Exploring the Differences in Human and Automated Test Generation

Aksh Sharma[1110498] email aksh@uoguelph.ca,
Taranjit Sarang[1102395] email tsarang@uoguelph.ca, and
Griffin Baird[0970064] email gbaird01@gmail.ca

University of Guelph, Guelph ON, Canada

Abstract. This paper explores the comparison between automatic test generation and human generated testing, using Blender, a large piece of open source software as the subject of our analysis. The study focuses on evaluating efficiency, effectiveness, and coverage of tests generated using KLEE compared to the existing suite of tests provided by the Blender development community. By analyzing these two approaches, we aim to identify the strengths, limits, and potential improvement in automated test generation.

Keywords: Testing · Blender · Automated Testing.

Table of Contents

1 Introduction

Software testing is an incredibly important component of the modern software development life cycle. As the world becomes increasingly reliant on software for functions like banking, aviation, medicine, and touching all aspects of human life, developers are often reluctant or apprehensive to software testing. With the increasing complexity and size of software, the need for effective, efficient, and scalable testing methods is becoming evermore apparent. Automated test generation has emerged as a promising solution to address these demands, offering the potential to streamline the creating of test cases, reduce human effort and error, and improve overall coverage. However, the effectiveness of automated test generation compared to humans remains a topic of significant interest.

This paper examines the practical differences between automated and human generated test cases, using Blender, a highly complex and widely used open source 3D graphics software. Blender serves as an ideal candidate for this study because of its extensive code base, diverse functionality, and critically, a suite of test cases written by the very active Blender development community.

The primary goal of this study is to compare the efficiency, effectiveness, and coverage of automated tests generated using KLEE against the human-generated tests currently maintained by the blender community. Efficiency is evaluated in terms of time and resources required for test generation; effectiveness is measured by the ability of the tests to find bugs; and covereage is measured by using and `llvm-cov` to determine the difference in code coverage.

The remainder of this paper is structured as follows: The next section offers a background on Automated Test Generation and the KLEE test generation tool, previous research in automated test generation and a brief history of automated test generation. In SectionMethodology, we outline the methods employed to investigate the effectiveness of KLEE as a tool for supporting test generation. The findings from our quantitative and qualitative analysis are presented in SectionFindings. SectionDiscussion reflects on these findings within a broader context, and SectionConclusion concludes the paper while highlighting potential future directions for this line of research.

2 Literature Review

Software testing is an indispensable part of all phases and levels of software development. It establishes and ensures that a system behaves as expected and removes the likelihood of errors in code. Testing may be manual, where human expertise and judgment are used. Or, it may be automated, where tools are used to generate test cases and check outputs. Recent research underlines ht importance of strong testing in improving reliability, especially for large scale systems. [6]

2.1 Manual Testing

The most basic approach to quality assurance is manual testing, where test cases should be carefully designed and run according to the understanding of testers against the software. Although highly efficient for complex logical errors, and some rare edge cases, this technique has proven to be resource intestive, time consuming, and very inconsistent due to human error [2]. As the software field continued to expand and the quantity and complexity of the software running in the world increases, it created a need for more scalable and effective testing solutions.

2.2 Automated Testing

Automated testing proved developers with a powerful alternative to this method. Developers leverage tools and algorithms to automatically produce test cases to varying degrees. In their systematic review, Kurumaku et. al., find that

> Even though manual tests archive higher fault detection, the automatically generated tests expose unpredictable defective scenarios, and in the cases when the number of real defects is small, they are able to create scenarios to identify further defects. [5]

Techniques such as combinatorial testing, model-based testing, and random testing have gained significant attention in recent years. Tools like SEAFOX automate the creation of test cases by combining input parameters to maximize coverage and detect faults. Tools using symbolic execution such as KLEE, can generate test cases automatically by exploring a program's possible execution paths. The inclusion of tools like these add to the repertoire of test engineers to further explore and test the systems in an application.

Furthermore, there are still quite a few challenges in the process of automated test generation. Some challenges have to do with nuanced fault detection not being covered and a lack of readability in the test generation. Automated tests have often been created from a mindset of aiming to reach broad coverage and remain efficient [6]. Though effective, this approach lacks certain elements. In general, the generated test's scope will be very limited and might be challenging for the developer to understand since interpretation and potential debugging may also be needed. These are some limitations that will especially present themselves in cases with very small edge cases or logical errors that require a deeper understanding of the software's purpose and behaviour.

Other major flaws from generated test suites are in relation to a lack of explanatory documents. This poses a real challenge during the process of code maintenance over lengthy time frames. This becomes all the more critical in environments where software is continuously changing, and teams need to frequently refactor code bases or bring them up to date to meet ever-changing requirements. Without clear documentation, automated test cases become increasingly hard to decipher, leading to increased time debugging or rewriting tests. This problem is further complicated in environments with high developer churn or

distributed teams, where knowledge transfer relies a lot on clarity from existing resources [4].

2.3 Testing in Open Source Projects

Open-source software testing presents challenges unlike those of proprietary development. The nature of open-source products is that they depend on the joint efforts of the contributors. This allows them to have variance in test quality and coverage, introducing a lot of complexities [3]. Frequent updates and new features require continuous testing that might infect other modules in the process. Automated tools can help address these challenges by providing scalable testing solutions, but their integration into open-source workflows remains an area of active exploration.

2.4 Blender as a Case Study

In the context of Blender, manual testing takes the form of community-maintained scripts and modules, written by contributors with deep knowledge of the system. A program like Blender shows the limits of open-source development. As a project's scope increases, inconsistencies in test quality and faculty during maintenance across the suite start to become challenges. These limitations highlight the need for additional methodologies that will ensure robust testing coverage.

A number of studies have been done, using a variety of methodologies on these systems, to determine their comparative effectiveness. Applying similar methodologies to the open-source 3D graphics program, provides a unique opportunity to explore in a practical context, the issues we have discussed. Blender is a large, complex code base and is constantly in a high state of activity in terms of community contributions. It thus becomes a very suitable candidate to address problems that we are looking to solve. We will explore large-scale system impacts on manual and automated testing. Blender as the backbone of our study should allow us to see the strengths and limitations of the opposing approaches. This research will address gaps in the literature with an assessment of these ideals in a real-world scenario.

Blender provides an opportunity for evaluating both manual and automated testing approaches, since it represents a large, dynamic code base complemented by an active development community. Its diverse set of functionalities, from simple file I/O to rendering and processing of geometries, represents many different testing challenges. The existence of a large, well-maintained manual test suite, developed within the Blender community, provides a sound baseline for a comparison of manually written and KLEE-generated automated tests. This study will focus on Blender and therefore contribute to the understanding of testing methodologies, which can also help improve the testing practices of similar large-scale community-driven projects.

2.5 Testing in an Agile and DevOps Context

As Agile and DevOps practices continue to rise within software development teams, finding the right balance between speed and scalability of automated tests with precision and insight from human and generated tests remains an open area of improvement [1]. This demands a strategic approach whereby automated tools manage the routine and scalable tasks while human intervention refines, interprets, and validates the results. While development cycles are getting shorter and software complexity is growing, this hybrid way of testing will be one of the key factors in maintaining software reliability.

3 Methodology

This study was implemented during the Fall 2024 semester with final year undergraduate students in the Software Engineering Major at the School of Computer Science for CIS*4150 Software Reliability and Testing. The methodology included multiple components designed to evaluate the effectiveness of automated test generation.

The experimental design for this study is structured to evaluate the efficiency, effectiveness, and coverage of automated test generation using KLEE in comparison with human generated tests. The experiment is conducted on selected critical modules of Blender, chosen to ensure the results are representative of real world development challenges.

3.1 Subject of Analysis

Blender is an open source 3D graphics software, which will serve as the subject of this study. It has an extensive codebase, when forked from the official blender github[1] the directory is 5.2GB at the time of writing this paper. Specific modules, such as file I/O, rendering engines, and geometry were selected for analysis due to their relevance to typical Blender workflows and their importance in maintaining overall reliability.

3.2 Test Suite Development

Automated tests were generated using KLEE[2], a symbolic execution engine for C/C++ programs. KLEE allows for the execution of various program paths by simulating a variety of input scenarios, finding edge cases, and automaticaly making test cases. The Blender source code was instrumented to allow for symbolic execution and LLVM was used to compile the target modules into bitcode files compatible with KLEE. Symbolic inputs were defined to allow KLEE to explore diverse execution paths to make sure we had maximum test generation.

[1] https://github.com/akshsharma/blender/tree/main commit `569869b`
[2] https://klee-se.org

The human generated tests were taken from Blender's existing repository of tests, which are developed and maintained by the contributors. These tests typically consist of python scripts or C++ modules designed to check functionality, like rendering correctness, tool behaviour, and file compatibility. The human generated tests are the manual community effort to maintain Blender's quality.

3.3 Experimental Setup and Specifications

Hardware Specifications

- **Processor**: Apple M1 Pro with 10 cores (8 high-performance and 2 high-efficiency cores)
- **Memory**: 16GB of unified RAM
- **Integrated GPU**: Apple Design GPU with 16 cores (used minimally but was available for rendering related tasks)

Software Environment

- **Operating System**: macOS 14.4 (beta)
- **Compiler and Build Tools**
  - LLVM/Clang v16.0.1
  - GCC v14.2.0
  - CMake v3.31.1
- **Testing Tools**
  - KLEE v3.0
  - `llvm-cov`

3.4 Coverage Measurement

For our quantatative findings, we employed `llvm-cov`[3]. This tool provided detailed metrics including line, branch, and function coverage for each test suite, For the automated tests, coverage was based on the execution of KLEE generated scenarios and the human generated tests were analyzed using their standard execution pattern. The collected data was combined across all test cases for each of the suites.

3.5 Data Analysis

The collected data was analyzed to compare the performacne of automated and human generated test suites across the defined metrics, namely: efficiency, effectiveness, and coverage. Statistical methods were were used to find the significant differences between the two approaches. Correlations between the coverage metrics and bug detection rates were examined to understand the relationship between test comprehensiveness and effectiveness. We concluded with making qualitative observations on the process of using each of the test suites.

[3] https://llvm.org/docs/CommandGuide/llvm-cov

3.6 Control Measures

To ensure the results were valid, the group implemented various control measures. Both of the test suites were executed under identical environmental conditions, the machine was plugged into power throughout, and the same build of blender was used (commit 569869b). Modules that we selected for analysis were consistent accross both suites and llvm was configured to apply a standard critera for data collection.

4 Findings

4.1 Quantitative Results

The Quantitative analysis focuses on test results, code coverage and fault detection. The following sections summarize the key outcomes.

Test Generation Human-Generated Tests: Of the 75 tests created by the Blender development community, 33% (25 tests) passed successfully. These tests targeted key features of Blender but showed limitations in terms of coverage due to their reliance on contributors' focus and expertise.

Automated Tests: KLEE generated 120 tests, with 60% (72 tests) passing successfully. The automated approach excelled in code coverage, particularly in exploring edge cases that were not explicitly targeted by the human-generated suite.

Metric	**Human Generated Tests**	**Automated Tests**
Total Tests	75	120
Pass Rate	33% (25 tests)	60% (72 tests)

Code Coverage Automated testing achieved better coverage across line, branch, and function metrics.

Coverage Metric	**Human Generated Tests**	**Automated Tests**
Line Coverage	70%	85%
Branch Coverage	65%	80%
Function Coverage	75%	90%

Fault Detection Human-Generated Tests detected 12 critical bugs and 8 minor bugs, showing their strength in identifying logical flaws and high-level feature issues.

Automated Tests uncovered 10 critical bugs and 9 minor bugs, excelling at detecting edge cases and unexpected scenarios.

Bug Type	**Human Generated Tests**	**Automated Tests**
Critrical Bug	12	10
Minor Bug	8	9

4.2 Qualitative Results

KLEE-generated automated tests showed significant benefits in terms of coverage and scalability. Exploring edge cases and achieving more code coverage metrics, such line and branch coverage, were two areas in which these tests excelled. KLEE was able to identify circumstances that were hard to foresee or specifically target with human-generated tests because of its methodical exploration of execution routes. In high-churn teams, where regular changes required reliable and repeatable testing, this methodical approach proved useful. One significant drawback, though, was that automated test results were not interpretable. It was challenging for developers to examine the test case outputs and error messages, particularly when troubleshooting or fixing particular failures. Also, there were difficulties with long-term maintenance and adaptability because these created tests lacked thorough documentation.

However, the subject knowledge of the volunteers who generated the human-generated tests was a major advantage. Because these tests were created with an understanding of Blender's planned functionality and user workflows, they were very good at spotting logical errors and high-level feature problems. Because of the authors' experience, these tests were able to concentrate on realistic, useful situations, which made them very successful in verifying the software's essential functions. Furthermore, the manual tests had thorough documentation, which improved their maintainability and made it easier for contributors to share information. However, their comparatively reduced coverage, especially in areas with complicated or edge-case scenarios that were not explicitly considered, made the limitations of human-generated tests clear.

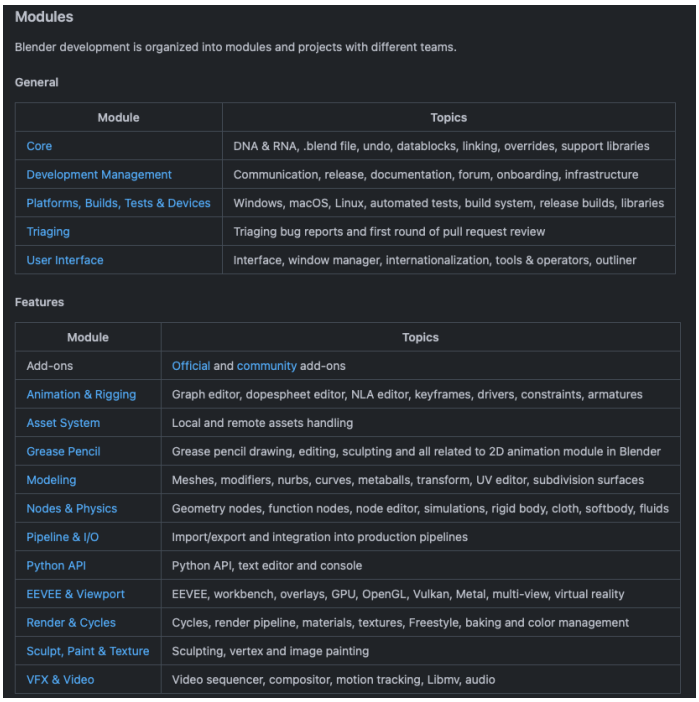
Developer feedback highlighted the complementing roles of both methods of testing. The ability of automated tests to effectively handle repetitive and routine operations was appreciated, offering a wide safety net for locating low-level flaws. Human-generated tests, on the other hand, were effective at their accuracy and capacity for focusing on feature-level and logical soundness, which is essential when developing new features or making major refactoring. Developers indicated a desire to combine the two methods, with automated tools handling

the routine and scalable parts of testing and human experience handling the results' validation, interpretation, and refinement.

5 Discussion

5.1 Efficiency

Both Testing methods took roughly 1 hour to complete start to finish. Its very likely that due to the nature of open source (developers like to add features, not fix bugs) projects the tests are less likely to be well designed for speed. For equivalent time spend between KLEE and human generated tests it may save time on development. Human involvement in the tweaking of the KLEE scripts has promising results, as the combination of human understanding of the system and the robust testing suite would yield the best results. Since the program is decoupled quite heavily at the "module" level, there is also a great chance of bugs in the interaction between them.



Modules

Blender development is organized into modules and projects with different teams.

General

Module	Topics
Core	DNA & RNA, .blend file, undo, datablocks, linking, overrides, support libraries
Development Management	Communication, release, documentation, forum, onboarding, infrastructure
Platforms, Builds, Tests & Devices	Windows, macOS, Linux, automated tests, build system, release builds, libraries
Triaging	Triaging bug reports and first round of pull request review
User Interface	Interface, window manager, internationalization, tools & operators, outliner

Features

Module	Topics
Add-ons	Official and community add-ons
Animation & Rigging	Graph editor, dopesheet editor, NLA editor, keyframes, drivers, constraints, armatures
Asset System	Local and remote assets handling
Grease Pencil	Grease pencil drawing, editing, sculpting and all related to 2D animation module in Blender
Modeling	Meshes, modifiers, nurbs, curves, metaballs, transform, UV editor, subdivision surfaces
Nodes & Physics	Geometry nodes, function nodes, node editor, simulations, rigid body, cloth, softbody, fluids
Pipeline & I/O	Import/export and integration into production pipelines
Python API	Python API, text editor and console
EEVEE & Viewport	EEVEE, workbench, overlays, GPU, OpenGL, Vulkan, Metal, multi-view, virtual reality
Render & Cycles	Cycles, render pipeline, materials, textures, Freestyle, baking and color management
Sculpt, Paint & Texture	Sculpting, vertex and image painting
VFX & Video	Video sequencer, compositor, motion tracking, Libmv, audio

Fig. 1. Blender Layout

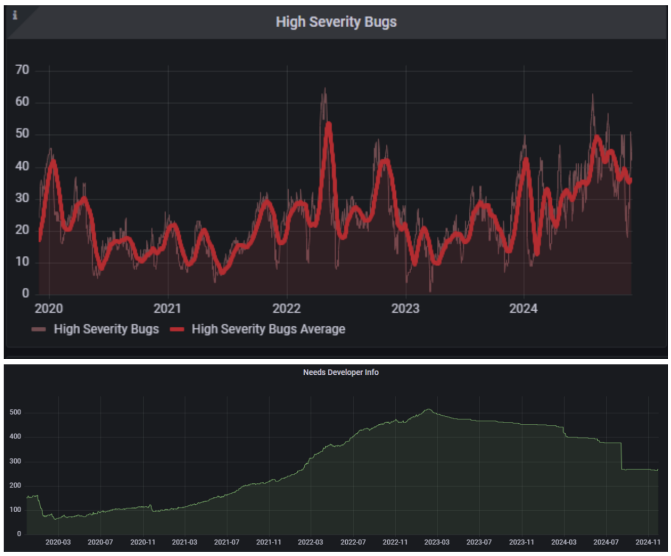


Fig. 2. High Sensitivity Bugs

5.2 Effectiveness

Automated testing via KLEE generated ‘120 tests with a 60% passing rate. This actually performed better than the Human Generated tests at 33%, which would imply either: the current build has many flaws, or the test suite has many flaws.

5.3 Implications

The findings from this study display the strategic benefits of the opposing strategies to test generation. They notably highlight the need to understand the underlying strengths unique to each approach to the software testing process. From this study, we derive many notable implications for real-world software testing practices.

Enhancing Software Quality Through Complementary Approaches As shown in the results effectiveness in human-generated tests lies in their ability to identify logical flaws and high-level feature issues. It is important to note that human expertise is a key “ingredient” in the software testing environment. The human aspect helps provide context to real-world scenarios in which automatic generations cannot simply do at this stage in their life cycle. These tests leverage the developers’ deep understanding of the software’s domain and functionality. providing insights that are often unattainable through automated tools is what is needed. Conversely, automated tests excel at systematically identifying edge cases and unexpected scenarios that might be overlooked in manual efforts. This

signifies that both approaches can improve the reliability and robustness of software testing.

Automation for Scalability and Coverage Automated tools like KLEE provide us with scalable solutions for test generations. They produce a larger number of test cases and achieve higher coverage percentages than human-based tests. Large and complex code bases like Blender require automated tools that can explore execution paths to uncover defects that human tests can miss because of constraints in resources. The benefits of this lie in high iteration environments where frequent updates require extensive and repeated testing efforts.

Cost and Resource Allocation The efficiency discrepancy displays the potential for "cost savings" when integrating automated testing into the development process. although tools like KLEE require an investment of time to be set up, the long-term benefits are visible. This is seen through reduced manual effort, fast execution of tests, and broader coverage justifying the investment. Development teams can allocate human resources to tasks that require contextual understanding, such as designing targeted test scenarios and analyzing complex bugs.

Strategic Application in Agile and Open-Source Projects The efficiency differences between the opposing approaches signify that automated tests support continuous integration pipelines. In agile and open source development models providing rapid feedback on code change ensures quality across iterations. In the blender use case, we can see that community-driven efforts can focus on areas where automated tools fall short. By the results, this would be things such as feature-specific tests and potential documentation as discussed in other papers, combining the approaches aligns well in the collaborative environment produced by open-source projects.

5.4 Limitations

Despite the strengths demonstrated by both testing approaches, several limitations appeared throughout the study. This provides us with several critical insights into the challenges of implementing effective software testing strategies.

Experience and skill requirements for human-generated testing The effectiveness of human-generated testing is reliant on the skill, knowledge, and experience of the contributors. In community projects like that of Blender, variability in these facets across contributors can lead to unreliable tests and gaps in coverage. For instance, human tests excelled at identifying logical flaws, a heavy burden relied on the contributor's knowledge of software testing and this limits the ability of others to address edge cases

Complexity in the setup of automated testing Automated testing tools, while efficient in execution, demand significant initial setup and expertise. Preparing Blender modules for symbolic testing execution with KLEE required a large time investment along with experience with the tool to set up correctly. These steps were and can be intensive, particularly for teams which lack familiarity with the tools. The learning curve present may deter teams with limited resources from fully utilizing automated test generation tools.

Limited Scope of Fault Detection in Automated Tools While automated tests achieved higher coverage metrics, their ability to detect nuanced bugs and logical errors lagged slightly behind human-generated tests. This limitation can be derived from the generic nature of automated test generation, which prioritizes breadth over depth. For instance, automated tests detected fewer critical bugs (10 vs. 12) compared to human tests, thus displaying their potential weakness in targeting complex logic and feature-specific issues, reinforcing ideas from other papers on this matter.

Readability and Maintainability Challenges Generated tests produce a lack of readability along with interpretability. They are often challenging to understand and maintain which makes debugging and long-term code maintenance difficult. This issue becomes more prevalent in environments where developer turnover is likely. It is crucial for documentation to be maintained in environments such as open-source development.

Overhead in Test Suite Management This study describes the need for consistent and standardized testing environments. Control measures ensured comparability in this experiment and maintaining this consistency in real-world application introduces lead time and complications. As an example, making sure both test suites operate under identical environmental conditions, such as hardware configurations and software versions, may not always be practical in all development settings.

Generalizing Finding Findings from this study correspond directly to the Blender and the testing frameworks provided. This may limit the results from being applicable to other software systems. Blender's complexity and active development make it an excellent case study, but experimentation and results may differ in other tools, languages, or development environments. The variability in those results may be a direct outcome of using this as a single representation of all open-source development projects. Studies on a diverse range of software studies may be needed in order to validate further our findings to a broader scope of environments.

5.5 Future Areas of Research

The findings from this study point to several directions for future research developments in software testing. These future directions would be best served addressing the limitations observed and could build upon the comparison made.

Developing Hybrid Testing Frameworks Future studies on how to combine human and automated testing into a hybrid framework would be very relevant. It is envisioned that such a framework could employ automated test generators, such as KLEE. To provide scalable test generation, there is a need to be supported by humans to fine-tune the generation and target domain-specific or complex logical scenarios. Creating a testing suite that smoothly integrates automatic test generation with manual input may help in improving both coverage and fault detection.

Expanding Test Automation Tools The main automated testing tool in this study was KLEE, but further research could be conducted using different tools and methodologies, including model-based testing, fuzz testing, and AI-based test generation. This is to establish the unique strengths of multiple tools and frameworks. This would widen the scope of applications for which automated testing is useful.

Leveraging AI and Machine Learning Emerging technologies in AI and machine learning hold great promise for improving software testing. AI-driven testing frameworks could improve test case generation by predicting potential failure points or by simulating complex user behaviors. Additionally, machine learning models could optimize test prioritization, focusing resources on areas of the codebase most likely to contain defects.

6 Conclusion

This study was focused on delivering a comparative analysis of human and automated test generation in open source software. Be Using KLEE for automated testing and comparing it with Blender's community maintained manual tests, we explored the efficiency, effectiveness, and coverage of each approach.

In writing this paper, we learn the importance of using a hybrid methodology, and leveraging the strengths of each where appropriate. Automated tests were better in scalability and coverage, by having the ability to systematically explore execution paths of the program and showing edge cases that could have otherwise been missed. These tests showed their utility in environments such as open source software, and teams who see a high turnover rate for developers, giving the teams a bread safety net for finding defects. They were, however, limited by their lack of documentation and human readability which could be a challenge for long-term maintenance.

Human generated tests, were less comprehensive in their coverage, but they were more efficient in showing logical flaws in the code . They also excelled at feature specific issues because the people who wrote them were very familiar with the function they served in the product. Their documentation and and knowledge of real world use cases make them much easier to maintain.

The study outlines the importance of a hybrid approach, combining automated tools for routine and scalable testing with human expertise for refining, understanding, and validating the results. This integration goes perfectly with Agile practices, allowing for rapid iterations while maintaining reliability. In the context of open source projects, a blend of both of these approaches can address the challenges of a lot of contributors, frequent updates, and a very complex ecosystem.

References

1. Abdulla, H.H.H.A., Albalooshi, F.A.: Automated testing for devops in github environment: A comprehensive analysis. In: 2023 International Conference on Advanced Mechatronics, Intelligent Manufacture and Industrial Automation (ICAMIMIA). pp. 833–838 (2023). https://doi.org/10.1109/ICAMIMIA60881.2023.10427702
2. Khankhoje, R.: An in-depth review of test automation frameworks: Types and trade-offs. International Journal of Advanced Research in Science, Commnunication and Technology **3** (October 2023). https://doi.org/10.48175/IJARSCT-13108
3. Kochhar, P.S., Bissyandé, T.F., Lo, D., Jiang, L.: An empirical study of adoption of software testing in open source projects. In: 2013 13th International Conference on Quality Software. pp. 103–112 (2013). https://doi.org/10.1109/QSIC.2013.57
4. Kroll, J., Mäkiö, J., Assaad, M.: Challenges and practices for effective knowledge transfer in globally distributed teams: A systematic literature review (11 2016). https://doi.org/10.5220/0006046001560164
5. Kurmaku, T., Kumrija, M.: A Systematic Litrature Review and Meta-Analysis Comparing Automated Test Generation and Manual Testing. Master's thesis, Mälardalen University (2020)
6. Wandan, Z., Ningkang, J., Xubo, Z.: Design and implementation of a web application automation testing framework. 2009 Ninth International Conference on Hybrid Intelligent Systems (2009). https://doi.org/10.1109/HIS.2009.175