

Building effective tests with generative AI

This chapter covers

- Three popular generative AI tools for Python test generation
- Using AI to create both `unittest` and `pytest` test suites
- Setting up in-memory databases for isolated testing environments
- Crafting effective prompts to generate high-quality test code
- Evaluating the strengths and weaknesses of each AI testing approach

Software testing is vital for reliability and security, but it's often rushed due to development pressures. Generative AI tools are changing this boring, repetitive process. They can automatically create detailed test suites, spot edge cases, and generate boilerplate code. This chapter shows how GitHub Copilot, Tabnine, and Blackbox AI can enhance your testing workflow. They can help you build strong, maintainable test suites quickly, while upholding quality.

High-quality tests are vital for reliable software. They catch bugs early and ensure new features don't break existing code. They also document how the code should behave. As a new developer, I struggled to grasp the value of testing. I thought creating tests for code I knew was good was a waste of time. It took one disaster to change my mind.

Testing is like an investment. You spend time building tests now to strengthen your product. This approach saves time and frustration by catching problems early. As your software evolves, each passing test reaffirms that your product still works properly. However, in today's fast-paced development world, testing often falls behind. This is where generative AI can aid developers as a valuable resource.

During the last decade, the fast-growing DevOps movement has pushed us to ship features at lightning speed. The pressure to produce faster often leads to less testing. Remember that if you don't have time to write tests, you won't have time to fix bugs.

When creating tests, you often write similar code with small changes. There's a lot of setup and boilerplate involved. Generative AI tools can greatly simplify this process. They analyze your code, infer method signatures, and predict the tests you need. These tools can write much of the repetitive code for you, saving hours of work. Smart people tend to get bored with repetitive tasks, which can lead to mistakes.

Generative AI not only speeds up the process; it also improves test quality. These tools suggest test cases you might overlook, ensuring broader coverage. I frequently use AI to create tests, which significantly cuts down development time. It's quicker than writing tests from scratch, and AI offers great suggestions. This makes thorough testing easier, reducing the risk of missing important tests.

Let's explore how to use generative AI tools to create tests for our application. We'll learn to use these tools to build more tests in less time. This strategy software remains dependable and well-structured as development speeds up. After all, the adoption of these tools is changing how management views timelines.

8.1 Why use generative AI for testing?

Generative AI tools dramatically reduce the time needed to create tests, and AI-generated scaffolding serves as a reliable foundation for efficient test development. The generative AI tools we're using may prove most valuable in test generation. Although we've spent this entire book generating new code, you may be the type of programmer who prefers their own code. You might feel these tools get in the way. Using them for testing, however, may change your mind entirely.

Many people criticize AI-generated code, and in many cases, their criticism is valid. Here are a few common complaints. AI-generated code

- Lacks context and understanding
- May contain bugs, vulnerabilities, and inefficiencies
- Creates overreliance problems with developers
- Can perpetuate biases
- Can create IP and copyright problems

These points are worth considering in any organization. They can be addressed with a combination of

- Careful prompt design
- Thorough two-person (minimum) code review
- A balanced approach to the use of AI

Generative AI tools have much less influence when it comes to testing. For instance, a unit test usually only requires the context of the single function or method it's testing. Tests themselves are far less complex than the code they're operating on. Biases and IP/copyright problems also have less impact as well.

You're still taking a risk, and you aren't getting something for nothing. However, the benefits of using generative AI for testing far outweigh the risks. It is possible to

- Build more tests in less time
- Improve test quality
- Reduce human error
- Have a faster feedback loop

Even if you don't find value with these tools in everyday development, you'll see that using them for tests is a great way to increase productivity. So, let's get to it. We'll look at three tools we've been working with throughout this book and learn how to use them to save time and improve the accuracy of our software.

8.2 **What are unit tests?**

Before diving into AI-generated tests, let's talk about the types of tests we will create. Unit tests are one of the most important parts of an application. While unit tests don't cover everything that can go wrong, they do a good job of ensuring your application is reasonably stable. They are short, focused tests that check one piece of a code at a time to make sure it works right.

In Python, unit tests often investigate program methods (functions) and classes. The overall goal is to take one piece of functionality and make sure it works as intended. You try to give the method different inputs and establish (assert) what you expect the output to be. For example, if you have a method that adds two integers, you will send it 2 and 2 and expect 4 as the answer. You might send it 5 and 1 or different variations to verify it's performing addition as expected.

Unit tests are useful for Python developers because they can help you find bugs quickly. Often, your application will appear to work fine, and you may publish it to the world. Then, one of your users enters some information, and the application breaks unexpectedly. With unit tests, you'll catch that problem before publishing.

8.3 **The tools we'll use for Python testing**

We will use three familiar tools for test generation: ChatGPT, Tabnine, and Blackbox AI. They all work well, and we'll compare the differences in approach for each and learn how to generate tests quickly.

8.3.1 Github Copilot

Copilot uses the traditional chat-style interface for building tests, as well as some shortcuts in the IDE. You can also interact with the API to automate test generation. We'll look closely at how to use GitHub Copilot to generate tests, within the IDE. Some advantages of using Copilot are

- World class model that contains many languages
- API for full customization
- The agent mode for automated tasks

8.3.2 Tabnine

Test generation has become a central focus in Tabnine's recent development efforts. You can use a traditional chat-style interface or integrate it into the UI of an IDE. There are multiple approaches that you can mix and match. Some advantages of Tabnine are the following:

- Integration into IDEs (JetBrains and VSCode)
- Chat interface that's aware of your application and context
- Different models available to generate code
- Provides security for your code so you can avoid IP/copyright problems
- Ability to run your own local models

8.3.3 Blackbox AI

Blackbox is similar to Tabnine as it runs in the IDE. You can also use a traditional chat interface or integrated IDE tools. I've been impressed by the results I've seen with Blackbox. Some advantages of Blackbox AI are

- Integration into VS Code
- Chat interface that's aware of your application and context
- Customized agents that help with tests

8.4 Writing unit tests with generative AI

There are various ways to generate tests with each tool, and it's important to know the differences. We will examine each tool to see its capabilities. All of them tend to default to `unittest` instead of `pytest`, so we'll see what they generate naturally and then push them into `pytest` to see how they work with each.

8.4.1 unittest or pytest?

Both `pytest` and Python's built-in `unittest` library are popular testing tools for Python. Each has its pros and cons. `unittest` has been a part of the Python standard library for a long time and has worked great for many years. `pytest` has recently grown in popularity because it is easy to use and has tons of great features.

We don't necessarily have to decide to go with one or the other here, but I want to explore `pytest` in more detail because it will be the way most Python developers build tests in the future. What makes `pytest` so appealing is its ease of use. The plain asserts mean we need less boilerplate code and ceremony when writing tests. It also has much stronger test discovery and a rich fixture system.

Personally, I prefer `pytest` for most projects. The standard unit tests have been around so long I'd be surprised if our tools didn't know about them. However, I want to see how well these tools work for what I think is the future of Python unit testing—`pytest`.

8.4.2 **Using Copilot for test generation**

Let's start our journey into testing with GitHub Copilot. We're going to use the Copilot plugin within the Visual Studio Code IDE. Note that Copilot works in many other IDEs as well. The first thing we need to do is set up our testing environment. We should already be in a Python virtual environment as it was set up at the beginning of the book. But here's a refresher just in case. In the folder containing your code, type in

```
Python3 -m venv hamradio
```

Based on your operating system, use the appropriate command. In Windows, enter

```
\hamradio\Scripts\activate.bat
```

With Windows PowerShell, use

```
\hamradio\Scripts\Activate.ps1
```

On Linux and Mac OSX Systems, enter

```
source /hamradio/bin/activate
```

Now that we're in our virtual environment, let's install `pytest`:

```
pip install pytest
```

Next, let's create a folder named `pytests` and a new file within that folder. The file should be named `test_models_questions.py`. If you're new to `pytest`, you might be wondering about the significance of this name. We want to start all tests with `test_` so `pytest` knows that it's a test to be run. We want to test functions in our `Questions` class, which is in `/models`. I renamed the remaining files so that they're easier for people to understand (figure 8.1).

The “`test+`” prefix is required to distinguish tests from code. The rest of the convention is optional, and you may have a better way, but this is what I prefer. Let's open the

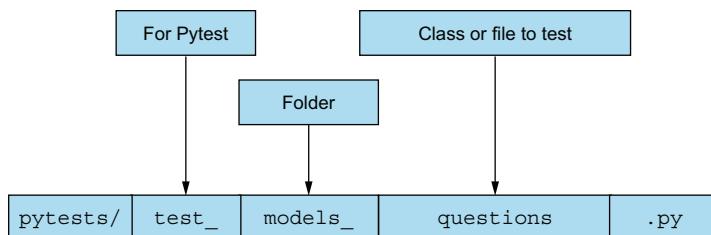


Figure 8.1 File naming convention to let `pytest` know it's a test and let the developer know what the test is intended to do

file in our Visual Studio Code with the GitHub Copilot extension installed. You will notice right away in this file that Copilot is inviting you to interact (figure 8.2).

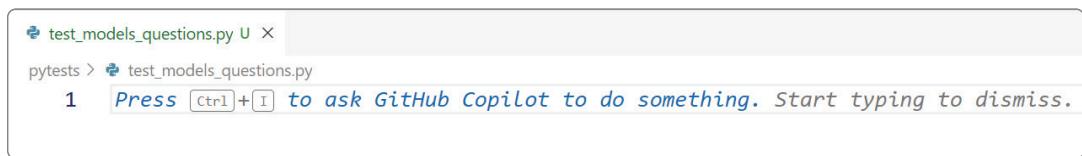


Figure 8.2 Copilot offers a hint for how to open a chat with Copilot, or you can start typing.

This gives us two options for invoking Copilot into our workflow. You can either press `Ctrl + I` or start typing. I'll try the `Ctrl + I` method first. It opens a window where you can either type in a prompt, like any other LLM in figure 8.3, or you can use shortcuts as shown in figure 8.4. The shortcuts help guide the tool with tests, fixes, documentation, or explanation.

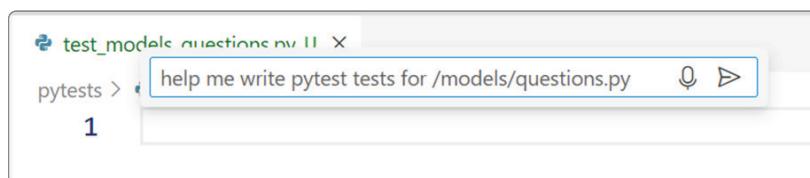


Figure 8.3 You can type in a prompt if you want to communicate with Copilot directly from the IDE. This is a good way to send specific requests.

Another option is to press the forward slash (/), which gives you several options. You can either document, fix, or explain the code (which we don't have yet) and finally generate tests for the code with `/tests`.

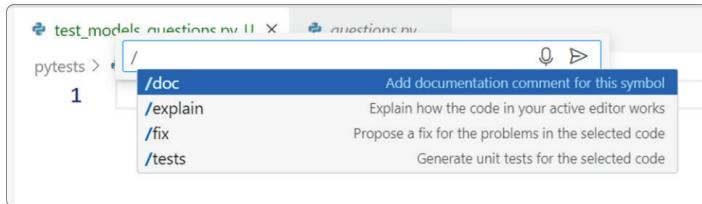


Figure 8.4 There is a set of commands with Copilot, which are shortcuts for common features. Rather than type a prompt for code generation, you can type one of these commands.

I will try just running `/tests` and see if Copilot will generate what I'm looking for. This type of shortcut is nice, because it's wasteful to spend time working on a prompt for something Copilot is preprogrammed to do. However, if you've come along this far in the book, you might worry about the assumptions Copilot will make when producing the output. As shown in figure 8.5, it creates a set of unittest tests for us instead of pytest.

A screenshot of a code editor showing the results of running `/tests`. The code generated is:

```

1 import unittest
2 from models import Question
3
4 class TestQuestionModel(unittest.TestCase):
5     def test_question_creation(self):
6         question = Question("What is the capital of France?", "Paris")
7         self.assertEqual(question.question_text, "What is the capital of France?")
8         self.assertEqual(question.answer, "Paris")
9
10    def test_question_str_representation(self):
11        question = Question("What is the capital of France?", "Paris")
12        self.assertEqual(str(question), "Question: What is the capital of France?")
13
14    def test_question_check_answer_correct(self):
15        question = Question("What is the capital of France?", "Paris")
16        self.assertTrue(question.check_answer("Paris"))
17
18    def test_question_check_answer_incorrect(self):
19        question = Question("What is the capital of France?", "Paris")
20        self.assertFalse(question.check_answer("London"))
21
22    def test_question_check_answer_case_insensitive(self):
23        question = Question("What is the capital of France?", "Paris")
24        self.assertTrue(question.check_answer("paris"))
25
26 if __name__ == '__main__':
27     unittest.main()

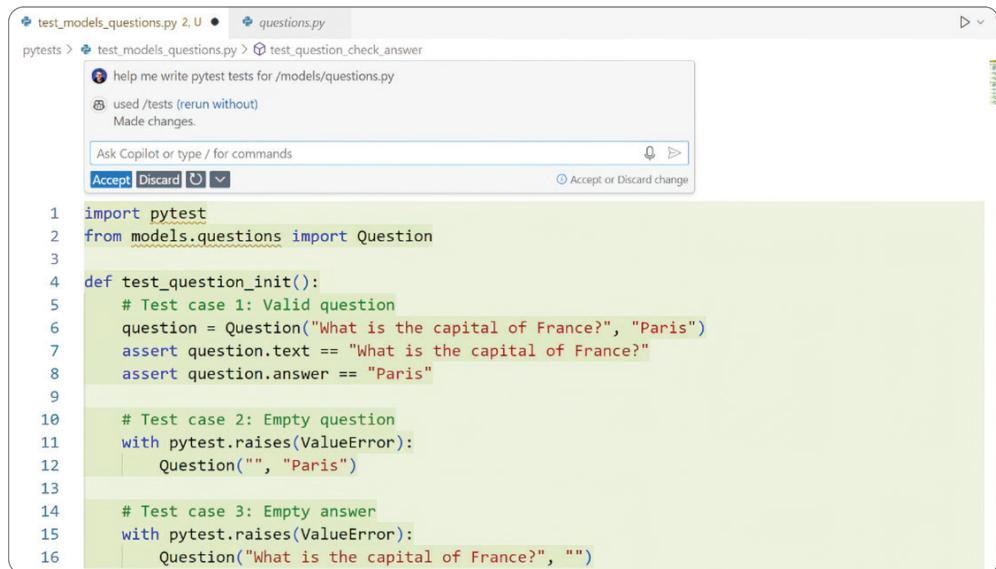
```

Figure 8.5 When running `/test` in the prompt window, Copilot assumed I wanted `unittest` tests instead of `pytest`.

Copilot assumed I wanted `unittest` tests, and I want `pytest` tests, so I will try specifying in a prompt what I would like Copilot to do instead:

JM Help me write pytest tests for `/models/questions.py`.

My hunch here is if I add more information into the prompt, it will give me better results. And it does produce `pytest` tests for this class (figure 8.6).



The screenshot shows the Copilot interface with a code editor. The file being edited is `test_models_questions.py`. The code generated by Copilot is as follows:

```

1 import pytest
2 from models.questions import Question
3
4 def test_question_init():
5     # Test case 1: Valid question
6     question = Question("What is the capital of France?", "Paris")
7     assert question.text == "What is the capital of France?"
8     assert question.answer == "Paris"
9
10    # Test case 2: Empty question
11    with pytest.raises(ValueError):
12        Question("", "Paris")
13
14    # Test case 3: Empty answer
15    with pytest.raises(ValueError):
16        Question("What is the capital of France?", "")

```

The interface includes a sidebar with file navigation, a message bar at the top, and a bottom bar with 'Accept' and 'Discard' buttons.

Figure 8.6 Copilot has generated some `pytest` tests as a result. By using a prompt, we can narrow down exactly what we're looking for and hopefully get better results.

I decided to accept the suggestion and insert it into my code. At first glance, I see that this set of tests will not run as they are, but we'll get to that. I would like to see how Copilot has examined the class and attempted to understand it enough to build tests. This may seem easy, but it's challenging for LLMs. Some of the biggest challenges to the folks working on LLMs is reasoning and understanding—they still aren't good at either yet. But let's look at the code generated:

JM import pytest
from models.questions import Question
def test_question_init():
 # Test case 1: Valid question
 question = Question("What is the capital of France?", "Paris")
 assert question.text == "What is the capital of France?"
 assert question.answer == "Paris"
 # Test case 2: Empty question

```

with pytest.raises(ValueError):
    Question("", "Paris")
# Test case 3: Empty answer
with pytest.raises(ValueError):
    Question("What is the capital of France?", "")
def test_question_check_answer():
    # Test case 1: Correct answer
    question = Question("What is the capital of France?", "Paris")
    assert question.check_answer("Paris") == True
    # Test case 2: Incorrect answer
    assert question.check_answer("London") == False
    # Test case 3: Case-insensitive comparison
    assert question.check_answer("paris") == True
    # Test case 4: Empty answer
    with pytest.raises(ValueError):
        question.check_answer("")
    # Test case 5: None answer
    with pytest.raises(ValueError):
        question.check_answer(None)

```

Copilot wrote out a set of tests to run, and from the very first test, `test_question_init()`: we can see a problem in understanding the context of this application. Copilot has assumed that the `Question` class is something you invoke, send it a question, and expect an answer. That's clear from the first test case:

```

# Test case 1: Valid question
question = Question("What is the capital of France?", "Paris")
assert question.text == "What is the capital of France?"
assert question.answer == "Paris"

```

This could happen for several reasons. Copilot could be relying on the name of the class to assume what it means. I can't rule out myself as a factor: maybe my prompt wasn't good enough. Or worse yet, maybe this is a bad design with terrible naming, and the LLM wasn't trained on code like this. Whatever the reasoning, these tests don't make much sense.

In the `test_question_check_answer()`: tests, things aren't any better. Copilot assumes the method `check_answer` takes in a string question and string answer:

```

# Test case 1: Correct answer
question = Question("What is the capital of France?", "Paris")
assert question.check_answer("Paris") == True

```

The problem is, there is no such method. This test is invalid as well. But what are we to do? We tried running `/tests`, then tried giving a better prompt. We're still stuck with tests that won't work. Once again, we must rely on our own knowledge of Python development to get this done. However, we can still get help from Copilot to generate tests and save us time. Here's how we'll do it.

CREATING A DIALOG WITH GITHUB COPILOT

If you've written unit tests in Python before, you know there are things you must do to set up your test environment. It rarely entails writing a few functions and calling it good. Let's see if GitHub Copilot can help us.

The first thing I'll do is select a method from our `Questions` class. Inside the IDE, I highlighted "self" and asked this question:

JM How can I generate a mock "self" in pytest for testing?

The prompt will be sent off to Copilot while using the existing context of this file. Then I can send the answer to a Github Chat window (figure 8.7).

```

40     self.cursor = cursor
41
42     def get_all_questions(self):
43         self.cursor.execute("SELECT id FROM questions")
44         return self.c how can I generate a mock "self" in pytest for
45             testing? Q ➔
46     def get_question(self, question_id):
47
48         self.cursor.execute('''SELECT id, correct, question, a, b, c, d
49         result = self.cursor.fetchone()
50
51         if result is None:
52             return None
53

```

Figure 8.7 Asking a question in the IDE so that Copilot understands the context better to produce a better result

After sending the prompt, I can open the Copilot chat, and I see a suggestion for how to do this. It's suggesting I put the test in the same file I'm working with, but other than that, the advice looks solid (figure 8.8).

One thing I would like to do is create a mock, in memory database for this application. This way, I can run tests against the software without worrying about affecting real data. Now, of course, this idea only works with a smaller application like ours. You can't load up an entire production database into a memory with an enormous enterprise application. But since I have faith our database will fit into memory, let's give it a shot.

I send the following query to GitHub Copilot in the chat:

JM How can I create an in-memory database for SQLite for testing that matches the database I have in `/data/questions.db`?

The screenshot shows the GitHub Copilot interface. At the top, it says "GitHub Copilot used /tests (rerun without)" and "Used 1 reference". Below that, a note says: "To generate tests for the `get_question` method in the `Questions` class, you can use the `pytest` library along with the `unittest.mock` module to create a mock `self` object. Here's an example of how you can do it:"

```

    test_questions.py app\models
    ┌─────────────────────────────────────────────────────────────────────────────────┐
    | 35 hidden lines                                         35 hidden lines |
    +-----+-----+
    36     assert question['c'] == "4"
    37-     assert question['d'] == "5"                                █
    ┌─────────────────────────────────────────────────────────────────┐
    | 35 hidden lines                                         35 hidden lines |
    +-----+-----+
    36     assert question['c'] == "4"
    37+     assert question['d'] == "5" import pytest
    38+ from unittest.mock import Mock
    39+ from models.questions import Questions
    40+
    41+ def test_get_question():
    42+     # Create a mock cursor object
    43+     cursor = Mock()
    44+
    45+     # Create an instance of the Questions class
    46+     questions = Questions(cursor)
    47+
    48+     # Set up the mock cursor's execute method
    49+     cursor.fetchone.return_value = (1, "Paris")
    50+
    51+     # Call the get_question method with a specific ID
    52+     question_id = 1
    53+     question = questions.get_question(question_id)
    54+
    55+     # Assert that the returned question matches expectations
    56+     assert question['id'] == question_id
    57+     assert question['question'] == "What is the capital of France?"
    58+     assert question['correct'] == "Paris"
    59+     assert question['a'] == "London"
    60+     assert question['b'] == "Paris"
    61+     assert question['c'] == "Berlin"
    62+     assert question['d'] == "Madrid"
  
```

At the bottom left, it says "Ask Copilot or type / for commands". On the right side, there are icons for search, refresh, and other navigation.

Figure 8.8 The results from GitHub Copilot when asked a direct question and with highlighted code I want it to look at. Copilot suggests creating a `test_questions.py` in the same directory as my `questions.py` class, but other than that, it looks closer to a real test.

While this sentence would be strange to say out loud, it's clearly conveying information we want Copilot to know:

- We want an in-memory database.
- We want to mimic SQLite.
- It will be used for testing.
- The existing database is `/data/questions.db`.

This prompt should convey the needed information to generate what we want. Copilot returns a helpful response:



- To create an in-memory SQLite database for testing that matches your existing database schema, you can follow these steps:
 - 1 **Connect to the in-memory database.**
 - 2 **Read the schema from the existing database.**
 - 3 **Apply the schema to the in-memory database.**
 - 4 **Copy the data from the existing database to the in-memory database (if needed).**

Here's a step-by-step plan in pseudocode:

- 1 Connect to the existing database '/data/questions.db'
- 2 Extract the schema using the 'sqlite_master' table.
- 3 Connect to the in-memory database.
- 4 Apply the extracted schema to the in-memory database.
- 5 Optionally, copy the data from the existing database to the in-memory database.

Now, let's implement this in Python:

```
```python
import sqlite3

def create_in_memory_db_from_existing(existing_db_path):
 # Connect to the existing database
 existing_conn = sqlite3.connect(existing_db_path)
 existing_cursor = existing_conn.cursor()

 # Extract the schema from the existing database
 existing_cursor.execute("SELECT sql
FROM sqlite_master WHERE type='table'")
 schema_statements = existing_cursor.fetchall()

 # Connect to the in-memory database
 memory_conn = sqlite3.connect(':memory:')
 memory_cursor = memory_conn.cursor()

 # Apply the schema to the in-memory database
 for statement in schema_statements:
 if statement[0]: # Ensure the statement is not None
 memory_cursor.execute(statement[0])

 # Optionally, copy data from the existing database to the in-memory
 # database
 for table_info in existing_cursor.execute(
 "SELECT name FROM sqlite_master WHERE type='table'"):
 table_name = table_info[0]
 if table_name != 'sqlite_sequence':
 data = existing_cursor.execute(
 f"SELECT * FROM {table_name}").fetchall()
```

```

 columns = [description[0]
for description in existing_cursor.description]
 placeholders = ', '.join('?' * len(columns))
 memory_cursor.executemany(
 f"INSERT INTO {table_name}"
 f"({', '.join(columns)}) VALUES ({placeholders})", data
)

Commit changes to the in-memory database
memory_conn.commit()

Close the existing database connection
existing_conn.close()

return memory_conn

Usage example
in_memory_db = create_in_memory_db_from_existing
('/data/questions.db')

```

This code will create an in-memory SQLite database that matches the schema of your existing database and optionally copies the data. You can use the 'in\_memory\_db' connection for your testing purposes.

This response looks helpful, so I will include the `create_in_memory_db_from_existing` function into our existing test file (`tests_models_questions.py`). At first glance, the code appears logical and readable, and it seems like it should function correctly. I'll build a quick test to connect to the database and attempt to run a query:

```

def test_connection():
 in_memory_db = create_in_memory_db_from_existing('../data/questions.db')
 in_memory_db.execute("SELECT * FROM questions")

```

This should extract our important data from the production SQLite database into memory and attempt to run a query. After running `pytest`, the test fails with the following error message:

```

FAILED test_models_questions.py::test_connection - sqlite3.OperationalError:
object name reserved for internal use: sqlite_sequence

```

This is a minor error. If we look at the queries in this function, we can see where it queries SQLite for all the table names:

```
existing_cursor.execute("SELECT sql FROM sqlite_master WHERE type='table'")
```

And then, it tries to load the data from these tables into memory:

```

for table_info in existing_cursor.execute
("SELECT name FROM sqlite_master WHERE type='table'"):

```

If the query attempts to access `sqlite_sequence`, it will be denied. Copilot attempts to rectify this problem with this line of code:

```
if table_name != 'sqlite_sequence': # Skip the sqlite_sequence table
```

However, it is still being queried, so we can rectify it with a small modification to our SQL code. I will add a clause to exclude the `sqlite_sequence` table from our initial query. So, I will change

```
existing_cursor.execute("SELECT sql FROM sqlite_master
WHERE type='table' ")
```

to

```
existing_cursor.execute("SELECT sql FROM sqlite_master
WHERE type='table' AND name IS NOT 'sqlite_sequence'")
```

and rerun the test. It is now successful (figure 8.9).

```
(hamradio) [jeremy@KATANA pytests]$ pytest
===== test session starts =====
platform linux -- Python 3.12.3, pytest-8.3.2, pluggy-1.5.0
rootdir: /mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/pytests
collected 2 items

test_models_questions.py .. [100%]
===== 2 passed in 0.10s =====
(hamradio) [jeremy@KATANA pytests]$
```

**Figure 8.9** Once we made a small modification to the code generated by Copilot, our tests pass successfully.

What can we learn from this small hiccup? Once again, your personal experience will help you tremendously when using generative AI tools. You can't rely on AI alone to build all the software. However, we also see just how close Copilot got to generating accurate code. We only needed a small tweak to the query, and it worked.

### In-memory testing benefits

Using in-memory databases for testing provides several advantages:

- *Speed*—In-memory operations are significantly faster than disk-based tests.
- *Isolation*—Tests are run in a contained environment without affecting production data.
- *Consistency*—Each test starts with a clean, predictable database state.
- *No cleanup*—The database disappears when tests complete.

**(continued)**

- *No configuration*—There is no need for separate test database credentials or connections.
- *Simplicity*—Test code reads more like production code, improving maintainability.

For larger projects, consider creating database snapshots that can be quickly loaded into memory rather than rebuilding schema and test data for every test run.

## BUILDING FUNCTIONAL UNIT TESTS

So far, we don't have any real tests here. We have one test that creates a mock and looks at the mock. The second test creates an in-memory version of our database and then runs a query against it. Let's create a real test for our software. We can modify our `test_get_question()` function to use our in-memory database. We'll query Copilot along the wall to help us out.

First, we'll clear the existing code in `test_get_question` and make it empty so it looks like this:

```
def test_get_question():
```

Now, let's take a step back for a minute. In our first test, `test_connection`, we have a line of code that connects to and builds our in-memory database. It looks like this:

```
in_memory_db = create_in_memory_db_from_existing('.../data/questions.db')
```

There's nothing inherently wrong with this line; however, as we create multiple tests, there is a problem. If we insert this line of code into every test, it will work just fine. However, we're repeating ourselves with every test. So, if we decide to change the location of the main database, for instance, we'll have to update each of these statements. There is a better way. It would be nice to create this connection one time and reuse it throughout each test, and that's exactly what we'll do.

### CREATING A FIXTURE

A fixture in `pytest` is a powerful tool. It helps prepare your environment for tests. It's setting up the stage before the play begins. In our case, it's going to call our function that sets up the in-memory database from our production database. This only needs to be done once, and if we require changes, we only need to implement them once. Creating fixtures makes for cleaner tests, saves resources, and most importantly your time. Let's see if Github Copilot can help us build this fixture. I'll start with a general prompt, and we'll get more specific if needed. Here's the prompt I used:

JM In this test suite, how can I create a `pytest` fixture for our database connection?

This prompt is very simple, with good reason. I want Copilot to use the entire file that's open, so I start with "In this test suite." The chances are good that Copilot will look at the code anyway, but this phrase ensures it. Then I mentioned "our database connection" because there is only one in the file right now. Then I ask how to create a fixture for that database connection, so Copilot will know exactly what I want it to do. Here is the result it came up with:



To create a fixture for your database connection in a test suite using 'pytest', you can use the '@pytest.fixture' decorator. This fixture will set up the database connection before each test and tear it down after each test.

Here's how you can modify your test suite to include a fixture for the database connection:

- 1 Import the necessary modules.
- 2 Define a fixture function that sets up the database connection.
- 3 Use the fixture in your test functions.

Here's the updated code:

```
```python
import pytest
import sqlite3
@pytest.fixture
def db_connection():
    # Setup: create the in-memory database connection
    memory_conn = create_in_memory_db_from_existing('../data/questions.db')

    yield memory_conn # This is where the testing happens
    # Teardown: close the database connection
    memory_conn.close()
def test_connection(db_connection):
    db_connection.execute("SELECT * FROM questions")
def test_get_question(db_connection):
    # Example test using the db_connection fixture
    result = db_connection.execute("SELECT * FROM questions WHERE id = 1")
    question = result.fetchone()
    assert question is not None
```

```

In this code:

- The 'db\_connection' fixture sets up the in-memory database connection and yields it to the test functions.
- After the test functions are done, the fixture ensures that the database connection is properly closed.

I'll update our tests accordingly. I'll create a new fixture and refactor the test functions to take the fixture as input. We're simply adding

```
def db_connection():
 # Setup: create the in-memory database connection

```

```
memory_conn = create_in_memory_db_from_existing('../data/questions.db')

yield memory_conn # This is where the testing happens

Teardown: close the database connection
memory_conn.close()
```

Then, for each of the methods, we change them accordingly:

```
def test_connection(db_connection):
def test_get_question(db_connection):
```

And after running the tests again, they pass. So, let's now build these to test our Questions class.

#### REFACTORING THE TEST CONNECTION

Our test connection function doesn't really test anything. It just runs a query, and if that query were to fail, the test would fail:

```
def test_connection(db_connection):
 db_connection.execute("SELECT * FROM questions")
```

This isn't incredibly useful. Instead let's check how many questions are in the questions table. This way, it still checks the connection to our in-memory database and makes sure our in-memory database contains all the questions it needs. This makes our test more useful and efficient. So, I decided to ask Copilot to help us out. Here's the prompt I sent:

 JM How can we make this method count the amount of records in the questions table and verify them in a test?

The results are exactly what I was looking for: a nice test to count the number of questions, then verify it against the known number of questions we expect. Here's the output:

 To modify the 'test\_connection' method to count the number of records in the 'questions' table and verify them in a test, you can follow these steps:

- 1 Execute a SQL query to count the number of records in the 'questions' table.
- 2 Fetch the result of the query.
- 3 Assert that the count matches the expected number of records.

Here's the updated code:

```
```python
def test_connection(db_connection):
    # Execute a SQL query to count the number of records in the questions table
    cursor = db_connection.execute("SELECT COUNT(*) FROM questions")

    # Fetch the result of the query
```

```

count = cursor.fetchone()[0]

# Assert that the count matches the expected number of records
expected_count = 10 # Replace with the expected number of records
assert count == expected_count, f"Expected {expected_count} records,
but found {count}"
```

```

In this code:

- The 'db\_connection.execute("SELECT COUNT(\*) FROM questions")' query counts the number of records in the 'questions' table.
- 'cursor.fetchone()[0]' fetches the count from the query result.
- The 'assert' statement checks if the count matches the expected number of records. Adjust 'expected\_count' to the actual expected number of records in your database.

I'll modify the function in the following way: I'll copy the code generated by Copilot and rename the function to make it more accurate. Next, I'll add our expected number of questions (411) because that's how many are in our production SQLite database:

```

def test_count_questions(db_connection):
 # Execute a SQL query to count the number of records in the questions table
 cursor = db_connection.execute("SELECT COUNT(*) FROM questions")

 # Fetch the result of the query
 count = cursor.fetchone()[0]

 # Assert that the count matches the expected number of records
 expected_count = 411 # Replace with the expected number of records
 assert count == expected_count,
f"Expected {expected_count} records, but found {count}"

```

This test is now more useful. It ensures that we can connect to our in-memory database and also verifies that the in-memory database has the proper number of questions. Now we know that we can test a method in our application to make sure it does what we expect.

#### TESTING OUR GET\_QUESTION\_SET FUNCTION

Now that we know our in-memory database is at least mostly functional, let's test a function in our `questions` class. I want to test the `get_question_set` function. Here's the prompt I sent to Copilot:

 How can I create a test for our `questions.get_question_set` function?

Again we're using a very basic and concise prompt that makes a lot of assumptions about what Copilot knows. We're expecting that Copilot

- Knows where `questions.get_question_set` is located
- Knows what it does

- Will use pytest
- Will utilize our fixture

These are a lot of assumptions, so let's see the results.

-  To create a test for the `get_question_set` function in your `questions` module, you need to follow these steps:
- Import the `questions` module.
  - Use the `db_connection` fixture to set up the database connection.
  - Call the `get_question_set` method with appropriate parameters.
  - Verify the results returned by the method.

Here's an example of how you can write the test:

```
import questions # Import the questions module
def test_get_question_set(db_connection):
 # Assuming get_question_set takes a database connection and some
 # parameters
 # Adjust the parameters as needed for your specific method
 question_set = questions.get_question_set(db_connection, some_parameter)
 # Verify the results
 assert question_set is not None
 assert len(question_set) > 0 # Assuming it returns a list of questions
 # Further assertions can be added based on the expected structure
 and content of question_set
 # For example:
 expected_question = {
 'id': 1,
 'question_text': 'What is the capital of France?',
 'answer': 'Paris'
 }
 assert question_set[0] == expected_question
```

In this code:

- The `test_get_question_set` function uses the `db_connection` fixture to set up the database connection.
- It calls the `get_question_set` method from the `questions` module with the necessary parameters.
- It verifies that the returned `question_set` is not `None` and contains at least one question.

Additional assertions can be added to check the structure and content of the `question_set` based on your specific requirements.

Make sure to replace `some_parameter` with the actual parameters required by your `get_question_set` method and adjust the expected results accordingly.

At a glance, this code makes sense, but after I inserted it and ran it, I found some problems. So, I refactored it, and here's how I set it up. Instead of importing `questions` with

```
import questions # Import the questions module
```

I am already importing previous questions, so I can simply create another instance of questions and pass in my cursor to the method. This is cleaner and simpler and will work. Here's the refactored method:

```
def test_get_question_set(db_connection):
 questions = Questions(db_connection.cursor())

 # Call the get_question_set method with a session_id of 1
 question_set = questions.get_question_set('1')

 # Verify the results
 assert question_set is not None
 assert len(question_set) > 0 # Assuming it returns a list of questions
```

This change ensures that our `get_question_set` method returns a set of questions. We aren't validating those questions, because they are generated at random. But it does verify that we are creating a list of questions for the test. I run the command and get two passing tests (figure 8.10).

```
(hamradio) [jeremy@KATANA pytests]$ pytest -s
=====
test session starts =====
platform linux -- Python 3.12.3, pytest-8.3.2, pluggy-1.5.0
rootdir: /mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/pytests
collected 2 items

test_models_questions.py ...

=====
2 passed in 0.17s =====
```

**Figure 8.10** Two passing tests that verify both the in-memory database and the `get_question_set` method. These tests ensure that the application is functioning as designed.

This is a look at the workflow for testing with GitHub Copilot. If you don't know a lot about testing, generative AI tools can be very helpful. If you're experienced with writing tests, it can help you write them faster. Here are some things I've noticed about using Copilot for tests:

- They're mostly accurate, and a few changes are usually needed.
- The `/tests` shortcut from the IDE has problems. It's better to go directly to the chat window to ask questions.
- It's good at grasping context and functionality.
- It will save time and is worth using.

Let's jump into another popular tool, Tabnine, and generate some tests with it.

## Crafting effective test prompts

When asking AI tools to generate tests, include these elements for better results:

- Specify the testing framework (`pytest` vs. `unittest`).
- Reference file paths to help the AI locate your code.
- Mention database requirements (in-memory vs. mocking).
- Describe edge cases you want tested.
- Request specific assertion styles.
- Include context about dependencies.
- Mention fixtures, whether they should be reused or created.
- Start general, then refine with follow-up prompts if needed.

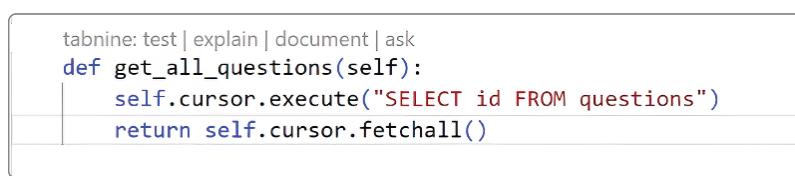
For example, instead of “create tests for my function,” try

“Create Pytest tests for the `get_question_set` method in `app/models/questions.py` using the in-memory database fixture. Include tests for both empty results and normal operation.”

### 8.4.3 Using Tabnine for test generation

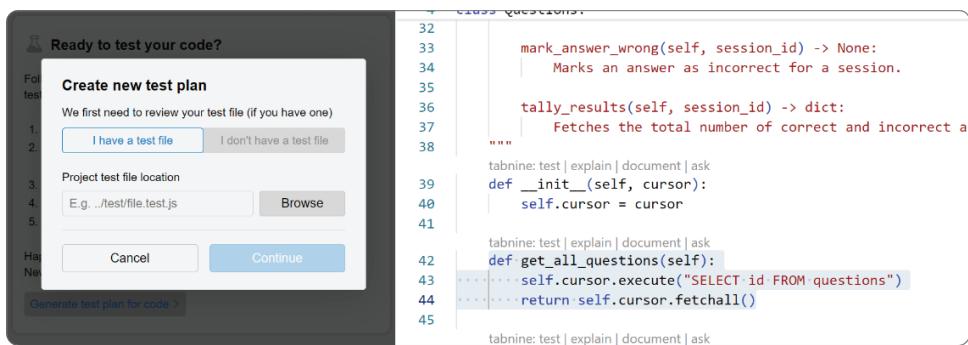
One thing you’ll find when using these tools is that the interface is nearly the same. And most of the time, it’s easier to jump into the chat window and send prompts. These tools are migrating toward focusing on the chat interface anyway, so it’s good to get into the habit of it. Let’s see how Tabnine handles tests.

Like Copilot, Tabnine has some shortcuts inserted into the IDE. You also have a chat window option. These shortcuts can be found near your functions (figure 8.11).



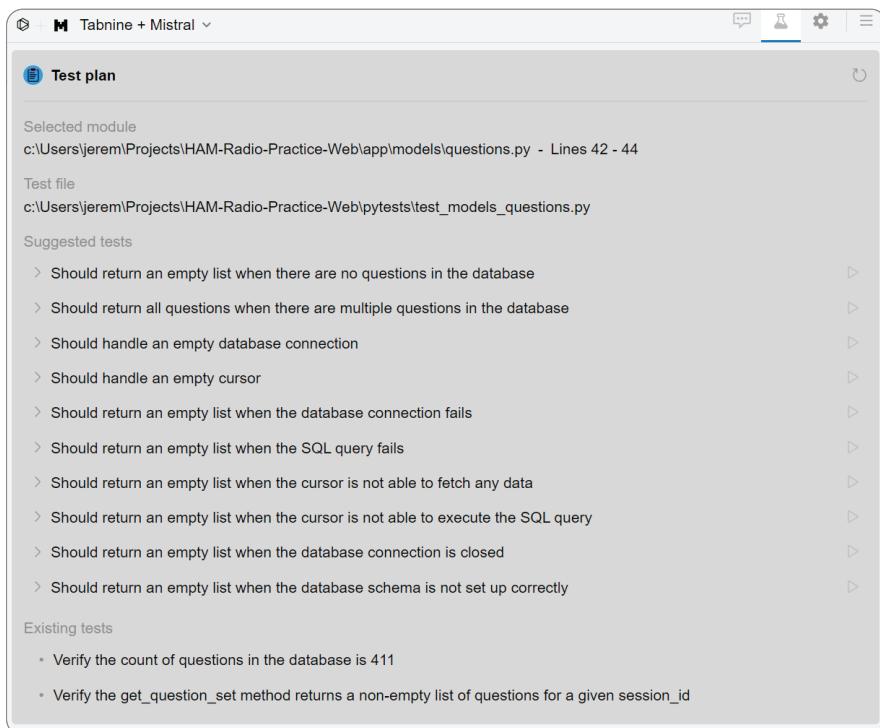
**Figure 8.11** You can see shortcuts above your functions with Tabnine. The options are `test`, `explain`, `document`, and `ask`. I've found they work very well.

Let’s pick a function in our `Questions` class and see how the process works in Tabnine. I’ll click the “test” shortcut above the `get_all_questions` method. Since we already have tests and a fixture set up, it will be interesting to see whether Tabnine infers context from that. As it turns out, as soon as I click `test`, a chat window opens and asks me that very question (figure 8.12): Do I have a test plan?



**Figure 8.12** When clicking the test shortcut for the first time, Tabnine will ask you for a test file to create a test plan. Since we already have tests built, we'll choose that.

After selecting the existing test file, you'll see something far different from Copilot. There you can see a test plan built, with suggested tests, and your existing tests. Tabnine analyzes your code and then generates ideas for tests (figure 8.13).



**Figure 8.13** Tabnine generates a full test plan for your application. It suggests several tests you might consider for your application.

This is a great feature from Tabnine, but how well do the tests work? Let's find out. I selected the top one, "Should return an empty list when there are no connections in the database?" Here is what it returned (figure 8.14).

The screenshot shows the Tabnine interface with a dark theme. At the top, it says "Suggested tests". Below that is a list item: "Should return an empty list when there are no questions in the database". The code block contains Python test code using unittest and Mock:

```

import unittest
from unittest.mock import Mock
from models.questions import Questions

class TestQuestions(unittest.TestCase):
 def setUp(self):
 self.cursor = Mock()
 self.questions = Questions(self.cursor)

 def test_get_all_questions_empty_list(self):
 # Arrange
 self.cursor.execute.return_value = []

 # Act
 result = self.questions.get_all_questions()

 # Assert
 self.assertEqual(result, [])

```

At the bottom of the code block are buttons for "+I Insert" and "Copy". Below the code block is a text input field with placeholder text "Describe how you'd like to modify the test" and a right-pointing arrow button.

**Figure 8.14** Tabnine returns a prebuilt test. Immediately, I can see that it's a `unittest` test, and not `pytest`. Also, it's using `Mock`, and I (bravely) would like to use the in-memory database instead.

One thing I noticed right away is this is a `unittest` test, and not `pytest`. It's also using `Mock`. However, there is an option here to "Describe how you'd like to modify the test." I will simply prompt that to make a `pytest` test and use our fixture to connect to the in-memory database instead of `Mock`. Here is the prompt I used:

- JM Create a `pytest` test instead and use the in-memory database provided in the existing fixture.

Instantly, I got the result I was looking for. It's a simple `pytest` test that uses our in-memory database. Here is the final test it produced:

```

def test_get_all_questions_empty_list(db_connection):
 # Arrange
 # Clear the questions table
 db_connection.execute("DELETE FROM questions")

```

```
db_connection.commit()
Act
result = Questions(db_connection.cursor()).get_all_questions()
Assert
assert result == []
```

This test will open our in-memory database just like the other tests did and delete all the questions from the questions table. Let's run it (figure 8.15).

```
(hamradio) [jeremy@KATANA pytests]$ pytest -s
=====
test session starts
=====
platform linux -- Python 3.12.3, pytest-8.3.2, pluggy-1.5.0
rootdir: /mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/pytests
collected 3 items

test_models_questions.py ...

=====
3 passed in 0.31s
=====
(hamradio) [jeremy@KATANA pytests]$
```

**Figure 8.15** We now have an additional passing test in our test suite. Tabnline created a functional test on the second shot.

The test ran perfectly. I realize that test ordering is important here, and I should run this test toward the end, but it was the first one on the list, so I decided to go for it. Bottom line: Tabnline created a functional test for our software in less than 5 minutes. Additionally, it provided us with many ideas for additional tests. Not all of them make sense to use for our application, but it's a great feature that Copilot doesn't automatically apply.

As a sidenote, I used the Tabnline + Mistral model. Tabnline allows you to choose from several models depending on your needs. You can balance performance, privacy, and accuracy by selecting multiple models. It's located at the top of the chat window, and figure 8.16 shows the models available at the time of this publication.

This is how simple it is to create tests with Tabnline. It has a bit of an advantage as we already had tests and a fixture available, and it fit right in and generated a working test within 5 minutes. It's a huge productivity booster.

Here's what I've noticed about using Tabnline for testing:

- The accuracy of the results is high. Many times, they're better than Copilot.
- I like the suggestions given for additional tests.
- Having a wide choice of models is great if you don't like the results you get from one.

Overall, I personally use Tabnline the most for testing. But let's check out Blackbox AI, another popular product.

The screenshot shows a list of models in the Tabnine Chat interface:

- Tabnine Protected** (Recommended): Better class of performance. Guaranteed privacy with zero data retention. Exclusively trained on permissive licenses to ensure no legal exposure.
- Codestral**: Best class of performance. No guarantee of privacy or deletion of data. May introduce IP liability risks.
- Tabnine + Mistral**: Best class of performance and fastest response time. Guaranteed privacy with zero data retention. May introduce IP liability risks.
- Claude 3.5 Sonnet**: Best class of performance. No guarantee of data privacy. May introduce IP liability risks.
- Command R**: Best class of performance. No guarantee of data privacy. May introduce IP liability risks.
- GPT-4o**: Best class of performance and significantly faster than GPT-4.0 Turbo. No guarantee of privacy or deletion of data. May introduce IP liability risks.

Each model entry includes a "Private" checkbox (checked for Tabnine Protected), a "Protected" checkbox (unchecked for Codestral, Mistral, Claude, Command R, and GPT-4o), and a "Best performance" checkbox (unchecked for Tabnine Protected). A "Recommended" badge is present next to Tabnine Protected.

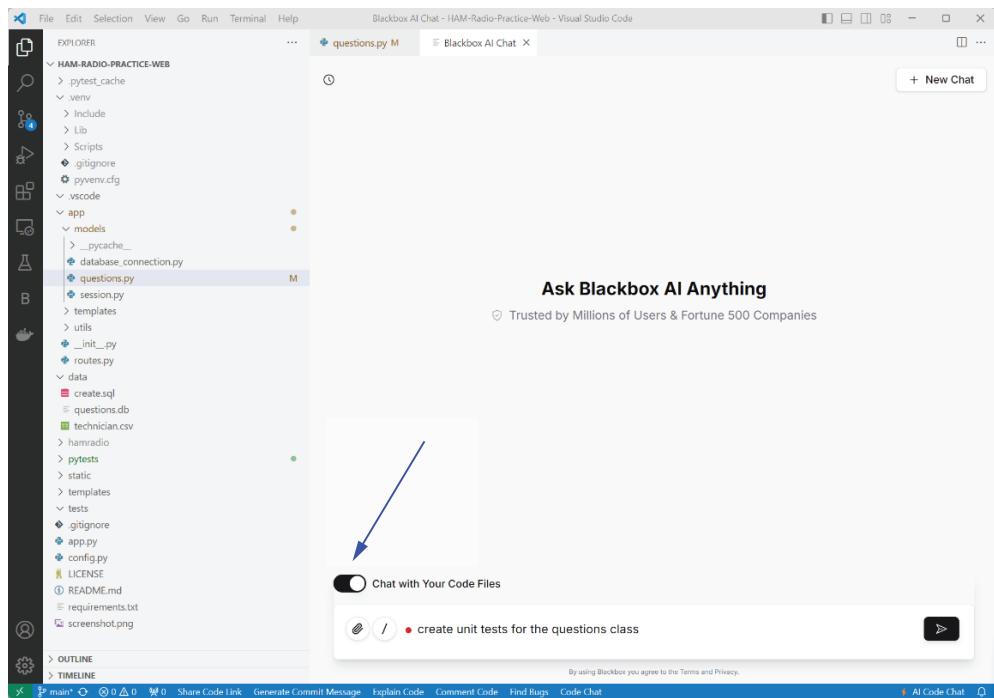
**Figure 8.16** Tabnine allows you to choose between many different models for test generation. If you need higher privacy, or simply want to browse different results, there are many options available.

#### 8.4.4 Applying Blackbox AI for test generation

Let's use our same `pytest` application and learn how test generation works with Blackbox AI. Its interface resembles the other two tools we discussed, and most of the work I do with it is in a chat window. You can conveniently put this chat window into a tab in Visual Studio Code, which I prefer to the sidebar interface. To start our tests, I will open this chat and ask Blackbox to create some tests. Notice there is a button “Chat with Your Code Files.” This feature enables using your application as context automatically when interacting with BlackBox AI (figure 8.17).

I'll start out fresh with our tests for now. I sent the following prompt:

JM Create unit tests for the `questions` class.



**Figure 8.17** With Blackbox AI, you can add the chat screen as a tab in Visual Studio. There's also a button to “Chat with Your Files” so you can include your code as context for the prompt.

Based on this simple prompt, I don't expect it will use pytest, but I'm curious to see what kind of tests it recommends. Here's what I got:

```

import unittest
import sqlite3
from questions import Questions # Import the Questions class
class TestQuestions(unittest.TestCase):
 def setUp(self):
 # Create a test database
 self.conn = sqlite3.connect(':memory:')
 self.cursor = self.conn.cursor()
 # Create the necessary tables
 self.cursor.execute('''
 CREATE TABLE questions (
 id INTEGER PRIMARY KEY,
 correct INTEGER,
 question TEXT,
 a TEXT,
 b TEXT,
 c TEXT,
 d TEXT
)
 ''')


```

```

 self.cursor.execute('''
 CREATE TABLE sessions (
 session_id INTEGER PRIMARY KEY,
 questions_correct INTEGER,
 questions_incorrect INTEGER
)
 ''')
 self.cursor.execute('''
 CREATE TABLE question_sets (
 session_id INTEGER,
 question_id INTEGER
)
 ''')
 # Insert some test data
 self.cursor.execute('''
 INSERT INTO questions (id, correct, question, a, b, c, d)
 VALUES (1, 1, 'What is the capital of France?',
 'Paris', 'London', 'Berlin', 'Rome')
 ''')
 self.cursor.execute('''
 INSERT INTO sessions (session_id,
 questions_correct, questions_incorrect)
 VALUES (1, 0, 0)
 ''')
 self.cursor.execute('''
 INSERT INTO question_sets (session_id, question_id)
 VALUES (1, 1)
 ''')
 self.conn.commit()
def tearDown(self):
 # Close the database connection
 self.conn.close()
def test_get_all_questions(self):
 questions = Questions(self.cursor)
 result = questions.get_all_questions()
 self.assertEqual(len(result), 1)
def test_get_question(self):
 questions = Questions(self.cursor)
 result = questions.get_question(1)
 self.assertEqual(result['id'], 1)
 self.assertEqual(result['correct'], 1)
 self.assertEqual(result['question'],
 'What is the capital of France?')
 self.assertEqual(result['a'], 'Paris')
 self.assertEqual(result['b'], 'London')
 self.assertEqual(result['c'], 'Berlin')
 self.assertEqual(result['d'], 'Rome')
def test_get_question_set(self):
 questions = Questions(self.cursor)
 result = questions.get_question_set(1)
 self.assertEqual(len(result), 1)
def test_get_answered_questions(self):
 questions = Questions(self.cursor)
 result = questions.get_answered_questions(1)
 self.assertEqual(result, 0)

```

```

def test_get_next_question(self):
 questions = Questions(self.cursor)
 result = questions.get_next_question(1)
 self.assertEqual(result, '1')
def test_store_answer(self):
 questions = Questions(self.cursor)
 result = questions.store_answer(1, 1, 1)
 self.assertEqual(result, 'test')
def test_mark_answer_wrong(self):
 questions = Questions(self.cursor)
 questions.mark_answer_wrong(1)
 self.cursor.execute('SELECT questions_incorrect
FROM sessions WHERE session_id = ?', (1,))
 result = self.cursor.fetchone()
 self.assertEqual(result[0], 1)
def test_tally_results(self):
 questions = Questions(self.cursor)
 result = questions.tally_results(1)
 self.assertEqual(result['questions_correct'], 0)
 self.assertEqual(result['questions_incorrect'], 0)
if __name__ == '__main__':
 unittest.main()

```

These tests cover all the methods of the `Questions` class. They create a test database and insert some test data before running the tests. Each test checks the expected output of a method and asserts that it is correct. If any of the assertions fail, the test will fail and an error message will be displayed.

Looking at the results, I must say I'm impressed. It created a set of `unittest` tests, as I expected, but Blackbox looked over my application and gathered context in a great way. I'll explain. In this part of the result:

*These tests cover all the methods of the Questions class. They create a test database and insert some test data before running the tests.*

Blackbox AI assumed this is how I want to run my tests. It doesn't insert mocks or anything irrelevant. Rather than telling it to create an in-memory database, Blackbox inferred that from looking at my existing application and how it was set up. Neither Copilot nor Tabnine did that. The code also looks great. I only needed a few modifications to make it run. I added these two lines as I did in the previous `pytest`:

```

Add the path to the 'app' directory to the sys.path
sys.path.insert(0, os.path.abspath(os.path.join
(os.path.dirname(__file__), '../app')))

Import the questions module
from models.questions import Questions

```

And then I ran the `unittest` test for this file:

```
python -m unittest test_questions.py
```

Figure 8.18 shows the results.

```
(hamradio) [jeremy@KATANA tests]$ python -m unittest test_questions.py
..1
....Correct answer is 1 and you selected 1
correct
..

Ran 8 tests in 0.001s

OK
(hamradio) [jeremy@KATANA tests]$
```

**Figure 8.18 With a small modification, Blackbox AI generated an in-memory database that looks like our production database and created tests for our functions. It was successful on the first run.**

We've proven that Blackbox AI can gather context and generate unit tests quickly and efficiently. So far, they look the best out of the tools we've examined. But what about using pytest? Let's see if Blackbox AI can wow us with code for this framework. I repeated the same test, but used a different prompt:

- JM Create pytest tests for the Questions class.

It immediately returns yet another pleasing result:

- Here is an example of how you can write pytest tests for the Questions class:

```
tests/test_questions.py
import pytest
import sqlite3
from models.questions import Questions

@pytest.fixture
def db():
 conn = sqlite3.connect(':memory:')
 cursor = conn.cursor()
 # Create the necessary tables
 cursor.execute('''
 CREATE TABLE questions (
 id INTEGER PRIMARY KEY,
 correct INTEGER,
 question TEXT,
 a TEXT,
 b TEXT,
 c TEXT,
 d TEXT
)
 ''')
 cursor.execute('''
 CREATE TABLE sessions (
 ''')
```

```
 session_id INTEGER PRIMARY KEY,
 questions_correct INTEGER,
 questions_incorrect INTEGER
)
```
cursor.execute('''
    CREATE TABLE question_sets (
        session_id INTEGER,
        question_id INTEGER
    )
```
Insert some test data
cursor.execute('''
 INSERT INTO questions (id, correct, question, a, b, c, d)
 VALUES (1, 1, 'What is the capital of France?',
'Paris', 'London', 'Berlin', 'Rome')
```
cursor.execute('''
    INSERT INTO sessions (session_id, questions_correct, questions_
incorrect)
    VALUES (1, 0, 0)
```
cursor.execute('''
 INSERT INTO question_sets (session_id, question_id)
 VALUES (1, 1)
```
conn.commit()
yield cursor
conn.close()

def test_get_all_questions(db):
    questions = Questions(db)
    result = questions.get_all_questions()
    assert len(result) == 1

def test_get_question(db):
    questions = Questions(db)
    result = questions.get_question(1)
    assert result['id'] == 1
    assert result['correct'] == 1
    assert result['question'] == 'What is the capital of France?'
    assert result['a'] == 'Paris'
    assert result['b'] == 'London'
    assert result['c'] == 'Berlin'
    assert result['d'] == 'Rome'

def test_get_question_set(db):
    questions = Questions(db)
    result = questions.get_question_set(1)
    assert len(result) == 1

def test_get_answered_questions(db):
    questions = Questions(db)
    result = questions.get_answered_questions(1)
    assert result == 0
```

```

def test_get_next_question(db):
    questions = Questions(db)
    result = questions.get_next_question(1)
    assert result == '1'

def test_store_answer(db):
    questions = Questions(db)
    result = questions.store_answer(1, 1, 1)
    assert result == 'test'

def test_mark_answer_wrong(db):
    questions = Questions(db)
    questions.mark_answer_wrong(1)
    db.execute('SELECT questions_incorrect FROM sessions
WHERE session_id = ?', (1,))
    result = db.fetchone()
    assert result[0] == 1

def test_tally_results(db):
    questions = Questions(db)
    result = questions.tally_results(1)
    assert result['questions_correct'] == 0
    assert result['questions_incorrect'] == 0

```

Note that I've used the `pytest.fixture` decorator to define a fixture that creates a test database and returns a cursor object. This fixture is used in each test function to create a new database connection.

I've also used the `assert` statement to make assertions about the expected behavior of the `Questions` class. If any of these assertions fail, the test will fail and an error message will be displayed.

You can run these tests using the `pytest` command. For example:

```
pytest tests/test_questions.py
```

This will run all the tests in the `test_questions.py` file and display the results.

These results look great at first glance, so I make the same folder modification to the test and run it as is. And all eight tests pass 100% with no errors or problems (figure 8.19).

```
(hamradio) [jeremy@KATANA pytests]$ pytest test_models_questions_2.py
=====
 test session starts =====
platform linux -- Python 3.12.3, pytest-8.3.2, pluggy-1.5.0
rootdir: /mnt/c/Users/jerem/Projects/HAM-Radio-Practice-Web/pytests
collected 8 items

test_models_questions_2.py .. .... [100%]

===== 8 passed in 0.06s =====
(hamradio) [jeremy@KATANA pytests]$
```

Figure 8.19 Here are the results of the tests as they were generated by Blackbox AI. It's nearly plug and play, though in real situations, we'd want to make modifications and double check everything.

So far, out of the three tools, Blackbox took the least amount of effort to generate tests for our application. Here is what I didn't do but would have done if the results weren't good.

MAKING MY PROMPT MORE SPECIFIC

For my prompt, I sent:

JM Create pytest tests for the Questions class.

This is a very general prompt, however if I didn't get the results I wanted, I could say something like

JM Create pytest unit tests for the questions class located at /app/models/questions.py. The application uses a SQLite database, and I would like to create an in-memory database that matches the production database for testing purposes.

USING A BLACKBOX AI AGENT

Blackbox AI has agents, which are specific to the language you're using. These agents are more focused on the language and environment, and not only do they have a Python agent, but a Flask agent as well.

You should use these agents when you want your request to be specific. For example, if you type in "build a unit test," Blackbox will see your Python code and infer that you mean a unit test for Python, but it's no guarantee. There are lots of languages that use unit tests. By specifying an agent, you are asking for more specific results (figure 8.20).

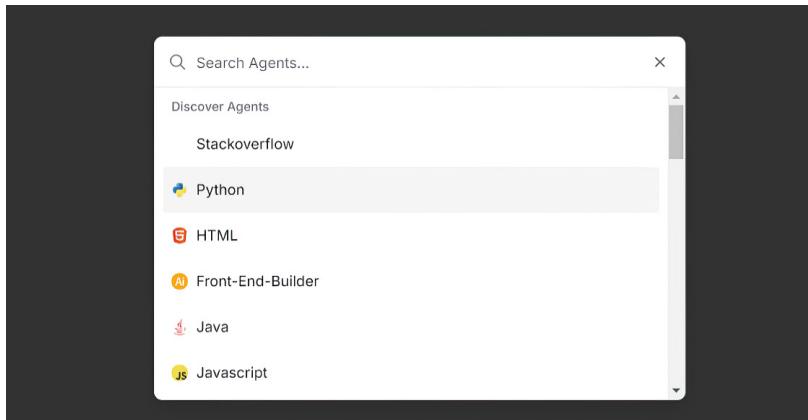


Figure 8.20 Blackbox AI has agents, specialized prompt-processing tools that focus on the language you're developing in. You can bring up an agent from the front slash (/) button in the prompt window in chat.

Thankfully, it produced exactly what I was looking for, but you never know when you might need to make some changes, and that's where an agent comes in handy. If you

need to get more specific, start with the prompt, or with Blackbox, and choose an agent.

8.4.5 Which tool should you use for testing?

Now that you've seen these tools in action, you can evaluate which one you might prefer for testing. Here are some things to keep in mind.

GITHUB COPILOT

Copilot is arguably the most popular generative AI coding tool today.

Pros

- Extensive language support: Excels in Python, JavaScript, TypeScript, Ruby, Go, C#, and C++
- Powered by OpenAI's GPT-4 Model, which is very advanced
- Integrates well with GitHub repositories

Cons

- Cloud-based solution, which may raise privacy concerns

TABNINE

Tabnine is a code completion tool that focuses on privacy and enterprise level security.

Pros

- Offers both cloud-based and on-premises solutions for enhanced security
- Trained on carefully vetted, high quality code repositories
- Several different models to choose from
- SOC-2 Compliant, ensuring security and privacy

Cons

- May have less language coverage than GitHub Copilot
- May have less advanced NLP capabilities

BLACKBOX AI

Blackbox is a newer entrant in the AI-powered coding assistant market, focusing on code generation and problem solving.

Pros

- Offers a free plan, with basic features and code chat
- Provides advanced features such as code creation and priority assistance with test plans
- Works with multiple IDEs
- Has incredible accuracy with code generation.

Cons

- Less established compared to Copilot and Tabnine

- Single model available currently
- Limited information available about its training data and model specifics

Check out all these tools and give them a shot. You'll likely find one you like that fits all your needs. There's no wrong choice here as they are all great.

AI test generation workflow

Follow this process to get the most effective AI generated tests:

- *Generate scaffold first.* Create basic test structure with fixtures and imports.
- *Review assumptions.* Check if the AI correctly understands your code's purpose.
- *Add test cases incrementally.* Generate one type of test at a time.
- *Verify edge cases.* Explicitly ask for tests covering boundary conditions.
- *Refine assertions.* Ensure tests verify the right behaviors.
- *Consolidate fixtures.* Look for opportunities to reuse test setup.
- *Run and debug iteratively.* Fix problems one at a time, using AI to help with errors.
- *Document test purpose.* Ask the AI to add comments explaining test coverage

Remember that AI can generate the structure and common cases, but you'll need to guide it toward complete test coverage.

We learned how generative AI tools can make testing easier and more helpful. Tools such as GitHub Copilot, Tabnine, and BlackboxAI each have their own strengths, but they all aim to help you write better tests faster. These tools don't just save time, but they also give you new ideas for tests you might not have thought of.

As you start building software faster, using AI in your testing can keep the quality high. The best approach is to combine what AI is good at (e.g., spotting patterns), with your own knowledge of how your code should work. Collaborating with AI empowers developers to create more robust test suites and deliver software that performs reliably across real-world scenarios.

Summary

- The use of generative AI tools allows Python developers to save valuable time, while maintaining high-quality standards for their applications.
- By applying refined prompt strategies, developers often see improvements in AI-generated tests that better reflect the nuances of their software environments.
- In-memory database fixtures enable comprehensive testing without affecting production environments
- Copilot, Tabnine, and Blackbox AI have different strengths, and you can choose the right testing companion based on your specific needs and constraints
- AI-generated tests can be incorporated into existing workflow, while maintaining appropriate review processes to ensure comprehensive test coverage