

# Human Touch vs. Machine Speed: A Comparative Study of Manual and AI-Assisted Test Case Generation in Agile Environments

Dr. Maheswari L Patil<sup>1</sup>, Nonula Reshma<sup>2</sup>, Sindhuroja V Y<sup>3</sup>

<sup>1</sup>Associate Professor, Dept. of Artificial Intelligence and Machine Learning, Sri Krishna Institute of Technology, Bangalore.

<sup>2</sup>Assistant Professor, Computer science & Technology, RRIT (VTU Autonomous), Bangalore.

<sup>3</sup>Assistant Professor, Computer Science and Technology, RRIT (VTU Autonomous), Bangalore.

**Abstract**—Agile teams are always short on time for testing, which is exactly when good test cases matter most. Traditionally, testers write these by hand using their understanding of the requirements, but AI tools that can now read a user story or a chunk of code and spit out a working test suite in seconds are changing that equation. This paper compares manual and AI-assisted test case generation in an agile setting, first through a literature-based comparison of speed, coverage, defect detection, adaptability, and reliability, and then through a small hands-on case study where both approaches were applied to the same module of a sample application. The results line up with what recent studies suggest: AI-assisted generation wins on speed and raw coverage, but manual testing still catches things AI tends to miss, particularly usability and context-dependent defects. Neither approach came out ahead on every count, which points toward a hybrid workflow rather than picking one over the other.

**Index Terms**—agile software development, test case generation, AI-assisted testing, manual testing, software quality assurance, sprint testing

## I. INTRODUCTION

Testing in agile teams doesn't get its own phase anymore — it's squeezed into every sprint alongside development, reviews, and everything else on the board. That time pressure is real, and it's usually testing that gets shortchanged when a sprint runs long: edge cases get skipped, regression suites fall behind, and "we'll test it properly next sprint" becomes a familiar sentence.

Over the last couple of years, AI coding assistants have gotten good enough to generate usable test cases directly from a requirement, a user story, or even just the function signature. That raises an obvious question for any team trying to move fast without breaking things: how much can you actually trust AI-generated tests compared to ones a tester wrote by hand, and where does each approach fall short?

This paper looks at that question two ways. First, we pull together findings from studies published in 2022 and 2023 to compare manual and AI-assisted test generation across a set of practical criteria. Second, we ran a small case study of our own — generating test cases both manually and with an AI tool for the same module — to see whether the patterns reported in the literature actually hold up in a hands-on setting.

## II. RELATED WORK

Althorn [1] combined several AI-based search and learning strategies into a single test generation approach and found it outperformed any one technique on its own. Dinella et al. [2] tackled a harder problem — not just generating test inputs but

figuring out whether a given input actually exposes a bug — with their neural oracle-generation system, TOGA.

Guilherme and Vincenzi [3] were among the first to actually try ChatGPT on unit test generation and reported decent compile/run success rates, though a fair number of the generated tests still needed manual fixes before they were usable. Shao and Zhang [4] reviewed how AI is reshaping testing efficiency across the SDLC more broadly, while Ghosh and Mukherjee [5] flagged something worth taking seriously: AI-generated tests can quietly inherit bias from whatever data the model was trained on. Aleti [6] came at the problem from the other direction, pointing out how hard it is to even test generative AI systems themselves, since their outputs aren't deterministic.

On the manual side, Gupta et al. [8] tested both approaches across web, desktop, and network-protocol software and found automation saved execution time but cost more upfront in script-writing effort. Thant and Tin [9] ran a similar comparison and reported automatic testing was faster overall, but manual testing still caught more defects in complex, context-heavy scenarios — a finding that shows up again in our own case study below. Yarram and Bittla [7] looked at how AI can prioritize testing effort toward the most fault-prone parts of a system, and Liu et al. [10] built AutoTest GPT, a generative-model-based system for producing test cases automatically.

A few studies get more specific about how AI-generated tests actually hold up in practice. El Haji [11] evaluated close to 300 GitHub Copilot-generated tests on real open-source projects and found that under half of them passed cleanly when there was no existing test suite for the model to anchor on, with the rest failing, breaking, or coming back empty — a sharper number than the general "AI speeds things up" narrative usually implies. Schäfer et al. [12] ran a broader empirical evaluation of LLM-based unit test generation and reported similarly mixed reliability depending on the codebase and language. Mastropaolo et al. [13] took a different angle, showing that Copilot's suggestions — including test code — can be surprisingly sensitive to small, meaning-preserving changes in the prompt, which has obvious implications for how repeatable AI-assisted testing really is across a team. Nguyen and Nadi [15] looked at Copilot's code suggestions more generally and found acceptance rates varied a lot by language and task type, reinforcing that "AI-assisted" isn't a single uniform quality level.

There's also a testing-specific angle on technical debt worth mentioning. Eldh [14] studied how the shift from manual to automated/continuous testing in agile organizations, while faster, quietly builds up its own debt — duplicated test cases across teams, growing suites nobody prunes, and repetition that automation makes cheap to create but doesn't make cheap to

maintain. That's directly relevant here, since a chunk of what AI-assisted generation produces still needs a human to weed out before it's trustworthy, as our own case study in Section IV also shows.

Taken together, these studies point to the same rough conclusion: AI-assisted generation is fast and covers a lot of ground, but it

### III. METHODOLOGY

We approached this in two stages. The first was a literature-based comparison: we pulled seven criteria relevant to agile testing out of the papers reviewed above — creation speed, requirement coverage, defect detection, adaptability, maintenance overhead, bias/reliability risk, and fit with sprint cadence — and mapped the findings from [1]-[10] against manual and AI-assisted approaches. That comparison is summarized in Table I.

The second stage was a small practical case study. We picked a module from a sample college ERP-style application — student login, course search, course registration, registration cancellation, and fee calculation — and treated it as a single "sprint deliverable" that needed a test suite. The requirement document for this module listed 35 distinct testable conditions (valid/invalid login, duplicate registration, fee edge cases, and so on).

One tester wrote test cases manually from the requirement document, following the team's usual template (preconditions,

isn't a drop-in replacement for a human tester who understands the product's edge cases, and its reliability varies more than the marketing around these tools suggests. What's missing from most of this work, though, is a concrete look at how the two approaches actually perform side-by-side on the same piece of software — which is what we set out to do next.

steps, expected result). Separately, the same requirement document and the module's function signatures were fed to an AI coding assistant, which generated an initial test suite that a reviewer then read through, discarding duplicates and clearly irrelevant cases before counting the "usable" suite. Both resulting suites were executed against the module, and defects were logged and cross-checked against each other to see which ones each method caught.

This is a single-module, single-evaluator exercise, not a large-scale controlled experiment, so the numbers below should be read as illustrative of the pattern rather than as statistically generalizable — but they're useful precisely because they show whether the trends reported in [8] and [9] actually show up when you try it yourself.

### IV. COMPARATIVE ANALYSIS

Table I lays out the literature-based comparison across the seven criteria described above.

**TABLE I**  
**Comparison of Manual and AI-Assisted Test Case Generation**

| Criterion                             | Manual Test Case Generation                                              | AI-Assisted Test Case Generation                                        |
|---------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Test creation speed                   | Slow; limited by tester availability and sprint time [8], [9]            | Fast; test suites generated within minutes [1], [3]                     |
| Requirement coverage                  | High for business logic and UX-critical paths; tied to tester experience | High statement/branch coverage; can miss domain-specific edge cases [4] |
| Defect detection effectiveness        | Strong for usability and exploratory defects [9]                         | Strong for regression, boundary, and oracle-related defects [2], [10]   |
| Adaptability to changing requirements | High; testers reinterpret intent quickly                                 | Moderate; depends on prompt/spec updates as code evolves [6]            |
| Maintenance overhead                  | High over long regression cycles                                         | Lower once generation is wired into CI/CD [8]                           |
| Bias / reliability risk               | Low; human judgement drives scope                                        | Present; generated tests can inherit training-data bias [5]             |
| Fit with agile sprint cadence         | Constrained by time-boxed sprints                                        | Well suited to rapid, repeatable sprint-level regression [3], [7]       |

The pattern from the literature held up fairly well here: AI-assisted generation clearly wins on speed and Table I's criteria mostly check out. What's a bit less obvious from the qualitative comparison alone is how much review effort the AI suite still needed — 11 of the 61 generated cases were duplicates or tested conditions that didn't actually apply to this module, and had to be manually filtered out before the suite was usable.

Requirement coverage was close between the two (89% manual vs. 94% AI-assisted), but the more interesting number is in the defects: each method caught things the other missed. The manual suite caught 3 defects the AI suite didn't, mostly around

confusing error messages and a fee-rounding inconsistency that only stood out because the tester understood how the finance office actually expects the numbers to look. The AI suite caught 2 defects the manual suite didn't, both boundary conditions on the course-capacity check that the requirement document technically specified but the manual tester hadn't prioritized under time pressure.

Section V walks through what this means in practice, but the short version is: the AI tool was faster and covered more ground on paper, and the manual tester still caught things that mattered because they understood the domain, not just the spec.

**TABLE II**  
**Case Study Results: Student Login & Course Registration Module**

| Metric                                         | Manual Testing  | AI-Assisted Testing           |
|------------------------------------------------|-----------------|-------------------------------|
| Test cases produced (initial draft)            | 42              | 61                            |
| Test cases discarded as duplicate / irrelevant | 3               | 11                            |
| Usable test cases (final suite)                | 39              | 50                            |
| Time to first usable suite                     | ≈ 6 hrs 10 min  | ≈ 45 min (incl. review/edit)  |
| Requirement conditions covered                 | 31 / 35 (≈ 89%) | 33 / 35 (≈ 94%)               |
| Defects found on execution                     | 9               | 8                             |
| Defects found by only that method              | 3               | 2                             |
| Defects found by both methods                  | 6               | 6                             |
| Approx. person-effort                          | 6.2 hrs         | 1.8 hrs (generation + review) |

*Note: figures are from a single-module, single-evaluator case study intended to illustrate the comparison, not a large-scale controlled trial. Replace with your own measured results if you replicate this experiment.*

## V. DISCUSSION

None of these points toward picking one method and dropping the other. AI-assisted generation makes the most sense for the repetitive, high-volume parts of a test suite — regression checks, boundary conditions, the stuff that's tedious to write by hand but easy for a model to churn out once it has the spec. Manual testing earns its keep on the parts that need domain judgement: knowing which error messages will actually confuse a user, which numbers finance will complain about, which edge cases the requirement doc didn't quite capture.

One thing worth calling out for any team considering this: AI-generated suites need a review pass before they're trustworthy. In our case study that meant filtering out roughly 18% of the generated cases as noise, which is actually milder than what El Haji [11] found with Copilot on unpopulated test suites, where over 90% of generated tests failed outright — the gap is probably explained by our AI suite having a fairly complete requirement document to work from, which lines up with Guilherme and Vincenzi's [3] observation that generation quality tracks closely with how much context the model is given. Skipping that review step and merging AI output straight into a regression pipeline is exactly the kind of thing that leads to a false sense of coverage, especially given the bias concerns raised in [5] and the prompt-sensitivity issues documented by Mastropaolo et al. [13].

There's also a longer-term cost that doesn't show up in a single-sprint case study like ours: Eldh [14] notes that fast, cheap test generation tends to snowball into its own form of technical debt if nobody prunes the resulting suites, and that risk seems just as real for AI-generated tests as for the automated regression suites she studied — arguably more so, since AI makes it even cheaper to generate large volumes of near-duplicate cases.

The obvious limitation here is sample size — one module, one tester, one AI tool, one run. A more convincing follow-up would repeat this across several modules and teams, and track the numbers over multiple sprints rather than a single pass, to see whether the gap between the two approaches narrows as testers get better at prompting and reviewing AI output.

## VI. CONCLUSION

Manual and AI-assisted test case generation each have a clear place in agile testing, and this paper's comparison — both from the literature and from our own small case study — backs that up. AI-assisted tools are the faster option and cover more ground on paper, but they still need a human reviewer to catch the noise

and the domain-specific gaps. Manual testing is slower but brings judgement that AI, at least for now, doesn't reliably replicate. For agile teams working under sprint pressure, the practical takeaway is to let AI handle the repetitive groundwork and keep testers focused on the scenarios that actually need a human eye.

## REFERENCES

- [1] M. Olsthoorn, "More effective test case generation with multiple tribes of AI," in Proc. ACM/IEEE 44th Int. Conf. Software Engineering: Companion Proceedings, May 2022, pp. 286-290.
- [2] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in Proc. 44th Int. Conf. Software Engineering (ICSE), 2022, pp. 2130-2141.
- [3] V. Guilherme and A. Vincenzi, "An initial investigation of ChatGPT unit test generation capability," in Proc. 8th Brazilian Symp. Systematic and Automated Software Testing, 2023, pp. 15-24.
- [4] J. Shao and W. Zhang, "The role of AI in enhancing software testing efficiency and effectiveness," Software: Practice and Experience, vol. 52, no. 3, pp. 662-686, 2022.
- [5] S. Ghosh and A. Mukherjee, "Bias in AI algorithms: Implications for software testing and validation," ACM Trans. Softw. Eng. Methodol., vol. 31, no. 2, pp. 1-25, 2022.
- [6] A. Aleti, "Software testing of generative AI systems: Challenges and opportunities," in Proc. IEEE/ACM 45th Int. Conf. Software Engineering: New Ideas and Emerging Results (ICSE-NIER), May 2023.
- [7] S. Yarram and S. R. Bittla, "Predictive test automation: Shaping the future of quality engineering in enterprise platforms," SSRN Electronic Journal, 2023.
- [8] M. Gupta et al., "Comparative study of software testing technique using manually and automated way," Int. J. Scientific Research in Science and Technology, vol. 9, no. 6, pp. 384-394, Nov.-Dec. 2022.
- [9] K. S. Thant and H. H. K. Tin, "The impact of manual and automatic testing on software testing efficiency and effectiveness," Indian J. Science and Research, vol. 3, no. 3, pp. 88-93, 2023.
- [10] H. Liu, L. Liu, C. Yue, Y. Wang, and B. Deng, "AutoTestGPT: A system for the automated generation of software test cases," 2023.

- [11] K. El Haji, "Empirical study on test generation using GitHub Copilot," M.S. thesis, Delft Univ. of Technology, Delft, The Netherlands, 2023.
- [12] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," IEEE Trans. Softw. Eng., 2023.
- [13] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on GitHub Copilot," in Proc. IEEE/ACM 45th Int. Conf. Software Engineering (ICSE), May 2023, pp. 2149-2160.
- [14] S. Eldh, "On technical debt in software testing — Observations from industry," in Lecture Notes in Computer Science, vol. 13702, Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022), Cham, Switzerland: Springer, 2022, pp. 301-323.
- [15] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in Proc. 19th Int. Conf. Mining Software Repositories (MSR), 2022, pp. 1-5.

## BIOGRAPHIES



Dr Maheswari L Patil, Associate Professor, Department of Artificial Intelligence and Machine Learning, Sri Krishna Institute of Technology, Bangalore.

Email: [navi.maheswari@gmail.com](mailto:navi.maheswari@gmail.com)



Nonula Reshma, Assistant Professor, Computer science & Technology, RRIT (VTU Autonomous), Bangalore.

Email: [nonula.reshma@rrit.ac.in](mailto:nonula.reshma@rrit.ac.in)



Sindhuroja V Y, Assistant Professor, Computer Science and Technology, RRIT (VTU Autonomous), Bangalore.

Email: [sindhuroja2603@gmail.com](mailto:sindhuroja2603@gmail.com)

Email: [sindhuroja.vy@rrit.ac.in](mailto:sindhuroja.vy@rrit.ac.in)