The Impact of GitHub Copilot on Test-First Development

Ilana Shapiro[1], Michael Peng[1] and Andrew Lara[1]

[1]UC San Diego

Abstract

In this work, we examine the impact of GitHub Copilot on Test-First Development (TFD), a common software development paradigm in which tests are written before implementation. In this way, developers are required to express problem requirements via tests, which then inform the implementation. We conducted a between-subjects (no Copilot vs Copilot) pilot study with four participants, which consisted of coding tasks and pre- and post-task surveys. In each coding task, participants iterated between writing a comprehensive API and writing a test suite for the given problem, with or without Copilot. We discovered that while Copilot greatly enhanced productivity, it resulted in superficial understanding of the problem requirements and decreased scope of the test suites. Developers should therefore be cautioned that when using Copilot for TFD, though they may gain initial increased productivity, Copilot ultimately leads to decreased problem comprehension and limits the coverage of their tests.

Keywords: Software Engineering. AI assistants. Generative AI. Test-Driven Development.

1 Introduction

First introduced in the Extreme Programming community, Test-Driven Development (TDD) is a workflow in which developers iterate between writing tests and implementing features in order to pass those tests. The original definition of TDD by Kent Beck requires practitioners to repeat the following steps [1]:

1. For a new feature, write only enough test code to fail.
2. Write only enough implementation code to pass the test.
3. Refactor the codebase while ensuring that all tests pass.

Case studies, controlled experiments, focus groups, interviews, and surveys have assessed the value of TDD over the past 20 years [2]. Nearly all high-rigor and high-relevance studies on TDD in industry settings have found that TDD improves external quality in the form of defect density in delivered software, reduces complexity, and is well-perceived by practitioners, but can potentially reduce productivity, which means it might increase development time [2]. However, TDD is not necessarily appropriate in every scenario. GUI development, for example, is a well-known application domain for which TDD is especially challenging [3]. Thus, more long-term case studies are still needed to further validate TDD for industry adoption and identify situations where TDD may be detrimental to software engineering objectives.

One of the reasons behind the inconclusiveness of past work on TDD is variance in ramp-up time [4]. Anecdotal and empirical evidence have shown that since TDD has a nontrivial learning curve, its benefits only manifest themselves after some initial investment and ramp-up time [4]. In this study, our subjects consist of students at UC San Diego, who are mostly novice programmers with little professional experience using TDD. Since we don't have the resources to give our subjects sufficient time to develop proficiency against probable reluctance to TDD [5], we opt to relax its constraints and study Test-First Development (TFD) instead, which only requires each feature's tests to be written before its implementation. Along with writing tests, we also investigate API design.

Recently, Generative AI tools have been applied to software development, with the advent of OpenAI Codex [6] and GitHub Copilot. These tools have promise to facilitate and automate tedious and predictable tasks. Given that TFD has been associated with reduced productivity, Generative AI can potentially mitigate the downsides of adopting TFD while retaining its benefits to external quality and complexity. To assess this potential, our study addresses how the use of GitHub Copilot in the API design and test writing phases of TFD would affect productivity and code quality. Our work complements recent efforts to evaluate Generative AI's ability to write implementation code in a TDD workflow [7].

In this paper, we report a controlled, between-subjects pilot study where four participants are asked to iterate between writing a comprehensive API and test suite for high-level programming tasks, either with or without GitHub Copilot. We perform thematic analysis on the results, record the time each participant took to complete each task, and analyze their resulting APIs and test suites. We find that although Copilot significantly reduced the time subjects took to write APIs and test suites according to our description, it actually hindered their understanding of the problem requirements. In conclusion, we warn that while Generative AI tools may be ideal for speeding up implementation of specifications [7], they ultimately serve as a crutch for developers in producing initial tests for a problem, resulting in a more surface-level understanding of the problem.

2 Method

We conducted a 1-hour pilot study with 4 participants. Each participant first completed a pre-task survey indicating their background with relevant topics (see Table 1 for details).

ID	JavaScript Experience	Jest Experience	Copilot Experience
P1	2	1	10
P2	2	1	3
P3	10	8	10
P4	10	9	7

Table 1. Participants' overview from pre-task survey. All experience metrics are self-reported ratings from 1 (least) to 10 (most).

Then, participants were given one or two tasks to complete, depending on what they were able to finish in the allotted 1-hour time frame. We designed three tasks that we randomly assigned to each participant (see Table 2 and Section 2.1 for details).

For each task, participants were given a high-level description for a JavaScript class. They were then tasked with writing a comprehensive API for the class using natural language in order to summarize their initial understanding of the problem. Then, they were asked to write psuedocode for a comprehensive Jest test suite for the class. We opted to have participants write pseudocode in order to minimize the confounding factor of uneven prior knowledge of JavaScript and Jest among participants (as shown in Table 1), particularly as our initial experimental results showed that participants with less JavaScript experience were unable to complete a single task if required to write executable JavaScript code.

Our study followed a between-subjects design, where participants were assigned to either have access to Copilot (P3, P4) or not have access to Copilot (P1, P2) when writing the tests. Neither group used Copilot to write the API. Finally, they were asked to iterate between modifying the API and the test suite. We particularly focused on participants' subsequent modifications to the English description of the task (i.e. the natural language API) after writing the test suite with or without Copilot. We observed the impact of Copilot on Test-First Development by examining both how writing the test suite impacts their understanding of the problem requirements via the API, and how comprehensive their test suites are, with or without using Copilot.

Finally, each participant was given a post-task survey to complete, which queried the perceived impact of writing the tests on their understanding of the problem requirements, and, in the Copilot group, the impact of using Copilot to write the test suite on their comprehension of the problem (see Section 3.3 for more details).

2.1 Task-Specific Prompts

2.1.1 API

The user prompt given for the API portion of each task is as follows:

T1. Please write an API for a class named TodoList that manages a to-do list ordered by due date. You should be able to add/remove items from the list, and modify existing items.

T2. Please write an API for a class named PrefixCalculator that implements a prefix calculator capable of performing basic arithmetic operations ($+$, $-$, $\times$, and integer division /). The calculator uses a stack-based approach to evaluate the prefix expression. For instance, `* 4 5` evaluates to 20.

T3. Please write an API for a class named URLShortener that implements the conceptual interface of a URL shortener website like is.gd. Users should be able to shorten new links, look up shortened links, list the shortened links they've made, check the number of visits to each shortened link, etc.

The API prompt for each task concludes with: "The API can be written in plain text. It should include a list of public method headers, their arguments, and a short description of what each method does."

Each participant was randomly assigned one or two of the tasks in Table 2:

ID	**Task ID Assignment**
P1 (no Copilot)	T1
P2 (no Copilot)	T3
P3 (Copilot)	T1, T2
P4 (Copilot)	T2, T3

Table 2. Randomized task assignment to each participant. Participants in the no Copilot group were initially assigned two tasks, but were only able to complete one, so we omit the tasks they did not do.

2.1.2 Test Suite

The user prompt given for the test suite portion of each task is: "Please write psuedocode for the [task_name] test suite (do not run the tests). Please make your tests as comprehensive as possible." task_name is defined as follows:

T1. task_name = ToDo List

T2. task_name = Calculator

T3. task_name = URL Shortener

In addition, basic starter code is given for each test suite, as well as a fake Jest test suite for syntax reference. Please refer to our GitHub repository[1] for details.

3 Results

3.1 Thematic Analysis of Participant Transcripts

Theme 1: Test-Driven Refinement of Problem Requirements Across Both Groups. All participants refined their understanding of problem requirements in some way as they wrote their test suites. For example, P2 (no Copilot) identified the need to add output statuses to the DeleteLink function in their API for T3 (URL Shortener) as they iteratively modified their tests and API to comprehensively encapsulate the problem requirements. Similarly, P1 (no Copilot), while revisiting their API after completing the test suite in T1 (ToDo List), realized they had overlooked a critical feature. When P1 was prompted to consider additional inputs or outputs for their tests, they added a method to retrieve all items in the ToDo list. Participants in the Copilot group also demonstrated refinement of their problem comprehension after writing their test suites. For instance, P4 (Copilot) added a parameter to their API in T3, modifying their URL lookup function after recognizing the need for a variant of the function that does not increment the lookup count. P4 articulated that this addition emerged primarily from their own reasoning, not from Copilot.

Theme 2: Functional vs Semantic API Changes Between Groups. We found that although all participants refined their understanding of the problem requirements via the test suite (i.e. Theme 1), after completing their test suite, those in the No Copilot group made additional *functional* changes to the API (i.e., altered its core functionality or added new features), while those in the Copilot group made minor *semantic* changes to the existing API (i.e., refined or adjusted the behavior of already existing functionality in the API). For instance, when prompted to revisit their API after completing

1 https://github.com/broad-well/tdd-copilot-study

the test suite for T1, when P1 (no Copilot) was prompted to think more critically about specific inputs or outputs they might want to verify, they realized "Oh wait, I did remember one thing that I forgot to add in the API", and subsequently added a new method description (i.e. additional functionality) to the API (`showItems`, i.e. get all items from the ToDo list) that their tests made them realize was missing. P1 also modified two other functions to throw exceptions (`checkNextItem`, `removeItem`) when the item did not exist in the list. In T3, P2 (no Copilot) made functional changes to their API when they realized "I should actually probably have these like output statuses [in the `DeleteLink` and `ShortenLink` functions]" as they wrote their tests. After completing their test suite, P2 then updated these functions in their API to have "success" or "failure" status output, and further refined their definition of `ListShortenedLinks`. After doing so, P2 further articulated "I feel like I messed up one of the tests" and returned to their tests to update their `shortenLink` test to reflect their modified API.

In contract, in the Copilot group, after writing the test suite for T3, P4 only made a very minor semantic change to their API after writing the test suite. They simply made their `Error` message more explicit and detailed by enumerating the possible `Error` types, and then updated the return signature of their `evaluate` function to reflect this (they did not, however, change the scope or functionality of the problem requirements). After completing the first iteration of both the API and test suite, P4 then articulated "I guess I made some changes as I was working on the tests... There could be a [another version of] lookup that doesn't increment the count, maybe, and then I might have to add some tests there." In other words, P4 merely added a variant of the existing `lookup` function to their API that had an "increment" parameter, and then relied on Copilot to autofill the test cases to reflect this semantic change.

Theme 3: Copilot Leads to Superficial Understanding and Tests. In the no Copilot group, participants demonstrated significantly deeper understanding of the problem requirements. P1 particularly exemplifies this in the No Copilot group. P1 modified their `removeItem` and `checkNextItem` functions in T1 to throw an exception if the item is not in the list. P1 subsequently modified all of his tests. They originally wrote their tests in a high-level psuedocode, but later rewrote them as formal specifications, with significantly greater detail. For instance, in the test for deleting an item from the list, P1 initially wrote the following psuedocode:

```
Description: in this test, I will remove an item from the todo list. To remove
an item, the test will ensure that the item does not exist in the todo list after
it is removed. this unit test will test todo lists that have the item and todo
lists that do not have the item.
```

After modifying the API, P1 updated this description to include the following:

```
Formal specification:
1. For all todo lists, if the deleted item is in the todo list before the delete
operation, the deleted item is not in the todo list after the delete operation.
2. For all todo lists, if the deleted item is not in the todo list before the
delete operation, there is an exception.
```

This addition starts to closely align with the level of detailed comprehension P1 would need when actually writing the implementation in a real-life TDD workflow.

In contrast, in the Copilot group, when writing a test for "should be idempotent to double-replacement" in T1, P3 simply said "this should do something," and then had Copilot generate the entire test, which they did not modify further. They also noted that if they had access to more sophisticated tools like Cursor AI, "By now, Cursor would have already got this done... it suggests if it sees me do this three times, it would just figure it out," but "this stuff is pretty boring stuff, so I don't think the Copilot's gonna mess it up." This demonstrates P3's reliance on Copilot to write the tests and assumption that Copilot will do it correctly. After completing T1, when asked if they considered that they needed to keep the list sorted by due date, P3 articulated "I did not think about that invariant," i.e. *they forgot to both write tests for this invariant and include this invariant in* *the API*, even though it was explicitly indicated in the task prompt. In contrast, P1 clearly included the due-date sort invariant in their API and in every relevant test they wrote. Finally, also in the Copilot group, P4 revealed that in T2, Copilot initially gave them "nonsensical" suggestions as Copilot didn't reference the API, but once they started writing tests, Copilot picked up on the patterns and auto-generated code for them. P4 said Copilot also frequently generated tests that were incorrect, leading them to eventually need to "read the code to make sure it was correct." Overall, P4 felt that Copilot "helped me with writing code but not really about understanding the requirements."

In addition to more superficial understanding, the test suites in the Copilot group were consistently more surface level and smaller in scope than the no Copilot group. For instance, consider T3. P2 (no Copilot) and P4 (Copilot) both included tests for checking if deleting a link returns a success code, listing all shortened links for a given URL works, looking up shortened links works, and shortening a URL returns a success code. However, P2 also included tests for checking if the number of visits to any version of a link is correct, newly shortened links are actually functional, deleted shortened links actually become nonfunctional, and that pre- and post-shortened links all go to the same destination. These are significantly more in-depth than P4's test suite. P4, however, did add additional *semantic*, rather than *functional* variations to their tests, which brings us back to **Theme 2**. For instance, P4 checks if trying to delete a nonexistant URL returns an error (this is merely a semantic variation of the generic delete function). This helps reveal that Copilot helps to detect edge cases of existing tests, but not to produce deeper, more comprehensive test suites.

3.2 Quantitative Analysis of Task Completion Time

Participant ID	Task ID	Task Completion Time (mins)
P1 (no Copilot)	T1	49.67
P2 (no Copilot)	T3	29.22
P3 (Copilot)	T1	20.03
	T2	19.07
P4 (Copilot)	T2	20.67
	T3	20.73

Table 3. Participants' time to complete each task, in minutes

Group	Average Task Completion Time	Standard Deviation
No Copilot	39.45	10.23
Copilot	20.13	0.6680

Table 4. Average and standard deviation of task completion times within the No Copilot and Copilot groups, in minutes

Overall, we observed a significant reduction in development time in the Copilot group compared to the No Copilot group. The standard deviation was quite large in the no Copilot group, but this may be due to the very small sample size (N=2). Even so, the shortest time in the No Copilot group was still approximately 9 minutes longer than the longest time in the Copilot group.

3.3 Post-Task Surveys

After completing their assigned task(s), each participant was asked

- How much did modifying the test suite contribute to your understanding of the problem requirements from 1 (least) to 10 (most)?
- If applicable, please elaborate how modifying the test suite impacted (1) your comprehension of the problem requirements and (2) how you modified the API.

In addition, the participants in the Copilot group were also asked

- How do you feel Copilot impacted (1) your ability to write the test suite and (2) your complete understanding of the problem requirements?

The quantitative ranking results from the first question are summarized in Table 5. Though P1 reported significantly lower impact of the test suite on their problem comprehension, their actions from Section 3.1 demonstrate otherwise, and this may be a result of skewed self-perception from P1, another inevitability from our very sample sample size.

ID	Impact
P1 (no Copilot)	3
P2 (no Copilot)	8
P3 (Copilot)	8
P4 (Copilot)	9

Table 5. Participants' rating of how much modifying the test suite impacted their understanding of the problem requirements, from 1 (not at all) to 10 (profoundly).

We also present a thematic analysis of participants' answers to the written survey questions.

Theme 1: Test-Driven Discovery of Problem Requirements Across Both Groups. After writing their tests, in the no Copilot group, P1 reflected that they "realized that a couple of my methods needed to throw exceptions and those exceptions should be described in the API." P2 reflected that modifying the test suite "forced me to think more deeply about how a real programmer would interact with the API." and helped them "better understand what satisfying the problem requirements would look like," as well as "helped me make modifications to the API to meet those requirements." In the Copilot group, P3 wrote that modifying the test suite "made me think about corner cases and how I could use the API." Specifically, in task T1 (ToDo List), they realized they "needed to able to differentiate two tasks that were otherwise identical." Finally, P4 reported that they "didn't realize edge cases until I worked on the test suite (e.g., parse errors, need to return count of URL lookup, need an option not to increment count, etc.)." Specifically, P4 recalled how they "changed the signature (number to `Result<number, Error>`) [for their evaluate function] to handle edge cases."

Theme 2: Copilot Leads to Superficial Understanding. P3 did not feel Copilot helped them understand the problem requirements. For instance, although P3 felt that Copilot "helped accelerate my writing of the test suite," they reported that "As far as understanding the problem requirements, I don't think copilot helped." In fact, P3 "missed testing the invariant for the ToDo task, and Copilot didn't suggest anything. I also had to manually tweak examples that Copilot generated to try and make them better tests." P3 did however did feel Copilot was good at picking up on the structure of the tests, which "allowed me to focus more on what good tests would look like, and less on the syntax of jest, and [more on] translating natural language to code," but this did not translate into P3 gaining enhanced understanding of the problem. Finally, P4 went on to report that although Copilot "helped me write test suite with fewer keystrokes" (i.e. more quickly), ultimately Copilot "didn't help me much about understanding the problem requirement."

4 Limitations

There are several limitations to our study. In particular, the tasks are limited in scope due to the time limit we imposed on each participant. A more complex task with greater time allocation would enable more realistic scenarios. P3 in particular noticed this as they worked on T1 (ToDo List): "I think that my process would probably be a lot more iterative if I were actually trying to build a ToDo list app or something you know and it was more informed by the UI." This continued as they completed T2 (Prefix Calculator), noting "you could write a full on, you know. Parse it into an AST and do a tree and stuff. But [this] seemed like this was simple enough that like this isn't even a parser really, it's just a lexer."

In addition, the differing levels of technical background among participants, as well as the small number of participants, poses limitations on the generalization of our results. We would ideally extend our study to a larger and more suitable subject pool in the future. Finally, recall that we chose to have participants write psuedocode for their Jest test suites, rather than executable JavaScript code. This decision was made due to the differing technical backgrounds as well as the time constraints, but inevitably limits the authenticity of our results. Ideally, participants would write executable test suites in a future version of this study.

5 Discussion

In summary, our analyses from Section 3 result in the following conclusions:

1. All participants refined their understanding of the problem requirements via writing the test suite.
2. Participants in the Copilot group made *semantic* changes to their existing API after writing their test suite, whereas participants in the no Copilot group made additional *functional* changes to their API after writing their test suite.
3. Participants in the Copilot group had significantly shorter development times than the no Copilot group, but demonstrated superficial understanding of the problem requirements and shallower scope of their test suite.

We arrive at these conclusions via thematic analysis of participants' live transcriptions that we sourced during the studies (Section 3.1), quantitative analysis of task completion time (Section 3.2), and written post-task surveys (Section 3.3). These findings inform us that developers using the TDD workflow may indeed benefit from using Copilot if they value coding speed above all else. However, they should be cautioned that using Copilot for their test suites will likely result in a superficial result, both in problem comprehension and test coverage, and may require more effort spent later, such as in the subsequent implementation stage, to resolve these shortcomings. In other words, Copilot can be seen as a crutch in the development process, speeding up initial progress but hindering quality. By addressing the limitations identified in Section 4, particularly pertaining to small sample size, task simplicity, and participants' use of pseudocode in our pilot study, we can further solidify these results and generalize them to broader populations of developers.

6 Future Work

In the future, we plan get IRB approval for this work and run our study on a much larger sample size. We intend to source participants with more evenly matched and significant JavaScript/Jest/Copilot backgrounds, such as from UCSD's software engineering courses. This base knowledge will enable us to ask all participants to write actual Jest code, instead of psuedocode, for their tests, which more accurately models a real-life TDD process.

We also plan to modify our tasks to have more complicated and nuanced edge cases, as we realize this may give greater insight into the impact of Copilot on participants' problem comprehension. We will also consider redesigning our tasks to more accurately reflect tasks engineers commonly face in the industry. This will also require conducting longer studies, as one hour is not long enough to accurately model a realistic software development task. Finally, we intend to supplement our current post-task survey to a semi-structured interview, which will lead to follow up questions and richer discussion. We will also modify our survey to query quantitative ranking of the impact on Copilot on the problem understanding (currently, we just ask about writing the tests in general).

7 Conclusion

Test-Driven Development is a widely studied software development process where developers write tests for desired features and subsequently make them pass [1]. Most studies on TDD in an industrial setting have reported that it improved the external quality of delivered software and reduced complexity, but some studies have reported that it reduced productivity as well [2]. With the nascent application of generative AI to software development [6], we investigate the potential for generative AI to enable developers to adopt TDD without sacrificing productivity. To do so in an environment where TDD proficiency among subjects is scarce, we relax the constraints to focus on Test-First Development (TFD), which only requires tests to be written before their corresponding implementation.

We focus on how the use of GitHub Copilot for API design and test writing affects both productivity and quality of work. We conducted a between-subjects pilot study with 4 computer science students at UC San Diego. Subjects were asked to write a comprehensive API and corresponding test suite in pseudocode for two abstract descriptions of tasks. They were randomly assigned to two groups: one encouraged them to use Copilot, while the other did not make Copilot available to them. Through thematic analysis, we discover that subjects in the No Copilot group made more profound changes to their API as they wrote their tests and that, most importantly, subjects in the Copilot group found that Copilot did not contribute to their understanding of problem requirements and led to a reduction in their test suites' scope.

In light of these preliminary findings, we warn developers against over-reliance on generative AI tools for requirement specification, API design, and test writing against an API, as they can lower the quality of produced work. If developers choose to use AI tools, they should treat generative AI output critically and remain engaged with the design process rather than trusting AI to understand their requirements. Finally, to further assess the role of generative AI in Test-Driven Development, we plan to conduct a more comprehensive study with a wider variety of participants, more realistic scenarios, and longer durations. We hope that our results can guide industry practice on upholding software quality in the era of generative AI.

References

[1] Beck, *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002, ISBN: 0321146530.

[2] H. Munir, M. Moayyed, and K. Petersen, "Considering rigor and relevance when evaluating test driven development: A systematic review," *Information and Software Technology*, vol. 56, no. 4, pp. 375–394, 2014, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.01.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914000135.

[3] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer, "Agile interaction design and test-driven development of user interfaces – a literature review," in *Agile Software Development: Current Research and Future Directions*, T. Dingsøyr, T. Dybå, and N. B. Moe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 185–201, ISBN: 978-3-642-12575-1. DOI: 10.1007/978-3-642-12575-1_9. [Online]. Available: https://doi.org/10.1007/978-3-642-12575-1_9.

[4] M. Ghafari, T. Gross, D. Fucci, and M. Felderer, "Why research on test-driven development is inconclusive?" In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '20, Bari, Italy: Association for Computing Machinery, 2020, ISBN: 9781450375801. DOI: 10.1145/3382494.3410687. [Online]. Available: https://doi.org/10.1145/3382494.3410687.

[5] K. Buffardi and S. H. Edwards, "Impacts of teaching test-driven development to novice programmers," *International Journal of Information and Computer Science*, vol. 1, no. 6, p. 9, 2012.

[6] M. Chen, J. Tworek, H. Jun, *et al.*, *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2107.03374.

[7] P. Cassieri, S. Romano, and G. Scanniello, "Generative artificial intelligence for test-driven development: Gai4-tdd," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2024, pp. 902–906.