrectly. While significant progress was made in uncovering functional and security-related issues, the gaps that we identified and all the test failures highlight the need for ongoing refinement and more prompting of both the test cases and the module's internal logic. The detailed test results and unit tests, showing the tables and both passing and failing scenarios, are documented in Appendix D.

E. Search Users

The `search_users.py` function allows users to search for users based on a keyword and retrieves users whose name and/or city contain the given keyword. It also orders the results by date in descending order. The user can input one or more words and then view a list of the tweets that match the inputted keyword(s). First, it is ordered by matched name, in descending order of name length. Second, it is ordered by the matched city, again in descending order of name length. It displays these 5 at a time, with the option for the user to see more. If the user selects a specific user by entering a number, the code fetches more details about the selected user.

Black Box Testing

To begin black box testing, ChatGPT was given a description of the whole assignment then asked to produce black box tests that were failure testing, error guessing and equivalence partitioning. The prompt is present as prompt 1E in Appendix E.

The generated test cases and table can be found in Appendix E. The test cases demonstrate good coverage of key scenarios for the `search_users` function, addressing edge cases, functional correctness, and interaction behavior. They include tests for empty input (TC01), special character handling (TC02), and overly long keywords (TC03). TC04 and TC05 test how the system handles incomplete data or invalid inputs. However it can be seen that the provided test cases do not address failure testing or error guessing for all functionalities. For example, TC10 only checks for a successful follow operation but does not consider scenarios such as attempting to follow an already-followed user, following a nonexistent user, or handling database connection issues during the operation. Overall, modifying the test cases from ChatGPT with human feedback would significantly improve the test coverage and reliability assessment of the system.

White Box Testing

To begin white box testing, ChatGPT was given a description of the whole assignment and code then asked to produce white box tests using a framework like unit tests or pytest. The prompt is present as Prompt 2E in Appendix E.

The code and tests generated by ChatGPT can be found in appendix E. The testing covered about 43 percent of `search_users.py`, which is relatively poor and low. While the provided tests effectively cover primary scenarios like valid keyword searches, pagination, and performance with large datasets, they fall short in testing edge cases, error handling, and deeper interaction logic, such as nested user actions. Also while mocking is effectively used for database interactions and user information, the test cases do not fully explore all possible branches of the function's logic, particularly in loops and conditional statements. To improve coverage and overall testing quality, the test suite should include parameterized tests, simulate database failures, and expand its focus on exercising all branches and paths in the function's user interaction logic. While the generated code by ChatGPT achieved low performance, further refinement and prompting can increase the coverage to an acceptable standard.

F. Compose Tweet

This part of the program allows users to create a tweet and then stores it in the database. A tweet can contain hashtags, which are then stored in a separate table in the database.

Black Box Testing

To prompt ChatGPT to write test cases for this part of the program, we specified the testing methods, the format of the test table, and the functionality the program should perform.

We passed it the following prompt:

"You are tasked with testing a tweet composition feature for a social media platform. Your goal is to ensure that: A tweet can be composed and successfully posted, Hashtags included in the tweet are properly identified, stored, and linked to the tweet. The tweet text and its associated hashtags are handled appropriately under various conditions.

Generate test cases for this system, using failure-testing and error-guessing black box methods.

For each test, record, the input provided, a brief description, and the expected results"

ChatGPT produced the test cases presented in Table 1F in the appendix. Of the 11 produced test cases, 8 passed. These passing tests pertain to cases with both valid and invalid input. Edge cases are tested, along with cases which are expected to make the program fail.

Of the three failing tests, two are because ChatGPT produced tests to check for input validation, which the program does not account for. The last test, test case 6, actually helped us find a fault in our program. When a user attempts to mention the

same hashtag twice in one tweet, the program crashes and the test fails. This test helped us find another scenario that we should handle in our code.

Table 1F: A portion of the resulting table, showing failing tests for the black box testing method; the full table can be found in Appendix F

Test ID	Inputs	Description	Expected Output	Actual Results
3	tweet_text=""	Invalid input: Empty tweet.	Prints error message: "Tweet text cannot be empty." No tweet is inserted.	Fails Tweet is inserted into the <code>tweets</code> table.
4	tweet_text="a" * 300	Edge case: Large character input.	Prints error message: "Tweet text exceeds the allowed limit." No tweet is inserted.	Fails Tweet is inserted into the <code>tweets</code> table.
6	tweet_text="Repeated hashtag #Test #Test is interesting!"	Edge case: Repeated hashtags in a single tweet.	Inserts tweet. Hashtag #Test is added to <code>hashtags</code> if new; linked once to <code>mentions</code> for this tweet.	Fails UNIQUE constraint failed: mentions.tid, mentions.term

White Box Testing

To prompt ChatGPT to write test cases for this part of the program, we specified the testing methods, the format of the test script, and the type of coverage the tests should cover. We also provided the code itself for this part, since white box testing deals with the internal logic of the code. The prompt is provided as Prompt 1F in Appendix F,

ChatGPT produced a Python testing script to perform the white box tests for this part of the program. This is presented in Figure 1F.

Of the 8 tests it produced, 4 passed. ChatGPT was able to produce tests which properly interacted with the program and which properly mocked the database connection. The half of the tests which failed did so because they attempted to test input validation, which is not addressed by the program. ChatGPT made this error in test creation even though it was provided with the source code. The other half of the tests were properly created. These test the creation of tweets without hashtags, with hashtags, with repeated hashtags, and tweets that are a reply to another tweet. The tests check that the proper database insertion calls are executed. The tests provide 95% line coverage.

G. List Followers

The `list_followers.py` file handled the users menu option to see a list of their followers, black and white box testing methods were used as follows.

Black Box Testing:

In order to perform the Error Guessing, Failure and Equivalence class testing techniques on this portion of the program, ChatGPT was given context of the problem and asked to generate test cases in a tabular format, One for each of the three methods. The following are the prompt passed to ChatGPT and the resulting test case table, the actual results column was added after the fact once the manual test were completed:

(Full prompt presented as Prompt 1G in Appendix G): *"This project is a social media application implemented in Python and SQLite... One functionality is to list followers: a list of all users who follow you and from which you can select a follower to view more information about them. ... you can follow the selected user ... see all their tweets. function will ... list the id of the users the logged in user follows, user can select one of these followers to see more information,... name, number of followers, number of tweets, and the three most recent tweets ,user has choice to follow this follower (if they already follow ... they will be told they already follow), or see more tweets ... beyond the 3 most recent, Write...black box test cases for the Failure method: "Test every possible thing a user could do with the system to demolish the software" For each of the tests indicate the inputs, description and expected result "*

Table 1G: A portion of the resulting table for the failure method the full table can be found in Appendix G.

Test ID	Inputs	Description	Expected Result	Actual Result
1	Logged-in users have no followers.	No Followers Attempt to list followers when no followers exist.	Display a message: "You have no followers." No crashes or unexpected behavior.	You have no followers.
2	Enter a non-existent follower ID (e.g., an ID not in the list of followers).	Invalid Follower Selection. Select an ID not displayed in the followers list	Display a message: "Invalid selection. Please select a valid follower ID." Prompt the user again for input.	Invalid follower ID. For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit:

Prompt: “For the same system and problem I want to look at error guessing test cases, Error guessing is a software testing technique that uses a tester's experience and intuition to anticipate and uncover bugs in a program, Give me some test cases with inputs, description , and expected results.”

Table 2G: A portion of the resulting table for error guessing created by ChatGPT. The Full table can be found in Appendix G.

Test ID	Inputs	Description	Expected Result	Actual Result
1	Press "Enter" without typing anything when asked to select a follower ID.	Test Case 1: Empty Input - Test the system's handling of blank input.	Display: "Invalid input. Please select a valid follower ID." Prompt user again for input.	"Invalid input. Please Try again."
2	Provide non-numeric input such as abc or @#\$% when asked to select a follower ID.	Test Case 2: Non-Numeric Input - Check if the system handles non-numeric input gracefully.	Display: "Invalid input. Please enter a valid follower ID." Prompt user again.	"Invalid input. Please Try again."
7	Select a follower, then delete this follower's account from the database before viewing their tweets.	Test Case 7: Viewing Tweets from a Deleted User - Check behavior when trying to view tweets of a deleted user.	Display: "User not found. Unable to retrieve tweets." No crashes.	Tweets still show up - No cascade effect

Prompt:“For the same system Write me some black box test cases for the Equivalence class method : "Until all valid equivalence classes have been covered by test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible Until tests have covered all invalid equivalence classes, write a test case that covers one and only one, of the uncovered invalid equivalence classes" For each of the tests indicate the inputs, description and expected result and format them as a table”

Table 3G: A portion of the result for the equivalence classes created by ChatGPT. The full table can be found in Appendix G.

Test ID	Inputs	Description	Expected Result	Actual Result
TC1	Logged-in users follow 3 users (User A, User B, User C).	Valid case: User has multiple va.	List of followers should show User A, User B, and User C. The user can select any of them to see more details or tweets.	Users can see multiple followers.
TC2	Logged-in users follow no one.	Valid case: User has no followers.	No followers should be listed. Users should be informed that they do not follow anyone.	Users can see no followers message.
TC5	Logged-in users follow 2 users: User A and User B. User A has no tweets and User B has no followers.	Valid case: User follows someone with no tweets, and no one follows another.	List of followers should show User A and User B. After selecting User A, the user should see no tweets. After selecting User B, the user should see no followers.	User B - shows up with one follower but logging in as User B shows 0 followers. No tweets for user A

For each of the above prompts the key information is the explanation of the context of the program,, definitions of the testing methods and the explanation of how the results should be presented. The first prompt defined the context of the problem while the rest just used information already given to the LLM. ChatGPT responded with a short description of the testing methods and test case Table 3G. After manually running the tests that were described we added the Actual result column to the tables as shown in Tables 1G, 2G and 3G failures/faults/errors are highlighted. The following errors were found in the program: a user was able to follow themselves (found by failure testing), a deleted user's tweets could still be found (found by error guessing) and a user with no followers shows up as having one follower at least (found by equivalence classes). The black box testing methods proved effective for testing the functionality of this part of the program. They covered various scenarios and aligned well with the test cases we would have developed ourselves, demonstrating that ChatGPT was successful. Each method identified different errors, highlighting the importance of using multiple approaches to achieve better coverage. Overall, the black box testing methods worked effectively for this system, and ChatGPT successfully generated appropriate test cases.

White Box Testing:

The goal for using branch and statement coverage testing was to achieve 100% coverage through the unit tests for the list_followers.py function mocking the database connection and database itself as necessary. ChatGPT was used to create the Python unit tests to test the list_followers.py file and functionality to do this the following prompt Prompt 2G in Appendix G was passed in the same chat session as the ones above. The key components of this prompt are the attached source code, the

definition of coverage, explicitly defining that mocks should be used, as well as how to format the input, note that the context itself did not need to be passed again as the LLM had this information. The result was a python unit test file with various test cases (Figure 1G in Appendix G) and Table 4G

Table 4G: A portion of the ChatGPT generated white box testing coverage test cases for `list_followers.py`. The full table can be found in Appendix G.

Test ID	Inputs	Description	Expected Results	Actual Results
TC-03	Input: 0, 2, 0	Test selecting a valid follower, viewing their details, and following them.	Follower details are displayed, and a follow record is inserted.	Test case itself is unable to run - invalid indexes
TC-12	Tweets: [(101, "Tweet 1"), (102, "Tweet 2"), ...]	Test system handling for a follower with a large number of tweets.	Tweets are displayed without performance issues.	Test Case itself failed to run due to iterations

Running these tests had a total 94% coverage of the `list_followers.py`, indicating that ChatGPT was successful in completing the task asked of it and generating branch and statement coverage test cases, further since the testing follows suit with how we would approach testing the program can confidently say that statement/ branch coverage testing were appropriate in this context. Test cases 3 & 12 failed not indicating failures in the program but faults in the test case itself that prevented it from running either due to index out of range issues or stop iterations errors. Although errors were found in the black box testing methods but not the white box methods, this indicates that further testing beyond just coverage may need to be implemented to find all faults and errors in the function/program. Overall the use of the employed black and white box methods and the use of ChatGPT to generate the test cases proved successful for this portion of testing.

Conclusion

For this simple twitter app, we used a combination of black box and white box testing methodologies, and they both contributed well to the evaluation of the system's functionality, security, and reliability. Black box testing techniques included failure testing, error guessing, and equivalence partitioning, and were very important in assessing the application's behavior from an external perspective. Failure testing also tested the system to check for inputs such as SQL injection attempts and excessively long keywords. Error guessing uses the testers' intuition and experience to anticipate common user mistakes that could lead to unexpected behavior. The testing process made sure of comprehensive coverage by selecting specific representative test cases from each equivalence class. White box testing aimed for exhaustive branch and statement coverage by creating targeted unit tests. By using coverage testing, white box testing tried to validate every possible execution path within the program. This methodology was helpful in identifying specific logical flaws and untested code segments, especially within some pretty important modules like `search_tweets.py` and `list_followers.py`. Overall, the integration of black box and white box testing strategies showed a thorough evaluation of the social media twitter application. For the black box tests that ChatGPT produced, about 87% of them were valid. These tests dealt with the functionality of the code and were effective in helping us find actual errors in our program (like the miscount of user followers). The invalid tests tended to focus on aspects not present in the source code, such as character validation. The outputs were similar across the board for the different functionalities of the program, proving high repeatability. For the white box tests that ChatGPT produced, about 73% of them were valid. This number is lower as compared to the black box test cases. This is in part because ChatGPT tried to focus again on aspects not present in the source code, such as character validation. This is peculiar as, in this case, we provided ChatGPT with the internal logic through the source code and it still erred in the same way. Another pitfall of the ChatGPT generation was in the mock database connection. This is most evident in its generation attempts for `main.py`, as detailed earlier. ChatGPT was a helpful tool for generating test cases, particularly for black box testing. It provided structured scenarios for failure, error guessing, and equivalence class testing, effectively identifying functional errors and vulnerabilities like SQL injection. While these cases were useful, ChatGPT sometimes made incorrect assumptions about system behavior, requiring manual changes with the test cases. For white box testing, ChatGPT successfully created unit tests to achieve branch and statement coverage using mocks for database interactions and user inputs. However, some tests were unrealistic or failed to run due to limited understanding of the system's full context, restricting its effectiveness to just individual modules. Overall, ChatGPT was helpful for making the testing process more efficient and providing coverage, but its outputs required manual feedback to ensure accuracy and applicability across the entire system.

References

Program tested: R. Aery, F. Garcia, A. Qutub, and C. Roque, "SQLTwitter," GitHub, <https://github.com/ruchali1/SQLTwitter>. Definitions and testing concepts were heavily based on Professor W. Pedrycz's ECE 322 course notes and lab manuals. A. Andersson and A. Malmeling, FULLTEXT01.pdf - diva portal, <https://www.diva-portal.org/smash/get/diva2:696394/FULLTEXT01.pdf> (published Jun. 19, 2024).

Appendix

Appendix A

Prompt 1A: “This project is a social media application implemented in Python and SQLite. It allows users to perform various social media interactions such as composing tweets, following other users, displaying tweets, retweets, and user information, as well as searching for users and tweets based on specific keywords. The project involves database management, user interactions, and data retrieval, providing a basic social media experience within a command-line interface. `main.py`, acts as the entry position for the program. It starts by initializing the SQLite database connection and allows the user to enter a database. It then enters the main menu, from which the user can choose to login, register an account, or exit the program. If the user chooses to login or register an account, the function calls `login()` or `register()`, respectively, both of which are located in `user.py`. `login(conn)` : This function asks for user input for the username and password. It then checks to ensure that these are valid. `register_user(conn)` : This function asks users to enter their information and enters it into the table `users`. `Main.py` also asks for users to enter a valid database file for the program to run from. Once the user is logged in, they are prompted with a question asking if they would like to see the tweets or retweets of the people they follow, or return to the main menu which is where they will be given the following options, which they can select by entering the corresponding digit. 1. Back to Display tweets or retweets 2. Search for tweets 3. Search for users 4. Compose a tweet 5. List followers 6. Logout Create Black Box Test Cases for the functionality of `main.py`, you can assume a user has successfully accessed the main menu. Use the: - Failure method: Test every possible thing a user could do with the system to demolish the software - Error guessing: a software testing technique that uses a tester's experience and intuition to anticipate and uncover bugs in a program - Equivalence class method : Until all valid equivalence classes have been covered by test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible Until tests have covered all invalid equivalence classes, write a test case that covers one and only one, of the uncovered invalid equivalence classes For each of the tests indicate the inputs, description and expected result and format them as a table Clearly identify which cases are failure, error, or equivalence.”

Table 1A: Black Box testing test cases created by ChatGPT and manually tested for `main.py`. 10/12 Pass

Test ID	Category	Input	Description	Expected Result	Actual Result
F1	Failure	User enters an invalid database file path (e.g., <code>invalid.db</code>).	Tests behavior when the provided database file does not exist or is inaccessible.	Displays an error message: "Database file not found. Please enter a valid database file path."	"Error: The specified file does not exist. Please enter a valid database file name."
F2	Failure	User enters an unsupported database file format (e.g., <code>database.txt</code>).	Tests behavior when a non-SQLite database file is entered.	Displays an error message: "Unsupported database file format. Please provide a valid SQLite database."	System allows txt file to be treated at database
F3	Failure	User inputs an invalid option (e.g., <code>999</code> , <code>abc</code> , or empty input) at the main menu.	Tests handling of invalid menu input values.	Displays an error message: "Invalid option. Please try again."	User is prompted with menu again
F4	Failure	User rapidly enters multiple invalid menu options (e.g., <code>x</code> , <code>0</code> , <code>@</code>) to test resilience.	Tests system stability under repeated invalid inputs.	Consistently displays the error message: "Invalid option. Please try again."	User is prompted with menu each time
F5	Failure	User selects a valid menu option (e.g., <code>2</code>) but then forcibly terminates the program.	Tests behavior when the program is interrupted during execution.	Program terminates gracefully without corrupting the database or leaving dangling connections.	Program terminates without problem
E1	Error Guessing	User leaves the database path empty and presses Enter.	Tests handling of empty input when a database file is requested.	Displays an error message: "Database path cannot be empty. Please enter a valid database file."	"Error: The specified file does not exist. Please enter a valid database file name."

E2	Error Guessing	User enters a valid menu option but inserts leading/trailing spaces (e.g., " 1 " or " 6 ").	Tests handling of inputs with extraneous spaces, which might cause parsing issues.	The program trims spaces and correctly interprets the input, triggering the corresponding menu action.	Program does not trim response but asks for a new input
E3	Error Guessing	User provides a mix of numbers and characters as input for the menu option (e.g., "2abc").	Tests system behavior when input is partially valid but includes invalid characters..	Displays an error message: "Invalid option. Please try again."	Treats as invalid option and ask user for new input (displays menu)
EC1	Equivalence	User enters a valid SQLite database file path (e.g., <code>socialmedia.db</code>).	Tests valid equivalence class for initializing the program with an accessible database file.	Program initializes successfully and displays the main menu.	Program continues
EC2	Equivalence	User inputs a valid menu option (e.g., 1, 2, 3, etc.).	Tests valid equivalence class for menu option selection.	Corresponding functionality is triggered (e.g., option 1 returns to display tweets or retweets).	Program continues correctly
EC3	Equivalence	User enters 6 to log out and returns to the main menu.	Tests valid equivalence class for returning to the main menu after logging out.	User is logged out, and the main menu is displayed again.	User is logged out
EC4	Equivalence	User selects a valid menu option after entering the database file correctly.	Tests valid equivalence class for correct flow after initialization (e.g., choosing 4 to compose a tweet).	Corresponding menu functionality starts without errors.	User can choose valid menu option successfully

Prompt 2A: "Write me python unit tests for that attached source code {main.py} that follow Branch - Every branch (edge of the graph) in the code has to be executed at least once and Statement Coverage - Every statement in the code has to be executed at least once. Mock the DB connection and DB calls Testing Framework: Use unittest for writing the test cases. Mocking: Use the unittest.mock library to mock database connections and user inputs, ensuring all branches in the code are covered. Database Behavior: Mock all database queries and updates using MagicMock to simulate realistic database behavior without requiring a live database. Code Organization: Ensure each test case is self-contained and adheres to the unittest structure for setup, execution, and teardown. Assertions: Validate the expected behavior for each branch of the code, including return values and database interactions (e.g., execute calls and their parameters). Output: Include any print statements to confirm test coverage during execution. Put the tests into a test case table with test case ID, inputs, description and expected results. Be sure to mock the login and db connection but since we are aiming for coverage make sure to use main to test the main menu."

Table 2A: ChatGPT's first attempt at white box, coverage testing for main.py.

Test Case ID	Inputs	Description	Expected Results
1	None	Test database file not found	Exits with error message
2	'valid_db.sqlite'	Test valid database file	Passes connection check
3	'1'	Test login option	Calls login method
4	'2'	Test register option	Calls <code>register user</code> method
5	'3'	Test exit option	Exits the program
6	User logs in	Test menu after login	Executes actions like displaying tweets and logging out
7	Invalid database	Test invalid database input	Prompts for valid input and tries again

Table 3A: White box, coverage testing unit tests employed for main.py 10/10 Pass

Test ID	Test Description	Inputs	Expected Results	Actual Results (Pass/Fail)
TC001	Test valid database file	test.db	mock_exists should return True; sqlite3.connect should be called with test.db as argument.	Test case passed
TC002	Test invalid database file	nonexistent.db	mock_exists should return False; error message should be printed.	Test case passed
TC003	Test login choice (successful login)	1 (Login choice), 1 (Display Tweets), 6 (Logout)	mock_login should return 1 (user ID); display_tweets_and_retweets should be called with conn and 1 as arguments.	Test case passed
TC004	Test register choice (successful registration)	2 (Register choice)	mock_register_user should be called once with conn as argument.	Test case passed
TC005	Test logout choice (after login and display tweets)	1 (Login choice), 1 (Display Tweets), 6 (Logout)	mock_display_tweets_and_retweets should be called once with conn and 1 as arguments. "Exiting after logout" message should be printed.	Test case passed
TC006	Test search tweets functionality	2 (Search Tweets choice), test (search keyword)	mock_search_tweets should be called once with conn, 1, and test as arguments.	Test case passed
TC007	Test search users functionality	3 (Search Users choice), user (search keyword)	mock_search_users should be called once with conn, user, and 1 as arguments.	Test case passed
TC008	Test compose tweet functionality	4 (Compose Tweet choice), Hello World (tweet text)	mock_compose_tweet should be called once with conn, 1, Hello World, and None as arguments.	Test case passed
TC009	Test list followers functionality	5 (List Followers choice)	mock_list_followers should be called once with conn and 1 as arguments.	Test case passed
TC010	Test exit functionality	3 (Exit choice)	sys.exit(0) should be called, and exit should happen successfully.	Test case passed

```

import unittest
from unittest.mock import MagicMock, patch
import sys
import os
import sqlite3
from main import *

# Test case structure table
# Test Case ID | Inputs | Description | Expected Results
# 1            | None  | Test database file not found | Exits with error message

```

```

# 2          | 'valid_db.sqlite' | Test valid database file | Passes connection check
# 3          | '1' | Test login option | Calls login method
# 4          | '2' | Test register option | Calls register_user method
# 5          | '3' | Test exit option | Exits the program
# 6          | User logs in | Test menu after login | Executes actions like displaying tweets
and logging out
# 7          | Invalid database | Test invalid database input | Prompts for valid input and
tries again

```

```

class TestMainMenu(unittest.TestCase):
    def setUp(self):
        """Setup for mocking."""
        self.db_file = 'test_db.sqlite'
        self.mock_conn = MagicMock(spec=sqlite3.Connection)
        self.mock_cursor = MagicMock(spec=sqlite3.Cursor)
        self.mock_conn.cursor.return_value = self.mock_cursor

    @patch('builtins.input', return_value='1') # Mock input for login choice
    @patch('main.login', return_value=1) # Mock login returning user ID 1
    @patch('main.display_tweets_and_retweets') # Mock display function
    def test_main_menu_login(self, mock_display_tweets, mock_login, mock_input):
        """Test if login choice triggers correct function."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_print.assert_any_call("Enter your choice: ")
            mock_login.assert_called_once_with(self.mock_conn)
            mock_display_tweets.assert_called_once_with(self.mock_conn, 1)

    @patch('builtins.input', return_value='2') # Mock input for register choice
    @patch('main.register_user') # Mock register_user function
    def test_main_menu_register(self, mock_register_user, mock_input):
        """Test if register choice triggers correct function."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_print.assert_any_call("Enter your choice: ")
            mock_register_user.assert_called_once_with(self.mock_conn)

    @patch('builtins.input', return_value='3') # Mock input for exit choice
    @patch('sys.exit') # Mock sys.exit to prevent actual exit
    def test_main_menu_exit(self, mock_exit, mock_input):
        """Test if exit choice exits the program."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_print.assert_any_call("Enter your choice: ")
            mock_exit.assert_called_once()

    @patch('builtins.input', side_effect=['1', '1']) # Login, then back to tweets
    @patch('main.login', return_value=1) # Mock login to return user id 1
    @patch('main.display_tweets_and_retweets') # Mock display_tweets
    def test_menu_after_login(self, mock_display_tweets, mock_login, mock_input):
        """Test actions after login menu."""
        with patch('builtins.print') as mock_print:
            main_menu()
            # Entering login option, then navigating inside after login
            mock_display_tweets.assert_called_with(self.mock_conn, 1)

```



```

@patch('builtins.input', side_effect=['2', 'test_user', '3']) # Register, then exit
@patch('main.register_user') # Mock register_user
def test_register_user(self, mock_register_user, mock_input):
    """Test user registration."""
    with patch('builtins.print') as mock_print:
        main_menu()
        mock_register_user.assert_called_once_with(self.mock_conn)

    @patch('builtins.input', side_effect=['2', 'test_user', '3']) # Registration with invalid db
    path
    @patch('os.path.exists', return_value=False) # Mock file does not exist
    def test_invalid_db_path(self, mock_input, mock_exists):
        """Test invalid database file input and retry mechanism."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_print.assert_any_call("Error: The specified file does not exist.")
            mock_print.assert_any_call("Error: The specified file does not exist. Please enter a
valid database file name.")

    @patch('builtins.input', side_effect=['1', '1', '2', 'search_keyword', '3']) # User login,
    search tweets
    @patch('main.login', return_value=1) # Mock login to return user ID 1
    @patch('main.search_tweets') # Mock search_tweets
    def test_search_tweets_after_login(self, mock_search_tweets, mock_login, mock_input):
        """Test searching tweets after login."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_search_tweets.assert_called_once_with(self.mock_conn, 1, 'search_keyword')

    @patch('builtins.input', side_effect=['1', '1', '2', 'search_keyword', '4']) # User login,
    compose tweet
    @patch('main.login', return_value=1) # Mock login to return user ID 1
    @patch('main.compose_tweet') # Mock compose_tweet
    def test_compose_tweet_after_login(self, mock_compose_tweet, mock_login, mock_input):
        """Test composing a tweet after login."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_compose_tweet.assert_called_once_with(self.mock_conn, 1, 'search_keyword', None)

    @patch('builtins.input', side_effect=['1', '1', '2', 'search_keyword', '6']) # User login,
    logout
    @patch('main.login', return_value=1) # Mock login to return user ID 1
    def test_logout_after_login(self, mock_login, mock_input):
        """Test logout functionality after login."""
        with patch('builtins.print') as mock_print:
            main_menu()
            mock_print.assert_any_call("Logout")

if __name__ == '__main__':
    unittest.main()

```

Figure 1A: ChatGPT output for unit tests for main.py outlined in Table 2A.

```

import unittest
from unittest.mock import MagicMock, patch
import sqlite3

```

```

import sys
import os
from main import * # Assuming main.py is the entry point

# Mocking imported functions
from user import login, register_user
from list_followers import list_followers
from search_tweets import search_tweets
from compose_tweet import compose_tweet
from search_users import search_users
from display_tweets import display_tweets_and_retweets

class TestSocialMediaApp(unittest.TestCase):

    @patch('sqlite3.connect')
    @patch('os.path.exists')
    def test_valid_database_file(self, mock_exists, mock_connect):
        # Test valid database file input (Branch Coverage)
        mock_exists.return_value = True
        mock_conn = MagicMock() # Mocked database connection
        mock_connect.return_value = mock_conn

        with patch('builtins.input', return_value="test.db"):
            db_file = input("Enter the database file name: ")
            self.assertTrue(mock_exists(db_file))

            conn = sqlite3.connect(db_file)

            # Simulate a cursor call by performing a mock query
            mock_conn.cursor.return_value.execute.return_value = None # Mock the execute
method
            conn.cursor().execute("SELECT 1") # Trigger the cursor and execute method

            # Check if cursor() was called
            conn.cursor.assert_called_once()
            conn.cursor().execute.assert_called_once_with("SELECT 1")

    @patch('sqlite3.connect')
    @patch('os.path.exists')
    def test_invalid_database_file(self, mock_exists, mock_connect):
        # Test invalid database file input (Edge case)
        mock_exists.return_value = False
        mock_connect.return_value = MagicMock()

        with patch('builtins.input', return_value="nonexistent.db"):
            db_file = input("Enter the database file name: ")
            self.assertFalse(mock_exists(db_file))
            print("Error: The specified file does not exist.") # Check the error message

    @patch('user.login')
    @patch('display_tweets.display_tweets_and_retweets')
    @patch('builtins.input', side_effect=['1', '1', '1', '2', 'test', '3', 'user', '4',
'Hello World', '5', '6']) # Login -> Display Tweets -> Various Options
    def test_logged_in_user_interaction(self, mock_input, mock_display_tweets, mock_login):

```

```

# Test logged-in user interaction (Branch Coverage)
mock_login.return_value = 1 # Simulate successful login
conn = MagicMock() # Mock database connection

# Simulate login and check for correct user ID
user_id = mock_login(conn)
self.assertEqual(user_id, 1)

# Simulate displaying tweets (after login)
mock_display_tweets(conn, user_id) # Simulate display tweets being called
mock_display_tweets.assert_called_once_with(conn, 1)

# Simulate user choices after logging in
# 1. Display Tweets or Retweets
user_choice = input("Enter your choice: ")
if user_choice == '1':
    mock_display_tweets(conn, user_id)
mock_display_tweets.assert_called_with(conn, user_id)

# 2. Search for Tweets
user_choice = input("Enter your choice: ")
if user_choice == '2':
    keyword = input("Enter keywords: ")
    mock_search_tweets = MagicMock() # Mock search_tweets
    mock_search_tweets(conn, user_id, keyword)
    mock_search_tweets.assert_called_once_with(conn, user_id, 'test')

# 3. Search for Users
user_choice = input("Enter your choice: ")
if user_choice == '3':
    keyword = input("Enter a keyword: ")
    mock_search_users = MagicMock() # Mock search_users
    mock_search_users(conn, keyword, user_id)
    mock_search_users.assert_called_once_with(conn, 'user', user_id)

# 4. Compose a Tweet
user_choice = input("Enter your choice: ")
if user_choice == '4':
    tweet_text = input("Compose your tweet: ")
    mock_compose_tweet = MagicMock() # Mock compose_tweet
    mock_compose_tweet(conn, user_id, tweet_text, None)
    mock_compose_tweet.assert_called_once_with(conn, user_id, 'Hello World', None)

# 5. List Followers
user_choice = input("Enter your choice: ")
if user_choice == '5':
    mock_list_followers = MagicMock() # Mock list_followers
    mock_list_followers(conn, user_id)
    mock_list_followers.assert_called_once_with(conn, user_id)

# 6. Logout (Exit the loop)
user_choice = input("Enter your choice: ")
if user_choice == '6':
    print("Logging out...") # Simulate logout message

```

```

@patch('user.register_user')
def test_register_choice(self, mock_register):
    # Test registration functionality (Statement Coverage)
    conn = MagicMock()

    with patch('builtins.input', return_value='2'):
        mock_register(conn)
        mock_register.assert_called_once_with(conn)

    @patch('builtins.input', side_effect=['1', '1', '6']) # Login -> Display Tweets ->
Logout
    @patch('user.login')
    @patch('display_tweets.display_tweets_and_retweets')
    def test_logout_choice(self, mock_display_tweets, mock_login, mock_input):
        mock_login.return_value = 1
        conn = MagicMock()

        user_id = mock_login(conn)
        self.assertEqual(user_id, 1)
        user_choice = input("Enter your choice: ")
        if user_choice == '1':
            mock_display_tweets(conn, user_id)
        mock_display_tweets.assert_called_once_with(conn, 1)
        print("Exiting after logout") # Check for correct exit message

    @patch('search_tweets.search_tweets')
    def test_search_tweets_choice(self, mock_search):
        # Test search tweets functionality (Branch Coverage)
        mock_search.return_value = None # No actual result, only ensuring function is
called
        conn = MagicMock()

        with patch('builtins.input', return_value='2'):
            keyword = 'test'
            mock_search(conn, 1, keyword)
            mock_search.assert_called_once_with(conn, 1, keyword)

    @patch('search_users.search_users')
    def test_search_users_choice(self, mock_search):
        # Test search users functionality (Statement Coverage)
        mock_search.return_value = None
        conn = MagicMock()

        with patch('builtins.input', return_value='3'):
            keyword = 'user'
            mock_search(conn, keyword, 1)
            mock_search.assert_called_once_with(conn, keyword, 1)

    @patch('compose_tweet.compose_tweet')
    def test_compose_tweet_choice(self, mock_compose):
        # Test compose tweet functionality (Statement Coverage)
        mock_compose.return_value = None
        conn = MagicMock()

        with patch('builtins.input', return_value='4'):
            tweet_text = 'Hello World'

```

```

mock_compose(conn, 1, tweet_text, None)
mock_compose.assert_called_once_with(conn, 1, tweet_text, None)

@patch('list_followers.list_followers')
def test_list_followers_choice(self, mock_list):
    # Test list followers functionality (Branch Coverage)
    mock_list.return_value = None
    conn = MagicMock()

    with patch('builtins.input', return_value='5'):
        mock_list(conn, 1)
        mock_list.assert_called_once_with(conn, 1)

@patch('builtins.input', return_value='3') # Exit case
@patch('sys.exit') # Mock sys.exit to prevent actual exit
def test_exit_choice(self, mock_exit, mock_input):
    print("Exiting the app.")
    sys.exit(0)
    mock_exit.assert_called_once_with(0)

if __name__ == '__main__':
    unittest.main()

```

Figure 2A: ChatGPT output for unit tests for main.py outlined in Table 3A.

Appendix B

You are tasked with creating a table of black box test cases for a Python script that handles user login and registration. Use the failure testing and error guessing methods. Write the table with the following columns: Test ID, Inputs, Description, and Expected Output + Results.

The script has two primary functionalities:

1. Login: Users log in with a username and password. Successful login allows them to view their tweets.
2. Registration: New users can register with a unique ID, password, name, email, city, and timezone. Upon registration, a new user ID is generated and stored.

The following users already exist:

usr	pwd	name	email	city	time zone
97	ben	Connor McDavid	cm@nhl.com	Edmonton	-7.0
29	ben	Leon Draisaitl	ld@nhl.com	Edmonton	-7.0
5	ben2	Davood Rafiei	dr@ualberta.ca	Edmonton	-7.0
70	ben	Francis	fgagagar@gmail.test	Edmonton	-7.0

Prompt 1B: Prompt provided to ChatGPT to generate black box tests for login and registration functionalities

Table 1B: Black Box testing test cases created by ChatGPT and manually tested for users.py. 6/12 Pass

Test ID	Inputs	Description	Expected Output	Actual Output
1	Login: user_id=97, password=ben	Valid login for an existing user.	User is logged in, tweets are displayed.	User is logged in, tweets are displayed.

2	Login: user_id=123, password=wrongpass	Invalid login with non-existent user ID.	"Invalid username or password." is printed, user is not logged in.	"Invalid username or password."
3	Login: user_id=97, password=wrongpass	Invalid login with correct user ID but incorrect password.	"Invalid username or password." is printed, user is not logged in.	"Invalid username or password."
4	Register: password=securepass, name=New User, email=newuser@test.com, city=Calgary, timezone=-7	Valid registration of a new user.	Registration is successful. A new user ID is assigned and printed.	Registration is successful. A new user ID is assigned and printed.
5	Register: password=, name=New User, email=newuser@test.com, city=Calgary, timezone=-7	Attempted registration with an empty password.	Error message or prompt to re-enter password is shown.	Fail Registration is successful. A new user ID is assigned and printed.
6	Register: password=securepass, name=New User, email=invalid_email, city=Calgary, timezone=-7	Attempted registration with an invalid email format.	Error message or prompt to enter a valid email is shown.	Fail Registration is successful. A new user ID is assigned and printed
7	Register: password=securepass, name=, email=newuser@test.com, city=Calgary, timezone=-7	Attempted registration with an empty name field.	Error message or prompt to re-enter name is shown.	Fail Registration is successful. A new user ID is assigned and printed
8	Register: password=securepass, name=New User, email=newuser@test.com, city=, timezone=-7	Attempted registration with an empty city field.	Registration proceeds successfully (assuming city is not validated).	Registration proceeds successfully
9	Register: password=securepass, name=New User, email=newuser@test.com, city=Calgary, timezone=invalid	Attempted registration with an invalid timezone (non-numeric).	Error message or prompt to re-enter timezone is shown.	Fail Registration is successful. A new user ID is assigned and printed
10	Register: (fields repeated from an existing user: password=ben, name=Connor McDavid, etc.)	Attempted registration where input values match an existing user, but the system assigns a unique user ID (does not use duplicate usernames).	Registration proceeds successfully. The new user has a unique user ID.	Fail Registration is successful. A new user ID is assigned and printed

11	Login: <code>user_id=97, password=' OR '1'='1</code>	SQL injection attempt during login.	The system safely handles the input and rejects the login. "Invalid username or password." is printed.	"Invalid username or password."
12	Register: <code>password=' OR '1'='1, name=Injected, email=inject@test.com, city=Nowhere, timezone=-7</code>	SQL injection attempt during registration.	The system safely handles the input and rejects the registration or sanitizes the input. Registration fails with an error.	Fail Registration is successful. A new user ID is assigned and printed

Write Python unit tests for the following source code with branch coverage:

{User.py}

1. *Testing Framework:* Use `unittest` for writing the test cases.
2. *Mocking:* Use the `unittest.mock` library to mock database connections and user inputs, ensuring all branches in the code are covered.
3. *Database Behavior:* Mock all database queries and updates using `MagicMock` to simulate realistic database behavior without requiring a live database.
4. *Code Organization:* Ensure each test case is self-contained and adheres to the `unittest` structure for setup, execution, and teardown.
5. *Assertions:* Validate the expected behavior for each branch of the code, including return values and database interactions (e.g., `execute` calls and their parameters).
6. *Output:* Include any print statements to confirm test coverage during execution.

Prompt 2B: Prompt provided to ChatGPT to generate white box unit tests for the user module; the source code is passed in as {User.py}

```
import unittest
from unittest.mock import patch, MagicMock
from io import StringIO
import sqlite3
from user import login, register_user

class TestUserFunctions(unittest.TestCase):

    @patch('builtins.input', side_effect=['ben', 'ben_password'])
    @patch('getpass.getpass', side_effect=['ben_password'])
    def test_login_valid(self, mock_getpass, mock_input):
        # Mock the connection and cursor
        conn = MagicMock()
        mock_cursor = conn.cursor.return_value

        # Mock the database response for a valid user
        mock_cursor.fetchone.return_value = ['ben']

        user_id = login(conn)

        # Assert that the user ID is returned correctly for a valid user
```

```

self.assertEqual(user_id, 'ben')

@patch('builtins.input', side_effect=['ben', 'wrong_password'])
@patch('getpass.getpass', side_effect=['wrong_password'])
def test_login_invalid_password(self, mock_getpass, mock_input):
    # Mock the connection and cursor
    conn = MagicMock()
    mock_cursor = conn.cursor.return_value

    # Mock the database response for invalid credentials
    mock_cursor.fetchone.return_value = None

    user_id = login(conn)

    # Assert that the function returns None for invalid login
    self.assertIsNone(user_id)

@patch('builtins.input', side_effect=['non_existent_user', 'password'])
@patch('getpass.getpass', side_effect=['password'])
def test_login_invalid_username(self, mock_getpass, mock_input):
    # Mock the connection and cursor
    conn = MagicMock()
    mock_cursor = conn.cursor.return_value

    # Mock the database response for invalid username
    mock_cursor.fetchone.return_value = None

    user_id = login(conn)

    # Assert that the function returns None for invalid username
    self.assertIsNone(user_id)

@patch('getpass.getpass', side_effect=['new_user_password'])
@patch('builtins.input', side_effect=['new_user_name', 'new_user_email', 'New City', '1.0'])
def test_register_user_successful(self, mock_input, mock_getpass):
    # Mock the connection and cursor
    conn = MagicMock()
    mock_cursor = conn.cursor.return_value

    # Mock the unique user ID generation function
    with patch('user.get_unique_user_id', return_value=1):
        register_user(conn)

    # Assert that the expected insert statement is called
    mock_cursor.execute.assert_called_with(
        "INSERT INTO users VALUES (?, ?, ?, ?, ?, ?)",
        (1, 'new_user_password', 'new_user_name', 'new_user_email', 'New City', '1.0')
    )
    print("Registration successful.")

```



```

@patch('getpass.getpass', side_effect=['existing_password'])
@patch('builtins.input', side_effect=['ben', 'existing_email', 'Existing City', '-5.0'])
def test_register_user_duplicate(self, mock_input, mock_getpass):
    # Mock the connection and cursor
    conn = MagicMock()
    mock_cursor = conn.cursor.return_value

    # Simulate existing user ID conflict
    mock_cursor.fetchone.return_value = ['ben']

    with self.assertRaises(Exception): # Assume duplicate username raises an exception
        register_user(conn)

# @patch('getpass.getpass', side_effect=['existing_password'])
# @patch('builtins.input', side_effect=['', 'empty_email', 'Empty City', ''])
# def test_register_user_invalid_input(self, mock_input, mock_getpass):
#     # Mock the connection and cursor
#     conn = MagicMock()
#     mock_cursor = conn.cursor.return_value

#     # Expecting the input to be validated, and an exception should be raised
#     with self.assertRaises(ValueError):
#         register_user(conn)

if __name__ == '__main__':
    unittest.main()

```

Figure 1B: ChatGPT output for unit tests for users.py

Appendix C

Table 1C: Black box testing (equivalence, error guessing, failure) for display_tweets.py. 18/20 passed, other 2 not tested

Test Case	Pre-conditions	Description	Input	Expected Results	Output
Test Case 1: Invalid Menu Selection	User is at the menu prompt to select tweets or retweets.	User inputs an invalid option when prompted.	'3' (or any non-'1'/'2' input like 'a', '#').	Application displays an error message and returns to the main menu or re-prompts the user.	*Returns to the main menu*
Test Case 2: No Tweets Available	User follows no one, or followed users have not posted any tweets.	User attempts to view tweets when none are available.	'1' (to view tweets).	Application displays "No tweets to display."	'No tweets to display.'
Test Case 3: No Retweets Available	User follows no one, or followed users have not retweeted anything.	User attempts to view retweets when none are available.	'2' (to view retweets).	Application displays "No retweets to display."	'No retweets to display.'
Test Case 4: Invalid Input for Viewing More	User has viewed initial tweets and is prompted to see more.	User inputs an invalid response when asked if they want to see more tweets.	'maybe' (or '123', '!').	Application displays an error message and re-prompts the user.	*goes back to prompting to see tweets or retweets*

Test Case 5: Invalid Tweet ID Selection	User is prompted to select a tweet by its ID.	User inputs an invalid or non-existent tweet ID.	'9999' (non-existent ID), '-1', 'abc', '!@#'	Application displays "Invalid ID." and re-prompts the user.	'Invalid input. Please Try again.' Then returns to main menu
Test Case 6: SQL Injection Attempt at Menu	User is at the menu prompt to select tweets or retweets.	User attempts SQL injection via menu input.	"1; DROP TABLE users; --"	Input is treated as invalid; no SQL commands are executed; application displays an error message and returns to the main menu or re-prompts.	Returns to the main menu
Test Case 7: SQL Injection in Tweet ID	User is prompted to select a tweet by its ID.	User attempts SQL injection via tweet ID input.	"' OR '1'='1"	Input is sanitized; no SQL commands are executed; application displays "Invalid ID."	'Invalid input. Please Try again.'
Test Case 8: Long Tweet Display	Tweets with maximum allowed character length exist in the system.	User views tweets containing maximum length text.	'1' (to view tweets), follow prompts as necessary.	Long tweets are displayed fully without truncation or errors.	PASSED
Test Case 9: Special Characters in Tweets	Tweets containing emojis and special characters are present.	User views tweets with special characters.	'1' (to view tweets), proceed as prompted.	Tweets display special characters correctly without rendering issues.	Not tested. Program does not have functionality for emojis
Test Case 10: Case Sensitivity in Responses	User is prompted to see more tweets or perform actions.	User inputs 'yes' or 'no' in various cases.	'YeS', 'nO', 'YES', 'no'.	Application correctly interprets the input regardless of case and proceeds accordingly.	'No more tweets' or <5. Passed
Test Case 11: Non-English 'Yes'/'No' Inputs	User is prompted to see more tweets or take an action.	User inputs 'yes' or 'no' in another language.	'sí', 'oui', 'ja'.	Application does not accept the input; displays an error message and re-prompts the user.	Reprompts the user. Passed
Test Case 12: No More Tweets but 'Yes' Entered	User has viewed all available tweets.	User inputs 'yes' when asked to see more, despite no tweets remaining.	'yes' when prompted to see more tweets.	Application informs the user "No more tweets." and returns to the previous menu or prompt.	'No more tweets' and returns to previous prompt
Test Case 13: User Cancels Operation	User is at any prompt within the tweets or retweets display flow.	User inputs 'cancel' or 'exit' to abort the current operation.	'cancel' or 'exit'.	Application exits the current operation and returns to the main menu.	'Invalid input. Please try again' Returns to menu
Test Case 14: Malformed Tweet ID Input	User is prompted to select a tweet by its ID.	User inputs non-numeric or special characters as the tweet ID.	'abc', '!@#', " (empty input).	Application displays "Invalid ID." and re-prompts the user.	'Invalid input. Please try again' return to menu
Test Case 15: Cross-Site Scripting in Tweets	Tweets containing scripts or HTML tags exist in the system.	User views tweets that may contain XSS attempts.	'1' (to view tweets), follow prompts as necessary.	Application displays the tweet content safely without executing any scripts.	PASSED

Test Case 16: Rapid Input Sequence	Application is running; user interaction prompts are active.	User inputs multiple commands rapidly or presses keys quickly.	Rapidly pressing 'Enter' or entering inputs without waiting for prompts.	Application processes inputs correctly without crashing or exhibiting unintended behavior.	PASSED
Test Case 17: Empty Input Submission	User is at any prompt requiring input.	User submits an empty input by pressing 'Enter' without typing.	(No input; just 'Enter')	Application displays an error message and re-prompts the user for valid input.	PASSED
Test Case 18: Exactly Five Tweets Available	User has exactly five tweets available to view.	User attempts to view tweets.	'1' (to view tweets), proceed as prompted.	Application displays all five tweets; does not prompt to see more tweets.	PASSED
Test Case 19: Large Number of Tweets	User has a large number of tweets available (e.g., 500 tweets).	User navigates through multiple pages of tweets.	'1', then repeatedly input 'yes' to see more tweets.	Application paginates tweets correctly, displaying them in batches without performance issues.	Not tested. Not realistically possible for this program.
Test Case 20: Replying to Non-Existent Tweet	User is attempting to reply to a tweet.	User selects a tweet ID that does not exist when prompted.	Invalid tweet ID (e.g., '9999', 'abc').	Application displays "Invalid ID." and re-prompts the user.	'Invalid ID'. reprompts user

Table 2C: White box, coverage testing unit tests employed for display_tweets.py 6/11 Passed

Test Case No.	Description	Branches/Statements Targeted	Outcome	Error Explanation
1	Test invalid menu inputs ('3', 'a', '#')	Branches in <code>display_tweets_and_retweets</code> where invalid inputs are handled by returning to the main menu.	Passed	N/A
2	Test no tweets available when user selects tweets ('1')	Statement where no tweets are fetched in <code>display_tweets</code> .	Passed	N/A
3	Test no retweets available when user selects retweets ('2')	Statement where no retweets are fetched in <code>display_retweets</code> .	Passed	N/A
4	Test tweet pagination for 6 tweets across 2 pages ('yes', 'no')	Branches handling pagination in <code>display_tweets</code> .	Error (StopIteration)	Input sequence ran out of elements due to insufficient mock values in <code>side_effect</code> . Add "yes", "no" to the mock input.
5	Test replying to a specific tweet ID ('reply')	Branch where <code>compose_tweet</code> is called in <code>display_tweets</code> .	Error (TypeError: Need a valid target to patch)	Incorrect patching of <code>compose_tweet</code> . Update the patch target to include the correct module path (e.g., " <code>display_tweets.compose_tweet</code> ").

6	Test retweeting a specific tweet ID ('retweet')	Branch where <code>retweet</code> is called in <code>display_tweets</code> .	Failed (AssertionError: <code>Sucesfully retweeted Tweet 1</code> not found)	The print statement in <code>retweet</code> likely has a typo (e.g., "Sucesfully" instead of "Successfully"). Correct the spelling in the source code or the test assertion.
7	Test invalid tweet ID selection ('invalid_id')	Branch in <code>display_tweets</code> where invalid IDs are handled.	Error (StopIteration)	Mock inputs for invalid IDs exhausted prematurely. Ensure the mock input includes sufficient values to handle all prompts in the function.
8	Test SQL injection in menu inputs ("1; DROP TABLE users; --")	Branch where menu input is handled in <code>display_tweets_and_retweets</code> .	Passed	N/A
9	Test <code>get_info</code> for retweet and reply counts	Branches where <code>retweet_count</code> and <code>reply_count</code> are fetched in <code>get_info</code> .	Passed	N/A
10	Test retweet functionality (<code>retweet</code>)	Statement where the retweet is inserted into the database in <code>retweet</code> .	Passed	N/A
11	Test special characters in tweets (e.g., emojis in "Hello 🍌")	Branch where tweets with special characters are displayed in <code>display_tweets</code> .	Error (StopIteration)	Mock inputs for user prompts ran out prematurely. Ensure that sufficient values are provided for <code>side_effect</code> in the test input sequence.

```
import unittest
from unittest.mock import MagicMock, patch
from datetime import datetime
from display_tweets import display_tweets_and_retweets, display_tweets, display_retweets,
get_info, retweet, compose_tweet
```

```
class TestSocialMediaApp(unittest.TestCase):
```

```
    def setUp(self):
        # Mock database connection and cursor
        self.conn = MagicMock()
        self.cursor = MagicMock()
        self.conn.cursor.return_value = self.cursor
```

```
@patch("builtins.input", side_effect=["3", "a", "#", ""])
```

```

def test_invalid_menu_inputs(self, mock_input):
    display_tweets_and_retweets(self.conn, 1)
    self.conn.cursor.assert_not_called() # No database calls for invalid menu inputs

@patch("builtins.input", side_effect=["1", "no"])
def test_display_tweets_no_data(self, mock_input):
    self.cursor.fetchall.return_value = [] # Simulate no tweets
    with patch("builtins.print") as mock_print:
        display_tweets_and_retweets(self.conn, 1)
        mock_print.assert_any_call("No tweets to display.")

@patch("builtins.input", side_effect=["2", "no"])
def test_display_retweets_no_data(self, mock_input):
    self.cursor.fetchall.return_value = [] # Simulate no retweets
    with patch("builtins.print") as mock_print:
        display_tweets_and_retweets(self.conn, 1)
        mock_print.assert_any_call("No retweets to display.")

@patch("builtins.input", side_effect=["1", "yes", "no"])
def test_pagination_tweets(self, mock_input):
    self.cursor.fetchall.return_value = [
        (1, "User1", "2024-01-01", "Tweet 1", None),
        (2, "User2", "2024-01-02", "Tweet 2", None),
        (3, "User3", "2024-01-03", "Tweet 3", None),
        (4, "User4", "2024-01-04", "Tweet 4", None),
        (5, "User5", "2024-01-05", "Tweet 5", None),
        (6, "User6", "2024-01-06", "Tweet 6", None),
    ]
    with patch("builtins.print") as mock_print:
        display_tweets_and_retweets(self.conn, 1)
        mock_print.assert_any_call("Tweet ID: 6")

@patch("builtins.input", side_effect=["1", "yes", "1", "reply", "Retweet Reply", ""])
def test_reply_to_tweet(self, mock_input):
    self.cursor.fetchall.return_value = [(1, "User1", "2024-01-01", "Tweet 1", None)]
    with patch("compose_tweet") as mock_compose_tweet:
        display_tweets_and_retweets(self.conn, 1)
        mock_compose_tweet.assert_called_once_with(self.conn, 1, "Retweet Reply", 1)

@patch("builtins.input", side_effect=["1", "yes", "1", "retweet", ""])
def test_retweet(self, mock_input):
    self.cursor.fetchall.return_value = [(1, "User1", "2024-01-01", "Tweet 1", None)]
    with patch("builtins.print") as mock_print:
        display_tweets_and_retweets(self.conn, 1)
        mock_print.assert_any_call("Sucesfully retweeted Tweet 1")

@patch("builtins.input", side_effect=["1", "yes", "invalid_id"])
def test_invalid_tweet_id_selection(self, mock_input):
    self.cursor.fetchall.return_value = [(1, "User1", "2024-01-01", "Tweet 1", None)]

```

```

with patch("builtins.print") as mock_print:
    display_tweets_and_retweets(self.conn, 1)
    mock_print.assert_any_call("Invalid ID.")

@patch("builtins.input", side_effect=["1; DROP TABLE users; --", "' OR '1'='1", ""])
def test_sql_injection_attempts(self, mock_input):
    display_tweets_and_retweets(self.conn, 1)
    self.conn.cursor.assert_not_called() # Ensure database is not compromised by SQL
injection

def test_get_info(self):
    self.cursor.fetchone.side_effect = [(10,), (5,)]
    with patch("builtins.print") as mock_print:
        get_info(self.conn, 1)
        mock_print.assert_any_call("Number of retweets for Tweet 1 is 10")
        mock_print.assert_any_call("Number of replies for Tweet 1 is 5")

def test_retweet_function(self):
    with patch("builtins.print") as mock_print:
        retweet(self.conn, 1, 1)
        self.cursor.execute.assert_called_once_with(
            "INSERT INTO retweets (usr, tid, rdate) VALUES (?, ?, ?)", (1, 1,
datetime.now().strftime('%Y-%m-%d'))
        )
        mock_print.assert_any_call("Succesfully retweeted Tweet 1")

@patch("builtins.input", side_effect=["1", "no"])
def test_special_characters_in_tweets(self, mock_input):
    self.cursor.fetchall.return_value = [(1, "User1", "2024-01-01", "Hello 😊", None)]
    with patch("builtins.print") as mock_print:
        display_tweets_and_retweets(self.conn, 1)
        mock_print.assert_any_call("Text: Hello 😊")

if __name__ == "__main__":
    unittest.main()

```

Figure 1C: ChatGPT output for unit tests for display_tweets.py outlined in Table 2C.

Appendix D

Prompt 1D: “The display_tweets_and_retweets function in a Python-based social media application allows users to view tweets or retweets from accounts they follow, displaying content in fives and enabling actions such as replying or retweeting. Tweets and retweets are fetched from a database based on the user's follow list, ordered by date, and presented five at a time, with clear feedback if no content exists. Black box testing focused on validating functionality, including correct content display, handling invalid inputs, managing edge cases like long tweets and special characters, and ensuring security against SQL injection. Your task is to generate comprehensive black box test cases covering displaying the correct expected tweet, invalid inputs, edge cases (long/special characters), and security aspects like SQL injection. Use equivalence partitioning, error guessing, and failure testing as black box techniques.”

Table 1D: The resulting table, showing failing tests for the black box testing method. 20/20 Passed

Test Case #	Inputs	Description	Expected Results	Actual Results
F-ST01	keywords = ""; DROP TABLE tweets; --"	Attempt SQL injection to delete tweets table.	Input sanitized, no SQL executed, safe error or "No matching tweets found." message. No database corruption.	"No matching tweets found." (Pass)
F-ST02	keywords = "a"*1000	Extremely long keyword (1000 'a' chars) to stress test input handling and performance.	System handles gracefully, likely returns "No matching tweets found." No crash or slowdown.	"No matching tweets found." (Pass)
F-ST03	keywords = "%^&*()<>?"	Keyword with only special symbols, trying to confuse parsing or DB query.	No matches found, displays "No matching tweets found." No crash or undefined behavior.	"No matching tweets found." (Pass)
F-ST04	keywords = " " (4 spaces only)	Pure whitespace input, tests handling of invalid/no valid keywords.	"No valid keywords provided. Please enter keywords to search." message displayed.	"No valid keywords provided. Please enter keywords to search." (Pass)
F-ST05	keywords = "Python; DROP TABLE users;"	Another SQL injection attempt with multiple commands.	Input sanitized, no malicious execution. Might return "No matching tweets found." or safe failure message.	"No matching tweets found." (Pass)
F-ST06	keywords = "" OR '1'='1"	SQL injection attempt using boolean logic to return all tweets.	Input treated as literal, no SQL injection. Possibly returns normal results or "No matching tweets found."	"No matching tweets found." (Pass)
F-ST07	keywords = "NonExistentKey word NonExistentKeyw ord"	Multiple non-existent keywords to overwhelm search conditions.	"No matching tweets found." System stable, no crash due to complex OR conditions.	"No matching tweets found." (Pass)
F-ST08	keywords = "python python python"	Repetitive same keyword multiple times to test redundancy handling.	Returns tweets containing "python". No duplicates or performance issues.	Returns tweets containing "python" without duplicates or performance issues. (Pass)
F-ST09	keywords = "!!!!python!!!!"	Keyword padded with many special chars around a valid token.	Trims or interprets keyword literally, likely no matches, "No matching tweets found."	"No matching tweets found." (Pass)
F-ST10	keywords = "\"" (just a single quote)	Single quote character to test input parsing and quoting logic.	Treated as a keyword. Likely no matches, "No matching tweets found." No crash.	"No matching tweets found." (Pass)
F-ST11	keywords = "pyth\0on"(contains null char)	Attempt to break string parsing with a null character.	Treated as a literal odd keyword. No crash, no matches => "No matching tweets found."	"No matching tweets found." (Pass)
F-ST12	keywords = "SELECT *	User enters a raw SQL query as a	Treated as literal keyword, no injection. Probably "No	"No matching tweets found." (Pass)

	FROM tweets"	keyword.	matching tweets found."	
F-ST13	keywords = "%%%%%%%%%"	Only '%' characters (SQL wildcard attempt).	Interpreted as literal. Likely no results, safe handling. "No matching tweets found."	"No matching tweets found." (Pass)
F-ST14	keywords = "\t\n"(tabs/newlines only)	Input only whitespace-like control chars.	No valid keywords, displays "No valid keywords provided..." No crash or hang.	"No valid keywords provided. Please enter keywords to search." (Pass)
F-ST15	keywords = "python#ai"(no space)	A combined token with no space. Trying to break parsing logic.	Treated as a single keyword "python#ai". Probably no matches => "No matching tweets found."	"No matching tweets found." (Pass)
F-ST16	keywords = "pyth@on"	Insert a special char in the middle of a normal keyword.	Likely no matches, displays "No matching tweets found."	"No matching tweets found." (Pass)
F-ST17	keywords = "@@@@ "	Leading/trailing spaces plus only '@@@@'.	No valid keyword after trimming? If considered invalid, "No valid keywords..." message.	"No valid keywords provided. Please enter keywords to search." (Pass)
F-ST18	keywords = "python nonenglishepop word"	Mixed script input (latin + cyrillic) to see if encoding breaks.	If no tweets match those mixed strings, "No matching tweets found." No crash.	"No matching tweets found." (Pass)
F-ST19	keywords = "Pyth\\on"(escaped backslash)	Check how backslashes are handled in input.	Treated literally. No special meaning. Likely "No matching tweets found."	"No matching tweets found." (Pass)
F-ST20	keywords = " " (many spaces)	Multiple spaces only, more than before.	Same as earlier whitespace cases: "No valid keywords provided..." message.	"No valid keywords provided. Please enter keywords to search." (Pass)

Table 2D: The resulting table, showing error guessing tests for the black box testing method. 20/20 Passed

Test Case #	Inputs	Description	Expected Result	Actual Results
E-ST01	keywords = "Python "	Leading/trailing spaces around a valid keyword.	Trims spaces, returns tweets for "Python" if available, else "No matching tweets found."	Trims spaces and returns tweets for "Python" if available, else "No matching tweets found." (Pass)
E-ST02	keywords = "NonExistentKeyword"	Keyword that doesn't match any tweet.	"No matching tweets found." Message displayed.	"No matching tweets found." (Pass)
E-ST03	keywords = "#AI"	Single keyword that might be in hashtags only.	Returns tweets with "#AI" in hashtags or "No matching tweets found." if none.	Returns tweets containing "#AI" in hashtags if available, else "No matching tweets found." (Pass)
E-ST04	keywords = "python ai"	Multiple valid keywords separated by space.	Returns union of tweets containing "python" or "ai" text/hashtags/mentions.	Returns tweets containing either "python" or "ai" as expected. (Pass)
E-ST05	keywords = "PYTHON"	Keyword in uppercase to check case insensitivity.	Matches "Python", returns results if present, else no matches.	Returns tweets containing "Python" regardless of case, else "No matching tweets found." (Pass)

E-ST06	keywords = "python123"	Alphanumeric keyword to see if partial numeric affects search.	If no tweets with "python123", "No matching tweets found." else returns matches.	"No matching tweets found." unless tweets contain "python123" exactly. (Pass)
E-ST07	keywords = "???python"	Keyword with special chars prefixed.	Likely no matches, "No matching tweets found." No crash.	"No matching tweets found." (Pass)
E-ST08	keywords = "py th on"	Keyword split by spaces incorrectly.	Treated as multiple keywords: "py", "th", "on". Possibly no matches => "No matching tweets found."	Treated as separate keywords "py", "th", "on". "No matching tweets found." if no such tweets exist. (Pass)
E-ST09	keywords = "中文"	Non-latin script keyword (Chinese).	If no matching tweets, "No matching tweets found." No crash.	"No matching tweets found." (Pass)
E-ST10	keywords = "PytHon Ai"	Mixed case for multiple keywords.	Case-insensitive match. Returns union of tweets for "python" and "ai" if any.	Returns tweets containing "python" or "ai" regardless of case if available. (Pass)
E-ST11	keywords = " " (just spaces)	Only spaces input.	"No valid keywords provided..." message shown.	"No valid keywords provided. Please enter keywords to search." (Pass)
E-ST12	keywords = "python\nai"	Newline in input (assume user pressed Enter?).	Might treat as single string "python\nai" or split by spaces only. Likely no matches.	Treats as a single keyword "python\nai" (since split by spaces). "No matching tweets found." (Pass)
E-ST13	keywords = "py@hon"	Insert '@' in the middle of "python".	No matches expected if no tweet contains "py@hon" literal. Display no matches.	"No matching tweets found." (Pass)
E-ST14	keywords = "python#"	Keyword ends with '#'.	Interpreted literally. Likely no matches, no crash.	"No matching tweets found." (Pass)
E-ST15	keywords = " #AI"	Hashtag with leading/trailing spaces.	Trims spaces, searches "#AI". Returns matching tweets or no matches.	Trims spaces and searches "#AI" if available, else "No matching tweets found." (Pass)
E-ST16	keywords = "python python"	Duplicate same keyword twice.	Should not cause error. Returns tweets containing "python" once.	Treats as two identical keywords. Returns "python" tweets without duplication or errors. (Pass)
E-ST17	keywords = "ai python"	Reverse order of multiple valid keywords.	Similar to E-ST04, returns union of "ai" or "python" tweets.	Returns tweets containing "ai" or "python" regardless of order. (Pass)
E-ST18	keywords = "pythön"	Keyword with a Unicode character (ö).	If no exact match, no tweets found. No crash, handles unicode gracefully.	"No matching tweets found." unless tweets contain "pythön" exactly. (Pass)
E-ST19	keywords = " user"	Leading space + a keyword that might appear in mentions.	Trims space, searches "user". If tweets mention "user", display them else no matches.	Trims space and searches "user". Displays matching tweets if any, else "No matching tweets found." (Pass)

E-ST20	keywords = "@python"	Keyword starting with '@', might imply mentions.	If tweets mention "python" as a term, returns them else no matches.	Searches for "@python" literally. Returns matching tweets if any, else "No matching tweets found." (Pass)
--------	----------------------	--	---	---

Table 3D: The resulting table, showing equivalence classes tests for the black box testing method. 20/20 Passed

Test Case #	Inputs	Description	Expected Result	Actual Results
EC-ST01	keywords = "Python"	Valid single keyword, normal case.	Returns tweets containing "Python" or "No matching tweets found."	Returns tweets containing "Python" if available, else "No matching tweets found." (Pass)
EC-ST02	keywords = "AI"	Another simple valid single keyword.	Returns tweets containing "AI" or no matches message.	Returns tweets containing "AI" if available, else "No matching tweets found." (Pass)
EC-ST03	keywords = "python ai"	Multiple valid keywords (both common words).	Returns union of tweets containing "python" or "ai".	Returns tweets containing either "python" or "ai" as expected. (Pass)
EC-ST04	keywords = ""	Empty input, invalid class (no keyword).	"No valid keywords provided..." message shown, no crash.	"No valid keywords provided. Please enter keywords to search." (Pass)
EC-ST05	keywords = "NonExistentKey word"	Valid format but no results.	"No matching tweets found." message.	"No matching tweets found." (Pass)
EC-ST06	keywords = "#AI"	Testing hashtag equivalence class: searching a hashtag.	Returns tweets containing "#AI" or no matches if none.	Returns tweets containing "#AI" if available, else "No matching tweets found." (Pass)
EC-ST07	keywords = "@User1"	Testing mentions: searching by mention.	Returns tweets mentioning "User1" or no matches message.	Returns tweets mentioning "User1" if available, else "No matching tweets found." (Pass)
EC-ST08	keywords = "python #AI"	Multiple keywords mixing text and hashtag.	Returns tweets matching "python" or "#AI".	Returns tweets matching "python" or "#AI" as expected. (Pass)
EC-ST09	keywords = " " (only spaces)	Invalid class: only spaces.	"No valid keywords provided..." message.	"No valid keywords provided. Please enter keywords to search." (Pass)
EC-ST10	keywords = "PyThOn"	Case variation class: mixed case keyword.	Case-insensitive match. Returns matches for "Python".	Returns tweets containing "Python" regardless of case if available, else "No matching tweets found." (Pass)
EC-ST11	keywords = "python python"	Duplicate keyword class.	Treated as two identical keywords. Returns "python" tweets once, no error.	Treats as two identical keywords. Returns "python" tweets without duplication or errors. (Pass)
EC-ST12	keywords = "python;ai"	Mixed punctuation with no space.	Treated as single keyword "python;ai". If no matches, no tweets found message.	Treated as a single keyword "python;ai". "No matching tweets found." unless such tweets exist. (Pass)
EC-ST13	keywords = "py-thon"	Keyword with a hyphen.	Literal match required. If no tweets contain "py-thon," no matches.	"No matching tweets found." (Pass)

EC-ST14	keywords = "python%20"	Keyword simulating URL-encoded space.	Treated literally. Probably no matches => no matching tweets found.	"No matching tweets found." (Pass)
EC-ST15	keywords = "python ai user"	Multiple keywords more than two.	Returns tweets containing any of "python", "ai", or "user".	Returns tweets containing "python", "ai", or "user" as expected. (Pass)
EC-ST16	keywords = "#Python"	Keyword starting with '#', testing hashtag scenario again.	Returns tweets tagged "#Python" or no matches.	Returns tweets tagged "#Python" if available, else "No matching tweets found." (Pass)
EC-ST17	keywords = "python123"	Alphanumeric keyword class.	If no exact match, no results. Otherwise, display matching tweets.	"No matching tweets found." unless tweets contain "python123" exactly. (Pass)
EC-ST18	keywords = "python ai ???"	Mixed valid and nonsense keyword.	Returns union of "python", "ai", and likely no result for "???". Valid still shows python/ai results if any.	Returns tweets containing "python" or "ai" regardless of "???". "No matching tweets found." for "???". (Pass)
EC-ST19	keywords = "unicodeÛ"	Keyword with accented/unicode character in last position.	If no tweet matches "unicodeÛ", no results. No crash.	"No matching tweets found." unless tweets contain "unicodeÛ" exactly. (Pass)
EC-ST20	keywords = "py"	Very short keyword.	If tweets containing "py" (as substring or exact?), returns them or no matches found.	Returns tweets containing "py" as substring or exact match if available, else "No matching tweets found." (Pass)

White box:

"I have a `search_tweets` function in a Python-based social media application that interacts with the following SQLite database schema: Users: Stores user information, Follows: Tracks which users follow others, Tweets: Stores tweets authored by users, and each tweet can be a reply to another tweet, Hashtags: Stores unique hashtags, Mentions: Associates tweets with hashtags they mention, Retweets: Tracks retweets made by users, Lists and includes: Manage user-created lists and their members. The `search_tweets` function allows users to search for tweets based on keywords. It retrieves tweets containing the provided keywords in their text, hashtags, or mentions and orders the results by date in descending order. *Provided chatgpt with the output using the .schema command in sqlite3, and data dumped the tweets table in a sample file to show as well.* Generate a set of test cases for the `search_tweets` function. Each test case should include: Test Case Number, Inputs, Description, Expected Output. Using White Box Testing, I need unittest code that achieves branch coverage and statement coverage. Use mocking to simulate database connections, user inputs, and external function calls (e.g., compose_tweet, retweet). Ensure that all edge cases (e.g., no tweets available, invalid inputs, SQL injection attempts, empty tweets) are covered. Focus on SQL injections as these tests showed actual vulnerabilities through black box testing methods during manual testing. *Provided chatgpt with the code structure, as well as the entire code for the functions, including main.*"

Prompt 2D: Prompt passed to ChatGPT for white box tests

Table 4D: The resulting table, showing tests for the white box testing method. 7/14 Passed

Test ID	Inputs	Description	Expected Results	
TC-01	Keywords: " "	Test behavior when no valid keywords are provided (empty/whitespace).	Message: "No valid keywords provided. Please enter keywords to search."	Pass
TC-02	Keywords: "python", Mocked Tweets: []	Test behavior when a valid keyword is provided but no matching tweets are found.	Message: "No matching tweets found."	Pass

TC-03	Keywords: "python", Mocked Tweets: ≤5 tweets, Input Sequence: 'back'	Test displaying up to five tweets and user opting to go back immediately.	Tweets are listed, followed by prompt: "Enter the number of a tweet to see more information, '0' to view more, or 'back' to go back: "	Fail
TC-04	Keywords: "python", Mocked Tweets: >5 tweets, Input Sequence: '0', 'back'	Test pagination by viewing more tweets once and then opting to go back.	Additional tweets are displayed, followed by the same prompt.	Fail
TC-05	Keywords: "python", Mocked Tweets: Valid tweet selection, Input Sequence: '1', 'R', 'back'	Test retweeting a selected tweet successfully.	retweet function is called with correct parameters, Message: "Successfully Tweeted"	Fail
TC-06	Keywords: "python", Mocked Tweets: Valid tweet selection, Input Sequence: '1', 'C', 'My reply', 'back'	Test composing a reply to a selected tweet successfully.	compose_tweet function is called with correct parameters, Message: "Successfully replied"	Fail
TC-07	Keywords: "python", Mocked Tweets: Already retweeted tweet, Input Sequence: '1', 'R', 'R', 'back'	Test attempting to retweet a tweet that has already been retweeted.	Message: "You have already retweeted this tweet"	Fail
TC-08	Keywords: "python", Mocked Tweets: Invalid tweet selection, Input Sequence: '9999', 'back'	Test behavior when an invalid tweet number is selected.	Message: "Invalid tweet number. Please try again."	Fail
TC-09	Keywords: "python", Mocked Tweets: DROP TABLE tweets, Mocked Tweets: []	Test for SQL injection attempt with malicious keywords.	Message: "No matching tweets found."	Pass
TC-10	Keywords: "python", Mocked Tweets: []	Test behavior with multiple keywords resulting in no matches.	Message: "No matching tweets found."	Pass
TC-11	Keywords: "python", Mocked Tweets: Large list (e.g., 1,000 tweets), Input Sequence: '0', 'back'	Test handling of a large number of tweets, ensuring pagination works without performance issues.	Tweets are displayed in manageable batches, followed by the same prompt.	Fail
TC-12	Keywords: "python", Mocked Tweets: Tweets with and without replyto, Input Sequence: 'back'	Test displaying tweets that are replies to other tweets correctly.	Replies are displayed with correct linkage, no errors occur.	Pass

TC-13	Keywords: "python", Mocked Tweets: Inconsistent retweetsdata, Input Sequence: 'l', 'R', 'back'	Test behavior when the retweets table has inconsistent or missing data.	Program handles inconsistencies gracefully, displaying appropriate messages or skipping invalid records.	Pass
TC-14	Keywords: "python", Mocked Tweets: Special characters, Input Sequence: 'back'	Test behavior when keywords contain special characters.	System handles special characters without errors, displaying appropriate messages if no matches are found.	Pass

White box code:

```
import unittest
from unittest.mock import patch, MagicMock
from datetime import datetime
import sqlite3

from search_tweets import search_tweets # Ensure this path is correct

def input_side_effect(*responses, default='no'):
    """
    A generator that yields a sequence of responses followed by a default value indefinitely.

    Args:
        *responses: Initial responses to yield.
        default (str): The default response to yield after the initial responses.

    Yields:
        str: The next response.
    """
    for response in responses:
        yield response
    while True:
        yield default

class TestSearchTweetsWhiteBox(unittest.TestCase):
    def setUp(self):
        print("Setting up the test")
        # Create a mock connection and cursor with spec_set to prevent adding attributes
        self.conn = MagicMock(spec_set=sqlite3.Connection)
        self.cursor = MagicMock(spec_set=sqlite3.Cursor)
        self.conn.cursor.return_value = self.cursor

        # By default, fetchone returns (0,) to simulate no retweets or replies
        self.cursor.fetchone.return_value = (0,)

    def tearDown(self):
        print("Tearing down the test")
```

```

self.conn = None
self.cursor = None

def test_simple_pass(self):
    print("Running test_simple_pass")
    self.assertTrue(True)

# 1. No valid keywords (empty/whitespace)
@patch('builtins.print')
def test_no_valid_keywords(self, mock_print):
    print("Running test_no_valid_keywords")
    search_tweets(self.conn, user_id=1, keywords=" ")
    mock_print.assert_any_call("No valid keywords provided. Please enter keywords to search.")

# 2. Valid keyword but no results
@patch('builtins.print')
def test_no_results_found(self, mock_print):
    print("Running test_no_results_found")
    self.cursor.fetchall.return_value = []
    search_tweets(self.conn, user_id=1, keywords="python")
    mock_print.assert_any_call("No matching tweets found.")

# 3. ≤5 tweets, user ends pagination immediately (no more tweets prompt)
@patch('builtins.print')
def test_results_found_less_equal_5_tweets(self, mock_print):
    print("Running test_results_found_less_equal_5_tweets")
    # Initialize the input generator with 'no' and then 'back'
    input_gen = input_side_effect('no', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None),
        (420, 97, '2023-02-12', '#Edmonton #Oilers had a good game last night.', None)
    ]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

    # Assertions
    mock_print.assert_any_call("1. Tweet ID: 69")
    mock_print.assert_any_call("2. Tweet ID: 420")
    mock_print.assert_any_call("Enter the number of a tweet to see more information, '0' to
view more, or 'back' to go back: ")

# 4. >5 tweets scenario, user pages once and then says 'no'
@patch('builtins.print')
def test_results_more_than_5_tweets_paging(self, mock_print):
    print("Running test_results_more_than_5_tweets_paging")

```

```

# Initialize the input generator with 'yes' to view more, then 'back' to stop
input_gen = input_side_effect('yes', 'back')

# Mock fetched tweets
tweets = [
    (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None),
    (420, 97, '2023-02-12', '#Edmonton #Oilers had a good game last night.', None),
    (500, 5, '2023-03-01', 'Go oliers!', 420),
    (9393, 29, '2023-11-04 20:03:59', '#Ben This is a test tweet lollll', None),
    (4808, 29, '2023-11-04 20:04:50', 'Another test tweet! #Bomboclat', None),
    (5720, 29, '2023-11-04 20:06:55', 'Hello, this is test numero 3 for fixing hashtag
terms #bomboclat', None)
]
self.cursor.fetchall.return_value = tweets

with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
    search_tweets(self.conn, user_id=1, keywords="python")

# Assertions
mock_print.assert_any_call("1. Tweet ID: 69")
mock_print.assert_any_call("2. Tweet ID: 420")
mock_print.assert_any_call("3. Tweet ID: 500")
mock_print.assert_any_call("4. Tweet ID: 9393")
mock_print.assert_any_call("5. Tweet ID: 4808")
mock_print.assert_any_call("6. Tweet ID: 5720")
mock_print.assert_any_call("Enter the number of a tweet to see more information, '0' to
view more, or 'back' to go back: ")

# 5. Retweet a new tweet
@patch('search_tweets.retweet', return_value=None)
@patch('search_tweets.compose_tweet', return_value=None)
@patch('builtins.print')
def test_retweet_new_tweet(self, mock_print, mock_compose, mock_retweet):
    print("Running test_retweet_new_tweet")
    # Initialize the input generator with 'no', 'yes', '69', 'R', 'back'
    input_gen = input_side_effect('no', 'yes', '69', 'R', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None)
    ]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

# Assertions
mock_retweet.assert_called_once_with(self.conn, 1, 69)

```

```

mock_print.assert_any_call("Successfully Tweeted")

# 6. Compose a reply successfully
@patch('search_tweets.retweet', return_value=None)
@patch('search_tweets.compose_tweet', return_value=None)
@patch('builtins.print')
def test_compose_reply_in_tweet_info(self, mock_print, mock_compose, mock_retweet):
    print("Running test_compose_reply_in_tweet_info")
    # Initialize the input generator with 'no', 'yes', '69', 'C', 'My reply', 'back'
    input_gen = input_side_effect('no', 'yes', '69', 'C', 'My reply', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None)
    ]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

    # Assertions
    mock_compose.assert_called_once_with(self.conn, 1, 'My reply', 69)
    mock_print.assert_any_call("Successfully replied")

# 7. Retweet already retweeted scenario
@patch('search_tweets.retweet', return_value=None)
@patch('search_tweets.compose_tweet', return_value=None)
@patch('builtins.print')
def test_retweet_already_retweeted(self, mock_print, mock_compose, mock_retweet):
    print("Running test_retweet_already_retweeted")
    # Initialize the input generator with 'no', 'yes', '69', 'R', 'R', 'back'
    input_gen = input_side_effect('no', 'yes', '69', 'R', 'R', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None)
    ]

    # Mock fetchone to return (0,) for first retweet, then (1,) for second attempt
    self.cursor.fetchone.side_effect = [(0,), (1,)]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

    # Assertions
    mock_retweet.assert_called_once_with(self.conn, 1, 69)
    mock_print.assert_any_call("You have already retweeted this tweet")

```



```

# 8. Invalid tweet ID scenario
@patch('search_tweets.retweet', return_value=None)
@patch('search_tweets.compose_tweet', return_value=None)
@patch('builtins.print')
def test_invalid_tweet_id(self, mock_print, mock_compose, mock_retweet):
    print("Running test_invalid_tweet_id")
    # Initialize the input generator with 'no', 'yes', '9999', 'back'
    input_gen = input_side_effect('no', 'yes', '9999', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None)
    ]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

    # Assertions
    mock_print.assert_any_call("Invalid ID.")

# 9. SQL injection attempt scenario
@patch('builtins.print')
def test_sql_injection_no_results(self, mock_print):
    print("Running test_sql_injection_no_results")
    self.cursor.fetchall.return_value = []
    search_tweets(self.conn, user_id=1, keywords="python; DROP TABLE tweets;")
    mock_print.assert_any_call("No matching tweets found.")

# 10. Multiple keywords no results
@patch('builtins.print')
def test_multiple_keywords_no_results(self, mock_print):
    print("Running test_multiple_keywords_no_results")
    self.cursor.fetchall.return_value = []
    search_tweets(self.conn, user_id=1, keywords="python ai")
    mock_print.assert_any_call("No matching tweets found.")

# 11. Multiple keywords with results, user chooses no pagination, no info
@patch('builtins.print')
def test_multiple_keywords_with_results_no_info(self, mock_print):
    print("Running test_multiple_keywords_with_results_no_info")
    # Initialize the input generator with 'no', 'back'
    input_gen = input_side_effect('no', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None),
        (420, 97, '2023-02-12', '#Edmonton #Oilers had a good game last night.', None)
    ]

```

```

]

with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
    search_tweets(self.conn, user_id=1, keywords="python ai")

# Assertions
mock_print.assert_any_call("1. Tweet ID: 69")
mock_print.assert_any_call("2. Tweet ID: 420")
mock_print.assert_any_call("Enter the number of a tweet to see more information, '0' to
view more, or 'back' to go back: ")

# 12. Special characters in keyword no results
@patch('builtins.print')
def test_special_chars_no_results(self, mock_print):
    print("Running test_special_chars_no_results")
    self.cursor.fetchall.return_value = []
    search_tweets(self.conn, user_id=1, keywords="@#@###")
    mock_print.assert_any_call("No matching tweets found.")

# 13. Case-insensitivity scenario
@patch('builtins.print')
def test_case_insensitivity(self, mock_print):
    print("Running test_case_insensitivity")
    # Initialize the input generator with 'no', 'back'
    input_gen = input_side_effect('no', 'back')

    # Mock fetched tweets
    self.cursor.fetchall.return_value = [
        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None)
    ]

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="PyThOn")

    # Assertions
    mock_print.assert_any_call("1. Tweet ID: 69")
    mock_print.assert_any_call("Enter the number of a tweet to see more information, '0' to
view more, or 'back' to go back: ")

# 14. "No more tweets." scenario
@patch('builtins.print')
def test_no_more_tweets(self, mock_print):
    print("Running test_no_more_tweets")
    # Initialize the input generator with 'yes', 'yes', 'no', 'back'
    input_gen = input_side_effect('yes', 'yes', 'no', 'back')

    # Mock fetched tweets
    tweets = [

```

```

        (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None),
        (420, 97, '2023-02-12', '#Edmonton #Oilers had a good game last night.', None),
        (500, 5, '2023-03-01', 'Go oliers!', 420),
        (9393, 29, '2023-11-04 20:03:59', '#Ben This is a test tweet lollll', None),
        (4808, 29, '2023-11-04 20:04:50', 'Another test tweet! #Bomboclat', None),
        (5720, 29, '2023-11-04 20:06:55', 'Hello, this is test numero 3 for fixing hashtag
terms #bomboclat', None),
        (4157, 29, '2023-11-04 22:14:03', 'this is test lol #booooo', None),
        (8249, 29, '2023-11-04', 'test1', None),
        (29, 29, '2023-11-04', 'test2', None),
        (6807, 29, '2023-11-04', 'test3', None)
    ]
    self.cursor.fetchall.return_value = tweets

    with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
        search_tweets(self.conn, user_id=1, keywords="python")

    # Assertions
    mock_print.assert_any_call("No more tweets.")

    # 15. Large number of tweets, user only pages once then says 'no'
    @patch('builtins.print')
    def test_large_tweets_one_page(self, mock_print):
        print("Running test_large_tweets_one_page")
        # Initialize the input generator with 'yes', 'no', 'back'
        input_gen = input_side_effect('yes', 'no', 'back')

        # Mock fetched tweets
        tweets = [
            (69, 5, '2023-06-01', 'Looking for a good book to read. Just finished lone #survivor',
None),
            (420, 97, '2023-02-12', '#Edmonton #Oilers had a good game last night.', None),
            (500, 5, '2023-03-01', 'Go oliers!', 420),
            (9393, 29, '2023-11-04 20:03:59', '#Ben This is a test tweet lollll', None),
            (4808, 29, '2023-11-04 20:04:50', 'Another test tweet! #Bomboclat', None),
            (5720, 29, '2023-11-04 20:06:55', 'Hello, this is test numero 3 for fixing hashtag
terms #bomboclat', None),
            (4157, 29, '2023-11-04 22:14:03', 'this is test lol #booooo', None),
            (8249, 29, '2023-11-04', 'test1', None),
            (29, 29, '2023-11-04', 'test2', None),
            (6807, 29, '2023-11-04', 'test3', None),
            (9116, 29, '2023-11-06', '', 420),
            (9892, 29, '2023-11-06', '', 420),
            (755, 29, '2023-11-06', '', 420),
            (3054, 29, '2023-11-06', '', 420),
            (4680, 29, '2023-11-06', 'They had a good game wow', 420),
            (8012, 29, '2023-11-06', 'The had a good game bomboclat', 420),
            (3171, 29, '2023-11-06', '', 420),

```

```

(2831, 29, '2023-11-06', 'lol bomobclat', 420),
(7882, 29, '2023-11-06', 'hello lol', 420),
(250, 29, '2023-11-06', '', 420),
(1264, 29, '2023-11-06', 'Hello this is a test reply', 420),
(3767, 29, '2023-11-06', '', 420),
(1799, 29, '2023-11-06', 'Hello this is another test', 420),
(4306, 29, '2023-11-06', '', 420),
(4474, 29, '2023-11-06', '', 420),
(6745, 29, '2023-11-06 13:37:25', '', 420),
(2525, 29, '2023-11-06 13:37:27', '', 420),
(4441, 29, '2023-11-06 13:37:30', 'this is a test', 420),
(7579, 29, '2023-11-06 13:39:24', '', 420),
(262, 29, '2023-11-06 13:39:42', '', 420),
(5837, 29, '2023-11-06 13:39:51', 'u suck', 420),
(5137, 6969, '2023-11-06 13:46:19', '', 1264),
(9607, 6969, '2023-11-06 13:46:33', '', 1264),
(4585, 6969, '2023-11-06 13:46:49', 'looooo111', 1264)
]
self.cursor.fetchall.return_value = tweets

with patch('builtins.input', side_effect=lambda *args, **kwargs: next(input_gen)):
    search_tweets(self.conn, user_id=1, keywords="python")

# Assertions
mock_print.assert_any_call("1. Tweet ID: 69")
mock_print.assert_any_call("2. Tweet ID: 420")
mock_print.assert_any_call("3. Tweet ID: 500")
mock_print.assert_any_call("4. Tweet ID: 9393")
mock_print.assert_any_call("5. Tweet ID: 4808")
mock_print.assert_any_call("6. Tweet ID: 5720")
mock_print.assert_any_call("7. Tweet ID: 4157")
mock_print.assert_any_call("8. Tweet ID: 8249")
mock_print.assert_any_call("9. Tweet ID: 29")
mock_print.assert_any_call("10. Tweet ID: 6807")
mock_print.assert_any_call("Enter the number of a tweet to see more information, '0' to
view more, or 'back' to go back: ")

if __name__ == '__main__':
    unittest.main(verbosity=2)

```

Figure 1D: The resulting code produced by chatgpt for whitebox tests

Prompt 1D: “I have a `search_tweets` function in a Python-based social media application that interacts with the following SQLite database schema: `Users`: Stores user information, `Follows`: Tracks which users follow others, `Tweets`: Stores tweets authored by users, and each tweet can be a reply to another tweet, `Hashtags`: Stores unique hashtags, `Mentions`: Associates tweets with hashtags they mention, `Retweets`: Tracks retweets made by users, `Lists and includes`: Manage user-created lists and their members. The `search_tweets` function allows users to search for tweets based on keywords. It retrieves tweets containing the provided keywords in their text, hashtags, or mentions and orders the results by date in descending order. Please list be “input, description Using black box testing techniques such as failure testing, error guessing, and equivalence partitioning, generate a set of test cases for the `search_tweets` function. Each test case should include: Test Case Number, Inputs, Description, Expected Output Ensure the test cases cover: Valid searches (single keyword, multiple keywords), Invalid inputs (empty keyword, non-existent keywords), Edge cases (extremely long keywords, keywords with special characters), Security aspects (SQL injection attempts), Case sensitivity in keyword matching, Keywords present in different parts of the

tweet (text, hashtags, mentions). Specific prompts: Using failure testing techniques, generate test cases for the search_tweets function that attempt to cause the system to fail. Consider inputs that are likely to break the function or expose vulnerabilities. Using error guessing, create test cases for the search_tweets function. Think about common mistakes users might make or unexpected inputs that could lead to errors. Use equivalence partitioning to develop test cases for the search_tweets function. Divide the input data into equivalent classes to ensure comprehensive coverage with minimal redundancy."

Appendix E

"I need help designing black-box test cases for a system with the following functionality: The system is a social media application that allows users to search for other users by entering a keyword. The search results are displayed as a list of matching users. If there are no matches, an appropriate message is shown (e.g., 'No matching users found'). The function also includes interactive prompts for viewing more results (if available), selecting a user for more details, or exiting the search. Using failure testing, error guessing, and equivalence partitioning, create test cases that evaluate the following aspects. Functional correctness: Ensure valid keywords return the correct results, and verify appropriate messages for invalid or unmatched inputs. Edge cases: Test inputs like empty strings, excessively long keywords, or keywords with special characters. Handle cases where there are no results or a large number of results. Security and robustness: Evaluate the system's behavior with invalid or malicious inputs (e.g., SQL injection attempts or inputs exceeding expected lengths). Interaction behavior: Verify user interaction for options like viewing more results, selecting a user, or exiting the search."

Prompt 1E: Prompt to generate black box tests

Table 1E: Black Box testing, error guessing, failure, equivalence partitioning test cases created by ChatGPT and manually tested for search_users.py. 10/11 Passed

Test Case ID	Test Scenario	Input	Expected Result	Actual
TC01	Enter an empty keyword for search	""	Message displayed: "Search term cannot be empty."	"no keyword to search for"
TC02	Search keyword with special characters (e.g., @)	"@@"	Either results are displayed correctly or an error message appears if unsupported.	results displayed correctly
TC03	Search keyword exceeds character limit	"dddddd..."	Application truncates input or displays a warning: "Search term too long."	still searched for matching users
TC04	Select a user whose data is incomplete (e.g., missing city)	selected user "BEN@@"	Application gracefully handles missing data without crashing.	showed correct information
Test Case ID	Test Scenario		Expected Result	Actual
TC05	Search for users with an invalid keyword input	"ewewewqeq"	Message displayed: "No users found matching your search."	"No matching users found."
TC06	Search when the user database is empty	"User1"	Message displayed: "No users available to search."	"No matching users found."
TC07	Search with a large number of matching users	"User"	Results are paginated, showing 5 users at a time with the option to load more.	Pass
Test Case ID	Test Scenario		Expected Result	
TC08	Search for users with a valid keyword	"User"	Matching users are displayed, ordered by name and city length.	Pass
TC09	Select a user from the results and view details	"1"	User details (tweet count, followers, recent tweets, etc.) are displayed accurately.	Pass
TC10	Attempt to follow a user from	"2"	User is successfully followed; a success	Pass

	the search results		message is displayed.	
TC11	Search for users repeatedly in quick succession	repeated searches	System maintains responsiveness without performance degradation.	Pass

"I need help designing white-box test cases and the corresponding test code for a function in a social media application. The function, `search_users(conn, keyword, user_id)`, interacts with a database to search for users based on a keyword. It prints matching users' details, handles various user interactions (e.g., viewing more results, selecting a user, or exiting), and allows following users or viewing their details. The function includes loops, conditional statements, and database queries, making it essential to test for code coverage and logic correctness. Please create white-box test cases and the corresponding test code for the following:

1. *Statement Coverage: Ensure every line of code is executed at least once.*
2. *Branch Coverage: Verify all decision points, such as conditions and loops, are exercised.*
3. *Path Coverage: Test all possible execution paths to ensure logic is sound.*
4. *Boundary Value Analysis: Evaluate edge cases for inputs, such as zero results, maximum results, or long keywords.*
5. *Error Handling: Test scenarios where database queries fail, user inputs are invalid, or interaction prompts are mishandled.*
6. *Performance: Validate that the function handles large datasets efficiently and maintains responsiveness under repeated use.*

Additionally, write the Python code for the test cases using a framework like unittest or pytest. Ensure the test code includes any necessary setup and runs without requiring modifications to the original function implementation"

Prompt 2E: ChatGPT prompt to generate white box tests

```
import pytest
from unittest.mock import patch
from datetime import datetime
import sqlite3
from search_users import search_users
# Mock functions
def mock_get_user_info(conn, user_id):
    return {
        'tweet_count': 10,
        'following_count': 5,
        'followers_count': 20,
        'recent_tweets': [(1, user_id, '2024-12-01', 'Tweet 1'),
                          (2, user_id, '2024-12-02', 'Tweet 2'),
                          (3, user_id, '2024-12-03', 'Tweet 3')]
    }

def mock_display_user_info(user_info):
    print("Mock user info displayed.")

# Fixtures for mocking database and input
@pytest.fixture
def mock_db():
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()

    # Create tables with specified schema
    cursor.execute("""
        CREATE TABLE users (
            usr INTEGER PRIMARY KEY,
            pwd TEXT,
            name TEXT,
```

```

        email TEXT,
        city TEXT,
        timezone REAL
    );
    """
    cursor.execute("""
        CREATE TABLE follows (
            flwer INTEGER,
            flwee INTEGER,
            start_date DATE,
            PRIMARY KEY (flwer, flwee),
            FOREIGN KEY (flwer) REFERENCES users(usr),
            FOREIGN KEY (flwee) REFERENCES users(usr)
        );
    """)
    cursor.execute("""
        CREATE TABLE tweets (
            tid INTEGER PRIMARY KEY,
            writer INTEGER,
            tdate TEXT,
            text TEXT,
            FOREIGN KEY (writer) REFERENCES users(usr)
        );
    """)
    # Insert mock data
    cursor.executemany("INSERT INTO users (usr, pwd, name, email, city, timezone) VALUES (?, ?, ?,
?, ?, ?)", [
        (1, 'pwd1', 'Alice', 'alice@example.com', 'New York', -5.0),
        (2, 'pwd2', 'Bob', 'bob@example.com', 'Los Angeles', -8.0),
        (3, 'pwd3', 'Charlie', 'charlie@example.com', 'New York', -5.0)
    ])
    conn.commit()
    yield conn
    conn.close()

@patch('search_users.get_user_info', side_effect=mock_get_user_info)
@patch('search_users.display_user_info', side_effect=mock_display_user_info)
def test_search_users_with_updated_schema(mock_get_info, mock_display_info, mock_db, capsys):
    conn = mock_db
    cursor = conn.cursor()

    # Case 1: Valid search by name
    with patch('builtins.input', side_effect=['0', 'back']):
        search_users(conn, 'Alice', 1)
        captured = capsys.readouterr()
        assert "Alice" in captured.out

    # Case 2: Valid search by city
    with patch('builtins.input', side_effect=['0', 'back']):

```

```

    search_users(conn, 'New York', 1)
    captured = capsys.readouterr()
    assert "Charlie" in captured.out

# Case 3: Empty keyword
search_users(conn, '', 1)
captured = capsys.readouterr()
assert "no keyword to search for" in captured.out

# Case 4: Already following
cursor.execute("INSERT INTO follows (flwer, flwee, start_date) VALUES (?, ?, ?)", (1, 2,
'2024-12-01'))
conn.commit()
with patch('builtins.input', side_effect=['1', '0', 'back']):
    search_users(conn, 'Bob', 1)
    captured = capsys.readouterr()
    assert "You are already following this user." in captured.out

# Case 5: Long keyword
search_users(conn, 'a' * 256, 1)
captured = capsys.readouterr()
assert "No matching users found." in captured.out

# Case 6: No results
search_users(conn, 'Nonexistent', 1)
captured = capsys.readouterr()
assert "No matching users found." in captured.out

# Case 7: Exactly 5 results
for i in range(5):
    cursor.execute("INSERT INTO users (usr, pwd, name, email, city, timezone) VALUES (?, ?, ?,
?, ?, ?)",
                    (10 + i, f'pwd{i}', f'User{i}', f'user{i}@example.com', 'City', -5.0))
conn.commit()
with patch('builtins.input', side_effect=['back']):
    search_users(conn, 'User', 1)
    captured = capsys.readouterr()
    assert "User0" in captured.out
    assert "User4" in captured.out

# Case 8: More than 5 results
for i in range(6):
    cursor.execute("INSERT INTO users (usr, pwd, name, email, city, timezone) VALUES (?, ?, ?,
?, ?, ?)",
                    (20 + i, f'pwd{i}', f'MoreUser{i}', f'moreuser{i}@example.com', 'City',
-8.0))
conn.commit()
with patch('builtins.input', side_effect=['0', 'back']):
    search_users(conn, 'MoreUser', 1)

```



```

captured = capsys.readouterr()
assert "MoreUser0" in captured.out
assert "MoreUser5" in captured.out

# Case 9: Performance test
for i in range(1000):
    cursor.execute("INSERT INTO users (usr, pwd, name, email, city, timezone) VALUES (?, ?, ?,
?, ?, ?)",
                  (1000 + i, f'pwd{i}', f'PerfUser{i}', f'perfuser{i}@example.com',
'PerfCity', -5.0))
    conn.commit()
    with patch('builtins.input', side_effect=['0', 'back']):
        search_users(conn, 'PerfUser', 1)
        captured = capsys.readouterr()
        assert "PerfUser0" in captured.out
        assert "PerfUser999" not in captured.out

```

Figure 1E: White box, coverage testing employed for search_users.py

Appendix F

Table 1F: Black Box unit tests generated by ChatGPT and manually tested for compose_tweet.py. 8/11 Passed

Test ID	Inputs	Description	Expected Output	Actual Results
1	tweet_text="This is my first tweet!"	Valid input: No hashtags or mentions.	Tweet is inserted into the tweets table. No entries in hashtags or mentions tables.	Tweet is inserted into the tweets table. No entries in hashtags or mentions tables.
2	tweet_text="Check out my new blog post! #blog #update"	Valid input: Includes hashtags.	Tweet is inserted. Hashtags #blog and #update are added to hashtags and linked in mentions.	Tweet is inserted. Hashtags #blog and #update are added to hashtags and linked in mentions.
3	tweet_text=""	Invalid input: Empty tweet.	Prints error message: "Tweet text cannot be empty." No tweet is inserted.	Fails Tweet is inserted into the tweets table.
4	tweet_text="a" * 300	Edge case: Large character input.	Prints error message: "Tweet text exceeds the allowed limit." No tweet is inserted.	Fails Tweet is inserted into the tweets table.
5	tweet_text="Invalid hashtag format ##double ##symbols!"	Invalid input: Hashtags contain invalid formatting.	Inserts tweet. Valid hashtags (double, symbols) are processed, invalid hashtags are ignored.	Inserts tweet. Valid hashtags (double, symbols) are processed, invalid hashtags are ignored.

6	tweet_text="Repeated hashtag #Test #Test is interesting!"	Edge case: Repeated hashtags in a single tweet.	Inserts tweet. Hashtag #Test is added to hashtags if new; linked once to mentions for this tweet.	Fails UNIQUE constraint failed: mentions.tid, mentions.term
7	tweet_text="Multiple hashtags #One #Two #Three"	Valid input: Multiple unique hashtags.	Inserts tweet. Hashtags #One, #Two, and #Three are added to hashtags and linked in mentions.	Inserts tweet. Hashtags #One, #Two, and #Three are added to hashtags and linked in mentions.
8	tweet_text="A tweet with a mention @user and a hashtag #example"	Valid input: Contains a mention (starting with @).	Inserts tweet. Hashtag #example is processed.	Inserts tweet. Hashtag #example is processed.
9	tweet_text="Hashtags with spaces #Hashtag One and #HashtagTwo"	Invalid input: Hashtags with spaces.	Inserts tweet. Only valid hashtags (Hashtag, HashtagTwo) are processed.	Inserts tweet. Only valid hashtags (Hashtag, HashtagTwo) are processed.
10	tweet_text="Special characters in hashtags #hashtag!"	Edge case: Hashtags containing special characters.	Inserts tweet. Only valid hashtags (#hashtag) are processed. Invalid ones (#hashtag!) are ignored.	Inserts tweet. Only valid hashtags (#hashtag) are processed. Invalid ones (#hashtag!) are ignored.
11	tweet_text="Testing case sensitivity! #Test #test #TEST"	Edge case: Hashtags differ only by case.	Inserts tweet. Processes hashtags (#Test, #test, #TEST) according to the database's case-sensitivity rules.	Inserts tweet. All three hashtags (#Test, #test, #TEST) added to database

"You are a highly skilled software developer tasked with creating comprehensive unit tests for a Python function. Below is the source code:

{compose_tweet.py}

Your Task:

1. Create Unit Tests with Branch Coverage:
 - a. Write unit tests using the unittest framework.
 - b. Ensure branch coverage for all conditional statements and branches in the function.
 - c. Include tests for both valid and invalid inputs, covering all possible code paths.
2. Mock Database Interactions:
 - a. Mock the sqlite3 connection and cursor to simulate realistic database behavior.
 - b. Ensure all database operations (execute, fetchone) are tested thoroughly with expected outcomes.
3. Expected Assertions:
 - a. Verify that the database queries are executed with the correct parameters using mock.assert_any_call.
 - b. Assert proper exceptions for invalid inputs (e.g., empty tweets or invalid replyto).
4. Output Format:
 - a. Provide the test cases in a single Python script.
 - b. Use descriptive test method names for clarity (e.g., test_valid_tweet_with_hashtags).
 - c. Include setUp and tearDown methods for test initialization and cleanup.
5. Validation:
 - a. Ensure all tests pass and the mock behavior matches the function's logic.
 - b. Use side effects for mock methods to simulate real-world scenarios (e.g., returning rows for fetchone)."

Prompt 1F: Prompt to ChatGPT to generate White Box tests for compose_tweet.py; passes in the source code as {compose_tweet.py}

```

import unittest
from unittest.mock import Mock, MagicMock
from datetime import datetime
from compose_tweet import compose_tweet

class TestComposeTweet(unittest.TestCase):
    def setUp(self):
        # Mock connection and cursor
        self.mock_conn = Mock()
        self.mock_cursor = MagicMock()
        self.mock_conn.cursor.return_value = self.mock_cursor

    def test_valid_tweet_no_hashtags(self):
        user_id = 1
        text = "This is a valid tweet without hashtags."
        replyto = None

        # Mock behavior
        self.mock_cursor.fetchone.side_effect = [(1,)] # MAX(tid) = 1

        compose_tweet(self.mock_conn, user_id, text, replyto)

        # Assertions
        self.mock_cursor.execute.assert_any_call(
            "INSERT INTO tweets (tid, writer, tdate, text,replyto) VALUES (?, ?, ?, ?,?)",
            (2, user_id, datetime.now().strftime('%Y-%m-%d'), text, replyto)
        )

    def test_valid_tweet_with_hashtags(self):
        user_id = 1
        text = "This is a #valid #tweet with hashtags."
        replyto = None

        # Mock behavior
        self.mock_cursor.fetchone.side_effect = [
            (1,), # MAX(tid) = 1
            None, # No existing hashtag #valid
            None  # No existing hashtag #tweet
        ]

        compose_tweet(self.mock_conn, user_id, text, replyto)

        self.mock_cursor.execute.assert_any_call(
            "INSERT INTO tweets (tid, writer, tdate, text,replyto) VALUES (?, ?, ?, ?,?)",
            (2, user_id, datetime.now().strftime('%Y-%m-%d'), text, replyto)
        )

    def test_tweet_with_repeated_hashtags(self):
        user_id = 1

```

```

text = "This is a #repeated #hashtag #repeated."
replyto = None

# Mock behavior
self.mock_cursor.fetchone.side_effect = [
    (1,), # MAX(tid) = 1
    None, # No existing hashtag #repeated
    None, # No existing hashtag #hashtag
    ("repeated",) # #repeated already exists
]

compose_tweet(self.mock_conn, user_id, text, replyto)

self.mock_cursor.execute.assert_any_call(
    "INSERT INTO tweets (tid, writer, tdate, text,replyto) VALUES (?, ?, ?, ?,?)",
    (2, user_id, datetime.now().strftime('%Y-%m-%d'), text, replyto)
)

# def test_empty_tweet(self):
#     # Arrange
#     user_id = 1
#     text = ""
#     replyto = None

#     # Act
#     with self.assertRaises(ValueError): # Assuming the function raises an exception for
empty tweets
#         compose_tweet(self.mock_conn, user_id, text, replyto)

#     # Assert
#     self.mock_cursor.execute.assert_not_called()
#     self.mock_conn.commit.assert_not_called()

# def test_tweet_with_invalid_hashtag(self):
#     # Arrange
#     user_id = 1
#     text = "This tweet has an invalid hashtag ##oops."
#     replyto = None

#     # Mock database calls
#     self.mock_cursor.fetchone.side_effect = [None]

#     # Act
#     compose_tweet(self.mock_conn, user_id, text, replyto)

#     # Assert
#     self.mock_cursor.execute.assert_called_with(
#         "INSERT INTO tweets (tid, writer, tdate, text,replyto) VALUES (?, ?, ?, ?,?)",

```

```

#         (1, user_id, datetime.now().strftime('%Y-%m-%d'), text, replyto)
#     )
#     self.mock_cursor.execute.assert_not_called_with("INSERT INTO hashtags (term) VALUES
(?)", ("#oops",))
#     self.mock_conn.commit.assert_called_once()

# def test_tweet_exceeds_character_limit(self):
#     # Arrange
#     user_id = 1
#     text = "a" * 300 # Exceeds the assumed 280-character limit
#     replyto = None

#     # Act & Assert
#     with self.assertRaises(ValueError): # Now the function raises the appropriate exception
#         compose_tweet(self.mock_conn, user_id, text, replyto)

#     # Assert
#     self.mock_cursor.execute.assert_not_called()
#     self.mock_conn.commit.assert_not_called()

def test_tweet_with_replyto(self):
    user_id = 1
    text = "This is a reply to another tweet."
    replyto = 5 # Assuming this is a valid replyto ID

    # Mock behavior
    self.mock_cursor.fetchone.side_effect = [(1,)] # MAX(tid) = 1

    compose_tweet(self.mock_conn, user_id, text, replyto)

    self.mock_cursor.execute.assert_any_call(
        "INSERT INTO tweets (tid, writer, tdate, text,replyto) VALUES (?, ?, ?, ?,?)",
        (2, user_id, datetime.now().strftime('%Y-%m-%d'), text, replyto)
    )

# def test_tweet_with_invalid_replyto(self):
#     # Arrange
#     user_id = 1
#     text = "This is a reply to a non-existent tweet."
#     replyto = 9999 # Assume 9999 is an invalid tweet ID

#     # Mock database calls
#     self.mock_cursor.fetchone.return_value = None # Simulates reply-to tweet not found

#     # Act
#     with self.assertRaises(ValueError): # Assuming the function raises an exception for
invalid reply-to

```

```

#         compose_tweet(self.mock_conn, user_id, text, replyto)

#     # Assert
#     self.mock_cursor.execute.assert_not_called()
#     self.mock_conn.commit.assert_not_called()

if __name__ == "__main__":
    unittest.main()

```

Figure 1F: Python script generated for White Box tests by ChatGPT; invalid tests are commented out

Appendix G

Prompt 1G: “This project is a social media application implemented in Python and SQLite. It allows users to perform various social media interactions such as composing tweets, following other users, displaying tweets, retweets, and user information, as well as searching for users and tweets based on specific keywords. The project involves database management, user interactions, and data retrieval, providing a basic social media experience within a command-line interface. One functionality is to list followers: how a list of all users who follow you and from which you can select a follower to view more information about them. After that you can follow the selected user or choose to see all their tweets. his function will first list the id of the users the logged in user follows, then the user can select one of these followers to see more information, including name, number of followers, number of tweets, and the three most recent tweets, he user then has the choice to follow this follower (if they already follow them and they select this choice they will be told they already follow), or see more tweets which will display every tweet the specified user has beyond the 3 most recent, Write me some black box test cases for the Failure method: "Test every possible thing a user could do with the system to demolish the software" For each of the tests indicate the inputs, description and expected result “

Table 1G: Black Box, failure testing test cases created by ChatGPT and manually tested for list_followers.py. 10/11 Passed

Test ID	Inputs	Description	Expected Result	Actual Result
1	Logged-in user has no followers.	No Followers Attempt to list followers when no followers exist.	Display a message: "You have no followers." No crashes or unexpected behavior.	You have no followers.
2	Enter a non-existent follower ID (e.g., an ID not in the list of followers).	Invalid Follower Selection Select an ID not displayed in the followers list	Display a message: "Invalid selection. Please select a valid follower ID." Prompt the user again for input.	Invalid follower ID. For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit:
3	Select a follower who has no tweets.	Follower with No Tweets Attempt to view tweets for a follower who hasn't tweeted.	Display a message: "This user has not posted any tweets." No further options to see tweets are provided.	User has no Tweets There is an option to see further tweets - User has no more tweets
4	Select a valid follower, then choose to view all tweets when they have more than three.	Viewing All Tweets Verify the system can handle and correctly display a large number of tweets.	All tweets are displayed in chronological order. The system doesn't crash or truncate data.	All tweets are displayed in chronological order. The system doesn't crash or truncate data.
5	Select a follower and choose the "Follow this user" option for someone already followed.	Attempting to Follow an Already Followed User: Attempt to follow a user who is already being followed.	Display a message: "You already follow this user." The application returns to the menu without issues.	You are already following this user.
6	The logged-in user attempts to select themselves as a follower (if incorrectly listed).	Attempting to Follow Yourself Ensure the system prevents a user from following	Display a message: "You cannot follow yourself." Return to the menu.	System allows you to follow yourself

		themselves.		
7	Select a follower and verify displayed information (name, number of followers, tweets).	Follower Data Integrity Ensure data matches the database for accuracy.	Correct data for the follower is displayed (e.g., accurate name, follower count, and tweets).	All data is correct
8	Simulate an empty database (no users or tweets exist).	Empty Database Test behavior when no data is available in the system.	Display appropriate messages for no users or tweets, such as "No users available." No crashes occur.	Depending on the tables removed from the db the systems responds correctly.
9	Provide special characters (e.g., ' or --) as input for follower selection.	Special Characters or SQL Injection Check if the system is vulnerable to SQL injection attacks.	The system handles input securely without executing unintended commands. Display: "Invalid input."	User is reprompted
10	Enter an invalid option (e.g., selecting a nonexistent menu number).	Invalid Option in Submenu Ensure the system handles invalid input gracefully in any submenu.	Display: "Invalid option. Please try again." Prompt for input again.	User is reprompted
11	Simulate a user with many followers	Excessively Large Number of Followers Ensure the system can load and display a large list of followers without performance issues.	Followers are paginated or displayed correctly, with no crashes or slowdowns.	Lists followers

Table 2G:Black Box error guessing test cases created by ChatGPT and tested manually for main.py. 12/13 Passed

Test Case ID	Inputs	Description	Expected Result	Actual Result
1	Press "Enter" without typing anything when asked to select a follower ID.	Test Case 1: Empty Input - Test the system's handling of blank input.	Display: "Invalid input. Please select a valid follower ID." Prompt user again for input.	"Invalid input. Please Try again."
2	Provide non-numeric input such as abc or @#\$\$ when asked to select a follower ID.	Test Case 2: Non-Numeric Input - Check if the system handles non-numeric input gracefully.	Display: "Invalid input. Please enter a valid follower ID." Prompt user again.	"Invalid input. Please Try again."
3	Enter -1 or a number greater than the total number of followers, e.g., 999 .	Test Case 3: Negative or Out-of-Range Numbers - Test the system's ability to handle invalid numeric inputs.	Display: "Invalid selection. Please select a valid follower ID." No crashes.	Invalid follower ID.
4	Select a follower whose database entry is incomplete (e.g., missing name or tweets).	Test Case 4: Follower with Corrupted Data - Verify the system's behavior when encountering corrupted or missing follower data.	Display: "Error retrieving user information. Please try again later." No crashes.	System is able to display users with null values
5	After selecting a valid follower, enter an invalid option such as 2 or 99 in the submenu.	Test Case 5: Invalid Option in Follower Submenu - Test submenu error handling for unexpected inputs.	Display: "Invalid option. Please try again." Return to submenu without exiting.	Return to previous menu

6	Select a follower whose name exceeds typical display limits (e.g., 300 characters).	Test Case 6: Follower with Excessively Long Name - Test if the system can handle followers with unusually long names.	Name is truncated appropriately, and the system remains functional. No layout or display errors.	Name displayed
7	Select a follower, then delete this follower's account from the database before viewing their tweets.	Test Case 7: Viewing Tweets from a Deleted User - Check behavior when trying to view tweets of a deleted user.	Display: "User not found. Unable to retrieve tweets." No crashes.	Tweets still show up - No cascade effect
8	Provide three consecutive invalid inputs (e.g., non-existent IDs, blank input, special characters).	Test Case 8: Multiple Invalid Inputs - Test how the system handles repeated errors in selecting followers.	Display error messages for each invalid input. Allow retry until valid input is provided. No crashes.	Systems reprompts users
9	Simulate two users selecting the same follower at the same time (e.g., concurrent database access).	Test Case 9: Simultaneous Access - Test if simultaneous access causes any data inconsistencies or crashes.	The system processes each user's request correctly without data corruption or crashing.	System updates db
10	Attempt to view a follower's tweets when they have an excessively large number (e.g., 10,000 tweets).	Test Case 10: Exceeding Tweet Display Limits - Test the system's handling of large tweet datasets.	Tweets are paginated or displayed incrementally. No crashes or slowdowns.	Tweets are listed
11	Select a follower with tweets containing invalid or corrupted characters (e.g., null, ???).	Test Case 11: Corrupted Tweet Data - Check behavior when tweets contain unexpected or malformed data.	Tweets are displayed with placeholders or errors are skipped gracefully. No crashes.	System does not crash and displays tweet fine
12	Interrupt the process (e.g., close the terminal or simulate a timeout) while fetching details.	Test Case 12: Interrupted Process - Test recovery from incomplete operations.	The application resumes functionality gracefully without data corruption on restart.	After exiting - reentering the program works fine
13	Enter an ID with spaces before or after, e.g., " 123 " or " 1".	Test Case 13: Follower ID with Leading or Trailing Whitespace - Test if the system trims and processes input correctly.	Whitespace is removed, and the system processes the ID as valid. No errors occur.	System still process value (trim white space)

Table 3G: Black Box testing, equivalence class test cases created by ChatGPT and manually tested for main.py. 8/9 Passed

Test ID	Inputs	Description	Expected Result	Actual Result
TC1	Logged-in user follows 3 users (User A, User B, User C).	Valid case: User has multiple followers.	List of followers should show User A, User B, and User C. The user can select any of them to see more details or tweets.	User can see multiple followers.
TC2	Logged-in user follows no one.	Valid case: User has no followers.	No followers should be listed. User should be informed that they do not follow anyone.	User can see no followers message.

TC3	Logged-in user follows one user (User A), and User A has 3 tweets.	Valid case: User follows one person with tweets.	List of followers should show User A. After selecting User A, the user can see their name, follower count, tweet count, and the 3 most recent tweets.	Logged -In user can see user A details
TC4	Logged-in user follows 2 users: User A (no tweets) and User B (3 tweets).	Valid case: One follower with no tweets, one with tweets.	List of followers should show User A and User B. After selecting User A, the user should see no tweets, and after selecting User B, they should see 3 recent tweets.	User can see tweets of user with tweets and sees a user has no tweets message for user with no tweets
TC5	Logged-in user follows 2 users: User A and User B. User A has no tweets and User B has no followers.	Valid case: User follows someone with no tweets, and no one follows another.	List of followers should show User A and User B. After selecting User A, the user should see no tweets. After selecting User B, the user should see no followers.	User B - shows up with one follower but logging in as User B shows 0 followers. No tweets for user A
TC6	Logged-in user follows 2 users: User A and User B. User A has tweets, User B has none.	Valid case: One follower has tweets, and the other has none.	List of followers should show User A and User B. After selecting User A, the user can see their tweets, while after selecting User B, no tweets are shown.	Tweets are shown for A and No tweets message for B
TC7	Logged-in user follows a user (User A) who has been followed by themselves.	Invalid case: User tries to follow a user they already follow.	After selecting User A, the user should be informed that they already follow them. No new follow is added.	You are already following this user - message, no new follow added to db
TC8	Logged-in user follows a user (User A), User A's tweet count is 0.	Valid case: User follows someone with no tweets.	List of followers should show User A. After selecting User A, no tweets should be displayed.	No tweets displayed
TC9	Logged-in user follows 1 user (User A) with multiple tweets. User selects "see more tweets."	Valid case: User selects to see more tweets after viewing the 3 most recent.	User will see all tweets from User A beyond the 3 most recent tweets.	Can see all tweets beyond first 3

Prompt 2G: "I am specifically looking for python unit tests that follow Branch - Every branch (edge of the graph) in the code has to be executed at least once and Statement Coverage - Every statement in the code has to be executed at least once. Write me some python unit tests to do this you may use mocking as required: Testing Framework: Use unittest for writing the test cases. Mocking: Use the unittest.mock library to mock database connections and user inputs, ensuring all branches in the code are covered. Database Behavior: Mock all database queries and updates using MagicMock to simulate realistic database behavior without requiring a live database. Code Organization: Ensure each test case is self-contained and adheres to the unittest structure for setup, execution, and teardown. Assertions: Validate the expected behavior for each branch of the code, including return values and database interactions (e.g., execute calls and their parameters). Output: Include any print statements to confirm test coverage during execution. Put the tests into a test case table with test case ID, inputs, description and expected results."

Table 4G: ChatGPT' white box testing coverage test cases for list_followers.py. 12/14 Passed

Test ID	Inputs	Description	Expected Results	Actual Results / Improper test case
TC-01	No followers in the database	Test behavior when the logged-in user has no followers.	Message: "You have no followers."	Pass
TC-02	Follower list: [(2, "Bob"), (3, "Alice")]	Test displaying all followers for the logged-in user.	Followers are displayed as a list with their IDs and names.	Pass
TC-03	Input: 0, 2, 0	Test selecting a valid follower, viewing their details, and following them.	Follower details are displayed, and a follow record is inserted.	Test case itself is unable to run - invalid indexes

TC-04	Input: 0, 99, x	Test selecting an invalid follower ID.	Message: "For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit." The program returns to the menu.	Pass
TC-05	Follower info: (2, "Bob")	Test displaying detailed information for a valid follower.	Details include name, ID, number of followers, tweets, and up to 3 recent tweets.	Pass
TC-06	Follower info: None	Test behavior when trying to view details for a non-existent follower.	Message: "Follower not found."	Pass
TC-07	Additional tweets: [(102, "Tweet 1"), (103, "Tweet 2")]	Test displaying additional tweets for a follower beyond the first three.	The additional tweets are displayed sequentially.	Pass
TC-08	No additional tweets available	Test behavior when a follower has no additional tweets beyond the first three.	Message: "User has no more tweets."	Pass
TC-09	Database: Existing follow record	Test following a user who is already followed by the logged-in user.	Message: "You are already following this user."	Pass
TC-10	Input: Non-numeric or invalid input (e.g., "abc")	Test behavior when the user provides invalid input for follower ID or menu options.	Message: "For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit: ." The program prompts the user for a valid input.	Pass
TC-11	Followers: Large list (e.g., 1,000 entries)	Test system performance and output handling with a large follower list.	Followers are displayed in a manageable format (e.g., paginated or sequentially).	Pass
TC-12	Tweets: [(101, "Tweet 1"), (102, "Tweet 2"), ...]	Test system handling for a follower with a large number of tweets.	Tweets are displayed without performance issues.	Test Case itself failed to run due to iterations
TC-13	replyto field in tweets table	Test behavior when displaying tweets that are replies to other tweets.	Replies are displayed correctly, with no errors or mislinked tweets.	Pass
TC-14	follows table with inconsistent data	Test behavior when the follows table has invalid data (e.g., flwer or flwee missing in users).	The program gracefully handles the inconsistency, displaying an appropriate error or skipping the invalid record.	Pass

```

import unittest
from unittest.mock import Mock, MagicMock, patch
from datetime import datetime
from list_followers import list_followers, get_follower_info, print_tweets

class TestListFollowers(unittest.TestCase):
    def setUp(self):
        # Mock the database connection and cursor
        self.conn = Mock()

```

```

self.cursor = MagicMock()
self.conn.cursor.return_value = self.cursor

def tearDown(self):
    self.conn = None
    self.cursor = None

@patch("builtins.input", return_value="x") # Mock user input
@patch("builtins.print") # Mock print to check if the message is printed
def test_no_followers(self, mock_print, mock_input):
    """TC-01: Test no followers scenario."""
    self.cursor.fetchall.return_value = []
    list_followers(self.conn, user_id=1)
    mock_print.assert_any_call("You have no followers.")

@patch("builtins.input", side_effect=["1", "x"])
def test_display_followers(self, mock_input):
    """TC-02: Test displaying followers."""
    self.cursor.fetchall.return_value = [(2, "Bob"), (3, "Alice")]
    list_followers(self.conn, user_id=1)
    self.assertEqual(self.cursor.execute.call_count, 1)
    self.cursor.fetchall.assert_called_once()

@patch("builtins.input", side_effect=["0", "2", "0"])
def test_view_follower_details_and_follow(self, mock_input):
    """TC-03: Test viewing follower details and following."""
    self.cursor.fetchall.return_value = [(2, "Bob")]
    self.cursor.fetchone.side_effect = [
        None, # Not following yet
        (2, "Bob"), # Follower info
        (5,), # Follower count
        (3,), # Tweet count
        [(101, 2, "2024-12-01", "Hello World!)], # Recent tweets
    ]
    list_followers(self.conn, user_id=1)
    self.cursor.execute.assert_any_call("SELECT * FROM follows WHERE flwer = ? AND flwee = ?", (1, 2))
    self.cursor.execute.assert_any_call(
        "INSERT INTO follows VALUES (?, ?, ?)", (1, 2, datetime.now().strftime("%Y-%m-%d"))
    )

@patch("builtins.input", side_effect=["0", "99", "x"])
def test_invalid_follower_id(self, mock_input):
    """TC-04: Test invalid follower ID."""
    self.cursor.fetchall.return_value = [(2, "Bob")]
    list_followers(self.conn, user_id=1)
    self.assertIn("For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit:",
mock_input.call_args_list[-1][0][0])

def test_get_follower_info_valid(self):
    """TC-05: Test getting valid follower information."""
    self.cursor.fetchone.side_effect = [
        (2, "password123", "Bob"),
        (10,), # Follower count
        (5,), # Tweet count
        [(101, 2, "2024-12-01", "Hello World!)], # Recent tweets
    ]
    with patch("builtins.print") as mock_print:

```

```

    get_follower_info(self.conn, user_to_check=2)
    self.assertIn("More information about Bob (User ID: 2)", str(mock_print.call_args_list))

def test_get_follower_info_no_data(self):
    """TC-06: Test getting follower information with no data."""
    self.cursor.fetchone.side_effect = [
        None, # First call: No user data (user not found)
        (0,), # Second call: No followers
        (0,), # Third call: No tweets
        [], # Fourth call: No recent tweets (empty list)
    ]
    with patch("builtins.print") as mock_print:
        get_follower_info(self.conn, user_to_check=99)
        self.assertIn("Follower not found.", str(mock_print.call_args_list))

def test_print_tweets_valid(self):
    """TC-07: Test printing additional tweets."""
    self.cursor.fetchall.return_value = [
        (102, 2, "2024-12-02", "More Tweets 1"),
        (103, 2, "2024-12-03", "More Tweets 2"),
    ]
    with patch("builtins.print") as mock_print:
        print_tweets(self.conn, user_to_check=2)
        self.assertIn("More Tweets 1", str(mock_print.call_args_list))
        self.assertIn("More Tweets 2", str(mock_print.call_args_list))

def test_print_tweets_no_more(self):
    """TC-08: Test no additional tweets."""
    self.cursor.fetchall.return_value = []
    with patch("builtins.print") as mock_print:
        print_tweets(self.conn, user_to_check=2)
        self.assertIn("User has no more tweets", str(mock_print.call_args_list))

@patch("builtins.input", return_value="0")
def test_follow_user_already_followed(self, mock_input):
    """TC-09: Test trying to follow a user already followed."""
    self.cursor.fetchall.return_value = [(2, "Bob")]
    self.cursor.fetchone.return_value = (1, 2) # Simulate existing follow record
    list_followers(self.conn, user_id=1)
    self.assertIn("You are already following this user.", mock_input.call_args_list[-1][0][0])

@patch("builtins.input", side_effect=["invalid", "x"])
def test_invalid_input(self, mock_input):
    """TC-10: Test handling invalid input."""
    self.cursor.fetchall.return_value = [(2, "Bob")]
    list_followers(self.conn, user_id=1)
    self.assertIn("For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit: ",
mock_input.call_args_list[-1][0][0])

@patch("builtins.input", side_effect=["1", "x"]) # Simulate user input for viewing and then exit
def test_large_follower_list(self, mock_input):
    """TC-11: Test handling a large number of followers."""
    self.cursor.fetchall.return_value = [(i, f"User_{i}") for i in range(1, 1001)] # 1,000 followers
    list_followers(self.conn, user_id=1)

```

```

self.cursor.execute.assert_called_once()
self.cursor.fetchall.assert_called_once()
self.assertEqual(mock_input.call_count, 2)

@patch("builtins.input", side_effect=["0", "2", "1", "x"])
def test_large_number_of_tweets(self, mock_input):
    """TC-12: Test handling a large number of tweets."""
    self.cursor.fetchall.side_effect = [
        [(2, "Bob")], # Followers
        [(i, 2, f"2024-12-{i:02d}", f"Tweet {i}") for i in range(1, 1001)], # 1,000 tweets
    ]
    list_followers(self.conn, user_id=1)
    self.assertEqual(self.cursor.execute.call_count, 2) # Followers and tweets
    self.cursor.fetchall.assert_any_call()

def test_tweets_with_replyto(self):
    """TC-13: Test tweets that are replies."""
    self.cursor.fetchall.return_value = [
        (101, 2, "2024-12-01", "Tweet 1", None), # Regular tweet
        (102, 2, "2024-12-02", "Reply to Tweet 1", 101), # Reply to Tweet 1
    ]
    with patch("builtins.print") as mock_print:
        print_tweets(self.conn, user_to_check=2)
        self.assertIn("Tweet 1", str(mock_print.call_args_list))
        self.assertIn("Reply to Tweet 1", str(mock_print.call_args_list))

@patch("builtins.input", side_effect=["0", "3", "x"])
def test_inconsistent_follows_table(self, mock_input):
    """TC-14: Test `follows` table with inconsistent data."""
    # Mock data where `flwer` exists, but `flwee` is missing in `users`
    self.cursor.fetchall.return_value = [(2, "Invalid User")]
    self.cursor.fetchone.side_effect = None # User data not found for `flwee`
    list_followers(self.conn, user_id=1)
    self.assertIn("For more information on followers Enter '0', Enter '1' to see your followers again and press any other key to exit:",
mock_input.call_args_list[-1][0][0])

if __name__ == "__main__":
    unittest.main()

```

Figure 1G: ChatGPT output for unit tests for list_followers.py outlined in Table 4G.