Can AI create unit tests?

An exploration of LLMs and the future of software testing.

Produced by

Bluefruit Software

Contents

Introduction

Unit tests validate the behaviour of individual components within software, helping developers catch bugs early, reduce defects, and provide a safety net when refactoring or enhancing code.

However, creating a comprehensive suite of these tests can be time-consuming and repetitive. In a well-tested codebase, it's common to have more test code than source code due to the detailed nature of unit testing. A device might require thousands of individual unit tests, which can take weeks to write and maintain as the codebase evolves. Additionally, achieving acceptable coverage requires testers to have detailed knowledge of the system, attention to edge cases, and a high degree of persistence. Given these constraints, the potential for AI to automate the generation of unit tests is of significant interest.

We set out to discover the effectiveness of large language models (LLMs) in generating unit tests, and assess their ability to produce reasonable, useful, and comprehensive test cases.

After several weeks of experiments, we determined that, while it’s still early days, AI already offers a promising solution to reduce the time and effort involved in test creation.

Experiment 1

Which LLM writes the best unit tests?

Our initial focus was to determine which LLMs produced the most effective unit tests, both in terms of code coverage and mutation scores.

Method

We evaluated multiple LLMs: ChatGPT-4o, Claude Opus, Llama 3.1 and Gemini 1.5, using the GoogleTest framework. The subject of our tests was TinyXML2, an open-source C++ XML parser, chosen for its compactness and availability of hand-written unit tests for comparison

We assessed the project's hand-written unit tests for code coverage and mutation score, and then got the LLMs to write replacement tests.

Tools used

Mull
Mutation testing tool using LLVM API to inject mutations at compile time. It accounts for code coverage, meaning only tested code is mutated.

llvm-cov
Code coverage utility that informs Mull which code to mutate. It also provides a secondary statistic on the quality of unit tests.

Cover Agent
A CLI tool that iteratively guided AI models by providing coverage feedback and refining the generated tests based on predefined goals (e.g., a target coverage percentage).

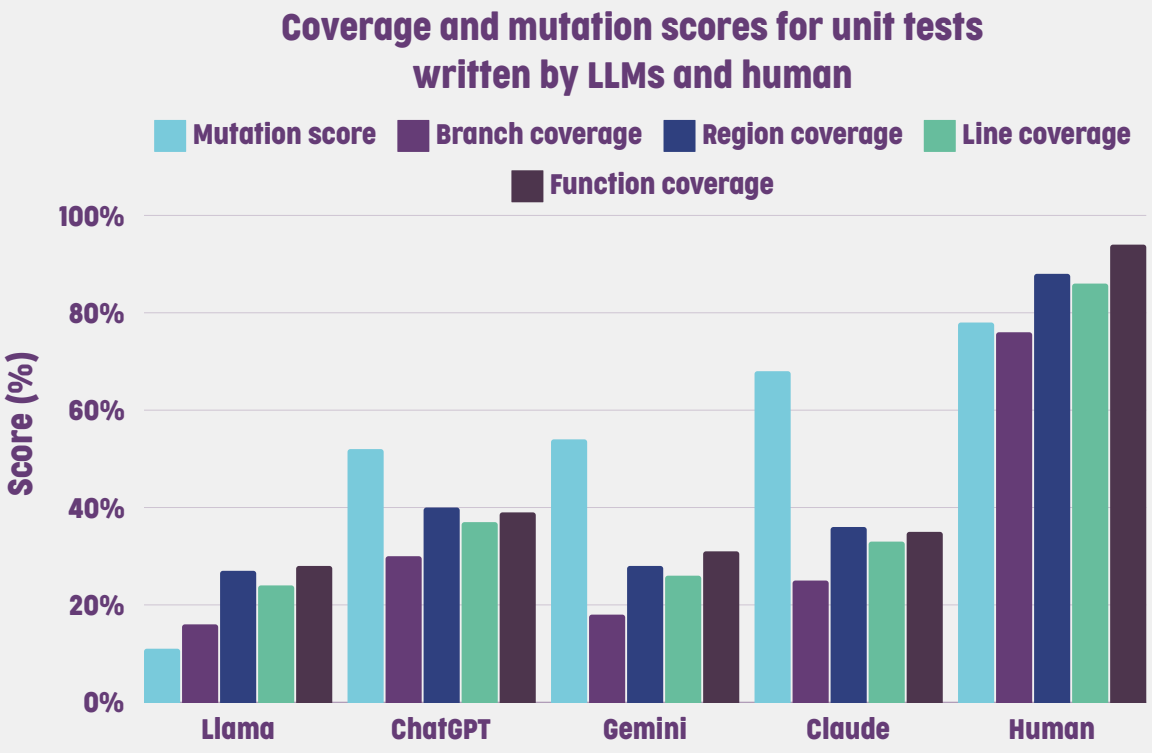
Google Test (gtest)
A specialised C++ testing framework developed that allows developers to write and run unit tests for their C++ code.

Results

The hand-written tests were superior to all four LLMs, scoring significantly higher for function coverage, line coverage, region coverage, and branch coverage.

Of the LLMs, ChatGPT achieved the highest coverage score, with Claude a close second. However, when it came to mutation scoring, Claude performed significantly higher than the other models, detecting almost as many bugs as the hand-rolled unit tests.

	Function Coverage	Line Coverage	Region Coverage	Branch Coverage	Mutation score
GPT	39.1%	36.9%	40.0%	29.6%	52%
Claude	35.2%	33.0%	36.2%	25.1%	68%
Llama	27.7%	23.7%	27.0%	16.4%	11%
Gemini	30.7%	26.4%	27.8%	18.0%	54%
Human	94.1%	85.9%	88.1%	76.2%	78%



Bluefruit Software

Experiment 2

Testing code with standards

For the next part of the study, we explored whether LLMs would create better unit tests based on a standard/specification rather than source code.

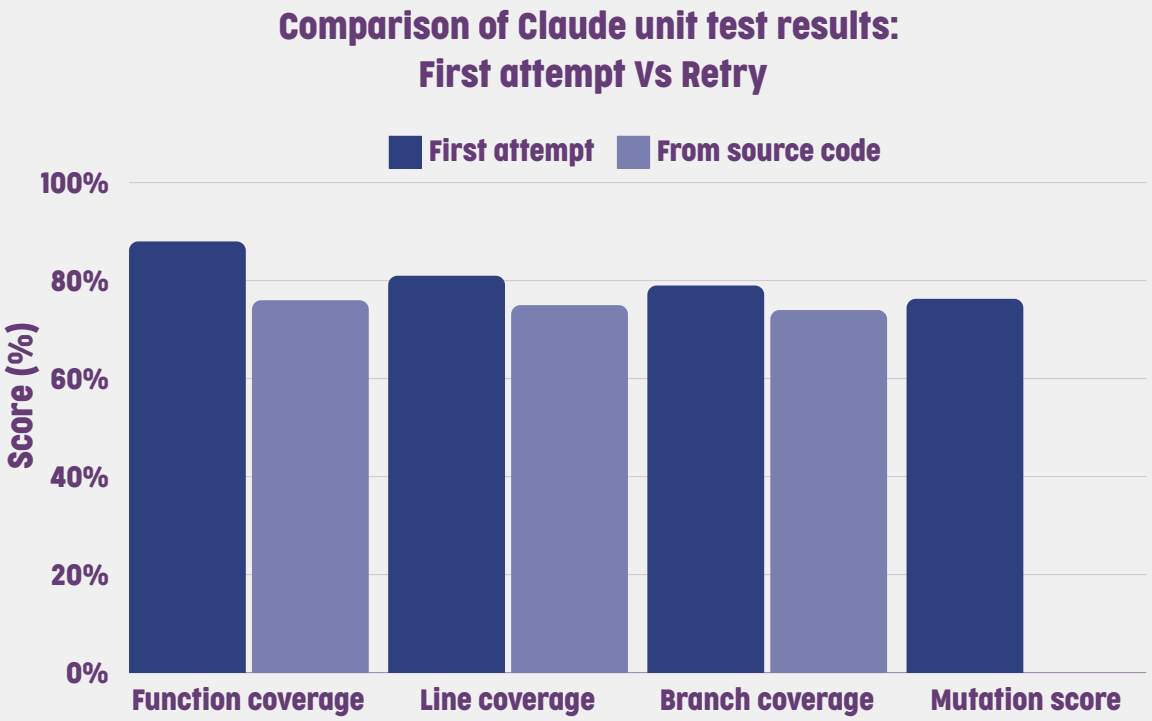
Method

For this experiment we selected TinyJSON, a lightweight JSON parser and generator for C and C++, with a single header/source, and written using TDD.

In one condition, we provided Claude with the JSON specification and header file and prompted it to write a set of unit tests using gtest to cover as much of the specification as possible. In another condition, we prompted Claude to create its unit tests based on the source code.

Results

Unit tests generated from the specification performed better than those generated from source code. They were readable and more comprehensive, covering key JSON features like parsing nested structures and handling various data types. However, Claude did require some coaching around broken tests.



Tests generated from source code tended to focus on lower-level functions, were less readable, and neglected critical functionality such as array handling.

Basing unit tests on a clear specification appears to be a more reliable method for achieving broad coverage and protecting against code mutations. This finding suggests that AI models may perform better when they are provided with high-level guidance that allows them to generate tests that cover a wider range of functional scenarios.

Like humans, AI would seem to be most effective while using a test-first approach.

The hand-written unit tests included with the TinyJSON repo still scored higher than Claude across the board.

Experiment 3

Improving results by providing feedback

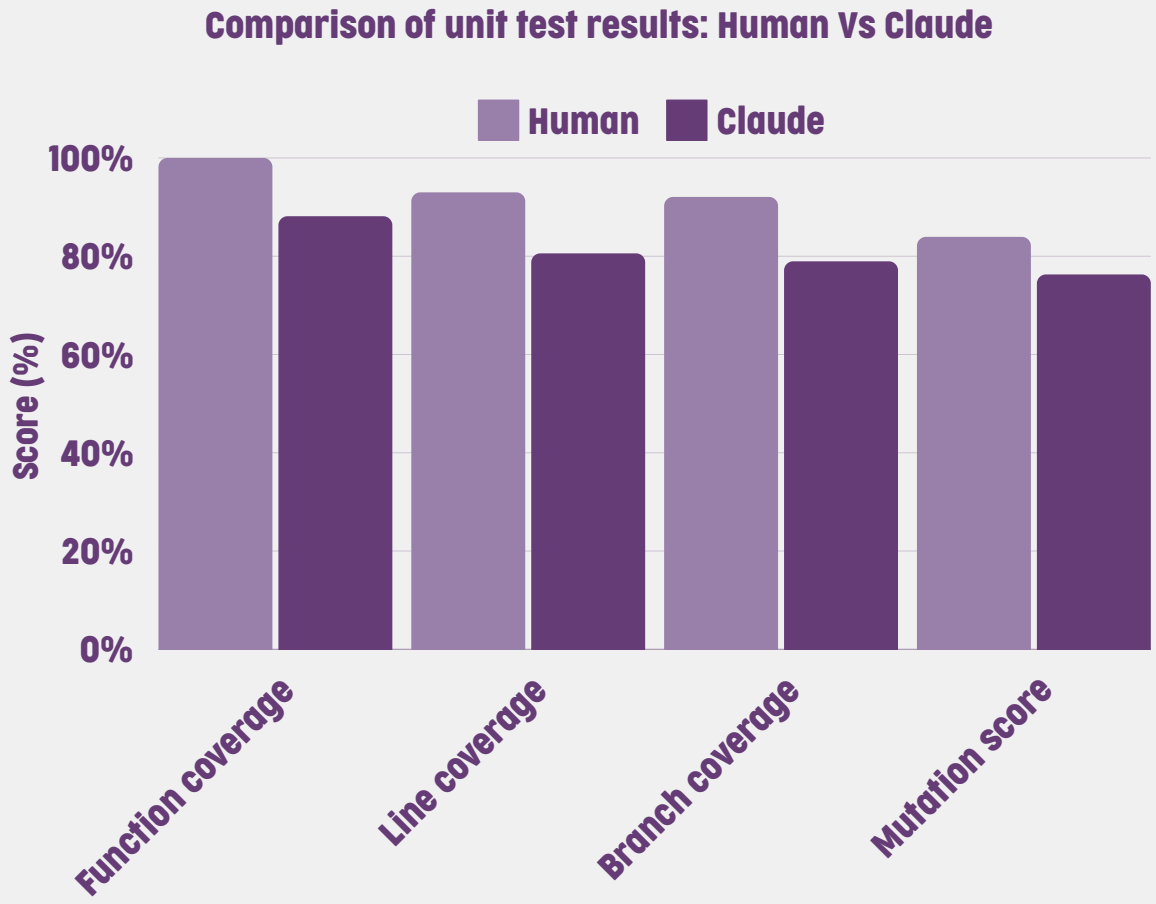
Our next study explored whether feedback could improve LLM performance in generating unit tests. Specifically, we examined whether allowing Claude to assess the results of its initial tests and then generate a second round of tests could improve coverage and mutation scores.

Method

Claude was given the results of its initial test suite, including coverage gaps and failed tests, and was prompted to improve its output. We then compared this to its previous output.

Results

While coverage improved slightly on a second attempt, the gains were modest. Claude required highly specific prompts to address the identified gaps, and additional improvements often led to test failures that required further back-and-forth refinement. Although this method shows potential, it suggests that LLMs in their current state still struggle when presented with more complex coding tasks.



Crucially, while hand-written tests offered greater coverage, Claude actually generated 0.6% unique tests, suggesting that an experienced human software tester, leveraging AI would provide the best overall result.

Experiment 4

Can an LLM make code more testable?

TinyJSON was written using TDD and is, therefore, inherently testable. However, in real-world scenarios where unit tests are required, it is unlikely that the source code will be written with unit testing in mind.

Can an LLM adjust the code to make it more test-friendly?

Method

For this study, we found a project on GitHub that had not been unit tested, MOS6502 Emulator in C++, and asked Claude Opus to write some tests for it and assess quality. We then asked the LLM to make adjustments to the code to improve testability before writing the tests again. Finally, we assessed and compared the results.

Results

This did not go well. Claude struggled to predict what the register values would be, it kept getting the program counter wrong, and couldn't fix these problems—even after several prompted iterations. In trying to correct its mistakes, Claude introduced significant complexity, creating long,

unreadable tests.

We were forced to conclude that complex programming subjects such as this may be beyond current LLMs.

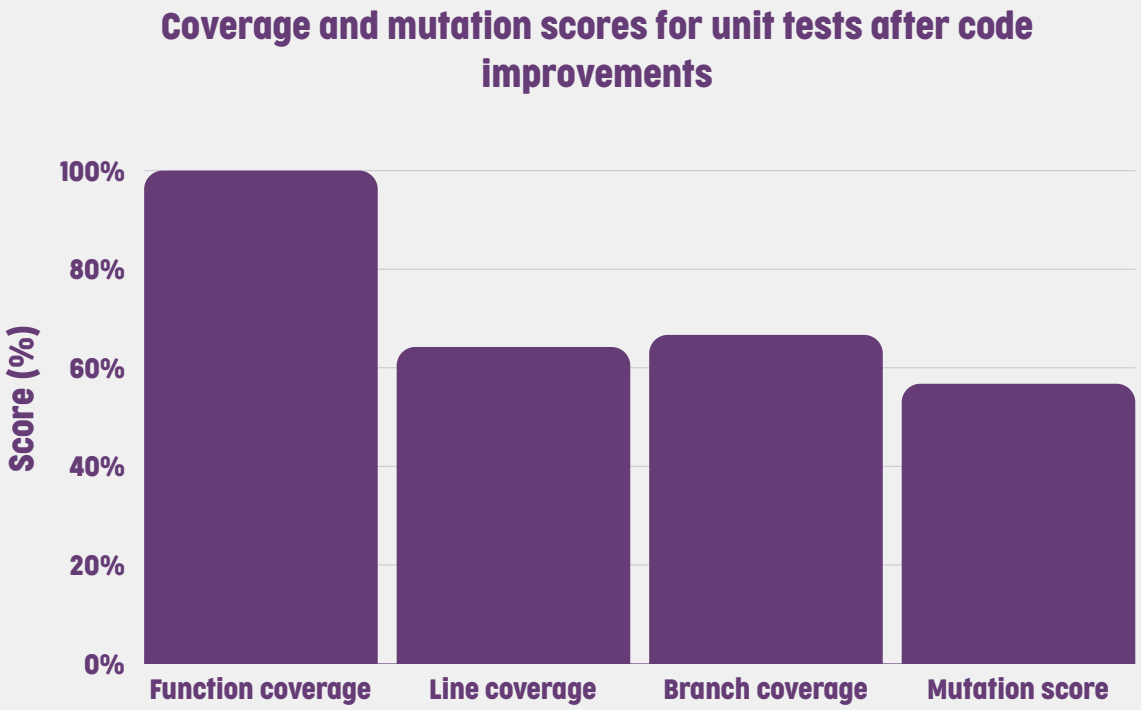
Experiment 4.5: Reduced complexity

Claude had failed to refactor TinyJSON, but how would it fare on something easier?

Method
We found a simpler piece of code—a Tetris game written in C++. Claude was asked to write some rests on an untested, no-unit-test class representing a tetramino (block). We then asked it to make adjustments to the code to improve testability before writing the tests again. Finally, we assessed and compared the results.

Results
Claude managed to achieve a function coverage of 100%, however this is likely due to the simplicity of the tetromino class. The line, branch, and mutation coverage remained low, indicating insufficient test depth and effectiveness.

It struggled with setting up mocks correctly, often assigning function names that did not match those in the class headers, leading to functions that were never called. The initial tests failed to execute properly, requiring manual corrections and multiple iterations to resolve these issues. Claude also seemed to have difficulty inferring the specific requirements and behaviours of the tetromino class, which impacted the relevance and accuracy of the generated tests. It might have benefited from having some requirements for this class.



Code recommendations

- Given the source code for the tetromino class and asked for improvements, Claude Opus came back with the following suggestions:

- Heavy reliance on global state: The class uses global constants like COLUMNS and ROWS, which makes it difficult to test the class in isolation.

- Lack of dependency injection: The class directly uses functions like get_tetromino() and get_wall_kick_data(), making it hard to mock these dependencies for testing.

- Complex methods with multiple responsibilities: Some methods, like rotate(), perform multiple operations and have complex logic, making them difficult to test thoroughly.

- Limited encapsulation: The class exposes internal state through methods like get_minos(), which can make it challenging to maintain invariants and test state changes.

- Absence of interfaces: The class doesn't implement any interfaces, which limits the ability to use polymorphism and create mock objects for testing.

- Tight coupling with the game matrix: Many methods take a constant reference to the game matrix, creating a strong dependency on the game state.

- Lack of clear separation between game logic and tetromino behavior: The class mixes concerns related to tetromino movement and game rules.

- No clear way to set up specific test scenarios: The class doesn't provide mechanisms to easily set up specific states for testing edge cases.

Code changes
Claude then performed the following code changes:

- Added dependency injection to facilitate better mocking.

- Replaced global constants with constructor parameters.

- Added an interface for the game board.

- Broke up some larger methods into more testable units

The results showed a 15% increase in line coverage, but branch coverage decreased slightly and obtaining a mutation score was not feasible due to numerous failing tests. Several iterations were undertaken to address these failures by consulting the model, but progress was limited. The model struggled with the system setup and introduced a dependency injection mechanism that it could not effectively use.

As the iterations to fix the issues continued, the source code began to change significantly, leading to decreased trust in the model's alterations. There were concerns that these changes might be breaking the original game logic. Without existing tests in the repository, it was impossible to verify if any functionality had been compromised. Despite efforts to enhance testability, the model continued to face difficulties, and its effectiveness diminished in subsequent attempts.

Experiment 5

Further analysis of tests from specification

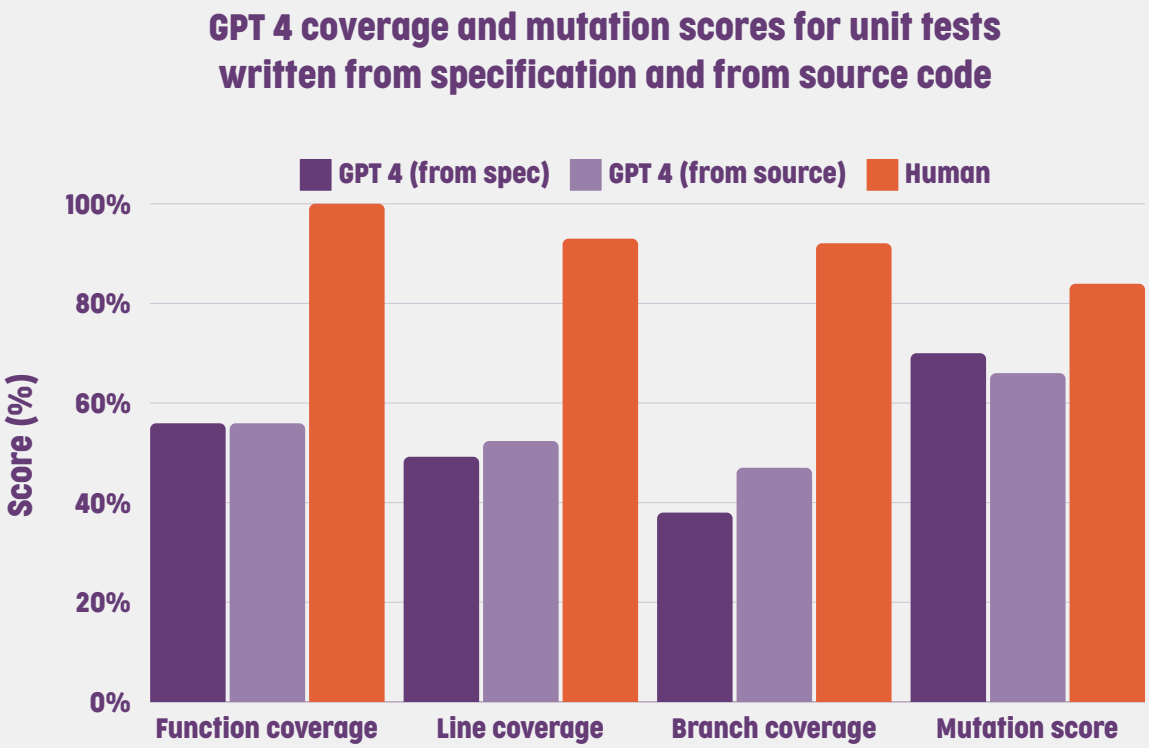
In previous studies, Claude seemed to perform better when asked to write tests based on specification, rather than from the source code. We explored whether the same was true of other LLMs.

Method

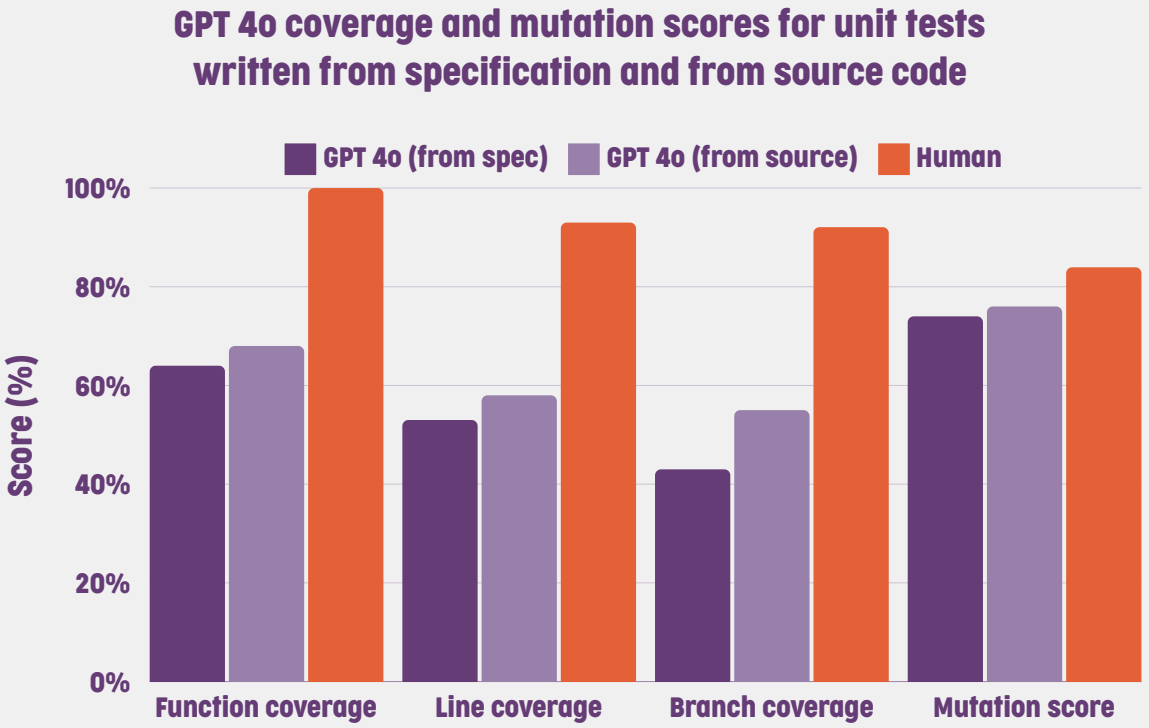
In one condition, we provided ChatGPT 4, ChatGPT 4o, Llama 3.1 70b, and ChatGPT o1-preview with the TinyJSON source code. In the other, we provided them models with the specifications. We then compared the results of the hand-written tests written by the developers using TDD.

Results

GPT4 performed slightly better when given the source code. The exception being the mutation score which was better when written from the specification. Overall, the performance was not very good.

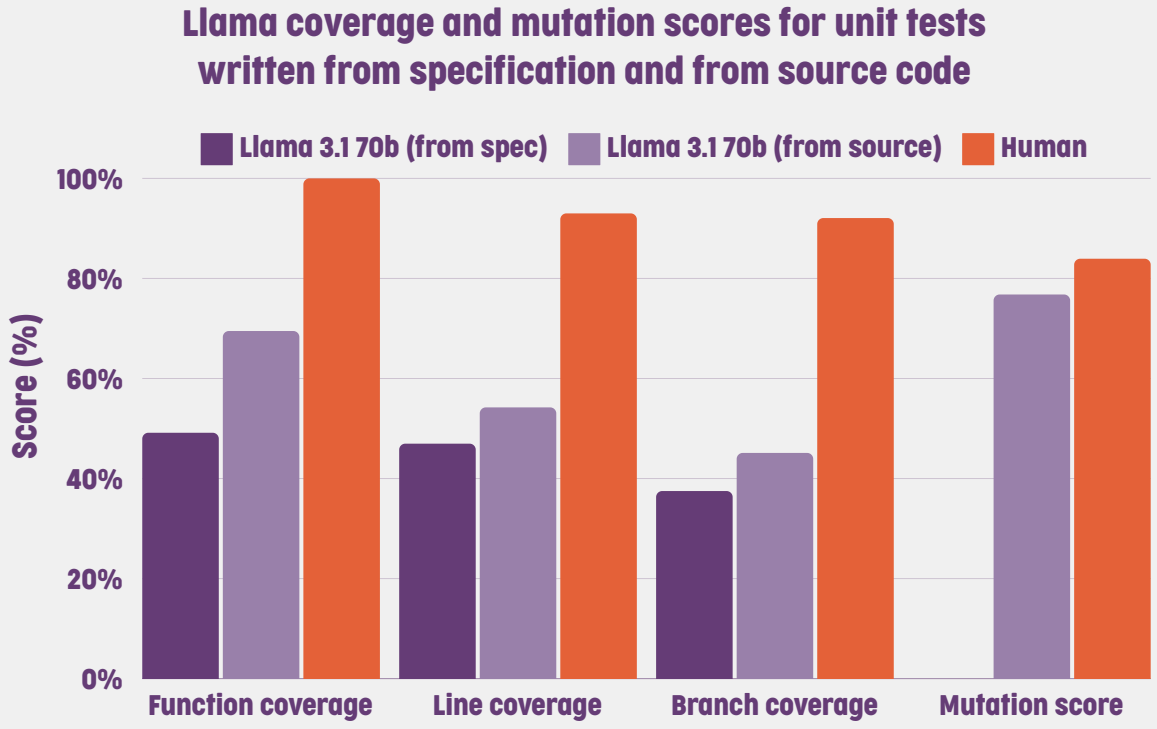


GPT 4o performed slightly better than GPT4 overall, with the source code condition achieving better scores across all effectiveness measures.



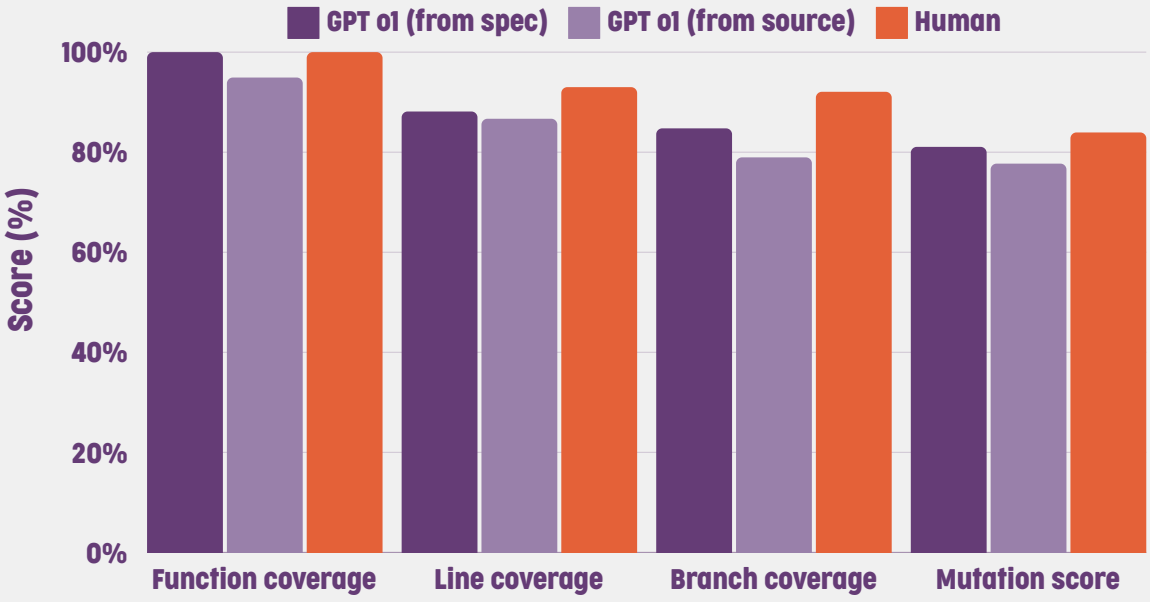
In the source code condition, Llama outperformed the GPT models in all but branch coverage.

In the specifications condition, Llama produced lots of failing tests. We went through a few iterations of feeding back errors and asking it to fix them, but this was not effective and, as such, the condition did not yield a mutation score.

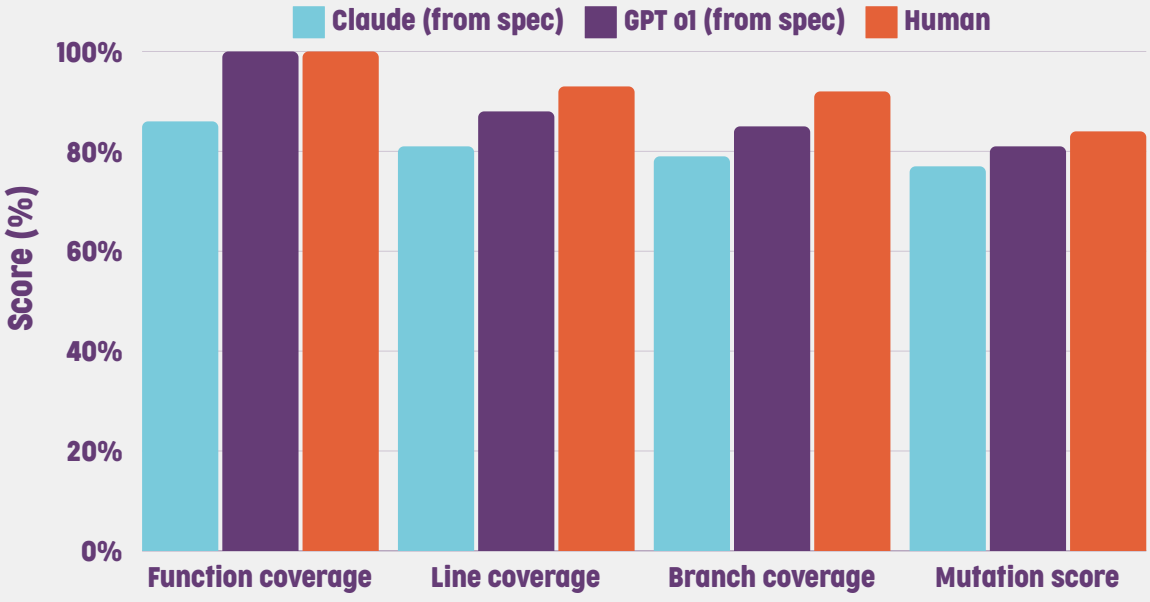


Released as we conducted this experiment, ChatGPT 01 produced by far the best results. Like Claude, it performed best writing tests from specifications rather than source code. However, its tests performed better than Claude's in every way.

GPT o1 Coverage and mutation scores for unit tests written from specification and from source code



Claude Vs GPT o1 coverage and mutation scores for unit tests written from specification



Conclusion

Our study demonstrates that AI has the potential to significantly streamline the process of creating and maintaining unit tests—and the technology is improving constantly. By leveraging advanced language models, we can already reduce the time and effort required to ensure software reliability, allowing teams to focus on innovation and development.

However, LLMs are still just tools. Without expert prompting and oversight, the work they produce has little value. Experienced software developers and testers are still essential to the creation of high-quality, testable, and thoroughly tested software.

If your organisation is looking to enhance its software testing processes, we can help. Get in touch to learn how we can support your software development and testing needs.

+44 (0) 333 577 7111

info@bluefruit.co.uk

www.bluefruit.co.uk