Comparative Empirical Evaluation of Software Test Automation using Artificial Intelligence and Manual Test Case Design

Jesús José Bone Caicedo, José Luis Carvajal Carvajal, and Victor Xavier Quiñonez Ku

***Abstract*—This study presents an empirical comparison between manually designed test cases and those automatically generated by generative artificial intelligence tools—ChatGPT and Diffblue Cover—applied to the Spring PetClinic system, developed in Java and Spring Boot. The fully automated experiment comprised 2,480 runs distributed across 12 test classes (6 human and 6 AI-generated), with 40 iterations per class and a total duration of 6.18 hours. Four main metrics—instruction coverage, branch coverage, mutation score, and execution time—were evaluated using descriptive and inferential analysis (Student's t, Welch, and Mann-Whitney U). At the aggregate level (N = 12), no significant differences were found ($p > 0.05$; $d < 0.30$), while at the complete level (N = 2,480), small effects ($r < 0.15$) were observed in all metrics, indicating marginal differences between the two approaches. The results confirm that generative AI can achieve quantitative performance comparable to that of human testing, albeit with more limited functional reasoning. This work provides empirical evidence on the current capabilities and limitations of AI in automated testing, highlighting its potential to accelerate repetitive tasks and improve productivity without replacing human analytical judgment.**

***Index Terms*—automated testing, Diffblue Cover, generative AI, software quality, test case generation**

I. Introduction

In the context of digital transformation and the growing incorporation of artificial intelligence in all stages of the software life cycle, test automation has taken center stage in agile development processes due to its potential to increase coverage, reduce time, and improve product reliability. This technical advance not only responds to a need for efficiency in engineering teams, but also to social demand for more reliable, secure, and highly available digital services, whose quality has a direct impact on sensitive sectors such as health, education, and finance. The emergence of generative AI—with large language models (LLMs) and associated tools—has renewed this field by promising faster and assisted test case generation, albeit with mixed and context-dependent results [1], [2], [3], [4]. In ecosystems dominated by Java and Spring Boot, it is pertinent to subject these solutions to rigorous evaluation to establish their true scope and limitations compared to traditional approaches [1], [2].

Despite recent advances (e.g., improvements in LLM-assisted unit testing in industrial contexts) [5], the literature shows that the landscape is still fragmented. Research such as that by Schäfer et al. [6] and Yang et al. [7] has demonstrated the ability of LLMs to generate automatic unit tests with promising results, but without establishing direct comparisons with human tests. Complementarily, Bhatia et al. [8] empirically evaluated the performance of ChatGPT against traditional tools such as Pynguin, showing comparable or superior coverage, but also a significant proportion of incorrect assertions and dependence on human supervision. Similarly, Kanth et al. [9] contrast conventional methodologies and AI-based approaches, focusing their analysis on efficiency and coverage, although without strict control over a common system under test (SUT). For their part, Kirinuki and Tanno [10] evaluated the interaction between ChatGPT and human testers in black-box testing, highlighting qualitative differences in reasoning and understanding of the system. However, recent systematic and grey literature reviews, such as that by Ricca et al. [11], agree that there is still a lack of empirical evidence combining standardized quantitative metrics—such as mutation score or coverage—with functional observations in the same experimental environment. While recent works have explored advanced metric-driven approaches, such as mutation-guided test generation [12], these efforts remain largely focused on quantitative effectiveness rather than functional interpretation.

In this context, an empirical gap persists: there is a lack of systematic and quantitative comparisons between AI-generated tests and manually designed tests under the same SUT and replicable evaluation framework [2]. Although the literature reports benefits—faster generation and automated exploration—it also warns of limitations associated with variable quality, false positives, and dependence on human supervision [13], [14], [15]. This combination of advances and gaps justifies the need for controlled studies that contrast the actual effectiveness of both approaches under equivalent conditions.

In addition to traditional quantitative indicators, this study incorporates a complementary observation related to functional understanding, understood in empirical terms as the differences observed in the way the compared approaches—generative AI and manual design—address the logic and functionality of the system under test. This dimension does not seek a formal cognitive analysis, but rather to contextualize the numerical results (coverage, mutation score, efficiency) based on how each approach interprets and structures the test cases [2], [4], [14], [15]. Its inclusion enriches the reading of the findings, offering a more complete perspective on the performance and functional quality of the automated testing

J. J. Bone Caicedo, J. L. Carvajal Carvajal, and V. X. Quiñonez Ku are with the Systems and Software Department, Pontifical Catholic University of Ecuador Esmeraldas Campus, Esmeraldas Ecuador (e-mail: jjbone@pucese.edu.ec; jose.carvajal@pucese.edu.ec; xavier.quinonez@pucese.edu.ec).

process.

This study aims to evaluate, in a controlled environment, the effectiveness of test cases automatically generated by generative AI compared to those manually created in Java projects, using a representative Java/Spring Boot project as the system under test (SUT). The overall objective of the study is to evaluate the effectiveness and functional understanding, from an empirical approach, of AI-generated test cases compared to those designed by humans, combining quantitative metrics—coverage (instruction/branch), mutation score, and operational efficiency—with empirical observations on their ability to adequately represent the functionality of the system [16].

The expected contribution integrates comparative evidence with replicable metrics and an analytical component that considers the empirical perspective of functional understanding, offering useful inputs for decision-making in educational and professional settings [1], [5]. In line with this technical and social relevance, the research adopts a comparative experimental approach, based on objective metrics and empirical observations, which allow for the examination of both the quantitative performance and the interpretive quality of AI-generated tests compared to manual tests in a controlled environment. Finally, the article is structured as follows: section 2 describes the materials and methods; section 3 presents the results and discussion; and section 4 develops the conclusions and future projections.

II. Materials and Methods

A. Research Design

The study was developed under a mixed-method empirical-comparative design, aimed at evaluating the effectiveness and functional understanding of test cases generated by generative artificial intelligence versus those developed manually. An applied approach was adopted, focused on measuring the actual performance of automatic verification techniques in a software quality assurance context. The experiment was conducted between May and December 2025 in a controlled academic environment, using the open-source Spring PetClinic project as the system under test (SUT) due to its modular structure, stability, and widespread use in empirical research [2]. Two test suites were generated from this SUT: one using generative AI tools (ChatGPT Plus and Diffblue Cover CLI) and another manually created by the researcher. Both were evaluated under equivalent conditions using standardized quantitative metrics (code coverage, mutation score, and execution time) and a qualitative analysis aimed at understanding the functional aspects of the system.

The methodological design followed contemporary guidelines for empirical studies with language models, which recommend documenting versions, configurations, prompts, and interactions to ensure transparency, reproducibility, and traceability [17]. Recent guidelines on the role of LLMs in software engineering research were also incorporated, emphasizing the importance of maintaining human oversight, explicitly declaring the use of AI-based tools, and mitigating the methodological risks associated with their use [18]. These principles made it possible to control external variables, strengthen the internal validity of the study, and ensure that the results were verifiable and replicable through the use of professional tools in controlled and documented versions.

B. Materials and Experimental Environment

The study was conducted entirely on the Spring PetClinic system (Java 17, Spring Boot 3.5.6, 3.5.0-SNAPSHOT), selected as the sole system under test (SUT) due to its adoption in recent research, its modular architecture, and its compatibility with automation frameworks in Java [2]. Since the research did not involve human subjects, the researcher's intervention was limited to the design and technical execution of manual tests, while the artifacts analyzed corresponded exclusively to test cases generated by artificial intelligence and manually designed tests under controlled conditions. The manual tests were developed using functional exploration with Postman and coded in JUnit 5 and MockMvc, while the AI-generated cases were obtained using ChatGPT (GPT-5 model) and Diffblue Cover CLI (Teams Edition), the latter executed without human intervention to ensure neutrality in bytecode-based generation. Additionally, automation scripts in PowerShell were used to sequentially and reproducibly execute the functional, unit, and combined test suites, avoiding manual intervention during execution.

Quantitative metrics were collected using JaCoCo 0.8.12 (instruction and branch coverage), PITest 1.16.0 (mutation score), and Maven Surefire 3.2.5 (total execution time), with automatic export of results in XML, CSV, and HTML formats. Python 3.12.6 was used for statistical analysis with the pandas, numpy, scipy, matplotlib, seaborn, and openpyxl libraries, allowing the calculation of descriptive measures, normality and variance contrasts, hypothesis tests (Shapiro-Wilk, Levene, Student's t-test, Mann-Whitney U) and effect sizes (Cohen's d and r). All runs were performed on a computer with an Intel Core i7-12700KF processor, 32 GB DDR4 RAM, and 1 TB NVMe SSD, running Windows 11 Pro 23H2, without parallel loads and with fixed random seeds to ensure determinism and reproducibility, in accordance with recent guidelines for AI-assisted empirical studies in software engineering [11], [17].

The experimental sample consisted of test suites produced using both approaches: functional and unit tests generated by AI (ChatGPT and Diffblue Cover) and equivalent tests designed manually on the same SUT components. This configuration allowed for an objective comparison of coverage, mutation, and operational efficiency metrics, complemented by qualitative observations on the functional consistency of the cases. All configurations, parameters, versions, and resulting artifacts were documented in a supplementary technical record to facilitate future replication of the experiment.

C. Experimental Protocol

The experimental protocol was developed in five consecutive phases, with the aim of ensuring methodological consistency, variable control, and reproducibility. In the planning phase, an initial literature review was conducted on artificial intelligence-assisted test automation techniques, and the study's comparison criteria were defined based on the selected metrics: instruction coverage, branch coverage, mutation score, and execution time. At the same time, the system under test (Spring PetClinic) was selected as the base environment for the empirical evaluation of manual and AI-generated approaches.

During the preparation phase of the experimental environment, the SUT was configured and the necessary measurement instruments for collecting metrics (PITest and Surefire) were installed, while JaCoCo was already part of the original project configuration. Subsequently, test cases were generated for the three approaches analyzed: (i) tests generated by ChatGPT GPT-5 using neutral prompts and providing only the source code of the method; (ii) unit tests automatically generated by Diffblue Cover CLI, without human intervention; and (iii) manual tests designed by the researcher with JUnit 5 and MockMvc, derived from the inspection of the source code and the behavior observed during the previous functional exploration with Postman.

The automated execution phase of the experiment was implemented using three PowerShell scripts, whose internal names were kept unchanged, supplemented with a brief description for clarity. The run_pitest_isolated_complete.ps1 script (responsible for executing all unit tests for both approaches) performed 40 consecutive iterations, recording the metrics from JaCoCo, PITest, and Surefire in each one. Similarly, the run-test-metrics.ps1 script (responsible for functional testing) ran 40 additional iterations under the same measurement conditions. Finally, the run_all_metrics_simple.ps1 script acted as an orchestrator, coordinating the sequential execution of the two previous scripts and recording the overall time of the process. In all cases, three preliminary runs (warm-up runs) were included to stabilize the JVM and mitigate the variability associated with the initial startup.

During the recording and organization of results phase, the outputs generated by each iteration—including XML files, CSV files, and HTML reports—were structured into separate directories by test type (functional or unit) and by approach (AI or manual), maintaining consistent names and fixed paths to ensure traceability. No human intervention was allowed during automated execution, and execution parameters, seeds, paths, and configurations were kept constant in order to minimize experimental noise and reinforce the internal validity of the study.

In the final phase of statistical analysis, the collected data were processed in Python using the pandas, numpy, scipy, seaborn, and matplotlib libraries. Descriptive measures (mean, median, standard deviation) were calculated, and statistical assumptions were verified using the Shapiro-Wilk (normality) and Levene (homogeneity of variances) tests. Depending on the fulfillment of these assumptions, parametric (Student's t-test or Welch's correction) or nonparametric (Mann-Whitney U) significance tests were applied. Effect sizes (Cohen's d and r) were also calculated, and graphical representations were generated to support the results obtained and facilitate their comparative interpretation.

The complete flow of the experimental protocol is presented in Fig. 1.

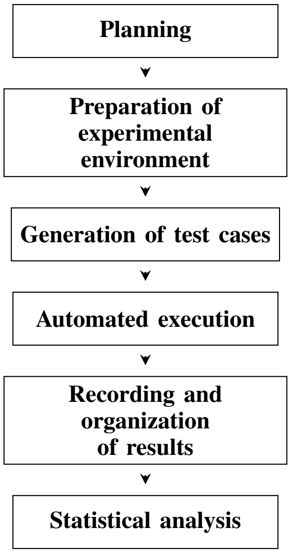
Planning
Preparation of experimental environment
Generation of test cases
Automated execution
Recording and organization of results
Statistical analysis

Fig. 1. Workflow representing the experimental protocol followed during the study.

D. Variables and Metrics

The study defined a set of independent and dependent variables aimed at comparing the behavior of test cases generated using artificial intelligence with those designed manually. The independent variable (IV) corresponded to the test generation approach, established with two levels: (i) manual generation and (ii) generation assisted by generative AI. The dependent variables (DV) were the results obtained from the quantitative and qualitative metrics used to evaluate the performance of both approaches, according to criteria identified in recent evaluations of AI-assisted testing tools [2], [11].

Among the quantitative variables, three main metrics were considered. Code coverage was evaluated using the indicators instruction coverage and branch coverage, in order to measure the percentage of instructions and paths executed by each test suite. The robustness of the test suite was measured using the mutation score, a widely used indicator for estimating the ability of tests to identify intentional modifications in the source code [2]. Finally, operational efficiency was defined as the total execution time of each test suite under controlled conditions, allowing for a comparison of the time load associated with each approach.

In addition to these quantitative metrics, a qualitative variable called functional comprehension was incorporated, related to the degree of consistency with which each approach represents the internal logic of the system under test. This dimension made it possible to contrast the structural and semantic adequacy of the test cases generated, taking into account recent findings on inconsistencies, omissions, or functional deviations observed in tests generated using language models [10], [13]. The conceptual structure linking the independent variable with the dependent variables, both quantitative and qualitative, is summarized in Fig. 2.

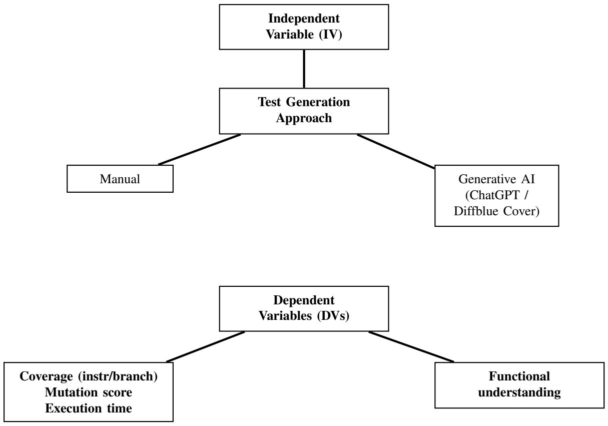
Independent Variable (IV)
Test Generation Approach
Manual
Generative AI (ChatGPT / Diffblue Cover)
Dependent Variables (DVs)
Coverage (instr/branch) Mutation score Execution time
Functional understanding

Fig. 2. Conceptual diagram showing the relationships between the independent variable (test generation approach) and the dependent variables (quantitative and qualitative metrics).

E. Data Analysis Methods

The data was analyzed using a quantitative and inferential approach, aimed at comparing the behavior of test cases generated by artificial intelligence with those designed manually. The metrics obtained—instruction coverage, branch coverage, mutation score, and execution time—were processed and organized into comparative tables, accompanied by statistical visualizations such as bar charts, box plots, and violin plots. These analyses were developed using Python and the pandas, numpy, matplotlib, and seaborn libraries.

First, descriptive statistics were applied to calculate measures of central tendency (mean and median) and dispersion (standard deviation) for each metric evaluated. Subsequently, an inferential analysis was performed to identify statistically significant differences between the test generation approaches. To this end, the assumption of normality was verified using the Shapiro-Wilk test and the homogeneity of variances using the Levene test.

The selection of statistical tests depended on the fulfillment of these assumptions. When the metrics met normality and presented homogeneous variances, the classic Student's t-test for independent samples was applied. In cases where normality existed but variances were not homogeneous, Welch's t-test was used. When the metrics did not meet the normality assumption in any of the groups, the nonparametric Mann-Whitney U test was used. In all cases, a significance value of $p < 0.05$ was adopted, accompanied by 95% confidence intervals.

Finally, effect sizes were calculated to complement the statistical interpretation. Cohen's d was used in comparisons made with Student's t-test, and the r measure was used for comparisons based on the Mann-Whitney U test. The combination of descriptive statistics, parametric and nonparametric tests, and effect size calculations ensured a robust, reproducible analysis consistent with best methodological practices in empirical software engineering studies [2], [11], [15]. The decision flowchart used to select the appropriate statistical test is shown in Fig. 3.

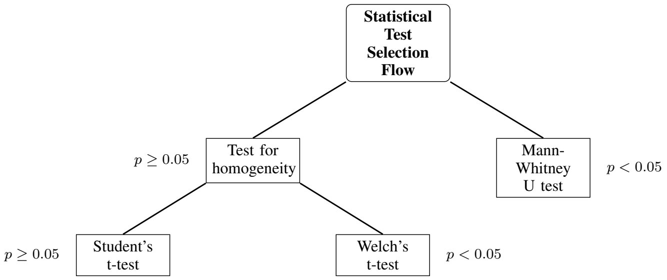

Fig. 3. Decision flowchart for selecting the appropriate statistical test based on normality and variance homogeneity assumptions.

F. Validity and Reliability Control

Internal validity was ensured by conducting the experiment in a controlled environment, keeping hardware, software, and system configuration conditions constant. The measurement tools—JaCoCo, PITest, and Surefire—operated autonomously with respect to the origin of the test cases, which made it possible to avoid instrumental biases and ensure a partially blind evaluation. The source code of the system under test was not modified, and the original tests of the project were excluded, preserving the experimental integrity. To reinforce construct validity, widely accepted metrics in software engineering—instruction coverage, branch coverage, and mutation score—were used, identified as consistent indicators for evaluating the quality and effectiveness of AI-assisted testing in recent studies [2], [11], [15], and consistent with contemporary methodological guidelines for empirical research with generative models [17].

In terms of reliability, controls were implemented to minimize researcher influence. The prompts used in ChatGPT were formulated in a neutral manner, without examples or guidelines, and the tests generated by Diffblue Cover and those designed manually were developed completely independently, avoiding code transfers or adaptations between approaches. Errors or failures observed during execution were recorded without subsequent corrections, ensuring traceability and consistency. External validity was supported by the use of Spring PetClinic, a representative and widely adopted system in empirical evaluations of testing tools for Java/Spring Boot environments, facilitating replication of the study and reinforcing the comparability of the results obtained.

G. Reproducibility and Ethics

The research was conducted under principles of transparency and traceability, ensuring that the experiment could be replicated in its entirety. All scripts, configurations, execution parameters, and intermediate results were documented and stored in a repository derived from the Spring PetClinic project, maintained under version control and structured to allow complete reproduction of the experimental flow. The system under test was used in accordance with its open source license (Apache 2.0) and without modifications to its source code, ensuring the consistency of the environment and the stability of the evaluation conditions. Likewise, the tools used—including ChatGPT and Diffblue Cover—were executed in accordance with their respective terms of use, avoiding any manipulation or intentional bias in the results.

From an ethical perspective, the study did not involve human participants or the processing of personal data, and therefore did not require informed consent procedures or review by an ethics committee. However, a strict commitment to academic integrity, transparent communication of findings, and the absence of selective exclusion of data was maintained. Generative artificial intelligence was used with neutral prompts and without interventions intended to influence the results, so that the test sets produced reflected the genuine behavior of each tool.

To ensure complete reproducibility of the study, the repository used during the experiment—containing all scripts, configurations, and results—is publicly available at: https://github.com/solveighty/spring-petclinic-test/

III. Results

A. Experiment Execution and Dataset Structure

The experimental execution was carried out using a fully automated process, designed to ensure consistency and eliminate any human intervention during data collection. Forty iterations were performed for each of the 12 test classes considered in the study—six designed manually and six generated using artificial intelligence tools (ChatGPT and Diffblue Cover)—yielding a total of 480 base measurement cycles. The entire execution process took 6.18 hours, considering the valid iterations and preliminary executions used to stabilize the JVM. Each cycle generated two separate records, corresponding to the compilation and execution phases, resulting in a final set of 2,480 raw records. For each record, four metrics were automatically stored: instruction coverage, branch coverage, mutation score, and execution time, extracted directly from the XML, CSV, and HTML artifacts generated by JaCoCo, PITest, and Surefire. Table I provides a summary of the experimental dataset structure.

TABLE I
Summary of the experimental dataset collected during the study.

Component	Cant.	Details
Total test classes	12	6 Manual + 6 AI-Generated
Iterations per class	40	Includes warm-up (3 initial iterations)
Total raw records	2,480	1,600 manuals and 880 AI
Metrics per record	4	instruction%, branch%, mutation%, time(s)

During the dataset integrity check, it was confirmed that the 2,480 expected measurements were generated correctly, with no missing records, duplicates, or format inconsistencies. The identification labels for each approach (manual and AI) remained consistent across all iterations, and the exported files retained the expected structure for statistical processing. Mutation score values equal to 0 were observed in some cases; however, these correspond to valid runs in which the test sets failed to kill any mutations generated by PITest. All values were retained in their entirety due to their analytical relevance, ensuring the completeness, consistency, and reliability of the dataset used in the following stages of the study.

B. Preparing the Dataset

The raw records generated during the experimental run—originally stored in twelve CSV files corresponding to the manual and AI-generated test classes—were consolidated into a single master file, following the procedure described in the experimental protocol. This integration was performed using automated Python scripts (pandas), which included format normalization, column name standardization, and data type verification, fully preserving the metrics collected in each iteration and ensuring the structural consistency of the dataset.

For the subsequent stages of the analysis, two levels of aggregation were defined. Table II illustrates the structure and distribution of records at both aggregation levels.

TABLE II
Total records for each test team.

Analysis level	Manual	AI	Description
N = 2,480	1,600	880	Individual observations per iteration
N = 12	6	6	Independent averages by test class

The complete level (N = 2,480) retained all individual records, allowing the intrinsic variability of the measurements to be captured. The aggregated level (N = 12) was obtained by averaging the 40 iterations associated with each test class, generating a balanced and statistically independent set at the class level.

C. Descriptive statistics of the metrics

Descriptive statistics were used to characterize the overall behavior of the metrics evaluated for both approaches (manual and artificial intelligence), both at the aggregate level (N = 12) and at the complete level of individual records (N = 2,480). For each metric, mean, median, and standard deviation values were calculated to describe the central tendency and variability associated with each test group.

Table III presents the descriptive measures at the aggregate level, where each value represents the average of 40 iterations per test class.

TABLE III
Average of 40 iterations per test class (Manual N = 6, AI N = 6).

Metrics	Group	Mean	Median	St Dev.
Instr Coverage	Manual	18.25	15.95	12.50
Instr Coverage	AI	17.67	16.13	11.36
Branch Coverage	Manual	14.58	11.88	12.62
Branch Coverage	AI	12.05	9.89	9.49
Mutation Score	Manual	18.52	15.28	17.71
Mutation Score	AI	14.76	15.11	11.61
Time Execution	Manual	0.082	0.071	0.069
Time Execution	AI	0.194	0.144	0.193

Table IV illustrates the descriptive measures at the complete level, reflecting the raw distribution across all 2,480 individual records collected during the experiment.

TABLE IV
INDIVIDUAL RECORDS PER ITERATION (MANUAL N = 1600, AI N = 880).

Metrics	Group	Mean	Median	St Dev.
Instr Coverage	Manual	19.94	21.85	12.11
Instr Coverage	AI	14.20	10.69	8.71
Branch Coverage	Manual	17.69	12.50	12.39
Branch Coverage	AI	13.50	16.25	6.75
Mutation Score	Manual	22.92	16.67	17.19
Mutation Score	AI	16.63	19.44	8.16
Time Execution	Manual	0.079	0.012	0.178
Time Execution	AI	0.114	0.003	0.232

The differences between means and medians observed in some metrics indicate that the data do not follow a completely symmetrical distribution. This behavior is reflected at both levels of analysis and is reported for consideration in subsequent statistical stages.

D. Verification of statistical assumptions

Statistical assumptions were validated in order to determine the relevance of applying parametric or nonparametric tests in inferential analyses. Two fundamental conditions were evaluated: normality of distributions and homogeneity of variances, both considered standard requirements for the use of Student's t-test and its variants.

1) Normality (Shapiro-Wilk): The normality of the metrics was assessed using the Shapiro-Wilk test, applied both at the complete level of individual records (N = 2,480) and at the aggregate level by test class (N = 12). At the complete level, none of the distributions evaluated had p-values greater than 0.05, indicating an absence of normality. In contrast, at the aggregate level, all metrics showed p-values greater than 0.05 for both approaches, fulfilling the assumption of normality.

Table V presents the Shapiro-Wilk test results at the complete level (N = 2,480), showing normality assessment for all metrics across both manual and AI approaches.

TABLE V
SHAPIRO-WILK NORMALITY TEST (MANUAL N = 1600, AI N = 880; FULL LEVEL, N = 2,480).

Metrics	Group	W	p-value	Normality
Instr Coverage	Manual	0.8160	2.34e-39	No
Instr Coverage	AI	0.7524	2.66e-34	No
Branch Coverage	Manual	0.8178	3.21e-39	No
Branch Coverage	AI	0.8697	1.96e-26	No
Mutation Score	Manual	0.8404	2.46e-37	No
Mutation Score	AI	0.8313	1.82e-29	No
Time Execution	Manual	0.4797	8.07e-56	No
Time Execution	AI	0.5354	4.67e-43	No

Table VI shows the Shapiro-Wilk test results at the aggregate level (N = 12), demonstrating that all metrics meet the normality assumption at this level of analysis.

TABLE VI
SHAPIRO-WILK NORMALITY TEST (MANUAL N = 6, AI N = 6; AGGREGATE LEVEL, N = 12).

Metrics	Group	W	p-value	Normality
Instr Coverage	Manual	0.9087	0.4281	Yes
Instr Coverage	AI	0.9038	0.3971	Yes
Branch Coverage	Manual	0.8807	0.2722	Yes
Branch Coverage	AI	0.9127	0.4541	Yes
Mutation Score	Manual	0.9132	0.4581	Yes
Mutation Score	AI	0.9720	0.9056	Yes
Time Execution	Manual	0.9296	0.5771	Yes
Time Execution	AI	0.8507	0.1596	Yes

These results indicate that only the aggregate level meets the normality assumption, while the complete level requires the use of nonparametric tests as it does not conform to normal distributions.

2) Homogeneity of variances (Levene): The homogeneity of variances between the manual and AI groups was evaluated using Levene's test. In three of the four metrics analyzed—instruction coverage, branch coverage, and mutation score—values of $p > 0.05$ were observed, indicating equality of variances between groups. In contrast, execution time presented $p < 0.05$, which shows heterogeneity of variances and requires the application of Welch's correction in the parametric analysis.

Table VII displays the results of Levene's test for homogeneity of variances at the aggregate level (N = 12).

TABLE VII
LEVENE'S TEST FOR HOMOGENEITY OF VARIANCES (N = 12).

Metrics	F. Stat	p-value	Homogeneity
Instr Coverage	0.1006	0.7577	Yes
Branch Coverage	0.1262	0.7298	Yes
Mutation Score	0.3930	0.5448	Yes
Time Execution	6.3679	0.0302	No

E. Parametric analysis (Student's t and Welch)

Parametric contrasts were applied only at the aggregate level (N = 12), given that this level met the assumption of normality and, for three of the metrics, the assumption of homogeneity of variances. For metrics with homogeneous variances, the standard Student's t-test was used, while for the Time (s) metric, which did not meet homogeneity, the Welch correction variant was applied.

Table VIII presents the results of the parametric tests (Student's t and Welch) applied at the aggregate level.

TABLE VIII
RESULTS OF PARAMETRIC TESTS (T-STUDENT AND WELCH).

Metrics	p-value	Cohen's d	Sig.
Instr Coverage	0.9350	0.048	No
Branch Coverage	0.7022	0.227	No
Mutation Score	0.6727	0.251	No
Time Execution	0.2293	-0.769	No

Figure 4 associated with parametric contrasts presents box plots illustrating the distribution of each metric for the two

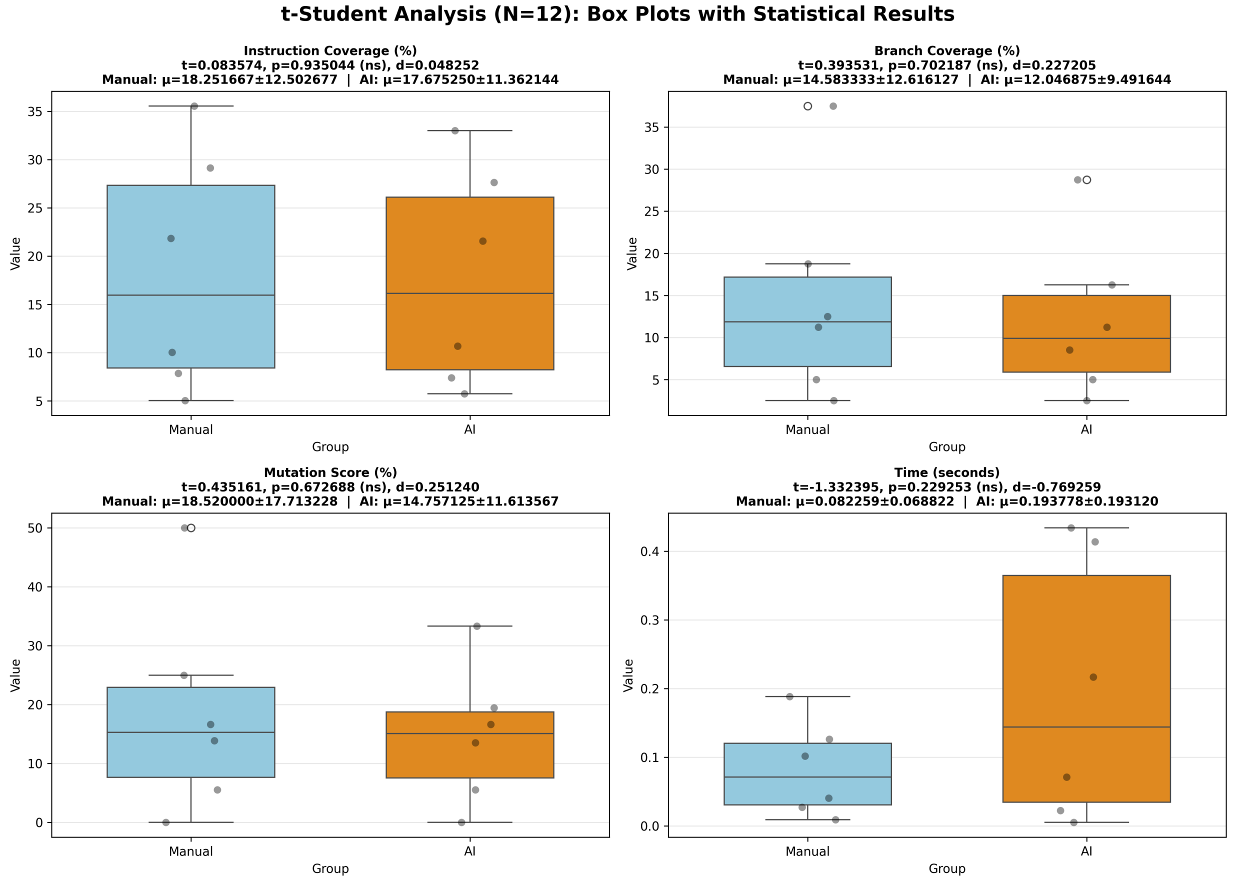

Fig. 4. Box plots from parametric analysis (t-Student and Welch) showing the distribution of metrics for both test approaches (Manual vs. AI) at the aggregated level (N = 12).

approaches evaluated. Its purpose is solely to provide a descriptive visualization of the central behavior and dispersion of the aggregate values (N = 12). This representation allows us to observe the overlap between groups, but it does not constitute additional evidence beyond the reported inferential results.

F. Non-parametric analysis (Mann-Whitney U)

Since the entire dataset (N = 2,480) did not meet the normality assumption, the nonparametric Mann-Whitney U test was applied to compare the distributions between the two approaches (manual and AI) for each metric. The contrasts were performed using the individual values per iteration, with the p-value and effect size r.

Table IX presents the results of the Mann-Whitney U test applied at the complete level (N = 2,480).

TABLE IX
RESULTS OF NON-PARAMETRIC TESTS (MANN-WHITNEY U).

Metrics	p-value	r	Sig.
Instr Coverage	3.20e-13	0.145	Yes
Branch Coverage	5.73e-10	0.123	Yes
Mutation Score	3.82e-03	0.057	Yes
Time Execution	3.74e-05	0.082	Yes

Figure 5 corresponding to the nonparametric analysis includes violin plots showing the complete shape of the distribution for each metric in both approaches (N = 2,480). This visualization is used solely for descriptive purposes to illustrate the density, variability, and possible asymmetries in the data. No additional analytical conclusions are drawn from these figures, as the inferential evidence is based solely on the results of the Mann-Whitney U test.

In summary, the results obtained show that, although both approaches—manual and AI-driven—produced comparable metrics across several indicators, subtle differences can be observed in both the distribution and consistency of the values. These variations suggest that automatic generation methods achieve similar levels of performance to manual methods in quantitative terms, but with different behavior patterns, which forms the basis for the interpretive analysis developed in the following section.

IV. How to Create Common Front Matter

The following sections describe general coding for these common elements. Computer Society publications and Conferences may have their own special variations and will be noted below.

A. Paper Title

The title of your paper is coded as:

```
\title{The Title of Your Paper}
```

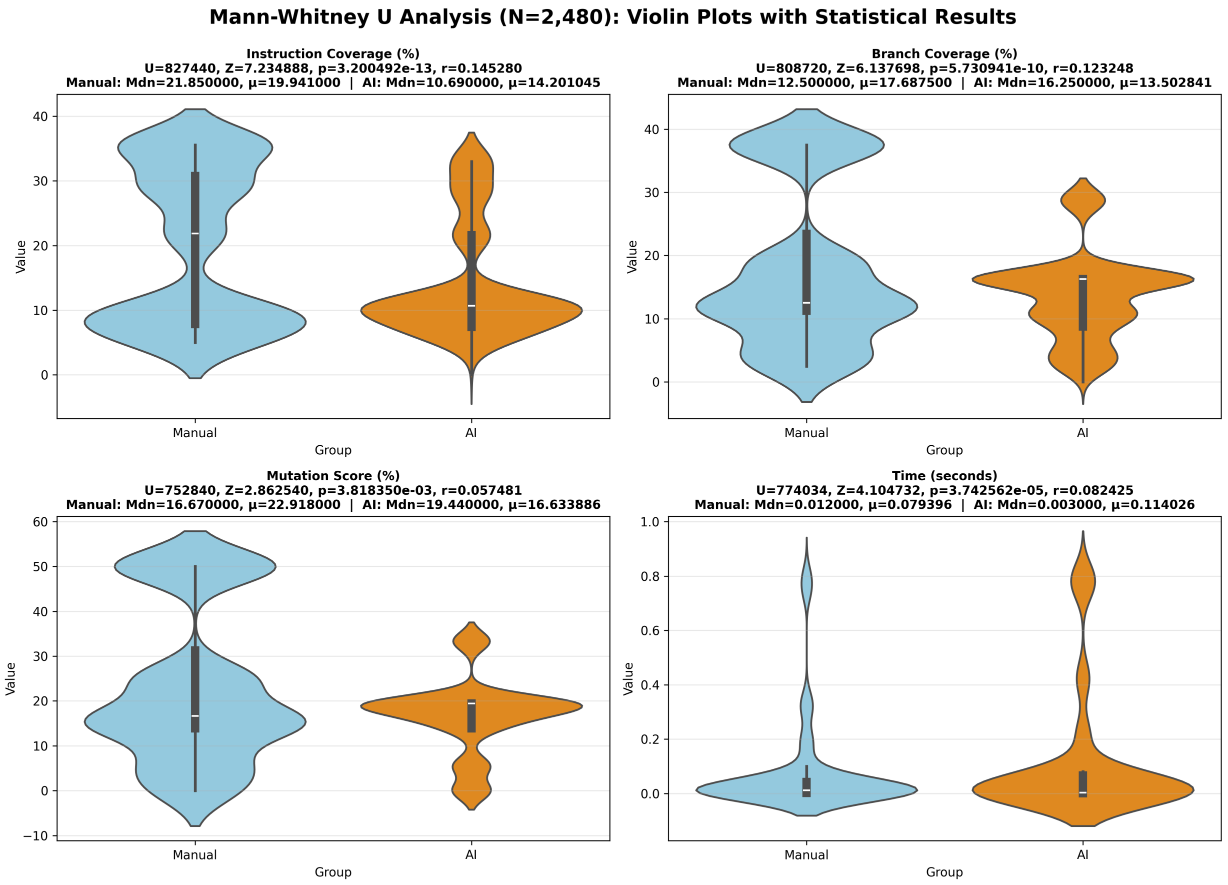

Fig. 5. Violin plots from non-parametric analysis (Mann-Whitney U) showing the distribution shape and density for each metric across both test approaches (Manual vs. AI) at the full individual level (N = 2,480).

Please try to avoid the use of math or chemical formulas in your title if possible.

B. Author Names and Affiliations

The author section should be coded as follows:

```
\author{Masahito Hayashi
\IEEEmembership{Fellow, IEEE}, Masaki Owari
\thanks{M. Hayashi is with Graduate School
of Mathematics, Nagoya University, Nagoya,
Japan}
\thanks{M. Owari is with the Faculty of
Informatics, Shizuoka University,
Hamamatsu, Shizuoka, Japan.}
}
```

Be sure to use the \IEEEmembership command to identify IEEE membership status. Please see the “IEEEtran_HOWTO.pdf” for specific information on coding authors for Conferences and Computer Society publications. Note that the closing curly brace for the author group comes at the end of the thanks group. This will prevent you from creating a blank first page.

C. Running Heads

The running heads are declared by using the `\markboth` command. There are two arguments to this command: the first contains the journal name information and the second contains the author names and paper title.

```
\markboth{Journal of Quantum Electronics,
Vol. 1, No. 1, January 2021}
{Author1, Author2,
\MakeLowercase{\textit{(et al.)}:
Paper Title}
```

D. Copyright Line

For Transactions and Journals papers, this is not necessary to use at the submission stage of your paper. The IEEE production process will add the appropriate copyright line. If you are writing a conference paper, please see the “IEEEtran_HOWTO.pdf” for specific information on how to code ”Publication ID Marks”.

E. Abstracts

The abstract is the first element of a paper after the `\maketitle` macro is invoked. The coding is simply:

```
\begin{abstract}
Text of your abstract.
\end{abstract}
```

Please try to avoid mathematical and chemical formulas in the abstract.

F. Index Terms

The index terms are used to help other researchers discover your paper. Each society may have it's own keyword set. Contact the EIC of your intended publication for this list.

```
\begin{IEEEkeywords}
Broad band networks, quality of service
\end{IEEEkeywords}
```

V. How to Create Common Body Elements

The following sections describe common body text elements and how to code them.

A. Initial Drop Cap Letter

The first text paragraph uses a "drop cap" followed by the first word in ALL CAPS. This is accomplished by using the `\IEEEPARstart` command as follows:

```
\IEEEPARstart{T}{his} is the first paragraph
of your paper. . .
```

B. Sections and Subsections

Section headings use standard LaTeX commands: `\section`, `\subsection` and `\subsubsection`. Numbering is handled automatically for you and varies according to type of publication. It is common to not indent the first paragraph following a section head by using `\noindent` as follows:

```
\section{Section Head}
\noindent The text of your paragraph . . .
```

C. Citations to the Bibliography

The coding for the citations are made with the LaTeX `\cite` command. This will produce individual bracketed reference numbers in the IEEE style. At the top of your LaTeX file you should include:

```
\usepackage{cite}
```

For a single citation code as follows:

```
see \cite{ams}
```

This will display as: see [**?**]

For multiple citations code as follows:

```
\cite{ams,oxford,lacomp}
```

This will display as [**?**], [**?**], [**?**]

D. Figures

Figures are coded with the standard LaTeX commands as follows:

```
\begin{figure}[!t]
\centering
\includegraphics[width=2.5in]{fig1}
\caption{This is the caption for one fig.}
\label{fig1}
```

Fig. 6. This is the caption for one fig.

```
\end{figure}
```

The [!t] argument enables floats to the top of the page to follow IEEE style. Make sure you include:

```
\usepackage{graphicx}
```

at the top of your LaTeXfile with the other package declarations.

To cross-reference your figures in the text use the following code example:

```
See figure \ref{fig1} ...
```

This will produce:
See figure 6 . . .

E. Tables

Tables should be coded with the standard LaTeX coding. The following example shows a simple table.

```
\begin{table}
\begin{center}
\caption{Filter design equations  ...}
\label{tab1}
\begin{tabular}{| c | c | c |}
\hline
Order & Arbitrary coefficients &
coefficients\\
of filter & $e_m$ &   $b_{ij}$ \\
\hline
1& $b_{ij}=\hat{e}.\hat{\beta_{ij}}$,
& $b_{00}=0$\\
\hline
2&$\beta_{22}=(~1,-1,-1,~~1,~~1,~~1)$ &\\
\hline
3& $b_{ij}=\hat{e}.\hat{\beta_{ij}}$,
& $b_{00}=0$,\\
\hline
\end{tabular}
\end{center}
\end{table}
```

To reference the table in the text, code as follows:

TABLE X
A SIMPLE TABLE EXAMPLE.

Order of filter	Arbitrary coefficients e_m	coefficients b_{ij}
1	$b_{ij} = \hat{e}.\hat{\beta_{ij}},$	$b_{00} = 0$
2	$\beta_{22} = (\ 1, -1, -1,\ \ 1,\ \ 1,\ \ 1)$	
3	$b_{ij} = \hat{e}.\hat{\beta_{ij}},$	$b_{00} = 0,$

```
Table~\ref{tab1} lists the closed-form...
```

to produce:

Table X lists the closed-form . . .

F. Lists

In this section, we will consider three types of lists: simple unnumbered, numbered and bulleted. There have been numerous options added to IEEEtran to enhance the creation of lists. If your lists are more complex than those shown below, please refer to the “IEEEtran_HOWTO.pdf” for additional options.

A plain unnumbered list

bare_jrnl.tex
bare_conf.tex
bare_jrnl_compsoc.tex
bare_conf_compsoc.tex
bare_jrnl_comsoc.tex

coded as:

```
\begin{list}{}{}
\item{bare\_jrnl.tex}
\item{bare\_conf.tex}
\item{bare\_jrnl\_compsoc.tex}
\item{bare\_conf\_compsoc.tex}
\item{bare\_jrnl\_comsoc.tex}
\end{list}
```

A simple numbered list

1) bare_jrnl.tex
2) bare_conf.tex
3) bare_jrnl_compsoc.tex
4) bare_conf_compsoc.tex
5) bare_jrnl_comsoc.tex

coded as:

```
\begin{enumerate}
\item{bare\_jrnl.tex}
\item{bare\_conf.tex}
\item{bare\_jrnl\_compsoc.tex}
\item{bare\_conf\_compsoc.tex}
\item{bare\_jrnl\_comsoc.tex}
\end{enumerate}
```

A simple bulleted list

- bare_jrnl.tex
- bare_conf.tex
- bare_jrnl_compsoc.tex
- bare_conf_compsoc.tex
- bare_jrnl_comsoc.tex

coded as:

```
\begin{itemize}
\item{bare\_jrnl.tex}
\item{bare\_conf.tex}
\item{bare\_jrnl\_compsoc.tex}
\item{bare\_conf\_compsoc.tex}
\item{bare\_jrnl\_comsoc.tex}
\end{itemize}
```

G. Other Elements

For other less common elements such as Algorithms, Theorems and Proofs, and Floating Structures such as page-wide tables, figures or equations, please refer to the “IEEEtran_HOWTO.pdf” section on “Double Column Floats.”

VI. How to Create Common Back Matter Elements

The following sections demonstrate common back matter elements such as Acknowledgments, Bibliographies, Appendicies and Author Biographies.

A. Acknowledgments

This should be a simple paragraph before the bibliography to thank those individuals and institutions who have supported your work on this article.

```
\section{Acknowledgments}
\noindent Text describing those who
supported your paper.
```

B. Bibliographies

References Simplified: A simple way of composing references is to use the `\bibitem` macro to define the beginning of a reference as in the following examples:

[6] H. Sira-Ramirez. “On the sliding mode control of nonlinear systems,” *Systems & Control Letters*, vol. 19, pp. 303–312, 1992.
coded as:

```
\bibitem{Sira3}
H. Sira-Ramirez. ``On the sliding mode
control of nonlinear systems,''
\textit{Systems \& Control Letters},
vol. 19, pp. 303--312, 1992.
```

[7] A. Levant.“Exact differentiation of signals with unbounded higher derivatives,” in *Proceedings of the 45th IEEE Conference on Decision and Control*, San Diego, California, USA, pp. 5585–5590, 2006.
coded as:

```
\bibitem{Levant}
A. Levant. ``Exact differentiation of
signals with unbounded higher
derivatives,''  in \textit{Proceedings
of the 45th IEEE Conference on
Decision and Control}, San Diego,
California, USA, pp. 5585--5590, 2006.
```

[8] M. Fliess, C. Join, and H. Sira-Ramirez. "Non-linear estimation is easy," *International Journal of Modelling, Identification and Control*, vol. 4, no. 1, pp. 12–27, 2008.
coded as:

```
\bibitem{Cedric}
M. Fliess, C. Join, and H. Sira-Ramirez.
``Non-linear estimation is easy,''
\textit{International Journal of Modelling,
Identification and Control}, vol. 4,
no. 1, pp. 12--27, 2008.
```

[9] R. Ortega, A. Astolfi, G. Bastin, and H. Rodriguez. "Stabilization of food-chain systems using a port-controlled Hamiltonian description," in *Proceedings of the American Control Conference*, Chicago, Illinois, USA, pp. 2245–2249, 2000.
coded as:

```
\bibitem{Ortega}
R. Ortega, A. Astolfi, G. Bastin, and H.
Rodriguez. ``Stabilization of food-chain
systems using a port-controlled Hamiltonian
description,'' in \textit{Proceedings of the
American Control Conference}, Chicago,
Illinois, USA, pp. 2245--2249, 2000.
```

C. Accented Characters in References

When using accented characters in references, please use the standard LaTeX coding for accents. **Do not use math coding for character accents**. For example:

```
\'e, \"o, \`a, \~e
```

will produce: é, ö, à, ẽ

D. Use of BibTeX

If you wish to use BibTeX, please see the documentation that accompanies the IEEEtran Bibliography package.

E. Biographies and Author Photos

Authors may have options to include their photo or not. Photos should be a bit-map graphic (.tif or .jpg) and sized to fit in the space allowed. Please see the coding samples below:

```
\begin{IEEEbiographynophoto}{Jane Doe}
Biography text here without a photo.
\end{IEEEbiographynophoto}
```

or a biography with a photo

```
\begin{IEEEbiography}[{\includegraphics
[width=1in,height=1.25in,clip,
keepaspectratio]{fig1.png}}]
{IEEE Publications Technology Team}
In this paragraph you can place
your educational, professional background
and research and other interests.
\end{IEEEbiography}
```

Please see the end of this document to see the output of these coding examples.

VII. Mathematical Typography and Why It Matters

Typographical conventions for mathematical formulas have been developed to **provide uniformity and clarity of presentation across mathematical texts**. This enables the readers of those texts to both understand the author's ideas and to grasp new concepts quickly. While software such as LaTeX and MathType® can produce aesthetically pleasing math when used properly, it is also very easy to misuse the software, potentially resulting in incorrect math display.

IEEE aims to provide authors with the proper guidance on mathematical typesetting style and assist them in writing the best possible article.

As such, IEEE has assembled a set of examples of good and bad mathematical typesetting. You will see how various issues are dealt with. The following publications have been referenced in preparing this material:

Mathematics into Type, published by the American Mathematical Society
The Printing of Mathematics, published by Oxford University Press
The LaTeXCompanion, by F. Mittelbach and M. Goossens
More Math into LaTeX, by G. Grätzer
AMS-StyleGuide-online.pdf, published by the American Mathematical Society

Further examples can be seen at http://journals.ieeeauthorcenter.ieee.org/wp-content/uploads/sites/7/IEEE-Math-Typesetting-Guide.pdf

A. Display Equations

A simple display equation example shown below uses the "equation" environment. To number the equations, use the `\label` macro to create an identifier for the equation. LaTeX will automatically number the equation for you.

$$x = \sum_{i=0}^{n} 2iQ. \tag{1}$$

is coded as follows:

```
\begin{equation}
\label{deqn_ex1}
x = \sum_{i=0}^{n} 2{i} Q.
\end{equation}
```

To reference this equation in the text use the `\ref` macro. Please see (1)
is coded as follows:

```
Please see (\ref{deqn_ex1})
```

B. Equation Numbering

Consecutive Numbering: Equations within an article are numbered consecutively from the beginning of the article to the end, i.e., (1), (2), (3), (4), (5), etc. Do not use roman numerals or section numbers for equation numbering.

Appendix Equations: The continuation of consecutively numbered equations is best in the Appendix, but numbering as (A1), (A2), etc., is permissible.

Hyphens and Periods: Hyphens and periods should not be used in equation numbers, i.e., use (1a) rather than (1-a) and (2a) rather than (2.a) for sub-equations. This should be consistent throughout the article.

C. Multi-line equations and alignment

Here we show several examples of multi-line equations and proper alignments.

A single equation that must break over multiple lines due to length with no specific alignment.

$$\begin{multline}\text{The first line of this example}\\ \text{The second line of this example}\\ \text{The third line of this example} \tag{2}\end{multline}$$

is coded as:

```
\begin{multline}
\text{The first line of this example}\\
\text{The second line of this example}\\
\text{The third line of this example}
\end{multline}
```

A single equation with multiple lines aligned at the = signs

$$\begin{align}a &= c+d \tag{3}\\ b &= e+f \tag{4}\end{align}$$

is coded as:

```
\begin{align}
a &= c+d \\
b &= e+f
\end{align}
```

The `align` environment can align on multiple points as shown in the following example:

$$\begin{align}x &= y & X &= Y & a &= bc \tag{5}\\ x' &= y' & X' &= Y' & a' &= bz \tag{6}\end{align}$$

is coded as:

```
\begin{align}
x &= y & X & =Y & a &=bc\\
x' &= y' & X' &=Y' &a' &=bz
\end{align}
```

D. Subnumbering

The amsmath package provides a `subequations` environment to facilitate subnumbering. An example:

$$\begin{align}f &= g \tag{7a}\\ f' &= g' \tag{7b}\\ \mathcal{L}f &= \mathcal{L}g \tag{7c}\end{align}$$

is coded as:

```
\begin{subequations}\label{eq:2}
\begin{align}
f&=g \label{eq:2A}\\
f' &=g' \label{eq:2B}\\
\mathcal{L}f &= \mathcal{L}g \label{eq:2c}
\end{align}
\end{subequations}
```

E. Matrices

There are several useful matrix environments that can save you some keystrokes. See the example coding below and the output.

A simple matrix:

$$\begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix} \tag{8}$$

is coded as:

```
\begin{equation}
\begin{matrix}  0 &  1 \\
1 &  0 \end{matrix}
\end{equation}
```

A matrix with parenthesis

$$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{9}$$

is coded as:

```
\begin{equation}
\begin{pmatrix} 0 & -i \\
 i &  0 \end{pmatrix}
\end{equation}
```

A matrix with square brackets

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{10}$$

is coded as:

```
\begin{equation}
\begin{bmatrix} 0 & -1 \\
1 &  0 \end{bmatrix}
\end{equation}
```

A matrix with curly braces

$$\begin{Bmatrix} 1 & 0 \\ 0 & -1 \end{Bmatrix} \tag{11}$$

is coded as:

```
\begin{equation}
\begin{Bmatrix} 1 &  0 \\
0 & -1 \end{Bmatrix}
\end{equation}
```

A matrix with single verticals

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \tag{12}$$

is coded as:

```
\begin{equation}
\begin{vmatrix} a &  b \\
c &  d \end{vmatrix}
\end{equation}
```

A matrix with double verticals

$$\begin{Vmatrix} i & 0 \\ 0 & -i \end{Vmatrix} \tag{13}$$

is coded as:

```
\begin{equation}
\begin{Vmatrix} i &  0 \\
0 & -i \end{Vmatrix}
\end{equation}
```

F. Arrays

The `array` environment allows you some options for matrix-like equations. You will have to manually key the fences, but you'll have options for alignment of the columns and for setting horizontal and vertical rules. The argument to `array` controls alignment and placement of vertical rules.

A simple array

$$\left(\begin{array}{cccc} a+b+c & uv & x-y & 27\\ a+b & u+v & z & 134 \end{array}\right) \tag{14}$$

is coded as:

```
\begin{equation}
\left(
\begin{array}{cccc}
a+b+c & uv & x-y & 27\\
a+b & u+v & z & 134
\end{array} \right)
\end{equation}
```

A slight variation on this to better align the numbers in the last column

$$\left(\begin{array}{cccr} a+b+c & uv & x-y & 27\\ a+b & u+v & z & 134 \end{array}\right) \tag{15}$$

is coded as:

```
\begin{equation}
\left(
\begin{array}{cccr}
a+b+c & uv & x-y & 27\\
a+b & u+v & z & 134
\end{array} \right)
\end{equation}
```

An array with vertical and horizontal rules

$$\left(\begin{array}{c|c|c|r} a+b+c & uv & x-y & 27\\ \hline a+b & u+v & z & 134 \end{array}\right) \tag{16}$$

is coded as:

```
\begin{equation}
\left(
\begin{array}{c|c|c|r}
a+b+c & uv & x-y & 27\\
a+b & u+v & z & 134
\end{array} \right)
\end{equation}
```

Note the argument now has the pipe ”|” included to indicate the placement of the vertical rules.

G. Cases Structures

Many times we find cases coded using the wrong environment, i.e., `array`. Using the `cases` environment will save keystrokes (from not having to type the `\left\lbrace`) and automatically provide the correct column alignment.

$$z_m(t) = \begin{cases} 1, & \text{if } \beta_m(t) \\ 0, & \text{otherwise.} \end{cases}$$

is coded as follows:

```
\begin{equation*}
{z_m(t)} =
\begin{cases}
1,&{\text{if}}\ {\beta }_m(t),\\
{0,}&{\text{otherwise.}}
\end{cases}
\end{equation*}
```

Note that the "&" is used to mark the tabular alignment. This is important to get proper column alignment. Do not use `\quad` or other fixed spaces to try and align the columns. Also, note the use of the `\text` macro for text elements such as "if" and "otherwise".

H. Function Formatting in Equations

In many cases there is an easy way to properly format most common functions. Use of the \ in front of the function name will in most cases, provide the correct formatting. When this does not work, the following example provides a solution using the `\text` macro.

$$d_{R}^{KM} = \underset{d_{l}^{KM}}{\text{arg min}}\{d_{1}^{KM},\ldots,d_{6}^{KM}\}.$$

is coded as follows:

```
\begin{equation*}
 d_{R}^{KM} = \underset {d_{l}^{KM}}
 {\text{arg min}} \{ d_{1}^{KM},
 \ldots,d_{6}^{KM}\}.
\end{equation*}
```

I. Text Acronyms inside equations

This example shows where the acronym "MSE" is coded using `\text{}` to match how it appears in the text.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_{i} - \hat{Y_{i}})^{2}$$

```
\begin{equation*}
 \text{MSE} = \frac {1}{n}\sum _{i=1}^{n}
(Y_{i} - \hat {Y_{i}})^{2}
\end{equation*}
```

J. Obsolete Coding

Avoid the use of outdated environments, such as `eqnarray` and $$ math delimiters, for display equations. The $$ display math delimiters are left over from PlainTeX and should not be used in LaTeX, ever. Poor vertical spacing will result.

K. Use Appropriate Delimiters for Display Equations

Some improper mathematical coding advice has been given in various YouTube™ videos on how to write scholarly articles, so please follow these good examples:

For **single-line unnumbered display equations**, please use the following delimiters:

```
\[ . . . \] or
```

```
\begin{equation*} . . . \end{equation*}
```

Note that the * in the environment name turns off equation numbering.

For **multiline unnumbered display equations** that have alignment requirements, please use the following delimiters:

```
\begin{align*} . . . \end{align*}
```

For **single-line numbered display equations**, please use the following delimiters:

```
\begin{equation} . . . \end{equation}
```

For **multiline numbered display equations**, please use the following delimiters:

```
\begin{align} . . . \end{align}
```

VIII. LaTeX Package Suggestions

Immediately after your documenttype declaration at the top of your LaTeX file is the place where you should declare any packages that are being used. The following packages were used in the production of this document.

```
\usepackage{amsmath,amsfonts}
\usepackage{algorithmic}
\usepackage{array}
\usepackage[caption=false,font=normalsize,
   labelfont=sf,textfont=sf]{subfig}
\u00sepackage{textcomp}
\usepackage{stfloats}
\usepackage{url}
\usepackage{verbatim}
\usepackage{graphicx}
\usepackage{balance}
```

IX. Additional Advice

Please use "soft" (e.g., `\eqref{Eq}`) or `(\ref{Eq})` cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BibTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BibTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Please do not use `\nonumber` or `\notag` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

X. A Final Checklist

1) Make sure that your equations are numbered sequentially and there are no equation numbers missing or duplicated. Avoid hyphens and periods in your equation numbering. Stay with IEEE style, i.e., (1), (2), (3) or for subequations (1a), (1b). For equations in the appendix (A1), (A2), etc..
2) Are your equations properly formatted? Text, functions, alignment points in cases and arrays, etc.
3) Make sure all graphics are included.
4) Make sure your references are included either in your main LaTeX file or a separate .bib file if calling the external file.

References

[1] A. Trudova, M. Dolezel, and A. Buchalcevova, "Artificial intelligence in software test automation: A systematic literature review," pp. 181–192, 2020.

[2] V. Garousi, Z. Jafarov, A. B. Keleş, S. Değirmenci, E. Özdemir Testinium AŞ, and R. Zarringhalami, "Ai-powered software testing tools: A systematic review and empirical assessment of their features and limitations," Tech. Rep., 5 2025. [Online]. Available: https://arxiv.org/abs/2409.00411

[3] A. Singh and O. Al-Azzam, "Artificial intelligence applied to software testing." Academy and Industry Research Collaboration Center (AIRCC), 11 2023, pp. 01–12.

[4] T. Li, C. Cui, R. Huang, D. Towey, and L. Ma, “Large language models for automated web-form-test generation: An empirical study,” *ACM Transactions on Software Engineering and Methodology*, 5 2025. [Online]. Available: https://doi.org/10.1145/3735553

[5] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, “Automated unit test improvement using large language models at meta,” in *FSE Companion - Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, 2024, pp. 185–196. [Online]. Available: https://arxiv.org/abs/2402.09171

[6] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” 12 2023. [Online]. Available: https://arxiv.org/abs/2302.06527

[7] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, “An empirical study of unit test generation with large language models.” 9 2024. [Online]. Available: https://arxiv.org/abs/2406.18181

[8] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, “Unit test generation using generative ai : A comparative performance analysis of autogeneration tools,” 2 2024. [Online]. Available: https://arxiv.org/abs/2312.10622

[9] R. Kanth, R. Guru, M. B. K, and D. Akshaya, “Ai vs. conventional testing: A comprehensive comparison of effectiveness &efficiency,” *Educational Administration: Theory and Practice*, 1 2023. [Online]. Available: https://kuey.net/index.php/kuey/article/view/7495

[10] H. Kirinuki and H. Tanno, “Chatgpt and human synergy in black-box testing: A comparative analysis,” 1 2024. [Online]. Available: https://arxiv.org/abs/2401.13924

[11] F. Ricca, A. Marchetto, and A. Stocco, “A multi-year grey literature review on ai-assisted test automation,” 1 2025. [Online]. Available: https://arxiv.org/abs/2408.06224

[12] G. Wang, Q. Xu, L. C. Briand, and K. Liu, “Mutation-guided unit test generation with a large language model,” 8 2025. [Online]. Available: https://arxiv.org/abs/2506.02954

[13] R. F. de Lima Junior, L. F. P. de Barros Presta, L. S. Borborema, V. N. da Silva, M. L. de Melo Dahia, and A. C. S. e Santos, “A case study on test case construction with large language models: Unveiling practical insights and challenges,” 12 2023. [Online]. Available: https://arxiv.org/abs/2312.12598

[14] S. Rehan, B. Al-Bander, and A. A.-S. Ahmad, “Harnessing large language models for automated software testing: A leap towards scalable test case generation,” *Electronics (Switzerland)*, vol. 14, 4 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/7/1463

[15] D. Huang, J. M. Zhang, M. Harman, Q. Zhang, M. Du, and S.-K. Ng, “Benchmarking llms for unit test generation from real-world functions,” 8 2025. [Online]. Available: https://arxiv.org/abs/2508.00408

[16] M. Gkikopouli and B. Bataa, “Empirical comparison between conventional and ai-based automated unit test generation tools in java,” p. 30, 6 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1764443&dswid=4942

[17] S. Baltes, F. Angermeir, C. Arora, M. M. Barón, C. Chen, L. Böhme, F. Calefato, N. Ernst, D. Falessi, B. Fitzgerald, D. Fucci, M. Kalinowski, S. Lambiase, D. Russo, M. Lungu, L. Prechelt, P. Ralph, R. van Tonder, C. Treude, and S. Wagner, “Guidelines for empirical studies in software engineering involving large language models,” 9 2025. [Online]. Available: https://arxiv.org/abs/2508.15503

[18] B. Trinkenreich, F. Calefato, G. Hanssen, K. Blincoe, M. Kalinowski, M. Pezzè, P. Tell, and M. A. Storey, “Get on the train or be left on the station: Using llms for software engineering research,” in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 7 2025, pp. 1503–1507. [Online]. Available: https://dl.acm.org/doi/10.1145/3696630.3731666

Jane Doe Biography text here without a photo.

IEEE Publications Technology Team In this paragraph you can place your educational, professional background and research and other interests.