International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 8-No.3 (2022) pp. 124-147
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

Benchmarking Autonomy: A Taxonomy of Human-in-the-Loop Checkpoints for AI-Generated Transformation Code

Jagan Ankam*

AI Data Engineering, IT Consulting, CGI Technologies and Solutions Inc., Miami Lakes, Florida 33016, USA
* **Corresponding Author Email:** jagan.ankam@gmail.com - **ORCID:** 0009-0006-4716-1014

Article Info:

DOI: 10.22399/ijcesen.5462
Received : 15 May 2022
Revised : 25 June 2022
Accepted : 30 June 2022

Keywords

AI-generated code;
data transformation;
SQL;
Apache Spark;
dbt;
data quality

Abstract:

Generative code systems can produce executable SQL, Spark, and dbt transformations with substantially less manual effort, but syntactic validity does not establish semantic correctness. A transformation may compile, complete successfully, and preserve an expected schema while silently changing the meaning of downstream data. This article presents a controlled pre-production benchmark and a taxonomy of human-in-the-loop checkpoints for governing that risk. The benchmark contains 12 realistic transformation tasks spanning relational SQL, distributed Spark processing, and dbt-style analytical modeling. Nine candidates contain seeded defects and three are correct controls. Three automated checkpoint families are evaluated independently and in combination: structural schema checks, declarative data-quality rules, and differential comparison against a trusted baseline. Schema checks and data-quality rules each detect one of nine defects (11.1%), and their detections overlap. Trusted-baseline comparison detects seven defects (77.8%). The union of all automated checkpoints detects eight defects (88.9%), leaving one low-magnitude semantic error that preserves schema, distributions, row counts, and tolerance-bounded aggregate values. The residual defect is identified only when the transformation logic and business intent are reviewed together. The findings support three conclusions. First, validation depth matters more than check quantity: semantic oracles materially outperform purely structural controls. Second, checkpoint portfolios exhibit diminishing returns when multiple rules target the same visible symptom. Third, human review should not be treated as an undifferentiated manual gate; it should be targeted toward transformations with monetary, temporal, incremental, rare-category, or tolerance-sensitive semantics. The article contributes a five-level checkpoint taxonomy, a reproducible defect taxonomy, task-level detection evidence, and a risk-based escalation policy suitable for pre-production DataOps and analytics engineering workflows. The analysis is artifact-centric and does not rank named code-generation models because generation logs and repeated model samples were not part of the benchmark. Because the benchmark is intentionally small, the numerical estimates are reported with uncertainty and are not generalized as population-level industry rates.

1. Introduction

Data transformation code occupies a deceptively dangerous position in modern information systems. It rarely owns the source data and rarely presents the final user interface, yet it determines how raw events become financial metrics, operational indicators, model features, regulatory reports, and executive decisions. SQL queries, Spark jobs, and modular analytical models routinely encode joins, filters, window functions, incremental-loading logic, temporal alignment, currency normalization, entity matching, and business classifications. A defect in any of these operations may not terminate execution. Instead, the transformation can complete normally and produce output that is structurally valid, statistically plausible, and semantically wrong.

The distinction between execution success and semantic correctness is central to data quality. Data-quality research has long argued that accuracy alone is insufficient and that fitness for use includes contextual, representational, and accessibility dimensions [1]–[4]. In transformation pipelines,

that principle means that a table can satisfy its declared type contract while violating the business interpretation that gives the table value. A revenue model can preserve a decimal column but duplicate rows through a many-to-many join. A retention model can return the expected number of columns while shifting the cohort boundary by one day. A Spark window can complete without error while partitioning over the wrong entity. An incremental model can pass freshness checks while omitting late-arriving events. These failures are difficult because the output often resembles a legitimate production fluctuation.

AI-assisted code generation increases the importance of this distinction. By the end of 2021, code-oriented language models had demonstrated substantial capability in representation learning, translation, synthesis, and repair [25]–[30]. CodeBERT, GraphCodeBERT, and PLBART showed that models can learn useful syntactic and structural properties of programs [25]–[27]. Large generative models subsequently demonstrated that natural-language specifications can be mapped to executable code, while also revealing limitations involving long operation chains, variable binding, and output reasoning [28], [29]. These capabilities create obvious productivity benefits, but they also change the economics of code production: more transformation logic can be generated faster than teams can manually inspect it. If validation practices remain shallow, the bottleneck moves from writing code to establishing that generated code preserves intended meaning.

Existing engineering practice commonly relies on three classes of automated controls. Structural checks verify columns, data types, nullability, keys, and contracts. Data-quality rules inspect ranges, uniqueness, row counts, distributions, freshness, and declared business constraints. Behavioral or differential checks compare a candidate output with a trusted implementation, a previous production version, or an equivalent execution path. Each class is useful, but each depends on a different oracle. Structural checks answer whether the output has the expected form. Data-quality rules answer whether selected properties remain within accepted bounds. Baseline comparison answers whether two implementations produce materially equivalent results for a defined input. None automatically establishes that the specification itself is correct or that the selected tolerance reflects business risk.

Problem Statement

Data-transformation failures are difficult to govern because execution success is a weak correctness signal. A SQL query, Spark job, or dbt model may compile, finish within its service-level objective, preserve the declared schema, and produce statistically ordinary output while implementing the wrong join cardinality, reporting boundary, NULL rule, partition key, category mapping, or exchange-rate lookup. The resulting defect is semantic rather than operational: the pipeline is available, but the data no longer represents the stated business rule.

Organizations already deploy schema contracts, declarative data-quality tests, and output reconciliation. The unresolved issue is their actual coverage when transformation code is generated or heavily assisted by AI. A contract can verify types without detecting a wrong predicate. A distribution rule can accept a rare-category swap. A baseline can miss a latent defect when the fixture does not activate the affected path, or it can suppress a real divergence through an inappropriate tolerance. Counting passed assertions therefore does not show how many distinct defect mechanisms were observed.

The research problem is to determine how checkpoint families should be composed, how overlapping detections should be measured, and when a human decision adds information that the automated portfolio cannot encode. The required unit of analysis is the transformation task and its seeded defect mechanism, not the number of individual assertions. The required outcome is a release decision supported by traceable evidence: specification, fixture, reference behavior, checkpoint results, tolerance rationale, and accountable approval.

This article addresses the problem with a controlled pre-production benchmark containing 12 realistic transformation tasks: nine faulty candidates and three correct controls. Each task is evaluated by schema validation, data-quality rules, trusted-baseline comparison, the union of automated evidence, and contextual adjudication. The benchmark is intentionally small; its purpose is to expose coverage, overlap, and residual risk under a transparent design, not to estimate an industry-wide defect-detection rate.

The manuscript consistently uses the term pre-production benchmark. It does not present the results as production incident prevalence, a vendor-model ranking, or proof that human reviewers are infallible. Those questions require a larger corpus, preserved model-generation logs, repeated samples, blinded adjudication, and multiple organizations.

Research Gap and Contributions

Research on data quality, software testing, machine-learning operations, and human-AI interaction supplies many of the components required to govern generated transformations, but

the components are usually studied separately. Data-quality systems focus on observed datasets and recurring anomalies [8]–[12]. Database testing research develops oracles for defects in database engines rather than defects in application-level transformation intent [17]–[21]. Big-data testing work generates inputs and exposes faults in data-intensive programs [22]–[24]. Code-generation research measures functional correctness on programming benchmarks [28], [29], while human-in-the-loop research characterizes when human feedback and oversight can improve automated systems [31]–[34]. The literature does not provide a unified, task-level benchmark that compares structural, statistical, differential, and contextual checkpoints for AI-generated SQL, Spark, and dbt-style transformations.

A second gap concerns the granularity of human involvement. Human review is often recommended as a generic safeguard, but such advice does not specify what a person should review, when escalation is justified, or which automated evidence should accompany the decision. Mandatory review of every generated transformation does not scale and may become ceremonial. Conversely, unconditional automation ignores the oracle problem: tests cannot detect requirements they do not encode [13]. A useful framework must allocate human attention according to residual risk rather than treating human participation as a constant.

A third gap is methodological. Reports of "tests passed" often count assertions rather than unique defect mechanisms. Ten rules that all detect the same row-count anomaly do not provide ten independent units of protection. The relevant outcome is unique defect coverage and marginal coverage added by each checkpoint family. Mutation testing and controlled defect benchmarks offer a disciplined way to measure this distinction [14], [15], but this logic has not been widely applied to generated data transformations.

This article makes four contributions:

1. A five-level taxonomy that organizes structural, statistical, behavioral, contextual, and operational checkpoints according to semantic depth, automation potential, and human accountability.
2. A controlled pre-production benchmark of 12 SQL, Spark, and dbt-style transformations containing nine seeded defects and three correct controls, with defects selected to preserve executability while varying observability.
3. A task-level comparison of schema checks, declarative data-quality rules, trusted-baseline comparison, their automated union, and a risk-based human escalation step, reported using defect recall, false-alarm observations, overlap, marginal gain, and uncertainty intervals.
4. A practical escalation policy that concentrates human review on high-semantic-risk transformations, particularly monetary, temporal, incremental, low-frequency classification, and tolerance-sensitive logic.

The contribution is intentionally bounded. The study demonstrates that a deeper oracle can dominate a larger number of shallow checks in a controlled setting. It does not establish a universal 77.8% baseline-comparator effectiveness rate, nor does it claim that all human reviewers will detect all residual errors. Those claims require larger benchmarks, multiple code generators, multiple organizations, blinded reviewers, repeated trials, and production incident data.

2. Related Work

Data quality and production validation

Foundational data-quality research established that useful data quality is multidimensional. Wang and Strong organized consumer-oriented dimensions into intrinsic, conextual, representational, and accessibility categories [1]. Pipino, Lee, and Wang later described practical assessment techniques that combine subjective and objective measures [2]. Batini and colleagues compared methodologies for assessment and improvement across data types, quality dimensions, and process phases [3]. ISO/IEC 25012 formalized a data-quality model that distinguishes inherent characteristics from system-dependent characteristics [4]. These works explain why transformation validity cannot be reduced to schema conformance: the meaning of a value depends on task context.

Production ML research converted these principles into executable controls. Sculley et al. described hidden technical debt created by data dependencies, configuration, and changes in external systems [5]. The ML Test Score proposed actionable tests for features, data, models, and infrastructure [6], while TFX integrated validation and pipeline components at production scale [7]. Schelter et al. developed large-scale verification techniques and the Deequ library for declarative constraints over Spark datasets [8], [9]. Breck et al. proposed data validation that infers schemas and identifies anomalies in ML inputs [10], and Caveness et al. demonstrated continuous analysis and validation using TensorFlow Data Validation [11]. Auto-Validate extended automatic inference to domain patterns in data lakes and explicitly addressed false positives that otherwise demand repeated human intervention [12].

These systems are highly relevant but are primarily designed to detect anomalous datasets, schema drift, or violated properties. A transformation can

be wrong while producing a dataset that looks historically normal. For example, a category swap between two rare labels can preserve cardinality and distribution. A one-day lag in exchange-rate selection can remain inside aggregate tolerances. The present work therefore treats data-quality rules as one checkpoint family rather than a complete semantic oracle.

Test oracles, mutation testing, and differential execution

The difficulty of deciding whether program output is correct is known as the oracle problem [13]. Explicit expected outputs are expensive to construct, and many domains lack a complete specification. Mutation testing addresses test adequacy by injecting controlled changes and measuring whether the test suite distinguishes the mutated program from the reference [14]. Defects4J demonstrated the value of curated, reproducible fault corpora for empirical software-engineering studies [15]. Metamorphic testing reduces dependence on exact expected outputs by checking relations that should hold across transformed inputs or executions [16]. These ideas directly motivate the benchmark design in this article: defects are intentionally seeded, and checkpoints are evaluated by unique defect detection rather than by assertion count.

Database-system testing provides strong examples of behavioral oracles. Pivoted Query Synthesis constructs queries expected to retrieve a selected row [17]. NoREC compares an optimizable query with an equivalent form intended to suppress optimization [18]. Ternary Logic Partitioning checks relations among partitioned query results [19]. SQUIRREL combines language validity with coverage feedback to generate meaningful DBMS inputs [20], while SQLCheck detects and diagnoses SQL anti-patterns using query and data analysis [21]. Although these approaches primarily test database engines or query quality, they demonstrate that equivalent or reference executions can reveal logic defects that do not crash the system.

For data-intensive application code, BigFuzz uses framework abstraction and schema-aware mutations to test Spark programs efficiently [22]. DiffStream compares stream-processing implementations for differential output equivalence [23]. White-box techniques for analytics with complex user-defined functions expose faults that conventional input generation misses [24]. These methods strengthen the case for behavioral validation, but they generally require specialized generators or program models. The benchmark studied here evaluates a simpler control that is already available to many engineering teams: execute the generated transformation and a trusted reference over the same deterministic fixture, canonicalize the outputs, and compare them at row, key, aggregate, and tolerance levels.

AI-generated code and human oversight

Pre-trained code models established the technical basis for AI-assisted transformation development. CodeBERT learned joint representations of programming and natural language [25]. GraphCodeBERT incorporated data-flow structure [26], and PLBART supported program understanding and generation through denoising pre-training [27]. Codex demonstrated natural-language-to-code generation on HumanEval, but its analysis also identified weaknesses involving long chains of operations and variable binding [28]. Program-synthesis experiments with large language models showed that scale improves task success and that human natural-language feedback can materially reduce error [29]. Early work on Codex-based program repair further showed that generated patches may be useful but require evaluation against real bug benchmarks [30].

Human–AI interaction research provides design principles for exposing system limitations, supporting correction, and enabling efficient oversight [31]. Human-in-the-loop surveys describe how human knowledge can intervene at data, model, and system levels [32]. Enterprise AutoML research identifies a recurring tension between speed and oversight [33]. Data-cascade research shows that upstream data practices can compound into downstream harm when responsibility and visibility are fragmented [34]. These findings support a checkpoint model in which humans do not merely approve a final artifact; they resolve uncertainty at specific points where automated oracles lack context.

Table 1 synthesizes the closest research streams. The key unresolved intersection is the combination of generated transformation artifacts, silent semantic defects, comparative checkpoint effectiveness, and risk-based human escalation.

Research Questions and Hypotheses

The pre-production evaluation is organized around four research questions. The hypotheses are directional because the benchmark is small and the primary objective is measurement design rather than population inference.

1. RQ1: How effectively do structural schema checks, declarative data-quality rules, and trusted-baseline comparison detect seeded defects in executable AI-generated transformation code?

2. RQ2: How much unique defect coverage is gained by combining checkpoint families, after accounting for overlapping detections?
3. RQ3: Which defect characteristics are associated with residual false negatives after all automated checkpoints are combined?
4. RQ4: How can human review be positioned as a risk-based escalation checkpoint rather than a universal manual gate?

H1. Trusted-baseline comparison will detect a larger proportion of seeded semantic defects than schema validation or declarative data-quality rules used independently. The rationale is that a reference execution observes behavioral divergence even when the candidate preserves structural and statistical properties.

H2. The union of heterogeneous checkpoint families will provide higher defect recall than any individual family, but the gain will be smaller than the sum of individual recalls because some detections overlap.

H3. Residual false negatives will be concentrated among low-magnitude, context-dependent defects that remain within configured tolerances and preserve common data-quality indicators.

H4. A risk-based escalation policy can reserve mandatory human review for transformations whose semantics are poorly represented by automated oracles, reducing manual workload while retaining an accountable control over residual risk.

3. Methodology

Study design

The study uses a controlled, repeated-measures benchmark. Every transformation candidate is evaluated by each automated checkpoint family over the same deterministic input fixture. The unit of analysis is a transformation task, not an individual assertion. A task is counted as detected when at least one check in a checkpoint family produces an actionable failure attributable to the seeded defect. Repeated alerts for the same defect do not increase the detection count. Figure 3 summarizes the repeated-measures workflow and the separation between task construction, checkpoint execution, and adjudication.

The benchmark contains 12 tasks distributed evenly across three implementation styles: four relational SQL transformations, four Spark-style distributed transformations, and four dbt-style analytical models. Three tasks are correct controls. Nine contain one seeded defect each. Single-defect isolation was chosen to make detection attribution unambiguous, following the logic of controlled fault benchmarks [14], [15]. Multi-defect interactions are reserved for future work because one defect can mask or amplify another.

A correct reference implementation was created for every task before the faulty candidate was finalized. Reference outputs were materialized from deterministic fixtures and independently inspected. Seeded defects were selected to satisfy four criteria: the candidate must execute successfully; the defect must change intended semantics for at least one admissible input; the output must remain superficially plausible; and the defect must represent a recurring class of transformation failure rather than a language syntax error. The design therefore excludes compile failures, missing dependencies, malformed SQL, and infrastructure outages.

The study is model-agnostic. It evaluates artifacts representative of AI-generated transformation code but does not compare code-generation systems. This choice isolates the validation question from model-specific prompting and sampling effects. It also creates an important limitation: model-level defect prevalence cannot be inferred from the benchmark. A future replication should record model identifier, version, decoding settings, prompt, context window, and number of attempts.

Experimental model scope and artifact-generation protocol

The experiment is artifact-centric rather than vendor-centric. The unit being evaluated is an executable transformation paired with a task specification and a known oracle. This design isolates the paper’ s primary question - how well checkpoints detect silent semantic defects - from a different question: which code-generation model produces more or fewer defects. No model-level performance claim is made because the source material does not include preserved generation transcripts, model versions, or repeated samples.

Three execution models are represented. The SQL condition covers relational joins, predicates, grouping, and numerical projection. The Spark condition covers distributed partitions, windows, temporal conversion, and deduplication. The dbt condition covers modular analytical models, incremental predicates, slowly changing dimensions, category mappings, and reference-data lookups. These are execution technologies, not competing AI models; every checkpoint is applied to the resulting artifact regardless of how it was authored.

A defensible model-comparison replication should stratify at least three code-model families documented by the end of 2021: decoder-only generative systems represented by Codex-style models [28]-[30], encoder-decoder program models

represented by PLBART [27], and encoder or structure-aware representations such as CodeBERT and GraphCodeBERT [25], [26] when used in retrieval, ranking, or repair workflows. For every generated candidate, the replication should store the exact model identifier and date, prompt and system context, decoding temperature and sampling parameters, random seed where supported, candidate rank, post-generation edits, dependency versions, and execution-engine version.

The experimental factors should be separated. Model is a between-system factor; task is a repeated factor; generation sample is nested within model and task; checkpoint family is a repeated measurement on every artifact. At least five independent samples per model-task pair are advisable for a comparative study because a single generation cannot characterize stochastic behavior. The present 12-task results should therefore be read as checkpoint evidence over a controlled artifact set, not as a defect-rate estimate for any named model.

Candidate-generation conditions for a future model comparison

A model-comparison extension must distinguish raw generation quality from the effectiveness of the validation framework. Each task should be issued from one frozen specification containing the input schema, output contract, business rule, allowed libraries, and examples that clarify edge conditions without revealing the seeded mutation. The same specification must be presented to every model. Prompt wording, system instructions, retrieval context, and conversation history are experimental inputs and must be versioned rather than treated as incidental operator behavior.

Two generation conditions are useful. A deterministic condition uses greedy decoding or the lowest supported temperature to measure repeatability under a fixed prompt. A stochastic condition produces multiple independent candidates at a declared temperature and top-p setting to estimate within-model variation. The first executable candidate should be retained even when semantically wrong. Syntax repair, test-guided repair, and human editing should be separate treatment arms because allowing one model unlimited correction while evaluating another at first pass confounds generation with intervention.

For each model-task pair, the dataset should preserve the unmodified response, extracted code, compilation status, runtime result, number of repair attempts, and final executable artifact. Non-executable generations belong in a generation-quality analysis but not in the silent-error recall denominator, because this paper studies transformations that run and return plausible data. A minimum of five candidates per model-task pair provides only a basic view of stochastic variation; larger repeated samples are needed when model rankings are a primary claim.

Model families should be reported by architecture and release identity rather than by a generic label such as "LLM." A defensible 2021-bounded replication could include decoder-only code generation, encoder-decoder program generation, and retrieval or repair systems. The analytical comparison would then separate: executable rate, semantic-defect prevalence among executable candidates, defect-type distribution, checkpoint recall conditional on defect type, and the amount of human intervention required before release. This separation prevents a highly executable but semantically weak model from appearing superior merely because fewer outputs fail to compile.

Operationalization of the SQL, Spark, and dbt execution models

The three execution technologies represent different semantic failure surfaces. SQL tasks emphasize relational algebra: join cardinality, predicate boundaries, NULL behavior, grouping grain, type precision, and date-keyed reference lookups. Spark tasks add distributed execution concerns, including partition keys, window specifications, ordering, deduplication, timezone conversion, and behavior across repeated batches. dbt tasks represent modular analytics engineering, where model dependencies, incremental predicates, slowly changing dimensions, category mappings, and tests are configured across a directed acyclic graph rather than in one statement.

Cross-technology comparison is performed at the artifact-output level. Every candidate consumes a deterministic fixture and produces a canonicalized result with stable ordering, normalized timestamps, explicit NULL representation, and fixed decimal precision. Checkpoints are defined against that canonical result and against engine-specific contracts. This avoids rewarding one technology merely because it emits rows in a different order, while preserving genuine differences in keys, values, boundaries, and state transitions. Engine version, adapter version, dependency lockfile, and session settings should be recorded because apparently identical code can behave differently under changed defaults.

The benchmark does not claim that four tasks per technology represent the full error distribution of SQL, Spark, or dbt. The balanced allocation is a design control that prevents one execution style from dominating the aggregate count. Future studies should expand within each stratum and report checkpoint recall by defect mechanism before reporting a pooled average. A pooled

number is useful only when readers can see which task mix produced it.

Defect taxonomy and benchmark tasks

The seeded defects span structural, relational, predicate, null-handling, partitioning, temporal, incremental, classification, and tolerance-sensitive mechanisms. The taxonomy is intentionally tied to transformation semantics rather than language syntax. A join fan-out defect, for example, can appear in SQL, a Spark join, or a dbt model. A timezone defect can appear through SQL conversion functions or Spark timestamp APIs. This mechanism-oriented framing supports cross-language replication.

Table 2 lists the complete benchmark. Correct controls are included to observe whether a checkpoint rejects valid transformations. The three correct tasks cover an aggregation, a normalization/deduplication flow, and a slowly changing dimension-style model. The nine faulty tasks each alter one semantically meaningful operation. Input sizes range from 10,000 to 100,000 rows, large enough to express duplicates, rare categories, late arrivals, nulls, and temporal boundaries while remaining inexpensive to execute repeatedly in pre-production.

Interpretation. The task set is balanced across SQL, Spark, and dbt, but the unit of inference is the seeded mechanism, not the engine. The activation-evidence column identifies the fixture condition that must be present before any behavioral comparator can observe the error. T04 and T12 intentionally illustrate different limits: T04 is hidden by an uninformative fixture, whereas T12 produces a visible delta that is incorrectly accepted by a generic tolerance. Correct controls T01-T03 test false alarms; they do not establish a population-level specificity estimate.

Checkpoint taxonomy

The taxonomy contains five levels. Level 0 structural checkpoints validate schema, type, key, and contract compatibility. Level 1 statistical checkpoints validate observable dataset properties such as null rates, uniqueness, ranges, distributions, row counts, freshness, and selected business constraints. Level 2 behavioral checkpoints compare executions or enforce relations among outputs. Level 3 contextual checkpoints require a person to interpret transformation logic, requirements, exceptions, or business impact. Level 4 operational checkpoints control release, monitoring, rollback, and post-deployment reconciliation.

The benchmark directly measures Levels 0–2 and uses Level 3 to adjudicate the residual false negative. Level 4 is represented in the practical escalation policy rather than experimentally measured. This separation is deliberate. Approval and monitoring can reduce impact, but they do not necessarily detect a defect before release. Treating deployment controls as detection tests would conflate prevention, detection, and response.

The human checkpoints are not restricted to source-code inspection. Specification review can identify ambiguity before generation. Oracle review can determine whether a baseline and its tolerance are fit for purpose. Exception adjudication can distinguish true defects from acceptable drift. Pre-release approval can make risk ownership explicit. Post-release reconciliation can detect delayed effects. The appropriate checkpoint depends on what information is missing from automation.

Metrics and analysis

For checkpoint family j, defect recall is defined as $Recall_j = TP_j / (TP_j + FN_j)$, where TP_j is the number of faulty tasks correctly flagged and FN_j is the number of faulty tasks that pass. With nine faulty tasks, each task changes recall by 11.1 percentage points. Correct-control false-alarm rate is $FP_j / (FP_j + TN_j)$, but the three-control denominator is too small for a stable specificity estimate. The manuscript therefore reports the observed control outcomes without presenting them as a population rate.

Automated-union coverage is the number of unique faulty tasks detected by at least one automated checkpoint family divided by nine. Marginal gain for a checkpoint is the number of previously undetected tasks it adds to an existing portfolio. Overlap is reported at the task level. This prevents multiple rules triggered by the same visible symptom from being misinterpreted as independent coverage.Because the sample is small, 95% Wilson score intervals are reported for defect recall. Exact paired comparisons are interpreted cautiously. For baseline comparison versus either basic checkpoint, the task-level discordance yields an exact McNemar two-sided p-value of approximately 0.070. The observed effect is large, but the benchmark is underpowered for conventional significance claims. The analysis therefore emphasizes effect size, mechanism, and reproducibility rather than dichotomous significance testing, consistent with empirical software-engineering guidance [35], [36].

The final adjudication classifies every alert and every pass against the known benchmark oracle. A false negative occurs when a faulty task passes a checkpoint family. A false positive occurs when a correct control fails. A checkpoint that raises a non-actionable warning unrelated to the seeded defect is recorded separately and does not count as detection.

Operational definitions of human checkpoints

A human-in-the-loop checkpoint is counted only when a named person or role receives defined evidence, answers a defined question, and records a decision that can alter the transformation lifecycle. Merely placing an engineer in a meeting or requiring an approval click does not constitute meaningful human oversight. The checkpoint must have an information boundary and an action boundary.

At the specification checkpoint, the reviewer determines whether the requested transformation is complete enough to generate and test. Required elements include source and target grain, join cardinality, null semantics, time zone, reporting boundary, incremental behavior, accepted mappings, numerical precision, and the authoritative owner of each business rule. Ambiguous requirements should be returned before code generation because downstream tests cannot reliably reconstruct omitted intent.

At the generated-code checkpoint, the reviewer examines semantic operations rather than coding style. The review focuses on key-preserving joins, filter inclusivity, NULL behavior, window partitions, aggregation grain, deterministic ordering, timestamp conversion, late-arrival logic, incremental cutoffs, mapping exhaustiveness, and rounding or tolerance. A style-only review can approve code that is readable and wrong; therefore the checklist must be tied to transformation failure mechanisms.

At the oracle checkpoint, the reviewer establishes whether a reference implementation is independent and whether the comparison relation is appropriate. Exact equality is suitable for deterministic classification and date-keyed lookup. Set equality is suitable when row order is irrelevant. Aggregate tolerance may be suitable for approximate algorithms but not for exact financial rules. Metamorphic relations are useful when an exact reference is unavailable, for example requiring that adding an unrelated customer cannot change another customer's balance.

At the exception checkpoint, the reviewer interprets a mismatch or anomaly. The decision categories should be defect, approved requirement change, valid source-data change, reference defect, test-fixture defect, or inconclusive. This classification prevents teams from weakening thresholds merely to make the pipeline green. Every accepted exception should be time-bounded and linked to an owner.

At the release checkpoint, the approver accepts residual risk based on the evidence packet. Approval should be restricted to accountable roles for high-impact outputs rather than delegated to the author of the generated code. Separation of duties is especially important when the same person wrote the prompt, selected the baseline, configured the tolerance, and reviewed the result.

At the post-release checkpoint, humans assess reconciliation differences, user reports, and monitoring alerts. This layer is not a substitute for pre-production validation. Its function is to limit the duration and impact of defects that escape earlier controls, trigger rollback, and feed newly observed failure mechanisms back into the benchmark catalog.

4. Experimental Setup

Data fixtures and execution protocol

Each task uses a deterministic synthetic fixture generated from a fixed random seed. Fixtures contain entity keys, timestamps, monetary values, categorical codes, nulls, duplicates, and update timestamps as required by the task. The SQL tasks are expressed in ANSI-compatible syntax with minimal dialect-specific functions. Spark tasks use DataFrame-style transformations. dbt tasks are represented as modular SELECT statements with model configuration and incremental predicates. The benchmark is designed to be portable rather than optimized for one vendor platform.

Every candidate and reference transformation is executed from a clean state. Incremental tasks receive an initial batch followed by a second batch containing on-time and late-arriving records. Temporal tasks include records immediately before, at, and after midnight and reporting boundaries. Rare-category tasks use class frequencies below 1%, ensuring that broad distribution tests are unlikely to trigger. Monetary tasks include a fixture-specific whole-dollar case for T04 and small day-over-day rate changes for T12.

Outputs are canonicalized before baseline comparison. Canonicalization applies deterministic row ordering, consistent timestamp formatting, explicit null representation, and fixed decimal precision. Comparisons are performed at four levels: row/key set equality, column-level value equality, aggregate reconciliation, and tolerance-bounded numeric difference. The default relative tolerance for aggregate monetary values is 0.5%, reflecting a common but potentially unsafe practice of allowing minor numerical drift. T12 is designed to test the weakness created by that tolerance.

All three automated checkpoint families run independently. An alert from one family does not alter the input or configuration of another. This avoids cascade effects in which an early failure prevents later evidence from being collected. The automated union is computed after all runs. Human adjudication receives the task specification,

generated code, reference code, automated results, and a concise output-difference summary. The reviewer is asked whether the candidate preserves intended semantics and whether any configured tolerance is appropriate for the business variable.

Automated checkpoint configuration

Schema validation checks column presence, order-insensitive names, declared data types, nullability, key columns, and decimal scale. Data-quality validation checks required non-null columns, primary-key uniqueness, accepted categorical domains, amount ranges, row-count ratios, freshness, duplicate ratios, and selected aggregate constraints. Rules are intentionally generic enough to resemble pre-production controls that can be reused across pipelines. They are not customized after observing the seeded defect.

Trusted-baseline comparison executes the reference and candidate on the same fixture. It compares canonicalized outputs and reports the smallest unit of divergence available: missing or extra keys, differing rows, differing column values, or aggregate deltas. For high-volume outputs, exact row comparison can be replaced by partitioned hashes plus sampled row-level diagnosis, but the benchmark uses exact comparison because the fixtures are modest.

Table 3 specifies the controls and their expected blind spots. The blind spots are analytically important. Schema checks cannot observe a semantically wrong filter when output types remain unchanged. Distribution rules can miss compensating errors, rare-class swaps, and low-magnitude drift. A baseline can miss a latent defect if the fixture does not activate it, and it can normalize away a type difference. A tolerance can also convert a real semantic divergence into an accepted pass.

Decision thresholds, model metadata, and oracle independence

Checkpoint thresholds are fixed before outcome inspection. Structural checks use exact decisions for column identity, declared type, nullability, key definition, and decimal scale. Data-quality rules use explicit thresholds recorded with the rule. Baseline comparison uses exact equality for business keys, categorical mappings, date assignments, and deterministic monetary lookups; tolerance is permitted only where the specification defines a genuinely approximate quantity. The T12 scenario deliberately violates this principle by applying a 0.5% aggregate tolerance to an exact date-keyed rate.Oracle independence is a design requirement. The reference should not be generated by copying or lightly rewriting the candidate because shared assumptions create correlated failure. Independence can be achieved through a separately authored implementation, a legacy production path, a declarative specification compiled by a different mechanism, a metamorphic relation, or a set of independently computed invariants. When none of these is credible, the architecture must mark the oracle as weak and route the change to a Level 3 checkpoint.

Model metadata belongs in the evidence packet even though the primary analysis is model-agnostic. A release record should identify whether the artifact was generated, completed, repaired, translated, or manually edited after generation. It should also preserve the model/version, prompt, sampling parameters, and candidate rank. These fields allow later analysis to distinguish failures introduced by the model from failures introduced by prompt ambiguity, human editing, dependency changes, or the execution engine.

The escalation policy is deterministic at the process level. Human review is mandatory when no independent oracle exists, when a mismatch affects a critical field, when acceptance depends on a tolerance not explicitly justified by the specification, when the transformation introduces new temporal or incremental semantics, or when the downstream impact is monetary, regulatory, or difficult to reverse. The reviewer may approve, require correction, classify the reference as defective, or request a stronger fixture; the decision and rationale are retained.

Framework application. The levels are cumulative rather than interchangeable. L0 prevents contract violations; L1 covers declared statistical and business constraints; L2 compares behavior with an independent oracle; the automated portfolio deduplicates overlapping alerts. L3 is invoked by a policy condition - weak oracle, unjustified tolerance, high business impact, or novel temporal/incremental logic - and must record a disposition. L4 does not replace pre-production detection: it limits exposure through canary execution, reconciliation, and rollback. Traceability connects every decision to the prompt, model, code, fixture, oracle, thresholds, and approver.

Reproducibility controls

The benchmark design includes reproducibility controls that should be retained in any implementation package: fixed input seeds; immutable task specifications; reference and candidate code versioning; environment capture; stable canonicalization; machine-readable check configurations; and a task-level outcome manifest. A checksum should be recorded for every fixture and output artifact. Repeated execution should produce identical results.

To prevent benchmark leakage, check rules are defined before the task outcomes are inspected. A replication involving an AI generator should also freeze the prompt and generation settings. Multiple samples from the same prompt should be treated as clustered observations rather than independent tasks. If human reviewers are evaluated, the review packet, time limit, reviewer expertise, and order of tasks should be controlled because learning and fatigue can alter performance.

The study does not use proprietary production data. This avoids privacy and access barriers but reduces ecological realism. The fixture generator is intended to capture relational and statistical properties relevant to the defect mechanisms, not to reproduce the full complexity of an enterprise data estate.

Dataset availability. The supplementary workbook and CSV package provide the authoritative task catalog, seeded-defect descriptions, checkpoint-outcome matrix, metric calculations, sensitivity scenarios, and data dictionary. The package does not contain proprietary production data. It also does not claim to reconstruct undisclosed raw execution logs or source code beyond the task-level specifications reported here. A strict independent replication must therefore reimplement the transformations and fixtures from the benchmark specification and should publish those executable artifacts.

Proposed architecture and oracle implementation patterns

Governance and specification plane

Figure 6 defines the proposed architecture. The proposed architecture begins with a governance plane. The immutable task specification records source and target grain, key semantics, temporal boundaries, null handling, mappings, precision, incremental behavior, and rule ownership. Semantic-risk metadata identifies monetary, regulated, temporal, identity, access-control, rare-category, or tolerance-sensitive logic. Prompt, model, code, fixture, and reference lineage are stored as versioned assets so that an approval can be reconstructed.

Dual execution and oracle plane

The execution plane performs isolated dual execution. The AI-generated candidate and an independently constructed reference operate on the same deterministic fixture in a clean environment. When a full reference is unavailable, a portfolio of metamorphic relations and invariants substitutes for direct equality. The reference should favor simplicity and independent reasoning over performance: a complex generated Spark job may be checked against a bounded SQL or Python implementation if that reduces correlated logic.

Layered checkpoint and evidence plane

The checkpoint engine separates structural, statistical, behavioral, and impact evidence. Contract-as-code checks schema, grain, key uniqueness, and precision. Data-quality rules assess nulls, domains, ranges, freshness, and selected aggregates. The behavioral comparator evaluates missing and extra keys, row-level differences, exact fields, tolerance-governed fields, and aggregate reconciliation. Impact analysis then identifies affected consumers and materiality without using impact as a substitute for defect detection.

Risk policy and human-decision plane

The evidence and decision plane packages the specification, lineage, fixture checksum, contract results, data-quality results, exact mismatching keys, value deltas, tolerance metadata, and impacted metrics. A policy engine applies risk tags and checkpoint outcomes. Level 3 reviewers answer bounded questions: Is the specification complete? Is the oracle independent? Is the tolerance appropriate? Is the mismatch a defect, an approved requirement change, a fixture issue, or an oracle defect? The reviewer records a decision that can block, repair, approve, or escalate the change.

Controlled release and learning plane

The release plane applies Level 4 controls after pre-production approval. Canary or shadow execution compares the candidate with the current path on recent production-shaped data. Reconciliation monitors critical keys and aggregates, and rollback remains available when differences exceed approved bounds. Any escape or disputed exception feeds back into the benchmark catalog, fixture generator, risk metadata, and review checklist. This closed loop converts incidents into measurable coverage improvements rather than isolated corrections.

A minimal baseline-comparison implementation follows seven steps: execute candidate and reference from identical clean inputs; normalize only explicitly permitted representations; compare primary-key sets; compare exact fields; apply field-specific rules to genuinely approximate values; reconcile critical aggregates; and preserve diagnostic samples for every difference. Defect evidence and impact evidence remain separate. A one-row exact mismatch can prove incorrect behavior even when the aggregate impact is small, while a large aggregate change may be legitimate after an approved rule change.

In a continuous-delivery workflow [37], generated code enters a protected branch and cannot reach the release gate until the evidence packet is complete.

High-risk transformations require an authorized reviewer independent of the generation step. Approved changes proceed through canary or shadow execution, production reconciliation, and rollback-ready deployment. The architecture therefore treats human oversight as an explicit, auditable control surface rather than an informal final glance at generated code.

5. Results

Task-level outcomes

Table 4 reports the full task-level outcome matrix. The three correct controls (T01-T03) pass every configured checkpoint, so no false alarm is observed in the control set. This is a sanity check, not a stable specificity estimate, because three controls are too few to represent the diversity of valid transformations. Among the nine faulty tasks, schema validation detects T04 and misses T05-T12. Data-quality validation also detects T04 and misses T05-T12. Their multiple assertions therefore collapse to one shared true positive rather than two independent sources of coverage.

Trusted-baseline comparison detects seven different semantic mechanisms. T05 exposes many-to-many join fan-out through duplicated business keys and inflated totals. T06 exposes an exclusive end-date error through missing boundary records. T07 exposes NULL-sensitive filtering through a changed key set. T08 exposes an incomplete Spark window partition through cross-entity contamination. T09 exposes reversed timezone conversion through events assigned to the adjacent business date. T10 exposes an incremental predicate that ignores late-arriving records. T11 exposes a rare-category mapping swap even though the overall distribution barely changes. These detections are behavioral: the candidate is flagged because its keyed result differs from the independently specified result, not because its output is obviously malformed.

Mechanism-level interpretation of the outcome matrix

T04 is the only defect detected by both basic checkpoint families. The candidate converts a monetary DECIMAL to an integer. Schema validation observes the loss of scale directly, while the monetary data-quality rule interprets the same representational change as a policy violation. Because all fixture values are whole dollars, the candidate and reference contain the same visible values; the baseline therefore supplies no additional evidence. This is overlap, not independent confirmation, and it shows why detection counts must be deduplicated by task.

T05-T08 demonstrate defects that preserve interfaces and broad statistics. Join fan-out (T05) creates valid rows and believable totals; an exclusive reporting boundary (T06) removes only boundary records; NULL-sensitive exclusion (T07) removes a subset without a runtime error; and the incomplete Spark window partition (T08) produces smooth numerical sequences. The baseline detects each because keyed or field-level outputs disagree with the reference. Generic schema and distribution checks are not designed to infer the intended business relation.

T09-T11 show why temporal, incremental, and rare-category logic require activated fixtures. The timezone error is visible only for records near the business-day boundary. The incremental defect is visible only after a second batch includes late arrivals. The category swap is visible only when the rare source codes are present and compared exactly. A benchmark that contains only average cases would incorrectly report these candidates as correct, regardless of comparator sophistication.

T12 is a decision-policy failure rather than an absence of numerical evidence. The candidate uses the prior day' s exchange rate, so a field-level difference exists. The configured 0.5% aggregate tolerance classifies that difference as acceptable even though the specification requires an exact date-keyed lookup. Contextual adjudication changes the decision rule, not the code comparison: the reviewer determines that this field is semantically exact and that aggregate materiality is the wrong acceptance criterion.

The resulting coverage sequence is therefore interpretable. Basic structural and rule checks detect one unique task. Trusted-baseline comparison adds seven distinct defects, bringing automated union coverage to eight of nine. Human adjudication resolves the remaining tolerance-masked defect, reaching nine of nine within this benchmark. The sequence does not imply that human review will always achieve complete recall; it demonstrates that a defined human checkpoint can contribute information when the automated decision policy is semantically incomplete.

Table note. Green cells indicate an automated detection; red cells indicate a false negative for a faulty task. The correct controls are not shaded as errors. The matrix shows unique mechanism coverage: schema and DQ overlap on T04; baseline comparison supplies seven additional detections (T05-T11); T12 remains a policy-mediated miss until contextual review requires exact equality for the date-keyed rate.

Statistical interpretation of the observed counts

The task-level confusion matrices make the differences concrete. Schema validation and data-quality rules each produce 1 true positive, 8 false negatives, 3 true negatives, and no false positives.

Their defect recall is therefore 11.1%, while their overall task accuracy is 33.3%. Trusted-baseline comparison produces 7 true positives, 2 false negatives, and 3 true negatives, yielding 77.8% recall and 83.3% accuracy. The automated union produces 8 true positives, 1 false negative, and 3 true negatives, yielding 88.9% recall and 91.7% accuracy.

Precision is 100% for every checkpoint in this benchmark because none of the three correct controls is flagged. That number should not be interpreted as evidence of a negligible false-alarm rate. With only three controls, one additional false positive would reduce specificity from 100% to 66.7%. The correct conclusion is limited: the configured checks did not reject these three valid transformations. A production-quality estimate requires many more correct controls, including legitimate schema evolution, seasonal distribution shifts, backfills, and source-system corrections.

Uncertainty is also large. The 95% Wilson interval for 1/9 recall is approximately 2.0%-43.5%; for 7/9 it is 45.3%-93.7%; for 8/9 it is 56.5%-98.0%; and for 9/9 it is 70.1%-100%. These intervals are not a defect in the calculation; they expose the limited information in nine faulty tasks. The evidence supports a strong within-benchmark ordering and a mechanism-level explanation, but it does not support precise industry percentages.

Marginal coverage is more informative than raw alert volume. Data-quality rules add no unique task beyond schema validation because both detect T04. Baseline comparison adds seven unique tasks. The automated union adds T04 to the baseline result, and contextual adjudication adds T12. Thus the value of a checkpoint is determined by the new failure mechanisms it covers after existing controls are considered, not by the number of assertions it executes or the number of duplicate alerts it emits.

The paired schema-versus-baseline difference is large but, with eight discordant tasks, the exact two-sided McNemar p-value is approximately 0.070. The benchmark is therefore better treated as an estimation and design study than as a conventional null-hypothesis test. The mechanism evidence explains why the difference occurs: baseline comparison observes keyed behavioral divergence, whereas basic checks mostly observe contracts and predefined summaries. Replication should prioritize more tasks and more independent defect instances rather than manufacturing significance from repeated assertions on the same task.

The two baseline misses require different remedies. T04 is an activation failure: whole-dollar fixtures prevent the value difference from appearing, so the remedy is boundary-rich fixture design plus an explicit precision contract. T12 is a policy failure: a real difference is observed but suppressed by a generic tolerance, so the remedy is field-specific tolerance governance. The automated union detects T04 through basic controls and T05-T11 through baseline comparison, reaching 8/9. Contextual adjudication adds T12 by rejecting the tolerance for an exact business lookup.

Aggregate detection performance

For the nine faulty tasks, schema validation records 1 true positive and 8 false negatives, giving 11.1% recall with a 95% Wilson interval of approximately 2.0%-43.5%. Data-quality rules produce the same counts and interval. Trusted-baseline comparison records 7 true positives and 2 false negatives, or 77.8% recall with an interval of approximately 45.3%-93.7%. The automated union records 8 true positives and 1 false negative, or 88.9% recall with an interval of approximately 56.5%-98.0%. Contextual adjudication identifies the remaining defect, producing 9/9 within this controlled benchmark; its approximately 70.1%-100% interval remains wide and does not imply that human reviewers are infallible.

The portfolio decomposition is more informative than the raw percentages. Schema validation contributes one unique detection, T04. Data-quality rules contribute zero additional detections after schema validation because they identify the same task. Trusted-baseline comparison contributes seven unique detections, T05-T11. Contextual adjudication contributes one unique detection, T12. Thus, the effective portfolio is not “three automated tools plus a reviewer”; it is a sequence of four distinct information gains: contract evidence, no additional gain from the overlapping DQ configuration, seven behavioral divergences, and one requirement-level correction. This decomposition is the central result because it identifies where engineering investment changes coverage rather than merely increasing check count.

The paired outcomes also show why effect size and statistical significance must be separated. Against schema validation, the baseline detects seven tasks that schema misses, while schema detects one task that the baseline misses. The exact two-sided McNemar p-value is approximately 0.070. That value does not erase the operational difference; it shows that a nine-defect benchmark is underpowered for population-level inference. The defensible conclusion is descriptive: the deeper behavioral oracle produced substantially broader mechanism coverage in this task set, and larger replications are required to estimate the effect precisely.Across all 12 tasks, including the three controls, observed classification accuracy is 33.3%

for schema validation, 33.3% for data-quality rules, 83.3% for trusted-baseline comparison, 91.7% for the automated union, and 100% after contextual adjudication. Each configured family records 3 true negatives and 0 false positives on the controls. Consequently, nominal precision is 100% for the alerts that were raised, but this number is unstable: a single false alarm in a larger control set would materially change it. Recall, unique coverage, and the reason for each miss are therefore more useful than a single accuracy score.

Overlap and marginal value

The overlap analysis demonstrates that assertion volume is not equivalent to assurance. Schema and data-quality validation contain several individual checks, yet all of their effective detections collapse to one task, T04. Their pairwise marginal gain is therefore zero. The baseline comparator contributes seven independent task detections because it observes semantic behavior at keys, rows, fields, and aggregates. The automated union improves over the baseline by one task, showing that structural contracts remain valuable even when a strong reference oracle exists.

The residual analysis distinguishes two false-negative classes. T04 is an evidence-generation failure for the baseline: the fixture never activates cents, so the faulty cast and reference yield the same canonical values. T12 is an evidence-decision failure: the difference is generated and measured, but policy suppresses it. These failure classes require different remedies. T04 requires stronger boundary fixtures or stricter type comparison; T12 requires requirement-specific tolerance governance and accountable oracle review.

No correct control triggers a failure in the configured evaluation. With only three controls, this observation should be treated as a sanity check rather than a specificity estimate. A larger benchmark should include correct transformations that exhibit legitimate drift, schema evolution, rare but valid values, and delayed arrivals. Those cases are necessary to measure the cost of false alarms and human exception handling.

The exact 95% confidence interval for specificity based on 3/3 accepted controls is approximately 43.9%-100%. The lower bound is the important part: the benchmark provides no credible basis for claiming a low operational false-alarm rate. That rate must be measured with substantially more correct transformations, especially legitimate schema evolution and distribution drift.

Sensitivity and counterfactual analysis

Two counterfactual configurations clarify why the reported rates are properties of the checkpoint design rather than immutable properties of the tools. First, if the baseline comparator requires exact equality for the exchange-rate output instead of accepting a 0.5% relative difference, T12 is detected. Baseline recall then increases from 7/9 to 8/9, and the automated union reaches 9/9. This improvement does not require a new algorithm; it requires a requirement-appropriate decision rule.

Second, if the T04 fixture includes at least one amount with a non-zero fractional component, the decimal-to-integer cast changes an observed value and the baseline comparator detects it. Baseline recall again increases to 8/9, although the automated union remains 8/9 under the original T12 tolerance. This result demonstrates input-activation sensitivity: a latent semantic defect can escape differential execution when the fixture does not exercise the distinguishing condition.

If both counterfactual changes are applied–fractional monetary values and exact exchange-rate comparison–the baseline alone detects all nine defects in this benchmark. That outcome should not be interpreted as proof that a baseline can replace other checkpoints. It occurs because the reference and fixture were strengthened specifically for the seeded mechanisms. Structural contracts still protect against future values and schema evolution, and data-quality rules still provide inexpensive monitoring when reference execution is unavailable.

A no-baseline configuration provides the opposite counterfactual. The automated portfolio collapses to the overlapping basic checks and detects only T04. This illustrates why organizations that generate or refactor transformation code without preserving an executable reference assume a large, often invisible semantic risk. The cost of maintaining a bounded reference may be lower than the cost of reconstructing intent after a production discrepancy.

The sensitivity analysis exposes a four-stage governance hierarchy. Fixture design determines whether distinguishing behavior is exercised. The comparator determines whether a difference is measured. The decision rule determines whether a measured difference is promoted to an alert. Human review determines whether the fixture, comparator, and decision rule actually encode the governing business requirement. With fractional monetary values, the baseline detects T04; with exact exchange-rate comparison, it detects T12; with both changes, it detects all nine tasks in this benchmark. That counterfactual does not prove that baseline comparison is universally complete. It demonstrates that observed recall is jointly produced by task activation, oracle construction, and policy configuration.

6. Discussion

Validation depth dominates check volume

The clearest result is that semantic depth matters more than the number of automated assertions. Structural and data-quality checks are indispensable for inexpensive, repeatable protection, but they mostly observe the shape of the output. The baseline comparator observes behavior. Seven defects change output behavior without violating schema or broad data-quality rules. This supports H1 and aligns with the general test-oracle literature: stronger oracles reveal faults that execution and shallow invariants cannot [13], [16].

The result should not be misread as an argument to abandon basic checks. T04 demonstrates why a portfolio is necessary. A reference comparison can miss a latent defect when the fixture does not activate the relevant input condition. The decimal cast is harmful even though whole-dollar values happen to match. Structural contracts detect a class of future risk that value comparison cannot see in the current fixture. The correct architecture is layered, not winner-take-all.

The portfolio also exposes a common anti-pattern in data engineering: teams may accumulate dozens of rules that target the same symptom, such as row-count movement, while leaving entire semantic mechanisms uncovered. Assurance should be reported as coverage over defect classes, critical business rules, and transformation paths, not raw test count. A small set of independent oracles can be more valuable than a large set of correlated thresholds.

The trusted baseline is powerful but not automatically trustworthy

The baseline comparator performs best, but "trusted" must be earned. A reference implementation can contain defects, encode outdated policy, or depend on the same flawed source assumptions as the candidate. If the candidate and reference share copied logic, their errors may be correlated. A historical production output can preserve a legacy defect. Baseline governance therefore requires provenance, independent construction where feasible, version control, review, and periodic revalidation.

Input activation is equally important. Differential comparison only reveals behavior exercised by the fixture. T04 passes value comparison because the fixture lacks cents. A robust comparator should be paired with boundary-oriented and property-based fixtures designed to activate risky paths: nulls, duplicate keys, negative amounts, fractional values, daylight-saving transitions, late arrivals, empty partitions, and rare categories. This is where mutation testing, metamorphic testing, and data-intensive fuzzing can extend the benchmark [14], [16], [22]–[24].

Tolerance design is the most consequential weakness observed. T12 passes not because the candidate equals the baseline, but because the difference falls below 0.5%. A tolerance is appropriate for nondeterministic approximations, floating-point variation, sampled computations, or accepted business materiality. It is inappropriate when the requirement is exact lookup by date and currency. Teams often reuse technical tolerances across semantically different metrics. Thresholds should therefore be typed by requirement: exact, set-equivalent, order-insensitive, monotonic, bounded-error, distributional, or human-adjudicated.

Human-in-the-loop does not mean human-everywhere

The residual error supports H3 but does not justify universal line-by-line review. Mandatory review of every generated transformation can become expensive and superficial, especially when reviewers repeatedly see low-risk boilerplate. Human attention should be allocated where automated evidence is weakest or the cost of an undetected error is high. The taxonomy in Figure 2 separates contextual review from operational approval so that organizations can decide what question the human is answering.

A specification checkpoint asks whether the task is unambiguous and testable. A code checkpoint asks whether joins, filters, windows, dates, nulls, increments, and mappings implement the requirement. An oracle checkpoint asks whether the reference output and tolerance are legitimate. An exception checkpoint asks whether observed drift is acceptable. An approval checkpoint records accountable risk acceptance. A post-release checkpoint determines whether production reconciliation or rollback is required. These are different cognitive tasks and should not be collapsed into a single "reviewed" checkbox.

Figure 6 proposes an escalation policy. Structural or data-quality failures block deployment and require repair. When those checks pass, a baseline comparator is used if available. Baseline mismatches go to adjudication rather than automatic rejection when legitimate redesign or expected change is possible. When no baseline exists, human review is mandatory. When the baseline matches or the difference is within tolerance, a semantic-risk classifier decides whether contextual review is still required. Monetary conversion, temporal boundaries, incremental logic, identity resolution, access-control filtering,

regulated classifications, and low-frequency categories are strong escalation candidates.

This design supports H4 conceptually, but the workload savings are not fully measured here. The benchmark demonstrates one residual defect and one successful adjudication. A rigorous evaluation of human review would require multiple reviewers, expertise stratification, blinded conditions, review-time measurement, inter-rater agreement, and comparison of full review against risk-targeted review.

Interpreting the "one in nine" basic-check result

The 11.1% result should be interpreted narrowly. In this benchmark, each basic checkpoint family detects one of nine seeded defects, and both detect the same defect. It does not follow that schema or data-quality checks detect only 11.1% of real production bugs. Detection depends on the defect distribution, rule configuration, data fixture, thresholds, and engineering maturity. A benchmark dominated by nulls, duplicates, and schema drift would favor basic checks. A benchmark dominated by semantic mappings and temporal logic would favor behavioral and contextual controls.

The value of the result is comparative and diagnostic. Under a mixed set of executable silent defects, shallow controls leave substantial blind spots. The benchmark makes those blind spots concrete and provides a procedure for measuring them. Organizations should construct their own mutation catalog from incidents, near misses, code-review findings, and business-critical rules. The resulting detection matrix will be more useful than adopting the percentages in this paper as industry constants.

Cost, risk, and the economics of review

Checkpoint design is an economic allocation problem. Let C_a denote the cost of automated validation, C_h the cost of human review, p_e the probability that a defect escapes, and I the expected impact of an escaped defect. A simplified expected-loss objective is $C_a + C_h + p_e \times I$. Full manual review increases C_h and may not eliminate p_e because reviewers miss defects. Minimal automation reduces immediate cost but can increase the escape term. The rational policy invests more validation effort when impact and semantic uncertainty are high.

The impact term is not proportional only to row count or percentage difference. A 0.2% error in a high-volume financial measure can be material, while a 20% change in a low-stakes development metric may be harmless. Impact assessment should consider monetary exposure, regulatory consequence, customer eligibility, operational dependency, reversibility, detection latency, and the number of downstream consumers. These factors are better escalation inputs than generic code complexity alone.

Human review cost also depends on evidence quality. Reviewing an entire transformation from scratch is expensive. Reviewing a focused packet that identifies changed joins, boundary conditions, mismatching keys, and tolerance decisions is cheaper and more reliable. Automated checkpoints therefore create value even when they do not fully decide correctness: they compress the search space for human judgment.

A practical risk score can combine semantic criticality, oracle weakness, novelty, and change magnitude. High scores arise when the transformation affects money or regulated classification, lacks an independent baseline, introduces new temporal or incremental logic, or uses a broad tolerance. Low scores arise for mechanically generated projections with stable contracts and exact reference agreement. The score should trigger a review policy, not replace judgment, and its weights should be validated against incident history.

The main opportunity cost of weak validation is not merely a data incident. It is organizational distrust. When business users repeatedly discover discrepancies, they create shadow spreadsheets, duplicate reconciliations, and manual sign-offs. The resulting control burden can exceed the time saved by AI-assisted code generation. Autonomy is economically valuable only when assurance scales with generation.

Threats to Validity

Construct validity. "Defect detected" is operationalized as an actionable checkpoint failure attributable to the seeded fault. This measure captures pre-release observability but not diagnosis time, repair quality, or downstream impact. The human-in-the-loop concept is broader than the single contextual adjudication performed here. The taxonomy addresses that broader construct, but the quantitative results mainly evaluate automated checkpoints.

Internal validity. The benchmark authors know the seeded defects, which can influence rule selection, fixture design, and adjudication. To reduce this risk, check families and thresholds should be frozen before outcome inspection, and reference implementations should be independently reviewed. The current manuscript describes those controls as part of the benchmark protocol, but an external replication with blinded evaluators would be stronger. T12 is sensitive to the selected 0.5% threshold; a stricter threshold would change the result. That sensitivity is part of the intended

mechanism, but it also means the aggregate recall is configuration-dependent.

External validity. Twelve tasks cannot represent the diversity of enterprise transformations, SQL dialects, Spark APIs, dbt packages, source systems, data volumes, or regulatory contexts. The tasks use deterministic synthetic data and single seeded defects. Production pipelines include correlated faults, upstream instability, nondeterministic execution, incomplete specifications, and organizational handoffs. The benchmark does not estimate defect prevalence for any code-generation model or industry.

Conclusion validity. With nine faulty tasks, confidence intervals are wide and paired hypothesis tests have low power. The numerical differences are descriptive. The study does not claim population-level statistical significance. Repetition over additional tasks and defect instances is necessary before estimating stable effect sizes.

Baseline validity. Trusted-baseline comparison assumes the reference is correct, independent, and semantically current. Shared code, copied logic, outdated policy, or defective fixtures can create correlated false negatives. The benchmark mitigates this through separately constructed references and deterministic outputs, but it does not eliminate the general risk.

Reviewer validity. The residual defect is identified through contextual review, but reviewer performance is not independently benchmarked. Expertise, fatigue, confirmation bias, time pressure, and the quality of the review packet can change detection. Human review can also introduce false positives or inconsistent decisions. Future studies should report inter-rater agreement and review cost.

Temporal validity. The literature review is intentionally capped at December 31, 2021, as required for this manuscript. That cutoff captures the emergence of modern code-generation models but excludes later empirical studies, tools, governance standards, and model capabilities. The review should be updated before journal submission if the venue expects current coverage; retaining the cutoff is appropriate only when it is a deliberate study constraint.

Practical Implications

The benchmark translates into the architecture in Figure 6. The central requirement is separation of duties: the generated candidate, independent oracle, fixture design, policy decision, and release approval must not collapse into one unreviewed act. The architecture records specification and model metadata, produces layered evidence, scores semantic risk, and routes weak-oracle or high-impact cases to an accountable reviewer.

Second, teams should make semantic risk explicit in metadata. Each transformation can be tagged for monetary impact, temporal boundaries, incremental loading, identity resolution, regulated classification, privacy filtering, rare-category mapping, nondeterminism, and tolerance sensitivity. These tags can drive mandatory review and stronger fixtures. A low-risk projection or rename may proceed after structural and regression checks. A currency conversion or eligibility filter should require exact oracle review and accountable approval.

Third, baseline comparison should become a standard pre-production pattern for migrations, refactoring, AI-generated rewrites, and performance optimization. The reference may be a prior implementation, a simple specification-first query, a spreadsheet calculation for a bounded fixture, or an independently written transformation. The comparator should generate diagnostic evidence rather than a single pass/fail flag. Missing keys, extra keys, value deltas, category changes, and boundary records are more useful to reviewers than aggregate percentages alone.

Fourth, tolerance governance should be treated as a first-class control. Every tolerance should record its owner, rationale, unit, scope, expiry or review date, and whether it reflects computational uncertainty or business materiality. Exact requirements should not inherit generic numerical tolerances. T12 demonstrates that a comparator can technically observe a difference while governance suppresses the signal.

Fifth, correct controls and legitimate-drift cases should be included in organizational benchmarks. A detector that catches every seeded defect but overwhelms engineers with false alarms will be bypassed. Controls should represent valid schema evolution, seasonal volume shifts, new categories, delayed sources, and expected reclassifications. Human exception cost is part of system performance.

Sixth, metrics should focus on unique risk coverage. Recommended dashboard measures include: seeded-defect recall by mechanism; correct-control false-alarm rate; unique coverage added by each checkpoint family; percentage of transformations with an independent baseline; percentage of high-risk transformations receiving contextual review; mean adjudication time; reopened defects after approval; and production reconciliation escapes. Assertion count and pipeline pass rate are insufficient.

A practical implementation sequence is as follows:

1. Inventory transformation types and identify business-critical outputs. Do not begin with every pipeline; begin with money, customer eligibility,

compliance, operational SLAs, and executive reporting.

2. Create a defect taxonomy from incidents and code-review findings. Seed representative defects into non-production copies and measure current checkpoint coverage.

3. Establish Level 0 contracts and Level 1 data-quality rules as reusable templates. Remove redundant rules that add no unique coverage unless they provide faster diagnosis.

4. Add Level 2 behavioral oracles for migrations, refactoring, and generated code. Build boundary-rich deterministic fixtures and compare at key, row, column, and aggregate levels.

5. Define semantic-risk tags and escalation rules. Require human review when no baseline exists or when exact business semantics can be hidden by thresholds.

6. Package evidence for reviewers: specification, generated diff, reference diff, failed checks, tolerance rationale, and impacted metrics. Review quality depends on evidence quality.

7. Use Level 4 release controls: canary runs, reconciliation, observability, rollback, and explicit ownership. Detection after deployment is inferior to pre-production prevention but better than silent persistence.

For analytics engineering teams, the immediate message is direct: adding more generic tests is not a substitute for constructing a semantic oracle. For managers, the implication is equally direct: approving AI-generated transformation code without funding reference implementations, fixtures, and review time shifts cost from development into incident discovery and decision correction. The apparent productivity gain is partly an accounting illusion if silent errors are discovered after reports, models, or financial processes have consumed them.

Minimum pre-production evidence packet

For each AI-generated transformation, the release record should preserve a minimum evidence packet. The packet should contain the immutable task specification; the prompt or generation request; model and version when disclosure is permitted; generated and final code; reviewer changes; source and target grain; input-fixture checksum; reference implementation and owner; contract and data-quality configuration; baseline comparison results; tolerance definitions; unresolved exceptions; risk tags; reviewer identity; approval decision; and rollback plan.

The packet serves three purposes. It enables a reviewer to make a defensible decision, enables an auditor or incident responder to reconstruct why the transformation was approved, and enables the organization to learn from escapes. When a production defect occurs, the failed mechanism should be added to the benchmark and linked to the checkpoint that missed it. This converts incidents into measurable improvements instead of one-off fixes.

Evidence retention should be proportionate to criticality but should not depend on the availability of a particular employee. Generated transformations are especially vulnerable to intent loss because the code author may not have manually reasoned through every line. The specification, oracle, and approval rationale therefore become part of the maintainable system, not temporary development artifacts.

Migration and change-data-capture release gates

The checkpoint policy extends beyond analytical models to database migration and change-data-capture cutovers. In a GoldenGate-based near-zero-downtime migration, low replication lag or a technically successful switchover is not sufficient evidence of semantic readiness. The release packet should combine independent reconciliation, backlog and checkpoint exposure, object-level validation, rollback readiness, and named approval. Gollapudi's risk-controlled Oracle migration framework similarly treats cutover as an evidence-based decision rather than a single infrastructure metric [38]. This operational case reinforces the central argument of the present study: execution success and apparently healthy telemetry are necessary but not sufficient conditions for release.

7. Conclusion and Future Work

This article introduced a controlled pre-production benchmark, a five-level human-in-the-loop checkpoint taxonomy, and a proposed closed-loop validation architecture for AI-generated transformation code. Across 12 SQL, Spark, and dbt-style tasks, nine contained seeded defects and three were correct controls. Structural schema checks and declarative data-quality rules each detected one defect, with complete overlap. Trusted-baseline comparison detected seven. The union of all automated controls detected eight, leaving one low-magnitude, tolerance-masked semantic defect. Contextual review identified that residual error by comparing the configured decision rule with the exact business requirement.

The result is not that automation fails or that every transformation requires exhaustive manual inspection. The stronger conclusion is that assurance depends on oracle depth and portfolio diversity. Structural controls detect contracts. Statistical controls detect abnormal properties. Behavioral controls detect divergence. Humans resolve ambiguity, context, thresholds, and

accountability. A mature pre-production process uses each for the question it can answer.

Future work should expand the benchmark along five dimensions. First, the task corpus should include hundreds of transformations, multiple defect instances, and multi-defect interactions. Second, code-generation models, prompts, and sampling strategies should be explicitly compared. Third, fixtures should combine synthetic boundary cases with de-identified production distributions and real incident replays. Fourth, human review should be evaluated with multiple reviewers, expertise levels, time measurements, and inter-rater agreement. Fifth, adaptive escalation policies should be tested to determine whether risk tags and automated evidence can reduce review effort without increasing escapes.

Additional research should compare exact baselines, metamorphic relations, property-based generation, query-equivalence techniques, and learned anomaly detection. It should also study baseline decay: a reference can become stale as business definitions change. Finally, organizations need longitudinal evidence linking pre-production checkpoint coverage to production incident frequency, time to detection, downstream correction cost, and user trust. The benchmark presented here supplies a reproducible starting point for that larger program of research.

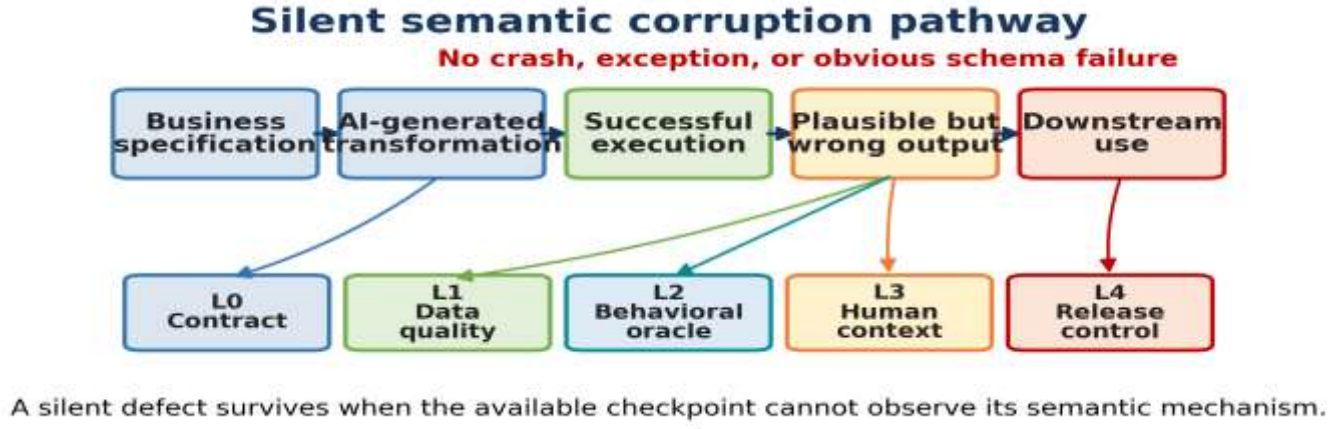

Figure 1. *Silent semantic-corruption pathway and checkpoint opportunities*

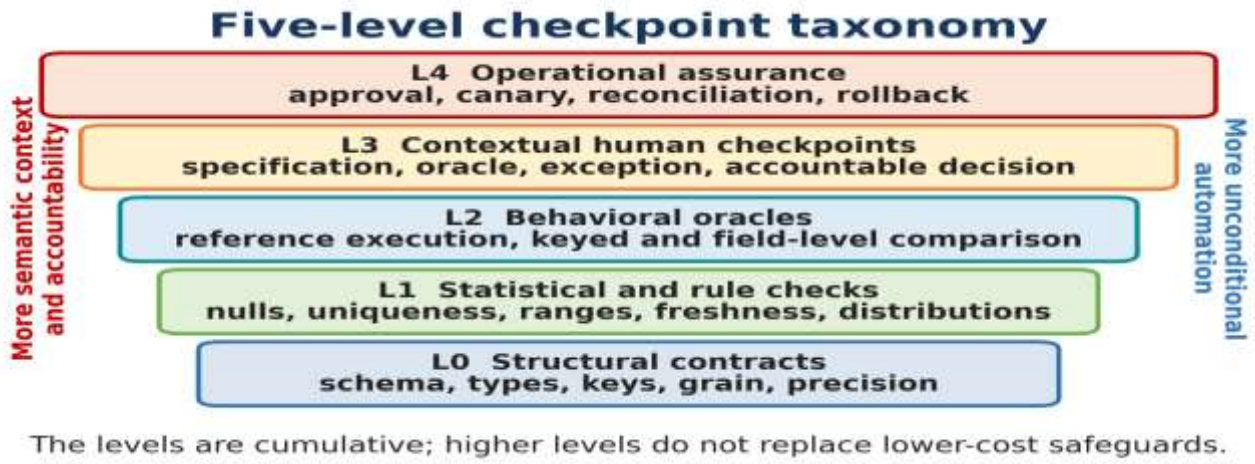

Figure 2. *Five-level checkpoint taxonomy; semantic depth increases from structural contracts to operational assurance.*

Table 1. *Related research streams and the unresolved intersection addressed by this study.*

Stream	Representative work	Primary evidence	Silent semantic errors	Human role	Limitation for generated transformations
Data-quality dimensions	[1], [2], [4]	Fitness-for-use dimensions and assessment metrics	Conceptually	Defines context and meaning	Does not specify executable checkpoint coverage.
Production	[8], [9], [10], [12]	Schemas, profiles,	Partly	Maintains rules	Plausible semantic

Stream	Representative work	Primary evidence	Silent semantic errors	Human role	Limitation for generated transformations
data validation		constraints, and learned patterns		and reviews exceptions	defects may remain inside learned bounds.
Software test oracles	[13], [16]	Expected output, specifications, and metamorphic relations	Yes	Defines or validates the oracle	Oracle construction can be costly and specification-dependent.
Controlled fault benchmarks	[14], [15]	Seeded or curated faults and test-suite kills	Yes	Validates fault realism	Not specialized for SQL, Spark, or dbt transformations.
Database differential testing	[17], [18], [19]	Equivalent or partitioned query results	Yes	Adjudicates divergent executions	Targets DBMS implementation rather than business intent.
Data-intensive program testing	[22], [23], [24]	Generated inputs and differential outputs	Yes	Interprets specifications and results	Framework-specific setup; limited business semantics.
AI code generation	[25], [26], [27], [28], [29], [30]	Benchmark tests and functional correctness	Yes	Prompts, repairs, and accepts artifacts	Limited evidence on production transformation checkpoints.
Human-AI interaction	[31], [32], [33], [34]	Calibrated feedback and accountable oversight	Yes	Intervenes under uncertainty	Does not quantify checkpoint-specific defect coverage.

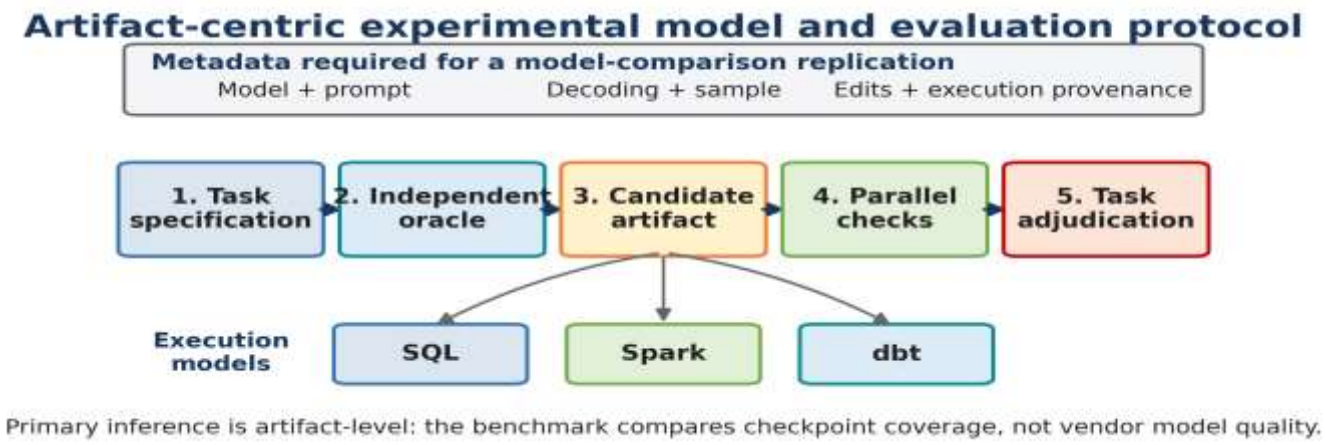

Figure 3. *Artifact-centric experimental model and evaluation protocol across SQL, Spark, and dbt execution models.*

Table 2*. Pre-production benchmark tasks and seeded defect mechanisms.*

ID	Engine	Objective	Status	Defect mechanism	Activation evidence	Why plausible	Primary risk
T01	SQL	Monthly revenue by customer and region	Correct	None	Standard aggregate fixture	Expected schema and aggregates	False-alarm control
T02	Spark	Normalize identifiers and deduplicate events	Correct	None	Duplicate/canonical-key fixture	Stable row count and keys	False-alarm control
T03	dbt	Customer dimension with	Correct	None	Two-version dimension fixture	Stable dimension cardinality	False-alarm control

ID	Engine	Objective	Status	Defect mechanism	Activation evidence	Why plausible	Primary risk
		current-status flag					
T04	SQL	Invoice and tax projection	Faulty	DECIMAL cast to integer	Fractional monetary value	Whole-dollar fixture masks value divergence	Lost cents and precision
T05	SQL	Orders-to-promotions revenue summary	Faulty	Many-to-many join fan-out	Multiple promotions per order	Believable increase in totals	Overstated revenue
T06	SQL	Reporting-period event filter	Faulty	Exclusive end boundary	Record on end boundary	Only edge records omitted	Incomplete reporting
T07	SQL	Blocked-account exclusion	Faulty	NOT IN with NULL	NULL in exclusion set	Subset disappears without failure	Unexplained undercount
T08	Spark	Customer rolling average	Faulty	Missing partition key	Shared key across regions	Smooth, valid-looking values	Cross-entity contamination
T09	Spark	Business-timezone conversion	Faulty	Offset applied in wrong direction	Records near midnight	Small day-level shifts	Misdated events/SLA errors
T10	dbt	Incremental transaction load	Faulty	Late arrivals excluded	Second batch with late timestamp	Freshness remains normal	Persistent missing history
T11	dbt	Reporting-category mapping	Faulty	Rare categories swapped	Rare codes in fixture	Distribution changes minimally	Priority-case misclassification
T12	dbt	Daily FX-rate translation	Faulty	Prior-day rate below tolerance	Adjacent dates with different rates	Aggregate remains within 0.5%	Systematic valuation error

Table 3. *Checkpoint configuration, decision rules, evidence, and known blind spots.*

Layer	Owner	Representative controls	Decision/evidence	Strength	Known blind spot	Escalation or human action
L0 Structural	Platform/data engineer	Columns, types, nullability, keys, grain, decimal scale	Exact contract decision and field-level diff	Fast and deterministic	Wrong joins, filters, mappings, and time logic	Define contract; approve intentional evolution.
L1 Statistical/DQ	Data owner/engineer	Nulls, uniqueness, domains, ranges, row ratios, freshness	Threshold decision plus failing metrics/keys	Reusable common-risk coverage	Plausible output, rare classes, compensating errors	Set thresholds; adjudicate legitimate drift.
L2 Behavioral	Independent test owner	Key-set, row, field, aggregate, hash, or metamorphic comparison	Material-divergence decision plus mismatches	Highest measured semantic reach	Defective oracle, unactivated path, unsafe tolerance	Validate oracle independence and field semantics.
Automated union	CI/CD policy engine	Deduplicated logical OR of L0-L2	Consolidated evidence packet and unique coverage	Broader than one family	Correlated rules and shared blind spots	Prioritize impact; remove redundant

Layer	Owner	Representative controls	Decision/evidence	Strength	Known blind spot	Escalation or human action
						noise.
L3 Contextual	Domain reviewer/accountable approver	Specification-code comparison, oracle and exception review	Approve, reject, repair, or strengthen fixture with rationale	Interprets intent and tolerance meaning	Fatigue and expertise inconsistency	Mandatory for weak oracle, high impact, or tolerance-dependent pass.
L4 Operational	Release owner/SRE/data operations	Canary, shadow, reconciliation, observability, rollback	Monitored reversible release and rollback record	Limits duration and impact	Detection may occur after exposure	Stop, rollback, reconcile, and convert incidents into cases.
Traceability	Governance/audit owner	Prompt, model, code, fixture, reference, dependencies, approvals	Complete immutable evidence packet	Enables reproducibility and learning	Missing metadata prevents causal analysis	No release on incomplete evidence.

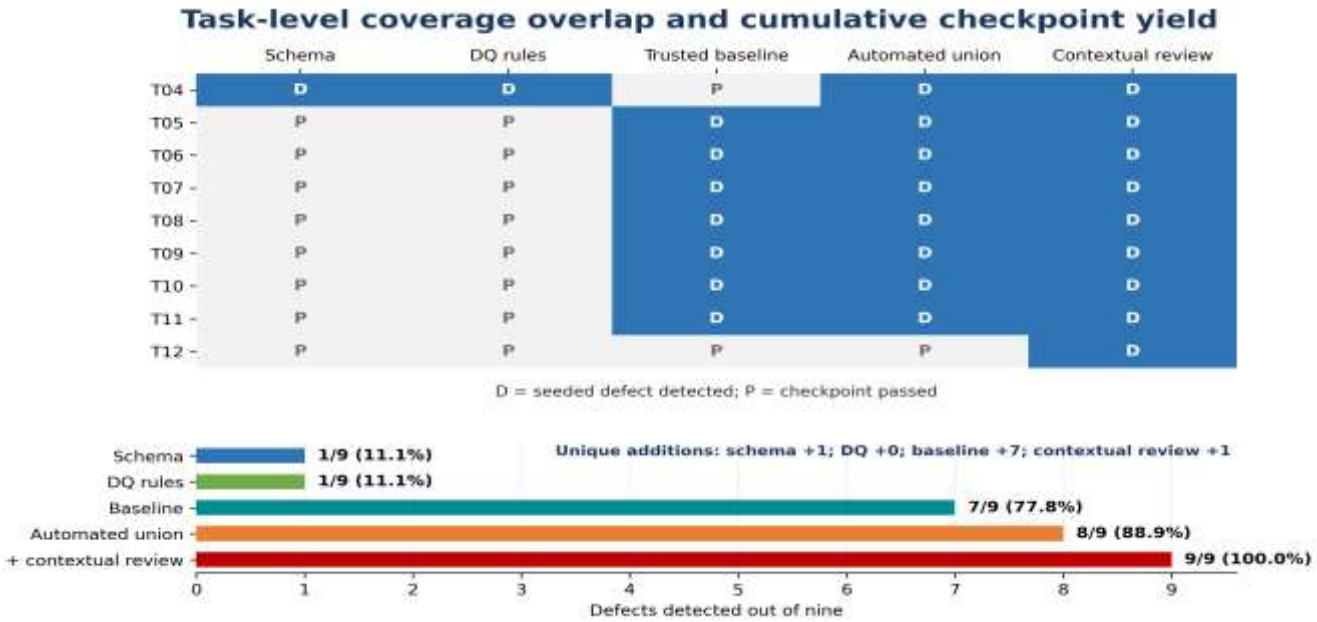

Figure 4. *Task-level coverage overlap and cumulative checkpoint yield.*

Table 4. *Task-level checkpoint outcomes (D = detected; P = passed).*

Task	Status	Defect class	Schema	DQ	Baseline	Union	Detection mechanism or residual cause
T01	Correct	Control	P	P	P	P	Valid aggregation accepted by all checkpoints.
T02	Correct	Control	P	P	P	P	Valid normalization and deduplication accepted.
T03	Correct	Control	P	P	P	P	Valid dimension logic accepted.
T04	Faulty	Precision/activation	D	D	P	D	Contract exposes lost scale; whole-dollar fixture masks value divergence.
T05	Faulty	Join cardinality	P	P	D	D	Key and aggregate differences expose many-to-many fan-out.
T06	Faulty	Temporal boundary	P	P	D	D	End-boundary keys are missing from the

Task	Status	Defect class	Schema	DQ	Baseline	Union	Detection mechanism or residual cause
							candidate.
T07	Faulty	NULL semantics	P	P	D	D	NULL-sensitive exclusion changes the business-key set.
T08	Faulty	Window partition	P	P	D	D	Rolling values diverge for shared identifiers.
T09	Faulty	Timezone logic	P	P	D	D	Events move to adjacent business dates near midnight.
T10	Faulty	Incremental predicate	P	P	D	D	Late-arriving keys are absent after second-batch execution.
T11	Faulty	Rare mapping	P	P	D	D	Exact categories differ although the distribution remains stable.
T12	Faulty	Tolerance policy	P	P	P	P	A measured difference is suppressed by 0.5% tolerance; L3 review rejects it.

D = defect detected; P = checkpoint passed. Correct controls contain no seeded defect.

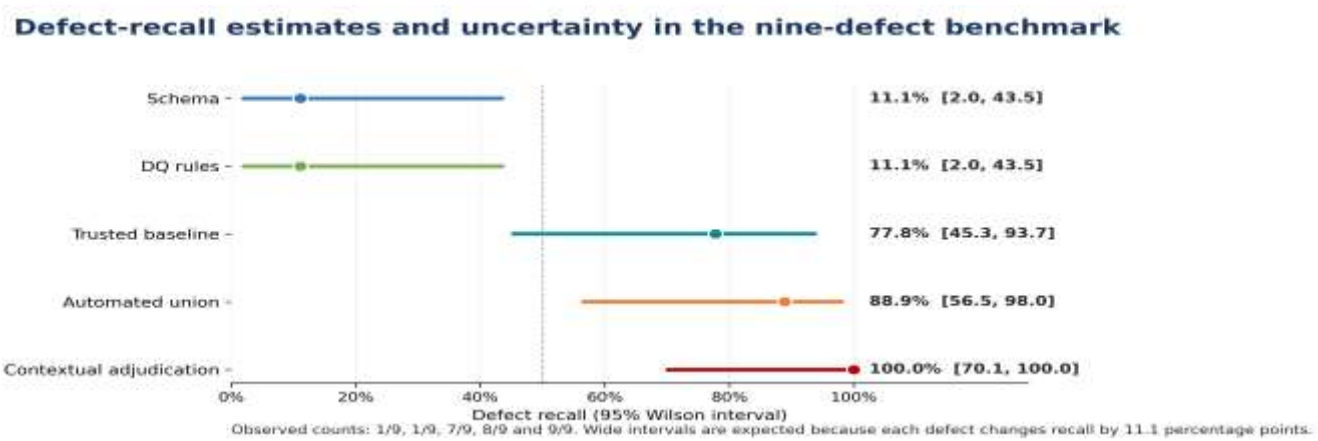

Figure 5. *Defect-recall estimates with 95% Wilson intervals.*

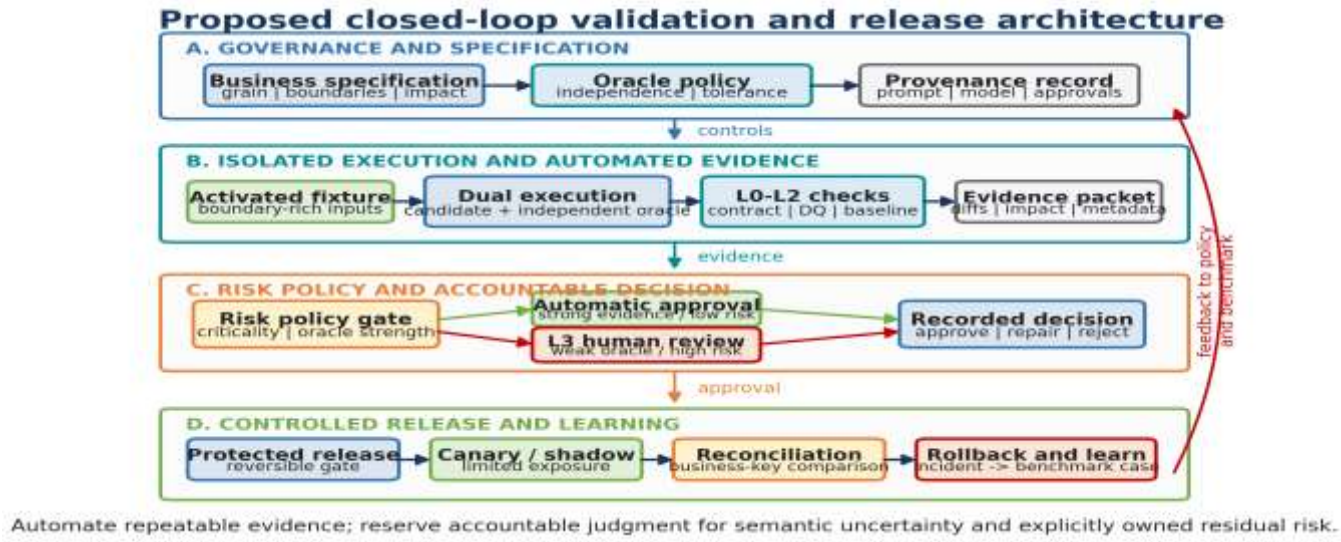

Figure 6*. Proposed closed-loop validation and release architecture.*

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.

- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," J. Manage. Inf. Syst., vol. 12, no. 4, pp. 5–33, 1996, doi: 10.1080/07421222.1996.11518099.

[2] L. L. Pipino, Y. W. Lee, and R. Y. Wang, "Data quality assessment," Commun. ACM, vol. 45, no. 4, pp. 211–218, 2002, doi: 10.1145/505248.506010.

[3] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," ACM Comput. Surv., vol. 41, no. 3, Art. no. 16, 2009, doi: 10.1145/1541880.1541883.

[4] D. M. Strong, Y. W. Lee, and R. Y. Wang, "Data quality in context," Commun. ACM, vol. 40, no. 5, pp. 103–110, 1997, doi: 10.1145/253769.253804.

[5] D. Sculley et al., "Hidden technical debt in machine learning systems," in Advances in Neural Information Processing Systems 28, 2015, pp. 2503–2511, doi: 10.5555/2969442.2969519.

[6] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in Proc. IEEE Int. Conf. Big Data, 2017, pp. 1123–1132, doi: 10.1109/BigData.2017.8258038.

[7] D. Baylor et al., "TFX: A TensorFlow-based production-scale machine learning platform," in Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, 2017, pp. 1387–1395, doi: 10.1145/3097983.3098021.

[8] S. Schelter et al., "Automating large-scale data quality verification," Proc. VLDB Endow., vol. 11, no. 12, pp. 1781–1794, 2018, doi: 10.14778/3229863.3229867.

[9] S. Schelter et al., "Unit testing data with Deequ," in Proc. 2019 Int. Conf. Manage. Data, 2019, pp. 1993–1996, doi: 10.1145/3299869.3320210.

[10] L. E. Lwakatare, E. Range, I. Crnkovic, and J. Bosch, "On the experiences of adopting automated data validation in an industrial machine learning project," in Proc. IEEE/ACM ICSE-SEIP, 2021, pp. 248–257, doi: 10.1109/ICSE-SEIP52600.2021.00034.

[11] E. Caveness et al., "TensorFlow Data Validation: Data analysis and validation in continuous ML pipelines," in Proc. 2020 ACM SIGMOD Int. Conf. Manage. Data, 2020, pp. 2793–2796, doi: 10.1145/3318464.3384707.

[12] J. Song and Y. He, "Auto-Validate: Unsupervised data validation using data-domain patterns inferred from data lakes," in Proc. 2021 ACM SIGMOD Int. Conf. Manage. Data, 2021, pp. 1678–1691, doi: 10.1145/3448016.3457250.

[13] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," IEEE Trans. Softw. Eng., vol. 41, no. 5, pp. 507–525, 2015, doi: 10.1109/TSE.2014.2372785.

[14] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE Trans. Softw. Eng., vol. 37, no. 5, pp. 649–678, 2011, doi: 10.1109/TSE.2010.62.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proc. Int. Symp. Softw. Testing Anal., 2014, pp. 437–440, doi: 10.1145/2610384.2628055.

[16] T. Y. Chen et al., "Metamorphic testing: A review of challenges and opportunities," ACM Comput. Surv., vol. 51, no. 1, Art. no. 4, 2018, doi: 10.1145/3143561.

[17] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in Proc. 14th USENIX Symp. Oper. Syst. Design Implement., 2020, pp. 667–682, doi: 10.48550/arXiv.2001.04174.

[18] M. Rigger and Z. Su, "Detecting optimization bugs in database engines via non-optimizing reference engine construction," in Proc. 28th ACM ESEC/FSE, 2020, pp. 1140–1152, doi: 10.1145/3368089.3409710.

[19] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," Proc. ACM Program. Lang., vol. 4, OOPSLA, Art. no. 211, 2020, doi: 10.1145/3428279.

[20] R. Zhong et al., "SQUIRREL: Testing database management systems with language validity and coverage feedback," in Proc. ACM CCS, 2020, pp. 955–967, doi: 10.1145/3372297.3417260.

[21] V. S. P. Dintyala, A. Narechania, and J. Arulraj, "SQLCheck: Automated detection and diagnosis of SQL anti-patterns," in Proc. 2020 ACM SIGMOD, 2020, pp. 2331–2345, doi: 10.1145/3318464.3389754.

[22] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "BigFuzz: Efficient fuzz testing for data analytics using framework abstraction," in Proc. 35th IEEE/ACM ASE, 2020, doi: 10.1145/3324884.3416641.

[23] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "DiffStream: Differential output testing for stream processing programs," Proc. ACM Program. Lang., vol. 4, OOPSLA, Art. no. 153, 2020, doi: 10.1145/3428221.

[24] M. A. Gulzar, S. Mardani, M. Musuvathi, and M. Kim, "White-box testing of big data analytics with complex user-defined functions," in Proc. 27th ACM ESEC/FSE, 2019, pp. 290–301, doi: 10.1145/3338906.3338953.

[25] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP, 2020, pp. 1536–1547, doi: 10.18653/v1/2020.findings-emnlp.139.

[26] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. ICLR, 2021, doi: 10.48550/arXiv.2009.08366.

[27] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in Proc. NAACL-HLT, 2021, pp. 2655–2668, doi: 10.18653/v1/2021.naacl-main.211.

[28] M. Chen et al., "Evaluating large language models trained on code," 2021, doi: 10.48550/arXiv.2107.03374.

[29] J. Austin et al., "Program synthesis with large language models," 2021, doi: 10.48550/arXiv.2108.07732.

[30] J. A. Prenner and R. Robbes, "Automatic program repair with OpenAI Codex: Evaluating QuixBugs," 2021, doi: 10.48550/arXiv.2111.03922.

[31] S. Amershi et al., "Guidelines for human-AI interaction," in Proc. 2019 CHI Conf. Human Factors Comput. Syst., 2019, pp. 1–13, doi: 10.1145/3290605.3300233.

[32] X. Wu, L. Xiao, Y. Sun, J. Zhang, T. Ma, and L. He, "A survey of human-in-the-loop for machine learning," 2021, doi: 10.48550/arXiv.2108.00941.

[33] A. Crisan and B. Fiore-Gartland, "Fits and starts: Enterprise use of AutoML and the role of humans in the loop," in Proc. 2021 CHI, 2021, doi: 10.1145/3411764.3445775.

[34] N. Sambasivan et al., "Everyone wants to do the model work, not the data work: Data cascades in high-stakes AI," in Proc. 2021 CHI, 2021, doi: 10.1145/3411764.3445518.

[35] C. Wohlin et al., Experimentation in Software Engineering. Berlin, Germany: Springer, 2012, doi: 10.1007/978-3-642-29044-2.

[36] B. A. Kitchenham et al., "Preliminary guidelines for empirical research in software engineering," IEEE Trans. Softw. Eng., vol. 28, no. 8, pp. 721–734, 2002, doi: 10.1109/TSE.2002.1027796.

[37] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," J. Syst. Softw., vol. 123, pp. 176–189, 2017, doi: 10.1016/j.jss.2015.06.063.

[38] R. Gollapudi, "Risk-controlled near-zero-downtime Oracle database migration using GoldenGate," Int. J. Comput. Exp. Sci. Eng., vol. 8, no. 3, pp. 113–123, 2022, doi: 10.22399/ijcesen.5382.