International Journal of Applied and Advanced Scientific Research (IJAASR)
Impact Factor: 5.655, ISSN (Online): 2456 - 3080
(www.dvpublication.com) Volume 8, Issue 2, 2023

DEFECT CHARACTERISTICS AND AUTOMATED PATCH CORRECTNESS ACROSS C AND JAVA BENCHMARKS

Divya Vishwanath* & Paul Johnson Asamoah**
* Janardhan Rai Nagar Rajasthan Vidyapeeth, Udaipur, Rajasthan, India
** Power Life University College, Accra, Ghana, West Africa

Cite This Article: Divya Vishwanath & Paul Johnson Asamoah, "Defect Characteristics and Automated Patch Correctness Across C And Java Benchmarks", International Journal of Applied and Advanced Scientific Research, Volume 8, Issue 2, July - December, Page Number 71-81, 2023.

Abstract:

This analysis examines whether defect characteristics provide stable signals of automated patch correctness in internationally used software benchmarks. Public annotations from ManyBugs and Defects4J yield 359 assessed defect and repair tool observations covering 126 defects across 13 C and Java projects. Descriptive proportions, Spearman associations, sample exclusions, and alternative weighting assess relationships and compositional sensitivity. Correct outcomes represent 13.37 percent of retained evaluations. File count shows almost no monotonic association with correctness, while reference repair extent shows a weak negative association and relevant test count a positive association. Excluding SPR and Prophet reduces the latter associations from negative 0.095 to negative 0.002 and from 0.170 to 0.067, respectively. Equal defect weighting reverses the small benchmark ordering. These patterns identify evaluation composition and unequal defect representation as consequential features of pooled comparisons. The contribution refines the search and validation perspective by distinguishing observed repair characteristics from the structure of assessment evidence. Benchmark maintainers and software

Key Words: Automated Program Repair; Benchmark Evaluation; Patch Correctness; Sensitivity Analysis; Software Testing.

1. Introduction:

An automatically generated patch can pass every available test and still leave software behavior incorrect. This gap between test satisfaction and correctness complicates decisions about accepting generated repairs, comparing repair techniques, and allocating verification effort. Automated program repair addresses the burden of correcting software defects, but its practical value depends on whether accepted changes preserve intended behavior beyond the inputs used during repair. Le Goues et al. 2019 identify weak specifications as a central challenge because available tests constrain candidate patches without fully describing the behavior that software must preserve. The substantive question therefore concerns both the characteristics associated with correct outcomes and the evaluation conditions under which those associations appear. The search and validation perspective explains why defect characteristics may shape repair outcomes. Candidate generation operates within a space of permitted program changes, while validation determines which candidates remain acceptable. A reference correction distributed across several files may indicate coordination requirements that restricted repair operators cannot readily reproduce. A correction involving more edited lines may indicate greater modification extent, although line count cannot distinguish repetitive edits from difficult semantic changes. Relevant tests provide behavioral constraints that may reject unsuitable candidates, but their number does not establish their diversity or assertion strength. These competing explanations make structural spread, modification extent, and test quantity distinct empirical measures rather than interchangeable indicators of repair difficulty. Gazzola et al. 2019 document differences across repair strategies and supported defect classes, reinforcing the need to interpret these measures within the capabilities of the techniques being evaluated.

Existing evidence establishes that generating an acceptable patch and repairing a defect correctly are different achievements. Martinez et al. 2017 report that the evaluated Java repair techniques generated patches for 47 of 224 defects, while manual assessment identified nine defects that could be correctly repaired. Their results demonstrate the consequences of using incomplete tests as correctness specifications. Motwani et al. 2022 examine patch quality using independent tests and identify a small positive relationship with test suite size alongside more complex relationships involving coverage. These findings support examining validation information, but they also show why test quantity alone cannot explain repair quality. Moreover, total test suite size and the number of tests relevant to files touched by a reference repair are related but different measurements. Their associations should not be interpreted as estimates of the same quantity. Correctness measurement introduces a further distinction between the quality of generated patches and the availability of evidence about that quality. Ye et al. 2021 show that additional generated tests can expose errors in earlier patch assessments, while Lin et al. 2022 investigate contextual code representations for distinguishing correct patches from over fitting candidates. These approaches address the credibility of correctness classification. A complementary issue concerns aggregation: a reported correctness proportion also depends on which generated outcomes receive assessment and how repeated evaluations of the same defect are weighted. The reviewed evidence does not establish whether associations involving reference repair file count, line count, and relevant

test count remain descriptively stable when project concentration, technique composition, and defect weighting change together within the available C and Java benchmark records.

This analysis aims to characterize the relationships between observed defect characteristics and automated patch correctness, and to assess how their interpretation changes with evaluation composition. The central research question is how correctness varies with developer patch file count, developer patch line count, and relevant test count among generated patches with available quality assessments. The specific objectives are to describe correctness across structural spread, examine its association with modification extent, and assess its relationship with relevant test quantity. Exploratory sensitivity comparisons examine whether the resulting patterns depend on particular projects, repair techniques, or the weighting of repeated defect evaluations. The empirical setting combines publicly available annotations for ManyBugs and Defects4J. ManyBugs provides reproducible C defects and baseline repair evaluations, supporting comparisons grounded in identifiable software projects and reference corrections, as documented by Le Goues et al. 2015. The consolidated source frame contains 409 defects, while the retained analytical sample comprises 359 defect and repair tool observations covering 126 distinct defects across 13 projects. The outcome records source assessed correctness, and the three explanatory measures retain their original count scales. Descriptive distributions, grouped correctness proportions, Spearman associations, and exploratory exclusions provide the analytical basis. The records form a cross section with repeated tool evaluations rather than annual observations; the period from 2010 to 2022 defines the literature boundary rather than a continuous defect observation window.

The completed analysis contributes evidence about the sensitivity of apparent relationships. File count exhibits almost no pooled monotonic association with correctness, while line count shows a weak negative association and relevant test count shows a positive association. The latter two weaken when SPR and Prophet are excluded, and removing the php project reduces the test count association. Equal weighting of distinct defects also reverses the small ordering between the C and Java correctness proportions. Together, these results show that defect measurements do not provide stable summaries independently of the evaluations contributing to them. The contribution is a descriptive integration of measurement, sample composition, and weighting sensitivity, without treating these comparisons as causal effects or isolated programming language differences. The practical relevance extends to benchmark maintainers, repair tool developers, and organizations evaluating generated code. Separating patch production from assessment availability makes the denominator of correctness explicit. Reporting both evaluation weighted and defect weighted proportions exposes the influence of unequal evaluation coverage, while project and technique exclusions reveal concentrated sources of pooled associations. These practices can inform comparisons across widely used open-source systems without assuming that benchmark proportions transfer directly to other software populations. The theoretical implication is that interpreting repair outcomes requires attention to both the constraints governing candidate generation and the composition of the evidence used to evaluate success.

2. Theory Conceptual Framework and Research Questions Development:

This analysis draws on the search and validation perspective of automated software repair. Repair tools generate candidate changes within a defined search space and use tests to determine which candidates remain acceptable. Defect characteristics constrain the changes required, while available tests constrain the evidence used to accept them. These processes interact because an acceptable patch must address the observed failure without disrupting required functionality. Le Goues et al. 2012 describe how GenProg combines program modification, test-based evaluation, and patch reduction within this process. This perspective supports examining whether observable defect characteristics correspond to differences in automated patch correctness. The focal explanatory construct is defect characteristics, represented by developer patch file count, developer patch line count, and relevant test count. These are observed measures rather than indicators of an estimated latent construct. The outcome is automated patch correctness as classified in the source data. Correctness receives a value of one when the source classifies a generated patch as correct and zero when it classifies the patch as incorrect or merely plausible. This distinction reflects the limits of validation evidence. Monperrus 2018 identifies test suites, contracts, models, and other correctness oracles as alternative foundations for automated repair. Their different information content means that passing a particular test suite cannot establish correctness across all intended behavior.

Developer patch file count measures the number of files containing substantive changes in the developer repair. It represents the structural spread of the reference correction. A repair distributed across several files may require coordinated changes across interfaces or interacting components. Such coordination can make a correct candidate harder to construct when a repair tool searches within restricted locations or applies limited modification operators. The theoretical expectation is therefore that greater structural spread may correspond to lower correctness among generated patches. A competing mechanism prevents treating this expectation as an established relationship. Several edited files may contain repeated, simple corrections, while a change within one file may require difficult semantic reasoning. File count captures repair spread rather than the difficulty of every required modification. The ManyBugs benchmark described by Le Goues et al. 2015 provides reproducible defects, tests, and reference repairs, supporting comparisons across defect characteristics. Its

benchmark structure does not establish that file count independently determines correctness. This analysis therefore uses an exploratory question that allows positive, negative, or inconclusive associations.

RQ1: How does the proportion of generated patches classified as correct vary with developer patch file count?

Developer patch line count measures the number of substantive lines edited in the reference repair. Whereas file count captures structural spread, line count captures the extent of the developer modification. More edited lines may indicate that a repair requires several coordinated transformations. Under restricted repair operators and search resources, this requirement may reduce the chance that a generated candidate reproduces the necessary behavior. This mechanism suggests a possible negative association between line count and correctness. However, reference repair length does not uniquely identify semantic difficulty. Developers may combine a correction with changes that an automated repair need not reproduce. Conversely, a short repair may depend on a difficult condition, boundary case, or data dependency. Gazzola et al. 2019 document substantial variation in repair approaches, modification strategies, and supported defect classes. Their synthesis supports treating the relationship between repair extent and correctness as dependent on the capabilities of the repair technique. Developer patch line count therefore serves as an observable proxy for repair extent, without assuming that longer reference repairs necessarily require longer automated patches.

RQ2: How does the proportion of generated patches classified as correct vary with developer patch line count?

Relevant test count measures the number of tests that execute at least one statement in a file modified by the developer repair. Its mechanism concerns the information available to constrain candidate acceptance. Additional relevant tests may expose more behaviors that a candidate must preserve. When these tests contain distinct inputs and effective assertions, they can reject candidates that remove functionality or address only the triggering failure. This mechanism suggests a possible positive association between relevant test count and correctness. Test quantity nevertheless differs from test adequacy. Several tests may exercise similar paths or contain assertions that leave important behavior unspecified. Larger projects may also contain more relevant tests without providing stronger evidence for the particular defect. Martinez et al. 2017 show that patches satisfying available tests can remain incorrect when assessed beyond test satisfaction. That finding supports distinguishing validation success from correctness, but it does not establish that test count alone predicts correctness. The analysis consequently examines the observed relationship without imposing a uniformly positive direction.

RQ3: How does the proportion of generated patches classified as correct vary with relevant test count?

The conceptual framework links the three observed defect measures to source classified automated patch correctness. File count represents structural spread, line count represents modification extent, and relevant test count represents the quantity of potentially constraining validation evidence. These measures remain separate because they capture different mechanisms and need not vary together. Their conceptual links denote associations to be examined rather than demonstrated causal effects. The analytical unit is a defect and repair tool pair with a generated patch and an available correctness classification. The 359 eligible observations support descriptive comparisons of correctness proportions across the three measures. Several tools can contribute observations for the same defect, so these records do not represent 359 independent defects. Comparisons also remain conditional on patch production and recorded assessment. C and Java identify benchmark contexts rather than an isolated language effect because language coincides with benchmark membership. The research questions retain competing expectations and permit inconclusive patterns without changing their wording in response to the observed results.

3. Data:

This analysis establishes source provenance, reconciles benchmark records, and defines the sample used to examine automated patch correctness. Observed defect measures and documented quality assessments support descriptive comparisons across generated patch outcomes.

3.1 Data Sources, Population, and Sample Construction:

The data comprise ManyBugs.csv and Defects4J.csv from the LASER UMASS Automated Repair Applicability Data repository. The repository documents extensions to ManyBugs version beta 2.1 and Defects4J version 1.1.0. ManyBugs contributes 185 defects from nine C projects, while Defects4J contributes 224 defects from four Java projects. These are the repository subsets represented in the workbook, rather than the complete populations of defects in either benchmark. Le Goues et al. 2015 establish the reproducible benchmark setting for ManyBugs, while Martinez et al. 2017 document the Java repair evaluations underlying the available Defects4J outcomes. The source files retain their original filenames, and the workbook records the extraction date, source URLs, and source row numbers. The repository identifies an MIT licence. The workbook does not retain a repository commit identifier, which limits exact reconstruction of the downloaded repository state. The period from 2010 to 2022 describes the literature window rather than the dates of individual defect observations. The records do not establish a continuous observation period within that window.

The target population comprises automated repair outcomes for software defects in C and Java projects. The accessible sampling frame comprises the 409 benchmark defects and their recorded evaluations across nine repair techniques. The empirical setting is open-source software engineering without a defined national

sampling boundary. The unit of analysis is a defect and repair tool pair. The resulting structure is a cross section with repeated evaluations of some defects, rather than a time series or panel. Measurements refer to individual repair instances and have no annual frequency. The source schema contains 1,782 potential defect and tool combinations. This number includes combinations explicitly marked as unevaluated. Inclusion requires a reported generated patch, an available quality classification, complete explanatory measurements, and agreement between patch production and quality fields. Table 1 reconciles the source frame with the retained sample.

Table 1: Sample construction

Stage	Distinct Defects Represented	Observations	Reason or Rule
Source frame	409	1,782	All defect and tool combinations represented by the source schema
Excluded unevaluated combinations	80	240	Patch production field contains NA
Excluded combinations without a generated patch or quality assessment	350	1,070	Patch production equals zero and quality contains NA
Excluded generated outcomes without quality assessments	40	111	Patch production equals one but quality contains NA
Records with an available quality classification	126	361	Source quality classified as correct, incorrect, or plausible
Excluded conflicting records	2	2	Quality recorded despite patch production equalling zero
Final descriptive sample	126	359	Complete measurements and consistent production and quality fields

Distinct defect counts within exclusion categories overlap and must not be added. The final sample contains 275 C observations covering 79 defects and 84 Java observations covering 47 defects. All 13 source projects remain represented. No estimation stage introduces additional sample losses.

3.2 Data Integration, Cleaning, and Quality Assurance:

Integration appends harmonized records from the two benchmark files. It does not match defects across benchmarks. Benchmark, project, and defect identifier jointly identify a defect, while the repair tool identifies its evaluation record. Tool specific production and quality columns are converted into separate records. File count, line count, and relevant test count remain attached to their original defect and therefore repeat when several tools evaluate that defect. An audit identified no duplicate defect keys in either raw source sheet and no duplicate defect and tool keys among the 359 retained observations. Each retained record was reconciled against its recorded source row. The audit verified project and defect identifiers, all three explanatory measurements, patch production status, source quality, and binary outcome coding. Every retained explanatory count was a positive integer.

Two AE records contain conflicting production and quality information. Their source quality is plausible despite patch production equalling zero. Both records remain documented in the exclusion sheet and are omitted from the analytical sample. This rule preserves the distinction between a generated patch and an internally inconsistent source annotation. Missing values retain their source meaning. An NA production field identifies an unevaluated combination. An NA quality field can indicate that no patch was produced or that a reported patch lacks a quality assessment. The production field distinguishes these situations. Missing quality is never recoded as incorrectness. The three explanatory measures and the outcome contain no missing values in the retained sample, and no imputation or interpolation was applied.

Counts retain their original units. No frequency conversion, currency adjustment, standardization, trimming, or winsorization was performed. Large line and test counts remain in the sample because they reconcile with the raw sheets. Their inclusion establishes source consistency rather than independent verification of the underlying code or tests. The workbook records the implemented exclusion rules but contains no dated preregistration establishing when those rules were selected. The consistency audit did not change the retained sample. Eligibility conditions create selection boundaries: correctness is observed only for outcomes with reported patches and available assessments. The descriptive sample consequently represents assessed generated outcomes rather than all attempted repairs.

3.3 Variable Measurement, Construction, and Validation:

Defect characteristics are represented by three separate observed counts. Developer patch file count measures structural spread, developer patch line count measures modification extent, and relevant test count measures the quantity of tests executing code in files touched by the reference repair. These measures describe different aspects of the repair setting and are not combined into an index. Developer patch measurements refer

to the human reference correction. They do not measure the size of the automatically generated patch. Relevant test count measures test quantity rather than assertion strength or behavioral coverage. This distinction matters because automated repair can use test results to guide candidate acceptance without obtaining a complete specification of intended behavior, as illustrated by Le Goues et al. 2012.

Table 2: Variable dictionary

Variable or Indicator	Analytical Role	Definition and Construction	Source and Identifier	Unit and Frequency	Transformation and Direction	Missing Value Rule	Coverage and Limitation
Automated patch correctness	Outcome	One for correct; zero for incorrect or plausible	Both source files; Patch Quality fields	Binary classification per defect and tool pair	Binary recoding; higher values indicate source assessed correctness	Exclude unavailable or conflicting assessments	359 observations; inherits source assessment methods
Developer patch file count	Explanatory measure	Files containing substantive edits in the reference repair	Both source files; File Count	Files per defect	Untransformed; higher values indicate wider structural spread	Require a valid count	359 observations; repeated across tools evaluating the same defect
Developer patch line count	Explanatory measure	Nonblank and noncomment lines edited in the reference repair	Both source files; Line Count	Lines per defect	Untransformed; higher values indicate greater modification extent	Require a valid count	359 observations; does not directly measure semantic difficulty
Relevant test count	Explanatory measure	Tests executing at least one statement in a file edited by the reference repair	Both source files; Relevant Test Count	Tests per defect	Untransformed; higher values indicate more relevant tests	Require a valid count	359 observations; does not establish test adequacy
Benchmark and language	Descriptive identifiers	ManyBugs corresponds to C; Defects4J corresponds to Java	Source filename and benchmark documentation	Category per record	Original categories retained	Require an identified source	Language and benchmark coincide
Project and defect identifier	Grouping identifiers	Jointly identify the original defect within a benchmark	Project and DefectId	Identifier per defect	Original identifiers retained	Require complete identifiers	126 distinct retained defects
Repair tool	Evaluation identifier	Technique represented by the source production and quality columns	Patch Found and Patch Quality column suffixes	Category per evaluation	Original tool labels retained	Require an identified tool	Nine techniques with unequal evaluation coverage

The retained outcome contains 48 correct classifications, 73 incorrect classifications, and 238 plausible classifications. The binary outcome therefore contains 48 ones and 311 zeros. Combining incorrect and plausible labels follows the repository distinction between established correctness and test satisfying outcomes without established correctness. The original labels remain available for inspection. Table 3 reports descriptive statistics at the defect and tool observation level. Standard deviations use the sample denominator. Because explanatory values repeat across tools, these summaries describe the retained evaluation records rather than an equally weighted distribution of distinct defects.

Table 3: Descriptive statistics

Variable	Valid Observations	Mean	Standard Deviation	Median	Minimum	Maximum
Developer patch file count	359	1.16	0.59	1	1	5
Developer patch line count	359	26.77	148.01	4	1	1,608
Relevant test count	359	2,205.20	3,360.79	90	3	7,624
Automated patch correctness	359	0.134	0.341	0	0	1

Validation consists of source reconciliation, identifier checks, admissible value checks, and production and quality consistency checks. These procedures verify the construction of observed variables. They do not establish an independent reassessment of patch correctness.

3.4 Analytical Framework, Identification, and Reproducibility:

The implemented analysis is descriptive. It reports sample composition, measurement distributions, and the proportion of retained outcomes classified as correct. The quantity of interest is correctness among generated patches with available and internally consistent assessments. This denominator differs from all attempted repairs, all benchmark defects, and all possible candidate patches. Equation 1 defines the descriptive correctness proportion:

$$\hat{p}_g = \frac{C_g}{N_g}$$

Here, $\hat{p}_g$ denotes the correctness proportion for group g, C_g denotes the number of retained outcomes classified as correct in that group, and N_g denotes all retained assessed outcomes in that group. For the complete analytical sample, the calculation is 48 divided by 359, yielding 13.37 percent. The same descriptive definition gives 13.45 percent for the C benchmark and 13.10 percent for the Java benchmark. These proportions describe the available records and do not identify a programming language effect. Benchmark composition, repair techniques, evaluation coverage, and assessment procedures differ alongside language.

No regression, causal estimator, predictive model, significance test, or robustness analysis has been implemented. The three research questions concern variation in observed correctness across defect characteristics, but the available overall summaries do not establish their direction or magnitude. Variation in repair strategies documented by Gazzola et al. 2019 also limits interpretation of pooled outcomes as evidence about a common repair process. Correctness classifications remain conditional on the validation procedures used by the source evaluations. Monperrus 2018 distinguishes the different correctness oracles used in automated repair, supporting interpretation of these labels within their documented assessment context. The analysis therefore treats correctness as a source assessed outcome rather than a formal proof of equivalent behavior.

Reproducibility rests on the preserved raw sheets, source filenames, source URLs, source row numbers, retained original quality labels, and documented exclusions. These records permit reconstruction of the sample and binary outcome from the workbook. The final measurements use consistent definitions and retain their original scales, while repeated evaluations and conditional assessment coverage define the limits of the descriptive evidence.

4. Methods:

4.1 Research Design and Analytical Scope:

This analysis uses a descriptive observational design to examine how automated patch correctness varies across developer patch file count, developer patch line count, and relevant test count. The analytical unit is a defect and repair tool pair with a generated patch and an available correctness assessment. The sample construction in Table 1 yields 359 observations covering 126 distinct defects. The measurement definitions in Table 2 govern all calculations. The principal quantity of interest is the proportion of assessed generated outcomes classified as correct. Additional quantities describe monotonic associations between each explanatory measure and correctness. Useful variation arises across defects, projects, benchmarks, and repair techniques. Because several techniques evaluate the same defect, explanatory measurements can repeat across observations. The design characterizes observed relationships without identifying causal effects. Project characteristics, technique capabilities, evaluation coverage, and assessment practices can jointly influence the recorded patterns. Language also coincides with benchmark membership. Benchmark comparisons therefore describe the available C and Java records rather than isolate a programming language effect. Conditioning on generated and assessed outcomes further restricts interpretation to that selected population.

4.2 Descriptive Specification and Association Measures:

The analysis retains the correctness definition and group proportion specified in Equation 1. Correct outcomes receive one, while incorrect and plausible outcomes receive zero. Proportions use retained assessed outcomes as their denominators and are multiplied by 100 when reported as percentages. Comparisons between percentages are expressed in percentage points. For each observed measure, the analysis calculates the valid count, arithmetic mean, sample standard deviation, median, minimum, and maximum. Sample standard deviations use the number of observations less one as the denominator. These summaries describe evaluation records rather than an equally weighted distribution of distinct defects.

Spearman coefficients summarize monotonic associations without requiring normally distributed explanatory counts. Each coefficient is calculated as the Pearson correlation between the ranked explanatory measure and ranked binary outcome:

$$\rho_{s,k,g} = \frac{\sum_{i \in g}\left[R_{ki} - \overline{R}_{k,g}\right]\left[S_i - \overline{S}_g\right]}{\sqrt{\sum_{i \in g}\left[R_{ki} - \overline{R}_{k,g}\right]^2 \sum_{i \in g}\left[S_i - \overline{S}_g\right]^2}}$$

Here, $\rho_{s,k,g}$ denotes the descriptive rank association for explanatory measure kwithin sample g. The index iidentifies an evaluation record. R_{ki} denotes its rank on explanatory measure k, and $S_i$denotes its rank on correctness. The barred terms denote the corresponding mean ranks within g. Tied observations receive average ranks. Ranks are calculated separately within the complete sample and each benchmark subset.

Grouped proportions complement the coefficients because a single rank association can conceal irregular patterns. File counts are grouped into one file and two to five files. Line counts are grouped into one, two to five, six to twenty, and above twenty lines. Relevant test counts are grouped into one to ten, eleven to one hundred, 101 to 1,000, and above 1,000 tests. These exploratory bands were selected during analysis to summarize the observed distributions. They are not prespecified thresholds or estimated decision boundaries. The coefficients and grouped proportions are descriptive. Significance tests, confidence intervals, and

independent observation assumptions are not used to interpret them. Their purpose is to identify direction, magnitude, and distributional patterns within the retained records.

4.3 Record Processing and Measurement Checks:

Processing begins with the original benchmark sheets and their production and quality fields. Benchmark, project, and defect identifier jointly identify each defect. Adding the repair technique identifies the evaluation record. Tool specific source columns are converted into separate observations, while defect measurements remain linked to their original identifiers. Eligibility requires a production value of one, a recognized quality classification, and complete explanatory measurements. Source NA values retain their meaning rather than becoming numerical zeros. Unevaluated combinations, outcomes without generated patches, and generated outcomes without quality assessments remain outside the correctness sample. Two records with quality annotations conflicting with production status are excluded and retained in the exclusion documentation.

Every analytical record is reconciled against its recorded source row. Checks verify identifier agreement, explanatory values, production status, original quality classification, and binary outcome coding. Duplicate checks use the defect key in the raw sheets and the defect and technique key in the analytical records. The audit also verifies that retained explanatory measurements are positive integers. All explanatory measures remain on their original scales. Large values are retained when they agree with the raw records. No imputation, interpolation, normalization, logarithmic transformation, or outlier removal is applied. Consequently, the descriptive statistics and rank associations use the same 359 observations. Calculations use Python 3.12.14. Counts, grouping, means, medians, and sample standard deviations use Python routines, while rank associations use scipy.stats.spearmanr in SciPy 1.17.0. Virtanen et al. 2020 document SciPy's scientific computing framework. The calculations are deterministic and require no random seed, optimization settings, or convergence criterion.

4.4 Sensitivity Analysis and Reproducibility:

Exploratory sensitivity checks examine three specific sources of interpretive dependence. Technique composition is assessed by removing SPR and Prophet and recalculating correctness proportions and rank associations. Project concentration is assessed by removing php and repeating those calculations. These exclusions change the sample, so resulting differences describe sensitivity to composition rather than adjusted effects holding the population constant. The retained samples contain 270 and 259 observations, respectively.

Unequal evaluation coverage is examined through equal defect weighting. The analysis first averages correctness across the observed techniques for each defect and then averages those defect means:

$$\hat{p}_D = \frac{1}{D}\sum_{d=1}^{D}\frac{\sum_{j\in J_d} Y_{dj}}{n_d}$$

Here, $\hat{p}_D$ denotes the correctness proportion under equal defect weighting, Ddenotes the number of distinct retained defects, and didentifies a defect. J_d denotes its set of retained technique evaluations, $n_d$denotes the size of that set, and Y_{dj}denotes binary correctness for technique evaluation j. Each defect therefore receives equal total weight regardless of its number of observed evaluations. Equation 3 is calculated for the complete sample and separately within each benchmark. Assessment availability is examined through logical bounds for all reported generated outcomes:

$$L = \frac{C}{G}, \qquad U = \frac{C + M}{G}$$

Here, Land U denote the lower and upper correctness bounds, Cdenotes observed correct outcomes, Gdenotes all reported generated outcomes, and Mdenotes generated outcomes without quality assessments. The implemented values are 48, 470, and 111, respectively. The lower bound assigns no additional correct outcomes to the unassessed records, while the upper bound assigns correctness to all of them. These bounds address missing assessments without assuming that assessed and unassessed outcomes have equivalent correctness distributions. They remain conditional on the recorded classifications and are not confidence intervals.

The grouped comparisons and sensitivity checks are exploratory. The available materials contain no preregistration establishing them as prior analytical commitments. Reproducibility relies on the preserved raw sheets, original quality labels, source row identifiers, analytical records, exclusion documentation, and deterministic calculation rules. These materials permit reconstruction of the reported summaries while preserving the distinction between generated, assessed, and correctly classified outcomes.

5. Findings:

This analysis examines how automated patch correctness varies with developer patch file count, developer patch line count, and relevant test count. The findings combine descriptive comparisons, rank associations, and exploratory sensitivity checks. All results concern source assessed correctness among generated patches with available quality classifications. The analytical sample contains 359 defect and repair tool observations representing 126 distinct defects across 13 projects. ManyBugs contributes 275 observations from 79 C defects, while Defects4J contributes 84 observations from 47 Java defects. Eligibility requires

reported patch production, an available quality assessment, complete explanatory measurements, and consistency between production and quality fields. The retained observations contain no missing values for the three explanatory measures or the binary outcome.

Correct patches account for 48 observations, equivalent to 13.37 percent of the retained sample. The remaining observations comprise 73 incorrect classifications and 238 plausible classifications. The binary outcome assigns one to correct patches and zero to both remaining categories. These categories preserve the distinction between demonstrated correctness under the source assessment and acceptance through available tests. Martinez et al. 2017 similarly distinguish patches satisfying benchmark tests from patches assessed as correct beyond those tests.

Table 4: Sample characteristics and measurement distributions

Measure	Observations	Mean	Standard Deviation	Median	Minimum	Maximum
Developer patch file count	359	1.16	0.59	1	1	5
Developer patch line count	359	26.77	148.01	4	1	1,608
Relevant test count	359	2,205.20	3,360.79	90	3	7,624
Automated patch correctness	359	0.134	0.341	0	0	1

The explanatory distributions show substantial concentration and asymmetry. Single file reference repairs account for 324 observations, equivalent to 90.25 percent of the sample. Developer patch line count has a median of four but a mean of 26.77, while relevant test count has a median of 90 but a mean of 2,205.20. These differences show that a minority of large values materially influence arithmetic averages. The comparisons therefore retain medians and rank associations alongside means. The first research question concerns structural spread. Correctness equals 13.27 percent among single file reference repairs and 14.29 percent among repairs involving several files. The difference is 1.01 percentage points. At individual file counts, correctness reaches 23.81 percent for two file repairs but equals zero among the 14 observations involving three to five files. This irregular pattern does not indicate a consistent decline as structural spread increases.

The pooled Spearman association between file count and correctness is 0.003, indicating almost no monotonic ordering in the retained records. Corresponding coefficients are 0.010 for C and negative 0.061 for Java. The Java comparison contains only two observations involving several files, which limits interpretation of its direction. RQ1 is therefore answered descriptively: correctness varies across file counts, but the observed distribution provides no consistent monotonic pattern. The second research question concerns modification extent. Correct patches have a mean developer patch line count of 7.96 and a median of three. Other assessed outcomes have a mean of 29.67 and a median of four. The difference in means is much larger than the difference in medians, showing that the descriptive separation depends partly on large reference repairs among outcomes without a correct classification.

Exploratory line count bands reveal a nonmonotonic relationship. Correctness is 14.56 percent for one line reference repairs, rises to 19.13 percent for repairs involving two to five lines, falls to 4.94 percent for six to twenty lines, and reaches 11.67 percent above twenty lines. The bands summarize the observed distribution and were selected during the exploratory analysis.

Table 5: Correctness across exploratory measurement groups

Measure and Group	Assessed Outcomes	Correct Outcomes	Correctness Percent
File count: one	324	43	13.27
File count: two to five	35	5	14.29
Line count: one	103	15	14.56
Line count: two to five	115	22	19.13
Line count: six to twenty	81	4	4.94
Line count: above twenty	60	7	11.67
Relevant tests: one to ten	26	3	11.54
Relevant tests: eleven to one hundred	159	8	5.03
Relevant tests: 101 to 1,000	74	13	17.57
Relevant tests: above 1,000	100	24	24.00

The pooled Spearman coefficient for line count is negative 0.095. The association is negative 0.115 within C and negative 0.015 within Java. RQ2 therefore yields a weak negative pooled association, with little monotonic separation within the Java records. Developer reference repair extent does not provide a uniform ordering of automated patch correctness. This interpretation also respects the distinction between reference repair size and generated patch size. Le Goues et al. 2012 describe candidate generation and patch reduction as operations within the automated repair process, rather than reproduction of every developer edit.

The third research question concerns validation information. Correct patches are associated with a mean of 3,877.88 relevant tests and a median of 4,018.50. Other assessed outcomes have a mean of 1,947.04 and a median of 77. Correctness reaches 24.00 percent above 1,000 relevant tests, compared with 5.03 percent between eleven and one hundred tests. The difference is 18.97 percentage points.

The pooled Spearman coefficient between relevant test count and correctness is 0.170. The corresponding coefficients are 0.203 within C and 0.104 within Java. These values indicate a positive descriptive association, although the grouped proportions do not increase uniformly across every test count band. All 100 observations above 1,000 relevant tests belong to the php project. Consequently, the highest test count category also represents a specific project context.

Table 6: Descriptive rank associations with correctness

Explanatory Measure	Complete Sample	C Benchmark	Java Benchmark
Developer patch file count	0.003	0.010	Negative 0.061
Developer patch line count	Negative 0.095	Negative 0.115	Negative 0.015
Relevant test count	0.170	0.203	0.104
Observations	359	275	84

RQ3 therefore identifies a positive association between relevant test quantity and correctness within the assessed sample. It does not establish that additional tests caused the difference. Test counts do not measure assertion strength, behavioral diversity, or specification completeness. Monperrus 2018 identifies several distinct correctness oracles in automated repair, reinforcing the need to interpret test quantity within the information supplied by the validation process.

Repair technique composition materially affects the pooled results. Correctness ranges from 3.45 percent for TrpAutoRepair to 34.88 percent for Prophet. SPR records 28.26 percent. Together, SPR and Prophet contribute 28 of the 48 correct outcomes, equivalent to 58.33 percent, while accounting for 24.79 percent of retained observations. These comparisons describe the available evaluations because techniques were not assessed on identical defect sets.

Removing SPR and Prophet leaves 270 observations and reduces overall correctness to 7.41 percent. The line count coefficient changes from negative 0.095 to negative 0.002, while the relevant test coefficient declines from 0.170 to 0.067. The file count coefficient becomes 0.078. These exploratory exclusions show that pooled relationships depend partly on repair technique composition. They accord with the variation in supported defects, search strategies, and repair operations documented by Gazzola et al. 2019.

Project composition also affects interpretation. The php project contributes 100 observations and 24 correct outcomes, accounting for half of all correct classifications. Excluding this project leaves 259 observations with correctness of 9.27 percent. The relevant test coefficient declines to 0.090, while the line count coefficient becomes negative 0.064. The direction of these two associations remains unchanged, but their magnitudes weaken. These results identify project concentration as part of the explanation for the pooled patterns.

Weighting provides a further sensitivity check. The principal correctness proportion gives equal weight to each retained defect and tool observation. Giving equal weight to each of the 126 defects, after calculating its average correctness across observed tools, produces a proportion of 15.15 percent. Under evaluation weighting, C records 13.45 percent and Java records 13.10 percent. Under equal defect weighting, C records 14.24 percent and Java records 16.67 percent. The reversal demonstrates that the small benchmark ordering depends on the weighting rule rather than a stable language ranking. The benchmark setting described by Le Goues et al. 2015 supports comparison within documented evaluation conditions, rather than unrestricted generalization across software populations.

Outcome availability places a separate boundary on the findings. The source records 470 generated outcomes, of which 111 lack quality assessments. Among all reported generated outcomes, the observed correct classifications imply a lower bound of 10.21 percent if none of the unassessed outcomes were correct. Assigning correctness to every unassessed outcome yields an upper bound of 33.83 percent. These are logical missing outcome bounds rather than confidence intervals. Their width shows that the assessed sample cannot establish a precise correctness proportion for all generated outcomes.

The findings answer the three research questions within the retained records. File count shows no consistent monotonic relationship with correctness. Line count shows a weak negative pooled association that largely disappears when SPR and Prophet are excluded. Relevant test count shows a positive association that weakens after project and technique exclusions. The empirical result is therefore a conditional descriptive pattern: correctness differences depend on defect measurements, evaluation composition, outcome availability, and the weighting of repeated evaluations.

6. Discussion:

Apparent differences in automated patch correctness depend on how benchmark evidence is assembled and weighted. Table 6 shows almost no monotonic association for developer patch file count, a weak negative

association for line count, and a positive association for relevant test count. These differences challenge the treatment of defect characteristics as interchangeable measures of repair difficulty. Structural spread, modification extent, and validation information capture distinct constraints, while Equation 1 makes correctness conditional on the assessed outcomes included in its denominator. The contribution of this analysis is the joint interpretation of these measurements alongside sample composition and weighting sensitivity. This extends the discussion of search, synthesis, and learning approaches presented by Le Goues et al. 2019 by showing how evaluation design affects the interpretation of their observed outcomes.

The positive test count association provides evidence consistent with stronger validation constraints, but its sensitivity identifies an important boundary to that interpretation. Table 5 associates larger test counts with higher correctness proportions, yet the highest test count category belongs entirely to one project. Removing that project weakens the association, as does excluding SPR and Prophet. The resulting pattern distinguishes a pooled empirical relationship from a demonstrated transmission mechanism: test quantity varies alongside project context and repair technique. Gao et al. 2021 demonstrate that vulnerability repair can operate through extracted crash constraints without requiring a large test suite. Their approach, considered alongside the observed sensitivity, redirects interpretation from the number of tests toward the behavioral information those tests or alternative constraints provide.

The concentration of correct outcomes reveals why pooled averages require interpretation beyond individual defect measurements. SPR and Prophet contribute more than half of the correct classifications while representing approximately one quarter of retained evaluations. Excluding them nearly eliminates the negative line count association reported in Table 6. These shares describe outcome concentration rather than contributions to a causal effect, but they show that the apparent relationship between reference repair extent and correctness depends on the techniques represented. Lin et al. 2022 incorporate contextual code information into patch correctness assessment, demonstrating an alternative to relying on isolated static features. The evidence here supports a related conceptual refinement: reference repair counts have limited meaning when detached from the project and technique contexts in which automated candidates are generated and assessed.

Outcome availability and repeated evaluations also change the substantive interpretation of correctness. Equation 1 yields different quantities when its denominator represents assessed generated outcomes, all generated outcomes, or equally weighted defects. The reversal of the small C and Java ordering under equal defect weighting demonstrates that benchmark comparisons can change without altering any underlying correctness label. Missing assessments further widen the range of correctness proportions compatible with the source records. These findings make denominator definition and assessment coverage part of the empirical result rather than administrative reporting choices. Ye et al. 2021 show that correctness assessments can differ across evaluators and that additional testing can expose classification errors. Together, these insights indicate that an interpretable benchmark result requires both traceable labels and explicit accounting for which outcomes receive assessment.

The international relevance concerns evaluation practices for widely used open source software rather than differences between national economies. Motwani et al. 2022 identify a small positive relationship between test suite size and patch quality while documenting differences across repair settings. The positive association in Table 6 adds a complementary perspective by showing its dependence on project and technique composition. Tables 4 and 5 consequently support reporting distributions and contextual comparisons alongside a headline correctness proportion. Benchmark maintainers should distinguish patch production from assessment availability, while tool evaluators should present results under both evaluation and defect weighting when defects receive unequal numbers of evaluations. These practices refine the search and validation perspective by making the composition of observed evidence explicit. The practical contribution is a more interpretable basis for comparing repair outcomes across projects and languages, with conclusions calibrated to the assessed sample.

7. Conclusion and Implications:

Trust in automated software repair depends on how success is measured and compared. This analysis shows that apparent relationships between repair demands, validation evidence, and assessed correctness change with the composition and weighting of benchmark evaluations. Its contribution is an empirical demonstration that pooled patterns can weaken, disappear, or reverse when influential evaluation groups are removed or repeated defects receive equal weight. This finding refines the search and validation perspective by making the structure of evaluation evidence part of the interpretation of repair outcomes. Simple measures of repair difficulty or testing effort cannot independently justify expectations about correctness across heterogeneous settings. Managers should therefore compare repair systems using documented assessment coverage, comparable defect sets, and explicit weighting rules before allocating verification resources. Benchmark maintainers should distinguish generated outcomes from assessed outcomes and preserve traceable classifications. Software assurance policies can incorporate these reporting requirements into procurement and evaluation protocols, supporting decisions about adoption without treating benchmark performance as deployment assurance. Operationally, teams should maintain separate records for candidate generation, test

acceptance, and correctness assessment. These practices provide a transferable basis for evaluating software used across organizations and jurisdictions, while supporting accountable decisions about where automated assistance requires further human verification before deployment.

The benchmark structure provides opportunities to test transferability through broader project coverage and comparable evaluations across languages. Independent reassessment of recorded classifications would strengthen measurement comparability, while completing unassessed outcomes would reduce selection uncertainty. Matched evaluations using identical defects and verification procedures could distinguish technique differences from sample composition. Repeated observations across software releases could examine whether descriptive relationships persist as systems evolve. Controlled changes to validation information could test the proposed behavioral constraints directly. These extensions would develop the present descriptive contribution into stronger evidence about mechanisms, without assuming that observed associations establish causal effects or universal deployment performance.

References:

1. Gao, X., Wang, B., Duck, G. J., Ji, R., Xiong, Y., & Roychoudhury, A. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. ACM Transactions on Software Engineering and Methodology, 30, issue 2, Article 14, 1-27. https://doi.org/10.1145/3418461
2. Gazzola, L., Micucci, D., & Mariani, L. 2019. Automatic software repair: A survey. IEEE Transactions on Software Engineering, 45, issue 1, 34-67. https://doi.org/10.1109/TSE.2017.2755013
3. Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., & Weimer, W. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Transactions on Software Engineering, 41, issue 12, 1236-1256. https://doi.org/10.1109/TSE.2015.2454513
4. Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. 2012. GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering, 38, issue 1, 54-72. https://doi.org/10.1109/TSE.2011.104
5. Le Goues, C., Pradel, M., & Roychoudhury, A. 2019. Automated program repair. Communications of the ACM, 62, issue 12, 56-65. https://doi.org/10.1145/3318162
6. Lin, B., Wang, S., Wen, M., & Mao, X. 2022. Context aware code change embedding for better patch correctness assessment. ACM Transactions on Software Engineering and Methodology, 31, issue 3, Article 51, 1-29. https://doi.org/10.1145/3505247
7. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., & Monperrus, M. 2017. Automatic repair of real bugs in Java: A large scale experiment on the Defects4J dataset. Empirical Software Engineering, 22, issue 4, 1936-1964. https://doi.org/10.1007/s10664-016-9470-4
8. Monperrus, M. 2018. Automatic software repair: A bibliography. ACM Computing Surveys, 51, issue 1, Article 17, 1-24. https://doi.org/10.1145/3105906
9. Motwani, M., Soto, M., Brun, Y., Just, R., & Le Goues, C. 2022. Quality of automated program repair on real world defects. IEEE Transactions on Software Engineering, 48, issue 2, 637-661. https://doi.org/10.1109/TSE.2020.2998785
10. Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods, 17, issue 3, 261-272. https://doi.org/10.1038/s41592-019-0686-2
11. Ye, H., Martinez, M., &Monperrus, M. 2021. Automated patch assessment for program repair at scale. Empirical Software Engineering, 26, Article 20. https://doi.org/10.1007/s10664-020-09920-w