Int. j. adv. multidisc. res. stud. 2026; 6(5):933-944

Received: 09-08-2026
Accepted: 19-09-2026

International Journal of Advanced Multidisciplinary Research and Studies

ISSN: 2583-049X

The Blind Spot of a Passing Suite: A Defect Taxonomy by Discovery Method in an Authoritative Records System

[1] Vernon Jones Mwaba, [2] Francis Hanyama

[1] Department of Computer Science, DMI- St. Eugene's University, Chibombo Campus, Lusaka, Zambia
Department of CSE-ICT, DMI- St. Eugene's University, Chibombo Campus, Lusaka, Zambia

Corresponding Author: **Vernon Jones Mwaba**

Abstract

Testing literature evaluates a suite by what it catches. This article evaluates one by what it cannot see. This article treats a test suite as a measuring instrument whose blind spot is a structural property rather than an accident, our study classifies every defect recorded during the development of the Catholic Diocese of Mansa Pastoral Management System, which is a sacramental records platform whose registers are, under canon law, the juridical proof of the sacraments, and not by the component in which each defect lived, but by the method that discovered it. Ten discovery methods and four severity classes are defined. Severity is anchored externally in canon law and in the Zambian Data Protection Act rather than by triage judgement, removing the usual circularity from severity coding. The system carries 1,840 automated tests; its guards were additionally subjected to 225 controlled mutations under a protocol naming the test obliged to fail before the guard was removed. Across 44 system defects the continuously executed suite accounts for five, while the methods in which nothing is executed account for 37. Of the recorded 24 defects capable of silently corrupting a canonical register, 22 were invisible to a green suite of over a thousand tests; the two it did catch were both newly introduced and both caught by structural assertions rather than by examples. We characterized six structural blind spots, including a documented, covered and unreachable guard that no coverage instrument could report as absent.

Keywords: Defect Taxonomy, Discovery Method, Mutation-Based Falsification, Test Oracle Problem, Authoritative Records Systems, Canon Law Informatics

1. Introduction

A passing test suite is the most widely trusted evidence in software engineering. It is generated automatically, it is cheap to repeat, and in most applications, it is the gate through which a change reaches production. It is also evidence of an unusual kind, because the instrument reports on itself. A suite that is silent about a class of behaviour is, at the moment of reading, indistinguishable from a suite that has examined that behaviour and found it sound. Green is the same colour either way. When we read the result, what we learn is that no assertion written so far has been violated and we would typically conclude that the system works. The gap between those two statements is the subject of our article.

That gap is not a failure of diligence, and it cannot be closed by writing more tests of the same kind. It is structural. A test can only observe what its harness instantiates, can only fail on a predicate somebody wrote, and can only witness the single execution schedule it happens to produce. Anything outside those three bounds is not merely untested; it is unobservable by that instrument, in the way that a thermometer is not merely inaccurate about pressure but blind to it. The useful question is therefore not how strong a suite is, but what shape its blindness has and whether that shape happens to coincide with the defects that matter most.

The established ways of interrogating a suite do not answer that question, because they were built to answer a different one. Coverage and mutation score are scalars: they compress the suite's competence into a single figure of merit and thereby discard exactly the structure under investigation here. The empirical literature on mutation testing concentrates on adoption and cost. Sánchez *et al.* (2022) [7], surveying its use in open-source practice, find the technique applied far more narrowly than the research literature implies, while work on test oracles has established that the difficulty of deciding what a test should assert remains the dominant open problem in the field. Taromirad and Runeson (2025) [9], synthesising 145 studies of assertions in

software testing, report that after the test oracle itself, test generation is the leading concern, and that proposed solutions are overwhelmingly evaluated on small prototype cases rather than in industrial settings. A parallel line of work has shown that the instrument is not even stable: Parry *et al.* (2022) [5], in their survey of flaky tests, document a literature devoted entirely to the problem of suites whose verdicts vary without any change to the system under test. Each of these tells us something about how far a suite can be trusted. None tells us which classes of defect a given suite is constitutionally unable to see.

Furthermore, the defect-classification tradition does not address it, even though at first sight it appears to. Orthogonal Defect Classification and its descendants classify a defect by what it was, that is, its type, its impact, the trigger under which it surfaced, and have proved durable enough that recent work automates the assignment of those attributes; Kumar *et al.* (2022) [4] classify defects by ODC impact category using machine learning over several thousand reports. This is valuable for reliability modelling, and the ODC trigger attribute is a genuine relative of what is proposed below. But the trigger describes the condition under which a defect manifested, and its vocabulary is drawn almost entirely from within an execution frame: workload, stress, recovery, configuration. It has no cell for a defect found by reading the database catalogue for a constraint that was never declared, by setting a specification beside the code it claims to describe, or by noticing that a suite will begin failing on a date certain for reasons having nothing to do with the software. A taxonomy built on the question *what was the defect?* cannot answer the question *which of our detection instruments is idle?* Those are different cuts through the same material, and only the second one audits the detection apparatus itself.

Our article takes the second cut. It classifies every defect recorded during the development of a working system by the method that discovered it, and then asks whether that method is independent of how severe the defect was. The setting is chosen deliberately. The system is an authoritative records system, one whose stored records are not a cache of some external truth but the artefact of record itself, with no upstream source from which a corrupted entry could be reconstructed. Such systems are common and under-theorised: land titles, civil registration, clinical records, academic transcripts, and the case examined here, the sacramental registers of a Catholic diocese.

The choice matters for a methodological reason beyond stakes. In most systems, defect severity is assigned by a triage board applying judgement, which makes any correlation between severity and discovery method partly an artefact of who did the classifying. In an authoritative records system governed by external law, severity is given from outside the project. Under the 1983 *Code of Canon Law*, the parish registers of baptism, marriage and death are the juridical instruments by which the sacraments are proved (CIC, 1983, can. 535 §1), and the baptismal register is the spine of the system: confirmation, marriage and holy orders are each notated back onto the baptismal entry (CIC, 1983, can. 535 §2; can. 895; can. 1122). A defect that silently severs a notation link is therefore not a data-quality nuisance to be scheduled behind feature work. It destroys the mechanism by which a later sacrament can be proved at all, and it does so invisibly, years before anyone requests the certificate that would reveal the loss. In parallel, the Data Protection Act No. 3 of 2021 classifies religious belief as sensitive personal data (Republic of Zambia, 2021, s. 2) [6], which makes every row in every one of these registers sensitive by definition rather than by degree. Severity, here, is not an opinion the researcher supplies.

The case system is the Catholic Diocese of Mansa Pastoral Management System (CDOM PMS), a sacramental records platform comprising 155 backend source files exposing 163 documented interface paths over 36 tables in a schema-per-parish arrangement spanning 33 database schemas, together with an offline-capable client. At the point of our analysis the system carried 1,840 automated tests, all passing. Beyond the suite, its guards were subjected to a falsification protocol comprising 225 controlled mutations, in which the test obliged to fail was named before the guard was removed; a stricter and more falsifiable arrangement than a conventional mutation score, which is satisfied by the death of a mutant at the hands of any test whatever.

Our contribution is fourfold. First, a discovery-method taxonomy of ten categories that is explicitly not confined to execution, and that can be applied retrospectively at negligible cost by any project that keeps a defect register. Second, a severity scale anchored in external law rather than internal judgement, which removes the usual circularity from severity coding. Third, an empirical cross-tabulation of 47 defects from a real system, showing that severity and discovery method are emphatically not independent. Fourth, a characterisation of six structural blind spots that accounts for the pattern, including a case, a guard that is documented as the operative rule, exercised by a passing test, and unreachable in every execution that no coverage instrument could have reported as missing, because from the coverage instrument's point of view nothing was missing.

Four questions organise and guide our study. RQ1: which methods actually discovered the defects in a system under continuous automated test? RQ2: are defect severity and discovery method independent? RQ3: what structural properties of the suite account for the defects it did not observe? RQ4: what does a name-the-test-first falsification protocol reveal about the evidential value of the suite's own green result? In Materials and methods, we set out the case, the taxonomy, the severity anchoring and the falsification protocol, and states the study's limits. In Results and discussion, we present the distribution, the cross-tabulation and the six blind spots, and argue the consequences for systems of this class. Finally in the last sections we conclude and offer three recommendations.

2. Materials and Methods

2.1 Research Design

The study is a single-case, instrument-centred empirical analysis conducted inside a design science research project. The artefact, a diocesan sacramental records platform, was built to solve a practical problem, and the design knowledge reported here is of the kind vom Brocke, Hevner and Maedche (2020) [10] describe as arising from the act of construction rather than from observation of a system already in service. The object of our study, however, is not the artefact. It is the detection apparatus that we used to establish the artefact's correctness: the automated suite, and the other methods that were applied alongside it. The system supplies the defect population; the analysis is about the instruments.

A single case cannot establish a distribution that generalises,

and no such claim is made. What a single case can do, when the population is complete rather than sampled, is establish existence and structure: that a particular pattern occurs, that it is not a sampling artefact, and that there is a mechanism which explains it. The defect register analysed here is the complete recorded population for the system, not a sample drawn from it, which removes selection bias at the cost of confining the result to one system.

2.2 The Case System

CDOM PMS serves the Catholic Diocese of Mansa in Zambia. Its core function is the maintenance of the sacramental registers, the *libri* of baptism, confirmation, first communion, marriage and the dead across the parishes of the diocese, together with the cross-parish workflows the registers require when a sacrament is celebrated away from the parish that holds the relevant baptismal entry.

The server component comprises 155 Python source files, organised into 41 application services behind 19 interface routers, exposing 163 paths. Persistence is PostgreSQL under a schema-per-parish multi-tenancy arrangement: 36 tables in total, of which 25 are diocese-wide and 11 are replicated per parish across 33 schemas, with the active schema selected from a validated token on each request. Schema state is managed by 44 ordered migrations. The client is an offline-capable mobile application, holding a local replica and reconciling it on reconnection.

That last property is not incidental to the analysis. An offline-capable client makes the device a second writer of record, and the invariants that protect a register must then hold across a partition rather than merely within a single transaction. Köhler *et al.* (2024) [3] treat precisely this problem, the enforcement of safety properties and invariants in local-first applications, and their framing is the one adopted here: an invariant that is enforced only at one of several writing sites is not enforced.

At the point of our analysis, the system carried 1,840 automated tests, all passing, exercising unit, integration and interface levels, and executed on every change.

2.3 Why an authoritative records system is a distinct class

Four properties distinguish this class from ordinary transactional software, and each of them changes what a defect costs. First, the record is the artefact of value rather than a derived view: the register entry is what proves the sacrament (CIC, 1983, can. 535 §1). Second, there is no upstream source of truth. A corrupted row in an order system can be rebuilt from the order; a corrupted baptismal entry cannot be rebuilt from the baptism, which is an event in the past attested by nothing else. Third, correction is itself regulated: a register is amended by annotation rather than overwrite, so a silent, undetected corruption is not merely a wrong value but a wrong value that the lawful correction procedure was never invoked to address. Fourth, retention is effectively permanent, which gives a latent defect an unbounded window in which to manifest, and means that the application in which an invariant is expressed will almost certainly be replaced several times over while the data it wrote survives.

The notation chain deserves a concrete description, because the S1 class is defined by reference to it and a reader outside the tradition will not otherwise see why a missing cross-reference is graver than a wrong one. When a person is baptised, an entry is made in the baptismal register of the parish where the baptism occurred (*CIC*, 1983, can. 877 §1). That entry, and not any central index, is thereafter the person's canonical identity. When the same person is later confirmed, very often in a different parish, sometimes in a different diocese, a second entry is made in the confirmation register there, and a notation must also be made back on the original baptismal entry (can. 535 §2; can. 895). The same obligation attaches to marriage (can. 1122) and to holy orders.

The consequence is that the baptismal entry accumulates the person's whole sacramental history, and the certificate issued from it is the instrument on which later decisions rest; whether a person is free to marry, most importantly. A notation that is never made does not produce an error. It produces a certificate that is internally consistent, correctly formatted, and silently incomplete. Such a certificate is worse than useless because it would look complete while concealing the fact of a marriage. That failure mode; complete-looking and wrong, discovered years later by the person who needs it to be right, is what the S1 class names, and it is why silence is written into the class's definition.

The fourth property deserves emphasis because it bears directly on where an invariant ought to live. A register outlives its software. Any rule enforced only in application code is a rule that expires when that code is retired, while the rows it was supposed to protect continue to accumulate under whatever writes them next.

2.4 Unit of analysis and construction of the defect register

Our unit of analysis is the defect: a discrepancy between the system's behaviour or structure and a requirement binding on it, whether that requirement originates in canon law, in statute, in the system specification, or in an internal invariant the design depends upon. Cosmetic divergences of style, and deliberate design compromises recorded as such, are excluded.

Defects were recorded contemporaneously in a resolution register maintained throughout construction, each entry carrying the observation, the analysis, the resolution and critically for this study, the circumstances under which it came to light. Entries open with a dated statement of that kind: *found by grepping for the renamed columns rather than assuming nothing read them*; *found by the first run of the suite*; *confirmed by reading*. Because these were written at the time of discovery and not reconstructed for this analysis, discovery-method coding rests on contemporaneous evidence rather than recollection.

The register holds 60 numbered entries, of which 31 are defects; the remaining 29 record a confirmation that something was checked and found sound, a design decision and its reasoning, or a ruling on an ambiguity. To these were added the six critical and 10 high findings of a structured review; five independent passes over security, persistence, canon-law correctness, the HTTP surface and test quality, in which, the review records, nothing was executed at all. The resulting population is 47: 44 defects in the system and three in the detection apparatus itself.

One boundary decision must be declared because it biases a row of the results. That review also carried a fourth band of findings, recorded as prose summaries grouped by reviewer rather than as individually numbered items, roughly three dozen observations across persistence, security, canon law,

interface design and maintainability. We excluded them here because they cannot be counted reliably: a summary sentence naming three related weaknesses is not resolvable into three defects without a judgement the original text does not support. The exclusion is not severity-neutral. That band is where most of the confidentiality findings sit: login timing that distinguishes account states, a second existence oracle reachable through a type error, a cross-parish audit control that raises before it writes, so the S2 count in Section 3 is an undercount, and no inference should be drawn from the sparseness of that row. The S1 and S3 rows are unaffected, because critical and high findings were enumerated individually. A placeholder for the excluded band would be worse than its absence, so none is supplied.

One coding rule governs the whole analysis and it is the principal defence against hindsight bias. A defect is credited to the earliest method that actually surfaced it, never to a method that could in principle have surfaced it. Almost every defect in the population is expressible as a test once it is understood; several were in fact given regression tests immediately after discovery. Crediting those to automated testing would make the suite appear to have found defects that it demonstrably did not find, and would render the central question unanswerable by construction. Where we wrote a test in response to a defect, the credit went to whatever prompted the test.

2.5 The discovery-method taxonomy

Ten discovery methods were defined, listed in Table 1. We deliberately did not confine the taxonomy to execution. Three of the ten; catalogue inspection, specification confrontation and temporal reasoning, describe methods in which nothing is run at all, and these have no counterpart in trigger-based classification schemes.

Table 1: Discovery methods

Code	Method	The defect was surfaced by
DM1	Pre-existing automated test	a test already in the suite turning red, with no change to the suite
DM2	Newly written test	the act of writing an oracle for an invariant nothing had asserted
DM3	Mutation-based falsification	removing a guard and observing that the named test did not fail, or failed elsewhere
DM4	Execution against production-equivalent infrastructure	running against the real database engine, where behaviour differs from the test harness
DM5	Code reading and structured review	a reader following a path through the source, with nothing executed
DM6	Schema and catalogue inspection	interrogating the database's own metadata for a constraint that was never declared
DM7	Specification–implementation confrontation	setting a specification, comment or docstring beside the code it claims to describe
DM8	Concurrency demonstration	deliberately constructing an interleaving of two operations
DM9	Temporal reasoning	reasoning about instants the suite has not yet reached
DM10	Field use	a user encountering it in service

We have to comment on two boundaries. DM3 is distinguished from DM1 and DM2 by what is mutated and what is observed: in DM3 we deliberately break the system and the suite is the object under test, so a DM3 finding is usually a defect in the detection apparatus rather than in the system. DM4 is distinguished from DM1 by the environment rather than by the technique: a defect is coded DM4 when the behaviour that exposed it is a property of the production database engine that the test harness did not reproduce.

2.6 Severity, anchored externally

We assigned severity by reference to external authority rather than by judgement of business impact, using the four classes in Table 2. The anchoring is what makes the central cross-tabulation meaningful: because we coded both severity and discovery method, any scale resting on subjective impact would leave the correlation open to the objection that severity was inferred from the difficulty of discovery. A canon or a statutory section cannot be adjusted to suit a finding.

Table 2: Severity classes and their external anchors

Class	Definition	External anchor
S1	A canonical register entry may be silently created wrongly, lost, altered, or severed from the baptismal entry to which it must be notated	*CIC* (1983) can. 535 §1–§2, 877 §1, 895, 1122: the register is the proof of the sacrament, and the baptismal entry its spine
S2	Sensitive personal data may be disclosed beyond its lawful audience, or a parish boundary crossed	Data Protection Act No. 3 of 2021, s. 2, s. 19; *CIC* (1983) can. 220, can. 535 §1
S3	Lawful work is blocked or corrupted loudly; the register survives, and somebody knows	Operational; no external instrument engaged
S4	No consequence for a record, a right or an operation	None

The S1 boundary is the one that carries our argument, so its test is stated explicitly: a defect is S1 if there exists an input the system accepts, or a state it can reach, in which a canonical register entry is created wrongly, lost, silently altered, or severed from the baptismal entry to which canon law requires it be notated, without any signal to the user or the administrator. Silence is essential to the definition. A defect that fails loudly is an availability defect; the register survives it.

2.7 The falsification protocol

A test that has never been observed to fail is not evidence about the system; it is an untested assertion about the system. To convert the suite's green result into evidence, each guard the design depends on was subjected to a four-step protocol: the test obliged to detect the guard's absence was named in advance; the guard was removed from the source; the suite was run and that named test was observed to fail; the guard was restored and the test observed to pass again.

Naming the test in advance is what separates this from a conventional mutation score. A mutation score counts a mutant as killed when any test fails, which is satisfied by a test that fails for an unrelated reason; a collapse in an unrelated fixture, an import error, a cascade from a shared factory. Requiring a specified test to fail turns each mutation into a refutable prediction about which assertion is doing the work. The protocol admits three outcomes. FALSIFIED: the named test failed and recovered, so the guard is genuinely defended. VACUOUS: the suite remained green with the guard removed, or failed somewhere other than the named

test, so the guard is not defended by the assertion believed to defend it. ERROR: the mutation could not be applied at all in every observed instance because the anchor text used to locate the guard did not occur exactly once in the target file, anchor counts of zero, two and five having been recorded.

The VACUOUS outcome is related to, but not identical with, the equivalent-mutant problem of classical mutation testing. An equivalent mutant is semantically indistinguishable from the original, so no test could kill it. Several vacuous results here arose instead from redundant defence: the guard was real and its removal did change behaviour for some inputs, but a second guard elsewhere refused the same inputs first, so a single-site mutation could not reach the difference. The distinction matters practically. An equivalent mutant is noise to be suppressed; a redundantly defended guard is a finding, because it means our project does not in fact know which of its two guards is load-bearing, and a later change that removes the other one will pass the suite.

In total we ran 225 mutations. The outcomes are reported in Section 3.4. Vacuous results were not counted as verifications; the corresponding invariants were recorded as asserted but unfalsified. Across the project's verification checklist of 206 rows, 114 had been carried through the full protocol at the time of analysis and 92 remained asserted but unfalsified, so that the difference between a verified invariant and a believed one stays visible.

2.8 Instruments and environment

The automated suite runs under pytest against the application's own service, schema and router modules. Two database environments were involved and the distinction between them carries much of our argument. The unit and integration suites run against an in-memory SQLite database constructed by the test fixtures. The production target is PostgreSQL 18; migration round trips and the schema-level findings reported below were executed against PostgreSQL across the diocesan schema and all 32 parish schemas, with rows seeded at an earlier migration so that every backfill had something to act on.

The falsification harness is a purpose-built script rather than an off-the-shelf mutation tool. It takes a case file naming, for each mutation, the source anchor to remove or alter and the test obliged to fail; it applies the edit, invokes the named test, records the outcome, restores the source and re-invokes. Its per-run record: mutation, test, output with the rule removed, output with it restored, and verdict, was appended to a log, and that log is the source of Tables 6 and 7. Using a bespoke harness rather than a standard mutation framework is a deliberate trade: it forfeits breadth, since it mutates only guards somebody has written a case for, in exchange for the named-test requirement that a mutation score cannot express. It also, as Section 3.4 reports, carried a defect of its own.

2.9 Analysis

We coded each defect on two axes, discovery method and severity, and the joint distribution tabulated. The central hypothesis, that severity and discovery method are independent, is evaluated by inspection of the contingency table rather than by a significance test. A formal test would be inappropriate on a complete population with small expected cell counts, and would in any case answer a question about sampling variability that does not arise when the population is not a sample. The claim advanced is the one available from a census: that the joint distribution is lopsided in a specific direction, that the lopsidedness is large rather than marginal, and that there is a structural mechanism which explains it and which also predicts the exceptions.

2.10 Threats to validity

Construct validity. Discovery method is recorded, not inferred, but we made the record, and the moment at which a defect is deemed to have been discovered is to some extent a judgement, that is, a suspicion formed while reading code and confirmed by execution could be coded either way. The earliest-method rule pushes such cases towards the non-test methods, which is the direction that favours our own thesis. Readers should treat the boundary between DM5 and DM4 as the softest in the scheme.

Internal validity. A single researcher coded both axes, and no inter-rater reliability statistic can therefore be offered. This is a genuine weakness and is not mitigated by the external severity anchoring, which constrains severity coding but not discovery-method coding. The mitigation actually available is transparency: the full register, the mutation log and the coding of every defect are retained with the project, so the coding can be audited and contested.

Internal validity, second threat. The register may under-record. Defects noticed and fixed within seconds of being introduced were generally not written down. Such defects are overwhelmingly of the kind that a compiler, a type checker or an immediately-run test catches, so their omission biases the population against the automated-test categories. The effect is to overstate our proposition in the low-severity bands, and readers should discount the S4 row of Table 4 accordingly. The S1 finding is less exposed, because a defect capable of corrupting a canonical register is not the kind that gets fixed in seconds and left unrecorded; but the claim there is about a ratio rather than an empty cell, and a ratio can move.

External validity. The system was analysed before general deployment, so the field-use category is necessarily empty and the population is a pre-release one. Findings therefore describe what the instruments found before users were exposed, which is the period of interest for our argument but is not the whole life of a defect population. More broadly, one system in one domain supports no claim about proportions elsewhere. The mechanism proposed in Section 3.5 is offered as the generalisable part, and it is a claim about the structure of test instruments, not about the diocese, the Catholic Diocese of Mansa.

Reliability. The falsification protocol was itself found to be defective during our study of the system, in a manner described in Section 3.4, and was corrected and re-run. That episode is reported rather than suppressed because it is a direct instance of the paper's subject: an instrument returning a confident reading that its own construction made unreliable.

3. Results and Discussion

3.1 What found the defects

Of the 60 numbered entries in the resolution register, 31 are defects. Adding the six critical and 10 high findings of the structured review gives a population of 47, of which 44 are defects in the system and three are defects in the detection apparatus itself, treated separately in Section 3.4. Table 3

gives the distribution of the 44 system defects by discovery method.

Table 3: System defects by discovery method

Code	Discovery method	Defects	% of 44	of which S1
DM1	Pre-existing automated test	4	9.1	2
DM2	Newly written test	1	2.3	0
DM3	Mutation-based falsification	1	2.3	1
DM4	Execution against production-equivalent infrastructure	1	2.3	0
DM5	Code reading and structured review	25	56.8	15
DM6	Schema and catalogue inspection	1	2.3	1
DM7	Specification–implementation confrontation	11	25.0	5
DM8	Concurrency demonstration	0	0.0	0
DM9	Temporal reasoning	0	0.0	0
DM10	Field use	0	0.0	0
	Total	44	100.0	24

The first thing to say about Table 3 is the thing most likely to be dismissed as an artefact, so it is worth stating precisely. The automated suite1,840 tests at the time of analysis, 1,107 at the time of the structured review, executed on every change without anyone deciding to execute it, accounts for five of the 44 defects, or 11.4%. The three methods in which nothing is executed at all: reading code, inspecting the catalogue, and setting a specification beside the code it describes, account for 37%, or 84.1%.

The obvious objection is survivorship: a suite's successes are invisible, because a defect a test prevents is never written down. The objection is correct and it is not a defence. It is true that the suite silently refused an unknown number of regressions, and nothing here measures that. But the population under study is not *regressions*; it is *defects that reached the register*, having survived the suite. What Table 3 measures is which instrument eventually caught what the suite had already let through, and on that question, survivorship cuts the other way: every one of these 44 defects is, by construction, one the suite did not prevent.

A second objection is that reading found more because more reading was done. This is closer to the mark, and it exposes what our argument is really about. The reading was episodic: five focused reviews over a period of weeks, undertaken because somebody decided to undertake them, answerable to no schedule and in no continuous-integration pipeline. The suite was continuous, automatic and mandatory. The asymmetry in Table 3 is therefore not that one instrument is better than another. It is that the project's *mandatory* instrument produced 11.4% of its findings and its *discretionary* instruments produced 84.1%, which means the correctness of this system rested on an activity that no process required and that could have been skipped entirely without any indicator turning red.

3.2 Severity and discovery method are not independent

Table 4 crosses discovery method with severity. Under independence, each method's findings would carry roughly the population's severity profile, that is, 24 of 44, or 55%, being S1.

Table 4: Severity by discovery method

Discovery method	S1	S2	S3	S4	Total
DM1 Pre-existing automated test	2	0	1	1	4
DM2 Newly written test	0	0	0	1	1
DM3 Mutation-based falsification	1	0	0	0	1
DM4 Execution against production-equivalent infrastructure	0	0	1	0	1
DM5 Code reading and structured review	15	1	8	1	25
DM6 Schema and catalogue inspection	1	0	0	0	1
DM7 Specification–implementation confrontation	5	0	2	4	11
Total	24	1	12	7	44

The pre-existing suite found two of the 24 register-destroying defects. Code reading found 15; specification–implementation confrontation found five; catalogue inspection and mutation found one each. Put the other way, 22 of 24 S1 defects accounting for 91.7% were invisible to a green suite of over a thousand tests, and were found instead by methods that a project under schedule pressure drops first.

Note what the table does *not* show, because overstating it would be the easiest error available here. The conditional severity of a DM1 finding is not low: two of the suite's four findings were S1, a rate no worse than code reading's. The suite is not biased towards trivia. It simply found very little, and what it found was concentrated in a particular place, described next. Nor should the S2 column be read as a finding: for the population-boundary reason declared in Section 2.4, most of the system's confidentiality defects sit in a band of summarised review findings that could not be counted item by item, and that column is an undercount of unknown size.

Table 5: The 24 register-destroying defects, and how each was found

Ref	Defect	Found by	Why the suite did not see it
C1	The commit path is not atomic and can burn a canonical reference	DM5	The harness passes one session object as both sessions, so the two-transaction split cannot occur
C2	The audit hash chain forks under concurrency, so the log is not tamper-evident	DM5	No concurrency anywhere in the suite: no second session, no gather, no threads
C3	A transfer confers care of souls immediately and never displaces the incumbent	DM5	The guard that was missing is a database index; absent, it executes nothing
C4	The cross-parish half of Can. 535 §2 is an honour-system flag	DM5	Tests asserted the flag was set, which it was; nothing asserted an annotation was written
C5	The mobile outbox carries no role guard, so an assistant may commit	DM5	No test asserts which authorisation guard is attached to which route
C6	The outbox submit action never submits and reports success	DM5	The response said ok, and the assertion read the response rather than the record's state
H2	Reference assignment rolls back the caller's transaction	DM5	A rollback on a shared session is unobservable when both sessions are one object
H4	The offline commit bypasses the review gate the online commit enforces	DM5	No test in the suite commits; the review gate is a property of committing
H5	A marriage commits with no baptismal provenance; Can. 1122 goes unmet	DM5	The absence of a check produces no failing path to assert on
H9	Nothing enforces one parish priest per parish; three docstrings assert it	DM5	The invariant is stated in three docstrings and expressed in no predicate
R9	A register enum column carried no CHECK constraint	DM6	A constraint that was never declared executes nothing
R11	No data moved from the dispensation column into the new permission column	DM7	The defect is in migrated content; no test inspects the values a migration leaves behind
R15	A rename silently changed four readers, three of which write canonical documents	DM5	The renamed readers still compiled and still returned a value; only the meaning changed
R16	The parish columns are mandatory in the specification and nullable in every migration	DM7	A nullable column accepts every value a test supplies, including the ones it should refuse
R20	A derived canonical reference is still accepted from the client	DM7	The client-supplied value was accepted, which is precisely what the test asserted
R22	Confirmation sponsors were optional on canonical grounds the diocese overrides	DM7	Mandatoriness was a diocesan ruling that had not yet been encoded anywhere to be tested
R31	A notification loop bound the subject's own name and never used it	DM5	The loop ran to completion; an unused binding changes no output the fixtures observe
R44	Four registers accept a minister's middle name that no column holds	DM1	Found schema-to-column parity test, on its first run against the new columns
R45	Nothing wrote the delivery column, so the cross-parish mailbox delivered nothing	DM1	Found integration tests of dispatch, once the delivery column was newly depended upon
R46	Cross-field rules enforced only on create are walked around by a PATCH	DM5	No test sent a PATCH that violated a cross-field rule, because the rule was believed to be global
R50	No priest anywhere could act on an out-of-diocese follow-up duty	DM5	Every test asked the inbox question the routing answers; none asked the question it does not
R52	The promotion recorder was never called by anything	DM5	Dead code is reached by nothing, and a test is a kind of nothing
R57	A status was asserted per register while its own docstring called it derived	DM7	The asserted value and the derived value agreed on every fixture the suite used
R60	The two-state date of birth diverged across registers; the refusal was unreachable	DM3	The refusal was unreachable, so the test covering it never reached the branch it named

3.3 The two the suite caught, and what they have in common

Earlier we asserted that the defects capable of destroying a canonical register were, without exception, not the ones a passing suite caught. The data refute the phrase *without exception*, and the refutation is more instructive than the original claim. Two S1 defects were caught by tests already in the suite.

R44; four registers accepting a minister's middle name that no column held, so the value was taken from the user and silently discarded, was caught by a schema-to-column parity test on its first run against the new columns. R45; nothing writing the column on which cross-parish delivery had just been made to depend, so that every notification declared itself undeliverable and the mailbox delivered nothing, was caught when the suite exercised dispatch after that dependency was introduced.

Both share two properties, and the properties are the finding. First, the defect was *new*: introduced by a change made into code the suite already exercised. Second, the assertion that caught it was *structural* rather than exemplary. R44's parity test does not check that a particular baptism is recorded correctly; it enumerates the fields of a schema, enumerates the columns of a table, and asserts the two sets correspond. It has no fixture and no example. It is closer to a type check than to a test.

The contrast with the 22 the suite missed is then sharp. Those were, with few exceptions, *old* defects sitting in territory the harness could not reach: two transactions where the harness had one object, a lock the harness compiled away, a concurrency the harness never produced, a constraint that was never declared and therefore never executed. A regression suite is, on this evidence, exactly what its name says, an instrument for detecting regression, the introduction of a new fault into ground already covered. It is not an instrument for establishing that ground is covered in the first place, and reading it as though it were is the error this article is about.

3.4 The suite examined: what falsification revealed

A suite that has never been observed to fail is a hypothesis, not a measurement. 225 mutations were run under the protocol of Section 2.7. Table 6 gives the outcomes.

Table 6: Falsification outcomes

Outcome	Runs	%	What it establishes
FALSIFIED	194	86.2	The named test failed with the guard removed and passed when it was restored. The guard is defended by the assertion believed to defend it.
VACUOUS-OR-BROKEN	20	8.9	The suite stayed green, or failed somewhere other than the named test. The belief about which assertion was load-bearing was wrong.
ERROR	11	4.9	The mutation could not be applied: the anchor text did not occur exactly once in the target file.
Total	225	100.0	Over 154 distinct mutations and 149 distinct tests.

The 194 falsified runs are the article's positive result and should not be lost in the argument: across 135 distinct tests, a named assertion was shown to fail when the guard it defends was removed and to recover when the guard was restored. That is a stronger statement than a mutation score, which is satisfied by any test failing for any reason, and it converts those assertions from belief into evidence.

The 20 vacuous runs are the more interesting half. Inspection shows them to divide into two kinds, and the division matters because the literature treats only one of them. Some are close to the classical equivalent-mutant problem: the guard removed made no observable difference to any input the suite supplies, as with a blank-reference check on a lookup that returns nothing either way. Others including four marriage guards concerning the Can. 1086 dispensation, the Can. 1124 permission, and the citation of a Catholic party's baptism were *redundantly defended*: the guard was real, its removal did change behaviour, but a second guard elsewhere refused the same input first, so no single-site mutation could reach the difference.

Table 7: The sixteen distinct vacuous mutations

Guard removed	Runs	Reading
A blank canonical reference stops short-circuiting the lookup	1	No input distinguishes
The same, from the baptism side: a blank reference is no longer a no-op	1	No input distinguishes
The citation format floor goes, and a bare '7' reaches the register	1	Redundantly defended
A bare number and a two-digit year both become acceptable citations	1	Redundantly defended
A Catholic party may marry without citing a baptism at all	2	Redundantly defended
A religion may be recorded as a Christian denomination	2	Redundantly defended
One marriage may assert a permission and a dispensation at once	2	Redundantly defended
An impediment is recorded as merely permitted rather than dispensed	2	Redundantly defended
Christian-name canonicalisation dropped: Mary stops agreeing with Maria	1	Unresolved
A diocese-scoped token reaches parish register routes	1	Unresolved
An age band is imposed on Can. 891, refusing adults confirmed at a visitation	1	Unresolved
A submission for a missing record falls through to a bare 404	1	Unresolved
A commit hook writes outside the transaction again	1	Unresolved
A mandatory parish field accepts an empty string	1	Unresolved
The list ceiling truncates eagerly, cutting lists that fit	1	Unresolved
An off-by-one in the ceiling drops a row from lists that fit	1	Unresolved
Total: 16 distinct mutations	20	

Eight of the sixteen remained unresolved at the time of analysis, and that figure is reported rather than tidied away because it is the honest state of the project's knowledge. Each unresolved row is an open question of the same form: the guard was removed, the system's behaviour ought to have changed, the named test did not notice, and nobody yet knows whether the cause is a second guard, an input the fixtures never supply, or a genuine hole. A conventional mutation report would have shown sixteen surviving mutants and invited exactly one interpretation, that is, write more tests which is the right response to perhaps half of them.

That is a finding about knowledge rather than about code. Two guards defending one rule is not a defect; believing that a particular one of them is load-bearing, when it is not, is. The consequence is concrete: a future change that removes the *other* guard will pass the suite, because the suite's evidence for that rule was never attributable to the assertion that we thought it was carrying. Our response was to record the redundancy in the test file itself rather than to count the invariant as verified, and of 206 checklist rows, 114 had been carried through the full protocol and 92 remained asserted but unfalsified.

Three findings in the population concern the detection apparatus rather than the system, and they belong here. H10: no test asserted which authorisation guard was attached to which route, so exchanging the guard on certificate approval for a weaker one left all 1,107 tests green. R54: an invariant's test passes with and without the guard it was written for, because no stored profile carries the value that would distinguish them. R59 is the sharpest: the falsification harness itself could report a true mutation as vacuous, because a same-length edit left compiled bytecode timestamps and sizes unchanged and a stale cache was loaded in preference to the mutated source. One case of eleven came back vacuous in a batch and falsified when run alone. The harness built to measure the suite's blind spot had one of its own, and it was found only because two runs of the same mutation disagreed.

3.5 Six structural blind spots

The pattern in Table 4 is not a coincidence requiring statistical explanation; it is a consequence of how the instrument was built. Table 8 sets out six structural

properties of the suite, each of which places a class of defect outside its range, together with the defects in this population that each accounts for.

Table 8: Six structural blind spots

Blind spot	Why the instrument cannot see it	Where it bit
B1 The harness is not the platform	The integration harness built one in-memory SQLite database. SQLite does not enforce VARCHAR length, compiles SELECT... FOR UPDATE away, and ignores foreign keys unless a pragma is set.	H1, H2, R55, and the referential-integrity failure of the appointment path, recorded as invisible for exactly this reason
B2 No oracle, no failure	A test can only fail on a predicate somebody wrote. An invariant asserted only in prose has no predicate anywhere.	H9, C4, R17, R57 invariants stated in docstrings and enforced nowhere
B3 Absence has no test	Tests exercise code that exists. A constraint that was never declared executes nothing, so no path reaches it and no assertion misses it.	R9, C3's missing partial unique index, and zero foreign keys across the five registers
B4 One schedule, one observation	A test observes the interleaving it happens to produce. The harness used one session, one connection and no concurrency at all.	C2, H2 both untestable as the suite was written
B5 The suite is dated	A suite fixes the future in literals. The verdict is a function of the date on which it is read, and that dependence is invisible while the date is early enough.	65 future-dated literals across 13 files, the nearest falling due on 30 September 2026
B6 Wiring is not behaviour	A guard can be correct, and tested, and attached to the wrong route. Both halves pass in isolation; only their correspondence is wrong.	C5, H10 swapping two route guards left all 1,107 tests green

B1 deserves the most attention because it is the most common and the most avoidable. The integration harness built one in-memory SQLite database and passed the same session object as both the parish session and the diocesan one. Four consequences were verified directly rather than inferred. No test in the suite commits: a search for a commit call across the whole test tree returns zero, everything running inside a single never-committed transaction. Row-level locking is compiled away, so deleting the lock clause from reference assignment entirely would not turn one test red. String length is not enforced, which is why a four-character column happily accepted the ten-character deanery codes that fail on PostgreSQL. And foreign keys are ignored unless a pragma is set, which is recorded in the project's own design notes as the reason an entire class of referential failure in the appointment path was, in that document's words, invisible.

C1 is worth tracing in full, because it shows how an ordinary arrangement becomes invisible. The commit path for each register acquires two database sessions: a parish session, whose search path reaches the parish schema, and a diocesan session for the shared tables. It commits the diocesan session first, sequence counter, search index, audit line, cross-parish notification and the parish session second, for the register row itself. Two sessions are two pooled connections and therefore two transactions. If the second commit fails, the first is already durable, and the resulting state is precise: sequence number N is consumed for a record that remains unapproved; the diocesan search index holds a row for an entry no register contains; the audit log asserts that a record was committed which was not; and a Can. 1122 notification has been dispatched to another parish about a marriage that is not in the book. The receiving parish will annotate its baptismal register on the strength of it.

The fixtures construct one session object and pass it as both the parish session and the diocesan one. There is no second transaction, so there is no second commit to fail; the divergent state is not merely untested but unconstructible. The engine's own docstring, meanwhile, states that if any part fails none of it happened and no sequence number is burned. Every instrument agrees the path is sound: the tests pass, the lines are covered, and the documentation confirms the property. The defect was found by three of five reviewers independently, from three different directions, reading.

The point is not that SQLite is a poor substitute for PostgreSQL, which everyone already knows. It is that the substitution converts a whole region of the system's behaviour from *untested* into *untestable*, and does so silently. An untested region can be found by a coverage report. An untestable one cannot: the lines execute, the assertions pass, and the coverage instrument reports the region as covered. Only 3 of 148 endpoints were exercised over HTTP at all, and those with authentication overridden, so the 2% figure understates the problem rather than describing it, because the remaining coverage was generated below the layer where the defects lived.

The suite's verdict is therefore a function of the date on which it is read, and that dependence is invisible for exactly as long as the date remains early enough at which point a green suite will turn red with no change to the system, and, worse, some of those literals encode calendar rules whose *correctness* will silently invert rather than fail. A suite is not only an instrument; it is an instrument with an expiry date that nobody prints on it.

3.6 Documentation as a defect surface

Five defects in this population share a shape: a document asserts an invariant that nothing enforces, and the assertion is believed. H9 is the plainest the rule that one parish has one parish priest is stated in three docstrings and expressed in no constraint, no guard and no test, while the function that depends on it raises an unhandled error the moment it is violated. C4's engine docstring promises that if any part of a commit fails, none of it happened and no sequence number is burned, which is the precise opposite of what the two-session commit path does. R17's column comment asserts a use that a search of the codebase shows does not exist. R57's status is described by its own docstring as derived and is in fact asserted independently in each register, free to diverge.

This is not a documentation-hygiene complaint. A docstring that describes an invariant is read by the next engineer as a statement about what the code guarantees, and is therefore load-bearing in exactly the way a comment is supposed not to be: it suppresses the question *is this actually enforced?* in the mind of the person best placed to ask it. Tan *et al.* (2024) [8] establish how ordinary and how durable this decay is, finding outdated code element references in 28.9 per cent of

the most popular GitHub projects at the moment of measurement and in 82.3 per cent at some point in their history, persisting for an average of 4.7 years. Their concern is references to elements that no longer exist; the variant here is more dangerous, because the prose is not stale it was never true, and nothing in the toolchain distinguishes the two cases.

3.7 The guard that cannot fire

One defect in the population deserves separate treatment because it defeats every automated instrument at once. R60 concerns a field that may be given in one of two mutually exclusive ways: a date of birth, or a declaration that the date of birth is unknown. The schema contained a guard refusing both at once, its docstring presented that refusal as the operative rule, and a test exercised it. Earlier in the same validation chain, however, a normalisation step silently set the unknown flag to false whenever a date was supplied so the contradiction the guard existed to refuse could never arrive at the guard.

Consider what each instrument reports. Coverage: the guard's lines execute, because the test reaches them by another path, so the region is covered. The test suite: green, and the test named after the rule passes. Code review: the guard is present, correct in isolation, and documented a reviewer checking that the rule is implemented finds that it is. Mutation of the guard itself: the guard's removal changes nothing, so the mutant lives, and in a conventional workflow this reads as an equivalent mutant and is suppressed as noise. Every instrument returns a clean reading, and the operative behaviour of the system is the opposite of its documented rule: a user who declares a date of birth unknown and also supplies one has the declaration silently overwritten in a canonical register.

What exposed it was mutating a *different* line the normalisation step and noticing that the register's behaviour did not change either. Two mutations, each individually vacuous, whose joint vacuity is only explicable if one dominates the other. This is the practical argument for treating vacuous outcomes as data rather than as noise to be filtered, and it is the reason the protocol's name-the-test-first requirement matters: a conventional mutation score would have recorded a surviving mutant and moved on.

3.8 Where an invariant has to live

A consistent mechanism underlies B2 and B3. An invariant can be expressed in prose, in application code, in a test, or in a database constraint, and these are not interchangeable. Prose is unenforced by construction. Application code enforces only for the writers that run it. A test enforces only for the inputs it supplies. A database constraint enforces for every writer, including the migration script, the correction procedure, the administrator at a console, and the application that replaces this one in fifteen years.

The case system's own figures make the gap concrete. The whole schema declares 11 foreign key constraints, 10 of them restricting and 1 cascading, with 0 set to null. Every one of them is in the diocese-wide schema. The five sacramental registers the tables whose contents are the juridical proof of the sacraments, replicated across 33 schemas carry zero between them. Their referential integrity is maintained entirely by application code, in a system whose client is explicitly designed to write while disconnected.

This is the point at which the local-first literature becomes directly relevant rather than merely adjacent. Köhler *et al.* (2024) [3] address the enforcement of safety properties and invariants in applications that write locally and reconcile later, and their framing that an invariant holding at one writing site is not an invariant describes the case system's exposure precisely. A rule enforced in the server's validation layer is not enforced against a device that queued an entry offline, nor against the reconciliation that applies the queue, unless it is re-expressed at each of those sites. R46 is the same observation in miniature and was found by reading: a rule enforced only on creation is a rule that a partial update walks around, because the update schema sees only the fields it was sent.

The recommendation that follows in Section 5 is therefore not a preference for belt and braces. For an authoritative records system it is a straightforward inference from retention: the data outlive the application, so an invariant expressed only in the application has a shorter life than the rows it protects.

3.9 Relation to prior work

The result complements rather than contradicts the mutation-testing literature. Sánchez *et al.* (2022) [7] find the technique applied narrowly in open-source practice; this study's experience suggests one reason beyond cost, which is that the conventional output a surviving mutant is ambiguous between a missing test, an equivalent mutant and a redundantly defended guard, and only the first is actionable as usually reported. Naming the obliged test in advance disambiguates them at negligible extra cost.

Against the defect-classification tradition the relationship is complementary rather than competitive. Kumar *et al.* (2022) [4] show that ODC attributes can be assigned automatically from defect reports with high accuracy, which is precisely the property a type taxonomy needs if it is to scale. Discovery method does not have that property and probably cannot: the report of a defect records what was wrong, not how somebody came to notice it, and no classifier can recover from the text what the text does not contain. This is an argument for capturing discovery method at the moment of discovery rather than inferring it afterwards the practice this case happened to follow, and the substance of the first recommendation. It also marks the limit of the proposal: a project that has not been recording the circumstances of discovery cannot reconstruct this analysis from its issue tracker, however large the tracker is.

The work on assertions reaches the same wall from the other side. Taromirad and Runeson (2025) [9] find the oracle to be the dominant problem across 145 studies. The blind spots catalogued here are, almost all of them, oracle problems wearing different clothes: B2 is the oracle problem stated directly; B3 is the oracle problem for properties of structures rather than executions; B6 is the oracle problem for the correspondence between two artefacts each of which is individually correct. What this case adds is that the missing oracles were not exotic. They were cheap structural assertions does every schema field have a column, does every route carry the guard its specification names, does every register table declare the references it depends on of exactly the kind that did catch the two S1 defects the suite found.

The review literature supplies the necessary caution against reading this article as an endorsement of review over testing.

Charoenwet *et al.* (2024) [2], analysing 135,560 review comments in two large projects, find that reviewers do raise security concerns across 35 of 40 weakness categories, but that 30 to 36 per cent of concerns raised were merely acknowledged and 18 to 20 per cent went unfixed amid disagreement. Review found a great deal here too; whether it would have found it a second time, under deadline, with a different reader, is precisely what cannot be claimed from one case. Reading is not a reliable instrument. It is an instrument whose blind spot happens to be nearly disjoint from the suite's, which is a different and more useful property.

Finally, Parry *et al.* (2022) [5] document the instability of the suite as an instrument. The present study adds a distinct failure mode to that picture: not a test whose verdict varies without cause, but a test whose verdict is stable, correct, and about something other than what its name says R53's clock guard, which searched for one spelling of a fault that was present in six places under another, is the clearest instance, and R60 the most consequential.

3.10 Transferability to other authoritative records

The diocesan setting is unusual only in that its governing law is canon law. The four properties set out in Section 2.3 the record as the artefact of value, the absence of an upstream source, regulated correction, and effectively permanent retention are shared by land title registries, civil registration of births, marriages and deaths, academic transcript systems, court records, and the clinical record. In each, an entry is proof rather than a convenient copy of proof; in each, a silent corruption is discovered years later by the person who needs the certificate; and in each, correction is a controlled procedure rather than an update statement.

Two consequences carry across. The first is that severity in such systems can be anchored externally, in the statute or instrument that makes the record authoritative, which makes the cross-tabulation proposed here reproducible by other researchers without importing the original project's judgement. The second is that the specific blind spot that dominated this case a test harness that substitutes an embedded database for the production engine, and thereby converts constraint enforcement, locking and referential integrity from tested into untestable is not a canon-law problem at all. It is the default configuration of a great many application test suites, and its effect is most severe precisely where the database is doing the most juridical work.

A caution belongs with the invitation. The proportions here should not be transported to those settings, and a registry whose harness runs the production engine, whose invariants are declared as constraints and whose suite provokes interleavings would be expected to show a quite different table which is the point. The taxonomy is proposed as an instrument for a project to turn on itself, not as a result to be cited in place of doing so.

3.11 What the study does not establish

Three limits bear repeating at the point where the conclusions are drawn. The proportions in Tables 3 and 4 are properties of one system's recorded population and should not be transported; a project with a PostgreSQL-backed harness and concurrency tests would have a different table, which is the entire recommendation. The system was analysed before general deployment, so the field-use row is empty by construction and the population describes what was caught before users were exposed rather than the whole life of a defect population. And one researcher coded both axes: the severity axis is protected by external anchoring, but the discovery-method axis rests on contemporaneous records interpreted by the person who wrote them. What survives these limits is not a distribution but a mechanism six structural reasons a test suite cannot observe certain classes of defect together with a demonstration that in at least one system those classes contained almost all of the gravest defects.

4. Conclusion

This study set out to characterise what a passing test suite cannot see, by classifying defects according to the method that found them rather than the component they lived in. Across 44 defects in an authoritative sacramental records system, the automated suite accounted for five findings and the methods in which nothing is executed accounted for 37. Of the 24 defects capable of silently corrupting a canonical register, 22 were invisible to a green suite of over a thousand tests.

The original conjecture that the register-destroying defects were *without exception* not the ones a passing suite caught is false, and its failure is the study's most useful result. The suite caught two, and the two are alike: both were newly introduced into ground the harness already reached, and both were caught by structural assertions over the code's own metadata rather than by examples. That is the shape of the instrument. A regression suite detects the introduction of new faults into covered ground, which is what its name claims and all it claims. The error is not in the suite. It is in reading its green result as a statement about the system as a whole, when it is a statement about the intersection of the system with the harness's reach and the harness's reach was narrowed, in this case, by substituting an embedded database for the production engine, one session object for two, and no concurrency for a system that runs offline on handsets and reconciles later.

The six blind spots in Table 8 are offered as the portable part of the finding. They are not a list of this project's mistakes; they follow from what a test is. A test observes what the harness instantiates, so any divergence between harness and platform is out of range. It fails only on a predicate somebody wrote, so an invariant living in prose has no failing path. It exercises code that exists, so a constraint never declared is unreachable by definition. It witnesses one schedule. It fixes the future in literals and therefore has an expiry date. And it checks artefacts, not the correspondence between them, so a correct guard attached to the wrong route passes twice over.

Two secondary results are worth carrying forward. First, the name-the-test-first falsification protocol produced information a mutation score cannot: of 225 runs, 194 converted an assertion from belief into evidence, while the 20 vacuous outcomes identified guards the project wrongly believed to be load-bearing a class that a conventional workflow discards as equivalent-mutant noise. Second, the instruments themselves carried defects, including the falsification harness, which could report a true mutation as vacuous because of a stale bytecode cache. An instrument that measures blind spots has one.

For authoritative records systems the practical consequence is specific. These systems hold data that outlive their

applications, admit no upstream source from which a corrupted entry could be rebuilt, and are governed by external law that fixes what a defect costs. In such a setting an invariant expressed only where the tests can see it is an invariant with a shorter life than the rows it protects and, in the system, studied here, the five registers that constitute the juridical proof of the sacraments declared zero referential constraints between them.

5. Recommendations

Three recommendations follow, ordered by cost. The first is nearly free, the second is a modest addition to an existing practice, and the third is a design constraint.

5.1 Record the discovery method alongside every defect

A defect register that records *what* was wrong supports reliability modelling. A register that also records *how it was found* audits the detection apparatus, and costs one column. Its value is in the empty cells: a method that has found nothing in six months is either unnecessary or not being applied, and no other artefact in a normal project will distinguish those two cases. Had the case system carried this column from the beginning, the concentration of severe findings in episodic, discretionary reading would have been visible months before it was at which point the reading could have been made mandatory rather than left to depend on somebody deciding to do it. Teams already running Orthogonal Defect Classification can add discovery method as a further attribute without disturbing their existing coding.

5.2 Falsify each guard against a named test, and keep the vacuous results

Before an invariant is recorded as verified, name the test obliged to detect its absence, remove the guard, watch that named test fail, restore the guard and watch it pass. Naming the test in advance is the whole of the discipline: it converts each check into a refutable prediction about which assertion is load-bearing, and it distinguishes a guard that is defended from one that merely coexists with a green suite. Vacuous outcomes should be retained and examined rather than filtered as equivalent-mutant noise, because a guard that is redundantly defended is a guard the project has misunderstood, and the misunderstanding will surface as a silent regression when the other defence is later removed. Where a vacuous result cannot be resolved, write the reason into the test file, so that the difference between a verified invariant and a believed one remains legible to the next reader.

5.3 Express canonical invariants as database constraints, not only as application guards

In an authoritative records system, every invariant on which the juridical value of a record depends should be declared in the schema as a foreign key, a unique or partial unique index, a check constraint, or a not-null declaration in addition to whatever the application enforces. Three arguments support this, and only the third is specific to records systems. A schema constraint holds against every writer, including migrations, corrections and the successor application; an application guard holds only for the writer that runs it, which in an offline-capable system is not the only one. A declared constraint is discoverable by catalogue inspection, so its absence can be detected mechanically, whereas the absence of an application guard is invisible to every instrument a project normally runs. And the records outlive the software: retention is permanent, the application is not, and an invariant written in application code expires with the code. The corollary for test harnesses is immediate a harness whose database does not enforce the constraints the production database enforces cannot observe this class of defect at all, and its green result should be read accordingly.

6. References

1. Catholic Church. Codex Iuris Canonici. Vatican City: Libreria Editrice Vaticana, 1983.
2. Charoenwet W, Thongtanunam P, Pham V-T, Treude C. Toward effective secure code reviews: An empirical study of security-related coding weaknesses. Empirical Software Engineering. 2024; 29(4):88. Doi: 10.1007/s10664-024-10496-y
3. Köhler M, Zakhour G, Weisenburger P, Salvaneschi G. Consistent local-first software: Enforcing safety and invariants for local-first applications. IEEE Transactions on Software Engineering, 2024. Doi: 10.1109/TSE.2024.3477723
4. Kumar S, Sharma M, Muttoo SK, Singh VB. Autoclassify software defects using orthogonal defect classification. In Computational Science and its Applications - ICCSA 2022 Workshops. Lecture Notes in Computer Science, vol. 13381. Cham: Springer, 2022, 313-322. Doi: 10.1007/978-3-031-10548-7_23
5. Parry O, Kapfhammer GM, Hilton M, McMinn P. A survey of flaky tests. ACM Transactions on Software Engineering and Methodology. 2022; 31(1). Doi: 10.1145/3476105
6. Republic of Zambia. The Data Protection Act No. 3 of 2021. Lusaka: Government Printer, 2021.
7. Sánchez AB, Delgado-Pérez P, Medina-Bulo I, Segura S. Mutation testing in the wild: Findings from GitHub. Empirical Software Engineering. 2022; 27(6):132. Doi: 10.1007/s10664-022-10177-8
8. Tan WS, Wagner M, Treude C. Detecting outdated code element references in software repository documentation. Empirical Software Engineering. 2024; 29(1):5. Doi: 10.1007/s10664-023-10397-6
9. Taromirad M, Runeson P. Assertions in software testing: Survey, landscape, and trends. International Journal on Software Tools for Technology Transfer. 2025; 27(1):117-135. Doi: 10.1007/s10009-025-00794-1
10. Vom Brocke J, Hevner A, Maedche A. Introduction to design science research. In vom Brocke, J., Hevner, A. and Maedche, A. (eds.) Design Science Research. Cases. Progress in IS. Cham: Springer, 2020, 1-13. Doi: 10.1007/978-3-030-46781-4