Volume 11, Issue 7, July – 2026 International Journal of Innovative Science and Research Technology
ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

MigBench: An Execution Certified Benchmark for Large Language Model Review and Repair of MongoDB Data Migrations

Ahmed F. Mohamed[1]

[1]Penny Software Arabia for Information Technology Company, Riyadh, Kingdom of Saudi Arabia

Publication Date: 2026/07/21

Abstract: Production data migrations run with write credentials, often while the application they serve continues to handle traffic, and their worst failure modes concern how they change data rather than whether the code runs. Language model reviewers are increasingly asked to gate such scripts, with little evidence about their reliability in this setting. This paper presents MigBench, a benchmark of 300 MongoDB migration scripts in which 100 are correct and 200 each contain exactly one defect from eight operationally defined categories. The dataset is generated deterministically from a single seed, and every label is certified by execution: each script runs against a disposable MongoDB replica set under five behavioral probes covering expected state and scope, repeated execution, a counter race against simulated live traffic, crash injection with an invariant across collections, and crash injection followed by resume. All 300 labels were confirmed by behavior before any reviewer ran. Four reviewer designs were then evaluated across two model tiers, along with a repair agent whose patches are executed through the same probes. Model tier mattered far more than architecture. The mid tier model reached an F1 of 96 to 97% in every configuration, its single call and hybrid variants were statistically indistinguishable (McNemar p = 1.0), and a multi-agent design roughly tripled cost, more than doubled latency, and lost recall on the one category that requires reasoning about interleaved execution. A weak verifier inverted the value of the hybrid design: the small tier model flagged 61 to 74% of correct scripts, and handing it static analysis candidates made it significantly worse than reviewing alone (p = 0.021). Detection and repair also came apart: every configuration found missing tenant isolation with perfect recall, yet only 8% of the attempted fixes passed the probes, because the correct tenant field name cannot be observed in the defective script. Reviews cost $0.17 to $1.62 per hundred scripts with 98 to 100% verdict agreement across repeated runs. The results suggest that agentic database tooling should invest less in reviewer orchestration and more in giving repair agents runtime access to schema and execution feedback.

Keywords: *Data Migrations; MongoDB; Large Language Models; Code Review; Automated Program Repair; Benchmarks; Fault Injection; Multi-Agent Systems.*

How to Cite: Ahmed F. Mohamed (2026) MigBench: An Execution Certified Benchmark for Large Language Model Review and Repair of MongoDB Data Migrations. *International Journal of Innovative Science and Research Technology*, 11(7), 513-522. https://doi.org/10.38124/ijisrt/26jul649

I. INTRODUCTION

A data migration is an unusual kind of program. It runs once, it holds write credentials to a production database, and it often executes while the application it belongs to continues to serve requests. Its failure modes are correspondingly unusual. A backfill that omits a tenant filter rewrites documents belonging to every customer on the platform. An update that increments a field without a guard corrupts data the moment an operator retries a job that timed out. A script that modifies two related collections without a transaction leaves them permanently inconsistent if the process dies between the two writes. Mistakes of this kind rarely break a unit test, because tests seldom rehearse retries, interruptions, or concurrent traffic. They surface weeks later, as support tickets.

Language model reviewers are moving into precisely this gap. Teams already gate pull requests with model generated reviews [1], [2], and agentic systems that promise to find and fix defects are an active research area [3], [4]. Whether these reviewers can be trusted with migration safety is a different question from whether they can fix general bugs, and it is difficult to answer with existing benchmarks. Most code review and repair datasets target functional correctness in application code, and many rely on a model's judgment somewhere in their labeling or scoring pipeline. For migration safety, ground truth of that kind is not enough. The properties being tested are behavioral, namely what happens on a second run, during a crash, or under concurrent writes, so the labels themselves should be established behaviorally.

MigBench is built on that principle. It contains 300 migration scripts for MongoDB written in the migrate-mongo

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

convention: 100 correct, and 200 that each carry exactly one defect from a taxonomy of eight categories drawn from production migration practice. No language model wrote, paraphrased, or labeled any script. The entire dataset derives from one integer seed, and every label is certified by execution. Each script runs against a disposable MongoDB replica set under up to five behavioral probes, including crash injection and a race against simulated application traffic, and the dataset ships only because all 300 labels were confirmed by that machinery. The same probes score the output of repair agents, so no model judges another model anywhere in the pipeline.

On top of the benchmark, four reviewer designs were evaluated, chosen to represent the main architectural patterns in current practice: a deterministic static analyzer, a single LLM reviewer, a multi-agent panel of three scoped specialists with a synthesis step, and a hybrid in which the static analyzer's candidate findings are verified and extended by one model call. Each LLM design runs on a small tier and a mid tier model of the same family, and a repair agent attempts to fix the defects the hybrid reviewer confirmed. Every provider call is logged to a spend ledger, comparisons use paired statistics, and stability is measured across repeated runs.

The paper makes three contributions:

MigBench, a deterministic, execution certified benchmark for MongoDB data migration safety, with eight defect categories, five behavioral probes, and labels confirmed 300 out of 300 before any reviewer ran (Section III).

A controlled comparison of four reviewer architectures across two model tiers, with bootstrap confidence intervals, paired McNemar tests, repeat agreement, and complete cost accounting (Sections IV to VI).

A repair study scored by execution rather than plausibility, with a failure analysis that separates fixes that fail for lack of information from fixes that fail for lack of skill (Section VII).

MigBench originated in engineering work on Qodara, a change management platform for MongoDB, where the question of whether a model should be allowed to approve a customer's migration arises daily. The benchmark itself is fully synthetic and self contained, and it shares no code or data with that platform.

II. RELATED WORK

➢ *Language Models for Code Review and Repair*

Automated code review has progressed from pretrained models fine tuned on review data [1], [2] to general purpose LLMs applied directly to diffs and pull requests. Repair research followed a similar arc: Defects4J established test based validation for classical program repair [5], and SWE-bench extended the idea to repository scale tasks resolved by agents [3], [4]. Execution based scoring also anchors the main code generation benchmarks, from HumanEval [6] to the hardened test suites of EvalPlus, which showed how easily plausible looking output passes weak checks [7]. MigBench borrows this execution first stance and applies it to a different artifact: not a failing test to make pass, but a state mutation program whose safety properties, scope, idempotency, atomicity, and resumability, are invisible to conventional test suites.

➢ *Multi-Agent Systems for Software Engineering*

Frameworks such as MetaGPT [8], ChatDev [9], and AutoGen [10] organize language models into cooperating roles, and ensembling work argues that adding agents helps on some tasks [11]. Recent analyses are more skeptical and catalogue the ways multi-agent pipelines fail in practice [12]. Controlled comparisons on a fixed task with paired statistics remain rare. The results here contribute a clean negative data point: on this benchmark, three specialists plus a synthesis step never beat one strong reviewer, and the synthesis step itself introduced the only recall losses observed at the mid tier.

➢ *Evaluation by Model Judgment*

Protocols that use a language model as the judge of another model's output [13] are convenient but carry documented biases [14]; a judge tends to reward output that looks right. The repair study in Section VII offers a concrete case where that tendency would have inverted a conclusion: many of the failed patches in this data are precisely the ones a judge would praise, with transactions, atomic operators, and guard clauses in all the right places, and a filter that matches nothing in the database.

➢ *Schema Evolution and Migration Safety*

Schema evolution has a long history in the database community, from the PRISM workbench for relational schema changes [15] to empirical studies of how schemas and application code evolve together [16]. For NoSQL stores, Scherzinger, Klettke, and Störl formalized schema evolution operations [17] and later surveyed data migration strategies [18], while practitioner literature codified migrations as database refactorings [19]. On the language model side, benchmarks such as Spider [20] and BIRD [21] evaluate query generation, and security work probes the vulnerabilities of generated SQL [22]. The artifact MigBench evaluates sits between these traditions: not a schema operation or a query, but an imperative program that mutates production data, reviewed for the failure modes operators actually fear. The concurrency and consistency hazards in the taxonomy echo long standing evidence that application level database code often lacks the coordination it needs [23], [24].

III. THE MIGBENCH BENCHMARK

Fig 1 shows the pipeline end to end: one seed produces the dataset, execution certifies the labels, and the same execution machinery later scores repairs.

➢ *Script Format and Defect Taxonomy*

Every script follows the migrate-mongo convention: a module exporting an id, a human readable description, and an asynchronous up(db, client) function that performs the

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

migration against version 6 of the MongoDB Node.js driver. The description is part of the reviewed artifact, not decoration. Several categories, tenant isolation above all, can only be judged against stated intent, which mirrors real review, where the pull request description tells the reviewer what the change is supposed to do. Table 1 lists the eight defect categories with their operational definitions and the probe that certifies each one. The taxonomy was assembled from failure modes that recur in production migration practice: writes that escape their tenant, deletes whose filter matches everything, updates that cannot survive a retry, counters allocated by reading a value and writing it back later, changes to two collections without a transaction, numeric coercions that silently destroy unparseable values, array updates that hit the wrong elements, and checkpointed batch jobs that advance the checkpoint before doing the work.

Injecting one known defect into an otherwise correct program is the central move of mutation testing [25], [26]. MigBench differs from classical mutation in that the injected faults are not syntactic mutants but semantic failure modes, each paired with a probe that demonstrates the failure at runtime.

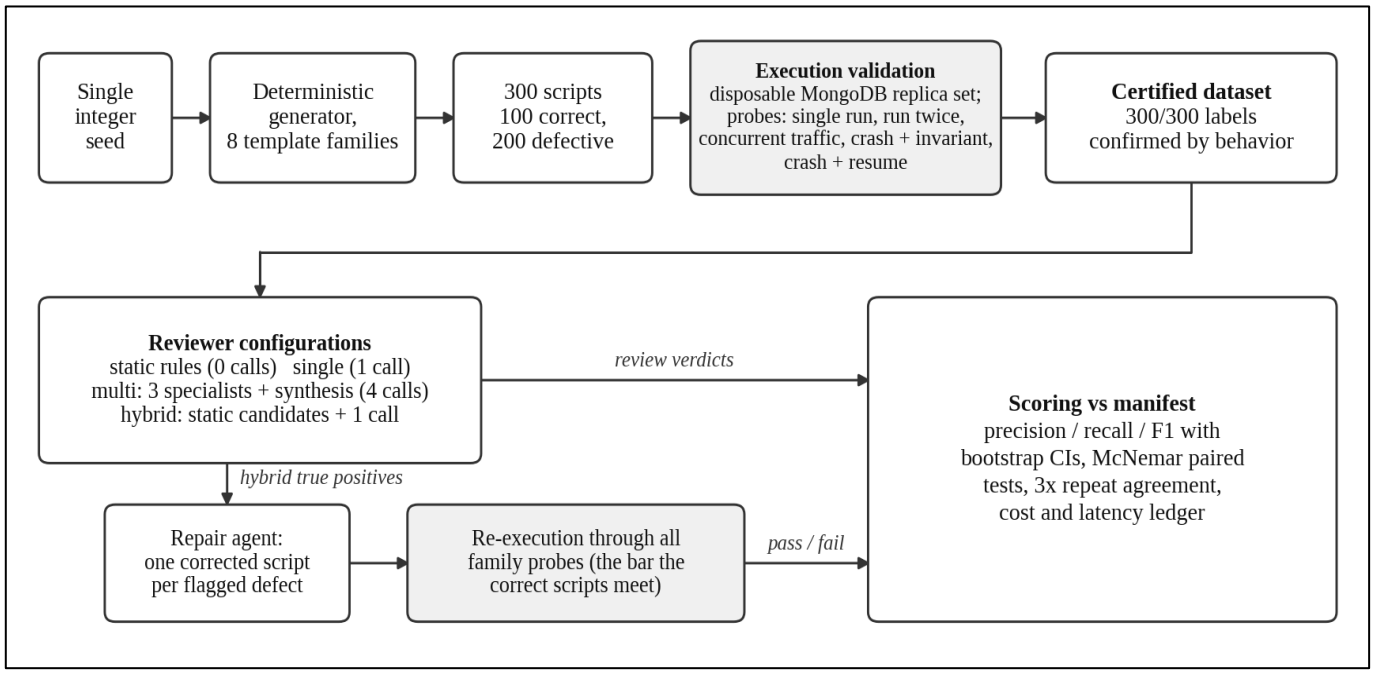

Fig 1 The MigBench Pipeline. A Seeded Generator Produces 300 Labeled Scripts; Execution Probes Certify Every Label Before Any Reviewer Runs, and the Same Probes Later Score Repair Attempts, So no Model Judges Model Output Anywhere in the Pipeline.

Table 1 Defect Taxonomy and the Probe that Certifies Each Category

#	Category	Operational Definition	Certifying Probe
1	Missing tenant isolation	Single tenant intent in the description, but the write filter lacks the tenant key, so documents of other tenants are modified	Single run scope check
2	Unbounded destructive operation	deleteMany with an empty or vacuous filter destroys documents outside the stated retention scope	Single run scope check
3	Non idempotent migration	Unguarded $inc or $push changes data again on every re-run	Run twice
4	Race prone counter allocation	Counter is read, incremented in application code, and written back later, colliding with concurrent allocations	Concurrent traffic
5	Partial multi collection update	Two business collections changed without a transaction; a crash between the writes breaks their shared invariant	Fault consistency
6	Data loss transformation	String to number conversion silently overwrites unparseable values instead of quarantining them	Single run on seeded invalid values
7	Unsafe array update	Positional $ or $[] update paths modify the wrong array elements instead of using arrayFilters	Single run per element check
8	Broken resume checkpoint	Checkpoint advances before the batch is applied; a crash followed by resume skips documents permanently	Fault resume

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

➢ *Deterministic Generation*

Each category has a template family that renders a correct and a defective variant from the same scenario parameters, so a defective script differs from its correct counterpart only in the injected flaw. All surface variation, including business domain, collection and field names, literal values, code shape, comment style, and one of six spellings of the tenant identifier, is drawn from a single seeded pseudorandom generator, which makes the dataset reproducible byte for byte from one integer. Ground truth lives only in a manifest written at generation time; the script files carry no markers that could leak a label to a reviewer. The composition is 100 correct scripts (12 or 13 per family) and 200 defective ones (25 per category), with sub variants inside several categories, such as empty versus vacuous delete filters and positional versus all elements array operators.

The six tenant key spellings deserve emphasis, because they turn a vague worry about missing context into a measurable quantity. The true key of each script (organizationId, tenantId, workspaceId, accountId, organization_id, or tenant_id) is observable only in the seeded data, never in the defective script. Section VII shows what that does to repair.

➢ *Ground Truth Certified by Execution*

The benchmark's credibility rests on the claim that labels reflect behavior rather than authorial intent. Every script executes against a freshly spawned local MongoDB instance, configured as a single node replica set so that multi document transactions are real, under the probes of its family:

Single run: one clean execution, after which the full expected end state is checked, including that documents outside the migration's scope (other tenants, non matching statuses, already migrated documents, non matching array elements) remain untouched. Defines categories 1, 2, 6, and 7.

Run twice: two sequential executions must produce the same end state, the classical idempotence requirement [27]. Defines category 3.

Concurrent traffic: the migration races a harness writer that plays the application, allocating ten documents from the same counter through atomic $inc operations; every issued sequence value must remain unique. Defines category 4.

Fault consistency: the database handle is a proxy that rejects the second write operation, simulating a crash between statements, and an invariant spanning the two collections is checked immediately afterwards. A transactional script sees the rejection inside withTransaction and aborts atomically. Defines category 5.

Fault resume: a crash is injected after the seventh write, mid batch, and the script is then re-run cleanly; every document must end up migrated exactly once. Defines category 8.

Crash injection as an oracle has precedent in fault injection research on data systems [28]. The validation gate requires every correct script to pass all of its family's probes and every defective script to fail exactly the probe that defines its category. The released dataset passes this gate 300 out of 300. As a second check, a stratified sample of 48 scripts (four defective per category plus two correct per family) was reviewed by hand and confirmed. Each validation or repair run spawns its own mongod process on a random port with a scratch data directory that is destroyed afterwards; by construction, the harness cannot be pointed at an external database.

IV. REVIEWER DESIGNS

All four designs receive the same platform context and must honor the same output contract, so the comparison isolates architecture. The context block states that the platform is a multi tenant SaaS application, that tenant identifier naming varies across collections, that the cluster supports multi document transactions, that bookkeeping collections for progress, quarantine, and counters are not tenant data, that scripts are judged against their stated description, and that migrations may be retried and may run under live traffic. The output contract demands strict JSON with a verdict of correct or defective plus findings drawn from the taxonomy, and instructs that a script is defective only when a finding has real production impact. A review that cannot be parsed counts as a verdict of correct with a recorded parse error, on the reasoning that an unusable review cannot block a migration. Table 2 summarizes the configurations.

Table 2 Reviewer Configurations

System	Calls	Description
static	0	Eight deterministic AST and pattern rules over the script source
single	1	Full taxonomy and context in one call
multi	4	Three scoped specialists in parallel plus one synthesis call
hybrid	1	Static candidates verified and extended by one call
repair	+1	Corrected script re-executed through all family probes

➢ *Static Analyzer*

Eight deterministic rules over the parsed source, one per category: single tenant intent in the description without a tenant key in the write filters, empty or vacuous deleteMany filters, unguarded $inc, $mul, or $push operations, counter reads followed by later writes, two business collections written without a transaction, numeric coercion with a silent fallback or missing validation, positional or all elements array update paths, and checkpoint writes that precede batch writes inside a loop. The analyzer makes no provider calls and costs nothing.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

➢ *Single Reviewer*

One call. The system prompt carries the full taxonomy and the context block; the user message carries the script.

➢ *Multi-Agent Reviewer*

Four calls. Three specialists run in parallel with narrowed mandates: data integrity (idempotency, lossy transformation, resume correctness), concurrency and atomicity (counter races, changes spanning collections), and tenant scope safety (isolation, blast radius, array targeting). Each returns findings only. A synthesis agent then sees the script together with all three reports, is told that specialists err in both directions, and issues the final verdict.

➢ *Hybrid Reviewer*

One call. The static analyzer's candidate findings accompany the script, and the model is told the analyzer is heuristic, produces false positives as well as misses, and that its job is to verify, discard, and extend.

➢ *Repair Agent*

One additional call for every true positive the hybrid reviewer flags. The model receives the script and the confirmed findings and must return a complete corrected script under an explicit safety contract: scoped to intent, idempotent, atomic where two business collections must stay consistent, resumable after interruption, free of read then write races, and lossless, with unparseable values quarantined rather than overwritten. The patched script then executes through all of its family's probes. Success requires passing every one, which is the same bar the 100 correct scripts meet.

V. EXPERIMENTAL SETUP

Models. The study uses two tiers of one commercial model family: claude-haiku-4-5 as the small tier and claude-sonnet-5 as the mid tier (haiku-4-5 and sonnet-5 in tables). No cross vendor comparison was run, and Section VIII discusses the resulting limits. Temperature 0 was requested for every call. The mid tier model rejects an explicit temperature parameter, which its API reports as deprecated for that model, so its calls ran at the provider default; run to run variability is therefore measured directly in Section VI. Mean token usage per script ranged from 871 input and 158 output tokens (small tier, single) to 4,182 and 781 (mid tier, multi-agent).

Scoring. Detection is scored at verdict level over all 300 scripts. Precision, recall, and F1 carry 95% percentile bootstrap confidence intervals from 2,000 resamples. Configurations are compared per model with exact two sided McNemar tests on paired verdict correctness [29], following standard practice for comparing classifiers on a shared test set [30]. Stability is measured as agreement across three repeated evaluations of a fixed 48 script subset. Every provider call is written to an append only ledger with token counts and list price cost; the whole study cost $17.96 across 4,784 calls. The mid tier model was billed at an introductory rate in effect through August 31, 2026, which is noted wherever cost appears.

One methodological detail deserves a warning label, because it silently distorted the first pass of this study. Reviews were initially capped at 700 output tokens. The mid tier model, which writes verbose JSON and cannot use assistant prefill, hit that cap on 6 to 15% of scripts depending on configuration. A truncated review fails to parse, a parse failure counts as a correct verdict, and so a token budget quietly turned verbose reviews into abstentions and depressed measured recall by 6 to 12 points. The affected rows were removed and re-run at a 1,500 token cap (3,500 for repairs); the small tier never approached the original cap, so its rows are first pass throughout. Benchmarks that pipe reviewers through token budgets should report truncation rates, because the artifact is easy to misread as model conservatism. After the fix, parse failures were rare: none for the small tier or for either single configuration, five for the mid tier multi-agent configuration, and three for its hybrid.

VI. RESULTS

➢ *Detection*

Table 3 reports the headline metrics and Fig 2 plots cost against F1. The mid tier model is accurate in every architecture: F1 of 96.9% (single), 95.8% (multi-agent), and 96.6% (hybrid), each with a 13% false positive rate on correct scripts and recall of 100%, 98.0%, and 99.5% respectively. The small tier reaches perfect recall in every configuration and pays for it, flagging 61 to 74% of correct scripts, which holds its F1 to 84 to 87%. In a review pipeline those two profiles behave very differently: a reviewer that misses nothing but blocks two thirds of legitimate migrations will not survive contact with an engineering team, while a 13% false positive rate is at least within the range a human gate can absorb.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

Table 3 Detection Performance Over 300 Scripts (200 Defective, 100 Correct); Bootstrap 95% CIs, 2,000 Resamples

Config	Model	Precision % [95% CI]	Recall % [95% CI]	F1 % [95% CI]	FPR %	Cat. Acc. %	Median Latency (ms)	Cost / 100 ($)
static	none	98.5 [96.6, 100.0]	100.0 [100.0, 100.0]	99.3 [98.3, 100.0]	3.0	100.0	~0	0.00
single	haiku-4-5	75.2 [69.7, 80.1]	100.0 [100.0, 100.0]	85.8 [82.2, 89.0]	66.0	70.0	2,192	0.17
multi	haiku-4-5	76.6 [71.4, 81.6]	100.0 [100.0, 100.0]	86.8 [83.3, 89.9]	61.0	73.5	4,972	0.64
hybrid	haiku-4-5	73.0 [67.7, 78.2]	100.0 [100.0, 100.0]	84.4 [80.8, 87.8]	74.0	100.0	1,669	0.17
single	sonnet-5	93.9 [90.5, 96.9]	100.0 [100.0, 100.0]	96.9 [95.0, 98.4]	13.0	95.0	3,661	0.52
multi	sonnet-5	93.8 [90.3, 97.0]	98.0 [95.7, 99.5]	95.8 [93.7, 97.6]	13.0	99.0	8,504	1.62
hybrid	sonnet-5	93.9 [90.3, 96.8]	99.5 [98.4, 100.0]	96.6 [94.6, 98.3]	13.0	99.5	4,159	0.62

Category accuracy is the share of true positives whose first reported category matches the injected defect. Costs use list prices in effect on July 11, 2026; claude-sonnet-5 was billed at an introductory rate (Section V). The static baseline shares an author with the dataset and is an upper bound (Section VIII).

The static baseline posts the best numbers of all, F1 99.3% with perfect recall at zero cost. That figure needs its caveat attached: the rules and the dataset share an author, so the number is an upper bound on template aware analysis, not evidence that static analysis solves the general problem. Section VIII returns to this point.

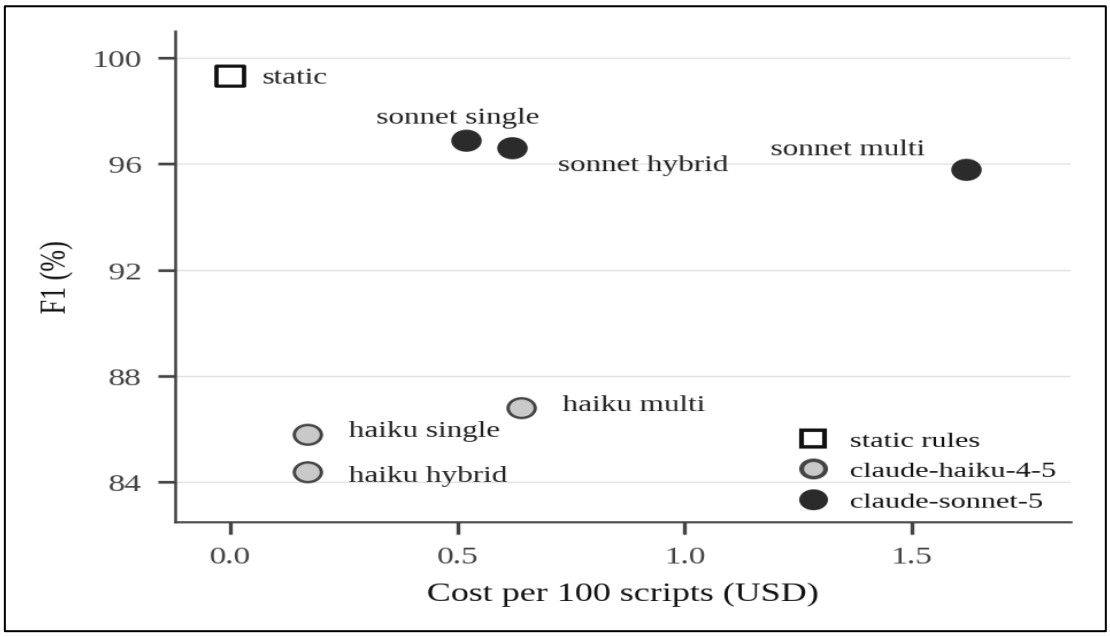

Fig 2 Review Cost Against F1 Over the 300 Script Benchmark. Marker Fill Distinguishes Model Tier. The Static Baseline Shares an Author with the Dataset and is an Upper Bound (Section VIII).

➢ *Architecture Effects*

Within the mid tier, architecture barely moves the numbers, and where it moves them it hurts. Single versus hybrid is statistically indistinguishable (one discordant pair, p = 1.0). Single versus multi-agent shows four discordant pairs, all favoring single, short of significance (p = 0.125), but the multi-agent configuration costs roughly three times as much ($1.62 versus $0.52 per hundred scripts), takes 2.3 times as long (median 8.5 versus 3.7 seconds per script), and is the only mid tier configuration that loses recall, dropping to 84% on the counter race category. The specialist reports in several of those cases were correct; the synthesis step overrode them. For the small tier the ordering turns into outright harm: hybrid is significantly worse than single (nine discordant pairs to one, p = 0.021) and worse than multi-agent (thirteen to zero, p < 0.001). Handed static candidates, the weak model confirms them and keeps inventing additional findings on clean scripts, 74 false positives against 66 for single. Verification is not a free lunch; a hybrid inherits the calibration of its verifier. Table 4 lists all pairwise tests.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

Table 4 Exact Two Sided McNemar Tests on Paired Verdict Correctness

Model	Comparison	Discordant	p
haiku-4-5	static vs single	66:3	<0.001
haiku-4-5	static vs multi	61:3	<0.001
haiku-4-5	static vs hybrid	74:3	<0.001
haiku-4-5	single vs multi	6:11	0.332
haiku-4-5	single vs hybrid	9:1	0.021
haiku-4-5	multi vs hybrid	13:0	<0.001
sonnet-5	static vs single	13:3	0.021
sonnet-5	static vs multi	17:3	0.003
sonnet-5	static vs hybrid	14:3	0.013
sonnet-5	single vs multi	4:0	0.125
sonnet-5	single vs hybrid	1:0	1.000
sonnet-5	multi vs hybrid	1:4	0.375

Discordant pairs a:b count scripts where the first configuration was right and the second wrong, and the reverse.

➢ *Category Structure*

After the token cap correction, every cell of the per category recall matrix is 100% except two: the mid tier multi-agent configuration at 84% and its hybrid at 96%, both on race prone counter allocation. That category is the only one whose defect is invisible in any single statement. The code reads a counter, computes in application space, and writes back later; no individual line is wrong, and the failure exists only under interleaving. It is also where the mid tier's one residual false negative appeared, a hybrid review that judged the pattern acceptable for a one off migration.

➢ *Stability*

Across three repeated evaluations of the 48 script subset, verdict agreement is 97.9 to 100% for every measured configuration, and agreement on verdict plus category is 95.8 to 100% (Table 5). The mid tier ran at the provider's default temperature, so this is not an artifact of forced determinism. At these prices an ensemble of three reviews per script would cost well under a cent per script, but the agreement rates suggest ensembling has little headroom to buy.

Table 5 Agreement Across Three Repeated Evaluations (48 Script Subset)

Config	Model	Verdict %	Verdict + Category %
single	haiku-4-5	100.0	95.8
hybrid	haiku-4-5	100.0	100.0
single	sonnet-5	100.0	97.9
hybrid	sonnet-5	97.9	97.9

➢ *Repair*

Under the full probe suite, repair succeeded on 66.3% of attempts for the mid tier (132 of 199) and 49.0% for the small tier (98 of 200). The mid tier attempted 199 rather than 200 because repair runs only on true positives its hybrid reviewer flagged, and that reviewer missed one race script. The aggregate hides the structure, which Table 6 and Fig 3 show: idempotency and checkpoint repairs succeed every time for both models, array targeting and multi collection repairs approach or reach 100% at the mid tier, while tenant isolation repairs succeed 8% of the time for both models and counter race repairs succeed at 0% and 8.3%. A seventeen point tier gap at the aggregate becomes, per category, a set of cliffs. Section VII walks through each one.

Table 6 Repair Success by Category (Succeeded / Attempted; Success Requires Passing All Family Probes)

Category	haiku-4-5	sonnet-5
Missing tenant isolation	2/25 (8.0%)	2/25 (8.0%)
Unbounded destructive operation	9/25 (36.0%)	14/25 (56.0%)
Non idempotent migration	25/25 (100%)	25/25 (100%)
Race prone counter allocation	0/25 (0.0%)	2/24 (8.3%)
Partial multi collection update	20/25 (80.0%)	25/25 (100%)
Data loss transformation	0/25 (0.0%)	14/25 (56.0%)
Unsafe array update	17/25 (68.0%)	25/25 (100%)
Broken resume checkpoint	25/25 (100%)	25/25 (100%)
Total	98/200 (49.0%)	132/199 (66.3%)

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

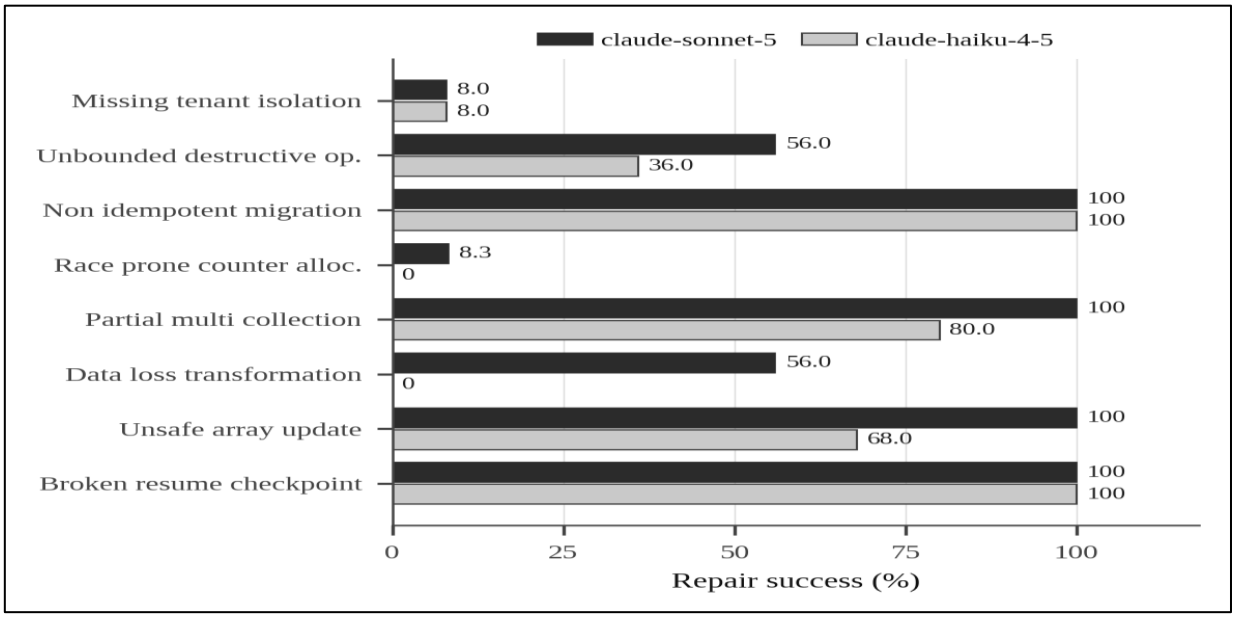

Fig 3 Repair Success by Defect Category Under the Full Probe Suite. Tenant Isolation and Counter Race Repairs Collapse for Both Tiers, for the Reasons Analyzed in Section VII.

VII. WHY REVIEWS AND REPAIRS FAIL

➢ *One Judgment Error Behind Every Mid Tier False Positive*

All thirteen false positives of the mid tier model, identically in the single, multi-agent, and hybrid configurations, are the same scripts: correct batched backfills flagged as data loss. Each computes String(doc.field).trim().toLowerCase() over documents selected by an existence filter. The model reasons that if the source field were missing, JavaScript would store the string "undefined". The hypothetical is coherent but sits outside both the seeded data and the taxonomy's stated bar of real production impact. The model reads the code correctly and misjudges materiality, a calibration failure at the taxonomy boundary rather than a perception failure. That the same thirteen scripts produce the same wrong call in three architectures is itself informative: adding specialists or static candidates does not touch judgment errors of this kind.

➢ *The Small Tier Ignores Stated Intent*

The small tier's dominant false positive mode is claiming missing tenant isolation on scripts whose descriptions state a platform wide maintenance intent, such as version bumps and counter normalizations that legitimately touch every tenant. It pattern matches on a bulk write without a tenant filter and does not consult the description. Its second mode misreads correct checkpoint ordering as broken. Both are failures to use context that was present in the prompt, and both explain why handing this model static candidates made it worse rather than better.

➢ *Tenant Isolation Repair Is Bounded by Information*

The defective script contains no tenant field, because that absence is the defect. The dataset draws the true key from six spellings, observable only in the seeded data, and the description calls the customer an organization. Both models, in every attempt, repaired with organizationId. Repair therefore succeeded on exactly the 2 of 25 scripts per model whose true key was organizationId and failed on the other 23, where the corrected filter matched zero documents. Detection recall on this category is 100% for every configuration: the models see the defect perfectly and cannot conjure a field name the script does not contain. No prompt fixes this. A repair agent needs schema introspection, the ability to inspect the database it is repairing, before this category becomes fixable. The probes also expose how quietly the failure presents: the patched script runs without error and modifies nothing, which a reviewer eyeballing the run could easily read as safe.

➢ *Race Repairs Fail on Driver Version Drift*

Eighteen of the mid tier's 22 failed counter repairs chose the correct construct, an atomic findOneAndUpdate with $inc, often inside a transaction, and then read the result as result.value.seq. That is the return shape of the MongoDB Node.js driver through version 5. Driver version 6, the version these scripts import, returns the document directly, so every probe run crashes with a TypeError. The small tier added missing counter documents and malformed calls on top of the same drift. This is version skew: patches written against the API a model knows best rather than the API in front of it, consistent with reports of models hallucinating or lagging library interfaces [31], [32]. The concept was right; the ecosystem had moved.

➢ *Data Loss Repairs and Prefix Parsing*

The mid tier's eleven remaining data loss failures added everything the safety contract asked for: quarantine collections, transactions, checks for NaN. They still convert the seeded European format price string "12,99" into 12, and then into 1200 cents, because parseFloat accepts numeric prefixes and the result passes a NaN check. Validation that is half right is the failure mode here, not validation that is absent.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

➢ *What Plausibility Scoring Would Have Hidden*

The failed tenant repairs look correct, with a well formed filter and the right update operator. The failed race repairs look exemplary, with transactions, atomic increments, and re-checked guards. A judge model grading these patches on inspection would have passed most of them [13], [14]. Only execution against seeded data exposed that one set modifies nothing and the other cannot run at all. Execution verified repair scoring is, in the author's view, the single most consequential design decision in this benchmark.

VIII. THREATS TO VALIDITY

Synthetic scripts. Template generated scripts with seeded surface diversity cannot reproduce the full mess of production migrations; defects appear one per script, and the scripts are short. Against this, the taxonomy comes from real practice, the scripts follow a real ecosystem convention, and every label is certified by behavior rather than by the generator's intent.

The static baseline shares an author with the dataset. Its rules are, in effect, aware of the templates, and its 99.3% F1 should be read as an upper bound rather than as a fair opponent for the LLM configurations. The defensible claim is narrower: where rules can be designed together with the defect classes, they dominate on cost and stability, and the value of a model begins where the defect surface cannot be anticipated by rules.

One model family. Two tiers of a single family leave vendor effects unmeasured; conclusions about architecture versus tier are within family.

Scoring judgment calls. Unparseable and truncated reviews count as abstentions and their rates are reported. Repair success requires the full probe suite, including a quarantine requirement stated in the repair contract; a laxer bar would raise the repair numbers.

Prompt sensitivity. One prompt set per architecture, untuned per model. Better prompting could move the absolute numbers, though it cannot supply the schema facts whose absence Section VII documents.

IX. CONCLUSION

For a well specified taxonomy of MongoDB migration defects, detection is close to solved at the mid tier: an F1 of 96 to 97% in every architecture, at $0.17 to $1.62 per hundred scripts, with 98 to 100% verdict agreement across repeated runs. Neither architectural pattern tested here improved on a single strong reviewer. The multi-agent panel roughly tripled cost, more than doubled latency, and introduced the only mid tier recall losses through its synthesis step, and the hybrid design turned actively harmful the moment its verifier was weak. If these results generalize, the practical advice is blunt: buy the better model before buying the more elaborate pipeline around it.

Repair is the open frontier, and the failure analysis says where the frontier runs. Tenant isolation repair fails for lack of information that no prompt can supply; the fix requires reading the schema. Counter race repairs fail on driver version drift; the fix requires executing the patch against the real dependency set. Data loss repairs fail on locale specific parsing; the fix requires running the code on the data it will face. All three point in the same direction. The affordances that made this benchmark's ground truth trustworthy, namely schema access, disposable execution, and behavioral probes for re-runs and interruptions, are exactly the affordances repair agents currently lack. Future work on agentic database tooling would do better to budget for those runtime capabilities than for additional reviewer orchestration, and MigBench will be extended toward schema aware repair and cross vendor comparison.

ACKNOWLEDGMENT

MigBench was developed in the context of Qodara, a change management platform for MongoDB with AI assisted governance features. The author thanks the platform's engineering team for discussions that shaped the defect taxonomy. The benchmark is fully synthetic and self contained; no production code, data, or customer information appears in it.

REFERENCES

[1]. Z. Li et al., “Automating code review activities by large-scale pre-training,” in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022. doi: 10.1145/3540250.3549081.

[2]. R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, “Using pre-trained models to boost code review automation,” in Proceedings of the 44th International Conference on Software Engineering (ICSE), 2022. arXiv:2201.06850.

[3]. C. E. Jimenez et al., “SWE-bench: Can language models resolve real-world GitHub issues?” in Proceedings of the 12th International Conference on Learning Representations (ICLR), 2024. arXiv:2310.06770.

[4]. J. Yang et al., “SWE-agent: Agent-computer interfaces enable automated software engineering,” in Advances in Neural Information Processing Systems 37 (NeurIPS), 2024. arXiv:2405.15793.

[5]. R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014. doi: 10.1145/2610384.2628055.

[6]. M. Chen et al., “Evaluating large language models trained on code,” arXiv preprint arXiv:2107.03374, 2021.

[7]. J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by ChatGPT really correct? Rigorous

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26jul649

evaluation of large language models for code generation," in Advances in Neural Information Processing Systems 36 (NeurIPS), 2023. arXiv:2305.01210.

[8]. S. Hong et al., "MetaGPT: Meta programming for a multi-agent collaborative framework," in Proceedings of the 12th International Conference on Learning Representations (ICLR), 2024. arXiv:2308.00352.

[9]. C. Qian et al., "ChatDev: Communicative agents for software development," in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), 2024, pp. 15174–15186. doi: 10.18653/v1/2024.acl-long.810.

[10]. Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversations," in Proceedings of the First Conference on Language Modeling (COLM), 2024. arXiv:2308.08155.

[11]. J. Li, Q. Zhang, Y. Yu, Q. Fu, and D. Ye, "More agents is all you need," Transactions on Machine Learning Research, 2024. arXiv:2402.05120.

[12]. M. Cemri et al., "Why do multi-agent LLM systems fail?" arXiv preprint arXiv:2503.13657, 2025.

[13]. L. Zheng et al., "Judging LLM-as-a-judge with MT-Bench and Chatbot Arena," in Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track, 2023. arXiv:2306.05685.

[14]. P. Wang et al., "Large language models are not fair evaluators," in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), 2024, pp. 9440–9450. doi: 10.18653/v1/2024.acl-long.511.

[15]. C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: The PRISM workbench," Proceedings of the VLDB Endowment, vol. 1, no. 1, pp. 761–772, 2008. doi: 10.14778/1453856.1453939.

[16]. D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2013, pp. 125–135. doi: 10.1145/2491411.2491431.

[17]. S. Scherzinger, M. Klettke, and U. Störl, "Managing schema evolution in NoSQL data stores," in Proceedings of the 14th International Symposium on Database Programming Languages (DBPL), 2013. arXiv:1308.0514.

[18]. U. Störl, M. Klettke, and S. Scherzinger, "NoSQL schema evolution and data migration: State-of-the-art and opportunities," in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), 2020, pp. 655–658. doi: 10.5441/002/edbt.2020.87.

[19]. S. W. Ambler and P. J. Sadalage, Refactoring Databases: Evolutionary Database Design. Boston, MA: Addison-Wesley, 2006.

[20]. T. Yu et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2018, pp. 3911–3921. doi: 10.18653/v1/D18-1425.

[21]. J. Li et al., "Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs," in Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track, 2023. arXiv:2305.03111.

[22]. X. Peng, Y. Zhang, J. Yang, and M. Stevenson, "On the vulnerabilities of text-to-SQL models," in Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2023. doi: 10.1109/ISSRE59848.2023.00047.

[23]. P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1327–1342. doi: 10.1145/2723372.2737784.

[24]. T. Warszawski and P. Bailis, "ACIDRain: Concurrency-related attacks on database-backed web applications," in Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, 2017. doi: 10.1145/3035918.3064037.

[25]. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," Computer, vol. 11, no. 4, pp. 34–41, 1978. doi: 10.1109/C-M.1978.218136.

[26]. Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, 2011. doi: 10.1109/TSE.2010.62.

[27]. G. Ramalingam and K. Vaswani, "Fault tolerance via idempotence," in Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2013, pp. 249–262. doi: 10.1145/2429069.2429100.

[28]. P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015. doi: 10.1145/2723372.2723711.

[29]. Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," Psychometrika, vol. 12, no. 2, pp. 153–157, 1947. doi: 10.1007/BF02295996.

[30]. T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning algorithms," Neural Computation, vol. 10, no. 7, pp. 1895–1923, 1998. doi: 10.1162/089976698300017197.

[31]. A. Eghbali and M. Pradel, "De-Hallucinator: Mitigating LLM hallucinations in code generation tasks via iterative grounding," arXiv preprint arXiv:2401.01701, 2024.

[32]. J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs," in Proceedings of the 34th USENIX Security Symposium, 2025. arXiv:2406.10279.