10.48047/jocaaa.2026.35.05.09

Trace-Bisimulation Equivalence for Regression Detection in Non-Deterministic Code-Generating Analytics Agents

Ravi Chandra Chodisetti
Independent Researcher, USA

Abstract

Code-generating analytics agents, systems that translate natural language queries into executable analytical programs, have become essential components of enterprise data infrastructure. However, their non-deterministic nature, producing structurally different but semantically equivalent programs across runs, renders conventional regression testing methods unreliable. Output-value comparison inflates false positives by flagging legitimate implementation variation, while syntax-based comparison produces false negatives by operating at the wrong level of abstraction. This article proposes a formal framework grounded in bisimulation equivalence from process algebra to address this gap. The framework introduces Analytical Dataflow Graphs as a normalized intermediate representation of program semantics, applies bisimulation to compare analytical behavior across agent versions, and implements a three-tier equivalence hierarchy comprising Result Equivalence, Analytical Bisimulation, and Intent Satisfaction. Evaluation across five hundred analytical queries and four model version transitions demonstrates a ninety-three percent true regression detection rate with a false positive rate below eight percent, reducing human review to below twelve percent of benchmark cases, against a seventy-four percent industry baseline reported by Pan et al. for production agents. These figures are drawn from different measurement contexts and are not directly comparable. The framework establishes a principled and scalable foundation for quality assurance in non-deterministic code-generating systems.

Keywords: Bisimulation Equivalence, Code-Generating Analytics Agents, Regression Testing, Analytical Dataflow Graphs, Non-Deterministic Program Equivalence

1. Introduction

Code-generating analytics agents represent a paradigmatic shift in enterprise data interaction. By transforming natural language queries into executable programs that operate on structured datasets, these systems eliminate the need for users to possess domain-specific programming expertise [1]. The maturation of large language models for code generation demonstrated by systems capable of producing competition-level solutions from natural language specifications has accelerated the deployment of such agents in production analytics environments [2]. Comprehensive surveys of the natural language to code problem confirm that the scope of these agents now encompasses a wide operational range, from simple aggregation queries to multi-step analytical workflows involving joins, conditional logic, and derived metric computation [3].

The quality assurance challenge introduced by these agents is structurally distinct from classical software testing. Because large language model outputs are non-deterministic and stochastic by design, the same natural language query may produce syntactically different programs across runs, model versions, or prompt configurations, while preserving identical analytical intent [4]. Regression testing, the process of verifying that a system update does not degrade previously correct behavior, is therefore complicated by the impossibility of relying on program identity as a correctness proxy. A systematic literature review of

10.48047/jocaaa.2026.35.05.09

large language models applied to software engineering confirms that evaluating functional correctness in generated code remains one of the most open and underspecified challenges in the field, with no consensus methodology for production deployment contexts [5].

This article proposes a framework that addresses this gap by grounding regression detection in bisimulation equivalence from process algebra. Rather than comparing programs by their syntactic structure or numerical outputs, the framework evaluates whether two programs exhibit the same observable analytical behavior, formalized through a directed acyclic graph representation of their dataflow semantics. The following sections establish the failure modes of existing approaches, define the formal representation and equivalence relation, describe the three-tier evaluation hierarchy, and report empirical results across a structured benchmark. The primary contributions of this article are (1) the formalization of Analytical Dataflow Graphs as a normalized, implementation-independent representation of analytical program semantics; (2) the adaptation of bisimulation equivalence from concurrent process verification to the comparison of generated analytical programs; (3) a three-tier equivalence hierarchy that integrates formal equivalence checking with calibrated evaluator judgment and confidence-gated human escalation; and (4) an empirical evaluation across five hundred queries and four model version transitions demonstrating that the framework achieves a ninety-three percent true regression detection rate with a false positive rate below eight percent, reducing human review to below twelve percent of cases in benchmark evaluation, compared to a seventy-four percent industry baseline reported for production agents.

2. The Regression Testing Problem for Code-Generating Agents

2.1 Why Output Comparison Produces False Positives

Output-value comparison is the most operationally natural regression test for code-generating systems: execute both the incumbent and updated agent's programs on the same dataset and assert result identity. For deterministic software under stable environments, this oracle is sufficient. For code-generating analytics agents, it systematically overestimates the regression rate because analytically equivalent programs can legitimately produce different numerical outputs depending on implementation choices that carry no analytical significance. Two illustrative failure modes are representative. First, ranking queries with composite metrics are subject to tie-breaking ambiguity: two implementations may sort by an identical primary metric but resolve ties through different secondary criteria, producing divergent output orderings that are both analytically valid. Second, floating-point arithmetic is non-associative, meaning that aggregation operations applied to rows in different processing orders produce results that diverge at machine precision, despite computing the same mathematical quantity. Neither case constitutes an analytical regression, yet output comparison flags both indiscriminately. Rigorous evaluation frameworks for code generation have established that augmented test suites reveal substantially more failure modes than pass-fail output comparison alone and that programs passing standard functional checks can still harbor latent semantic divergences detectable only through behavioral analysis [6].

At enterprise scale, the practical consequence of false positive inflation is severe. Empirical measurement of production agent deployments documents that seventy-four percent of production agents rely primarily on human evaluation for quality assurance [7]. When automated regression tests generate false positives at even a modest per-query rate, the resulting volume of flagged cases either consumes unsustainable human review capacity or desensitizes engineering teams to alerts, degrading the system's ability to surface genuine regressions. Evaluation methodology research confirms that automated LLM evaluation relying on output matching systematically conflates stylistic variation with functional failure, producing unreliable quality signals in production settings [8]. Differential analysis of program versions under output-based oracles has

10.48047/jocaaa.2026.35.05.09

further demonstrated that patched or updated programs appearing correct under such oracles frequently contain latent behavioral divergences that escape detection [9]. The root cause across all these cases is consistent: output comparison lacks the semantic abstraction necessary to distinguish implementation variation from analytical incorrectness.

2.2 Why Syntax Similarity Produces False Negatives

Syntax-based comparison metrics, abstract syntax tree edit distance, token-level Jaccard similarity, and related structural measures address the execution cost of output comparison by operating directly on code structure. Their fundamental limitation is a misalignment between the quantity they measure and the property of interest. Analytical correctness is a semantic property of program behavior; syntactic structure is a surface property of code representation. These two properties are largely orthogonal for the class of programs produced by large language models. The orthogonality manifests in both directions. Semantically distinct programs differing by a single token substitution, such as replacing a sum aggregation with a mean aggregation, may be syntactically near-identical yet produce entirely different analytical results across all non-trivial inputs. Semantically identical programs implementing the same filter-group-aggregate pipeline through pandas method chaining in one case and explicit loop-based accumulation in another may share almost no syntactic structure yet implement precisely the same analytical logic. Program dependency graph-based clone detection research has established that structural code similarity metrics achieve precision and recall rates fundamentally bounded by the many-to-one relationship between syntactic form and semantic content, with false negative rates exceeding thirty percent for semantically equivalent programs expressed through different idiomatic patterns [11]. The problem is compounded by the stylistic variability inherent in large language model generation. A single analytical operation, for instance, filtering a tabular dataset to rows satisfying a scalar predicate, admits at least four syntactically distinct Python realizations: bracket-based boolean indexing, query-method invocation with a string expression, lambda-based filtering, and list comprehension. All four realize the same relational selection operation. Benchmarking work on program equivalence reasoning confirms that syntactic surface features are unreliable predictors of semantic equivalence across all tested model families, with approaches relying on syntactic similarity achieving accuracy near chance level on semantically equivalent but syntactically dissimilar program pairs [19]. A regression testing system operating on syntax similarity will therefore generate false positives on harmless syntactic variation produced by ordinary sampling stochasticity and simultaneously fail to detect genuine analytical regressions embedded within syntactically stable code [4]. The combined effect is a test system that is uninformative in both directions and provides no actionable diagnostic signal about what actually changed analytically.

10.48047/jocaaa.2026.35.05.09

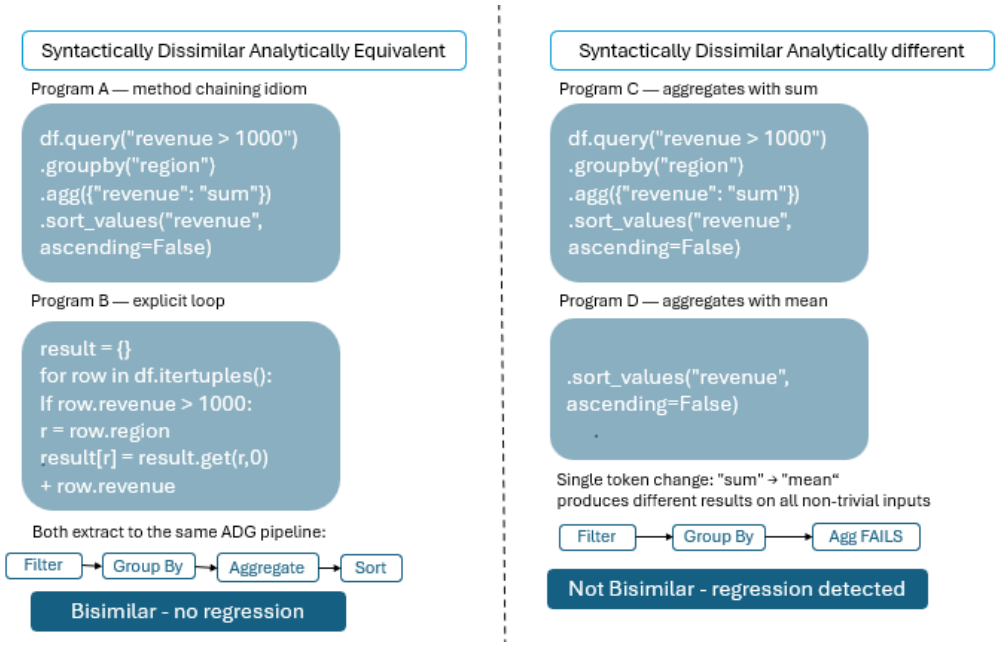

Figure 1: Syntactic Dissimilarity vs. Analytical Equivalence in Generated Analytics Programs

3. Analytical Dataflow Graphs: A Formal Representation

3.1 Graph Extraction from Generated Code

The framework introduces a formal intermediate representation, the Analytical Dataflow Graph, that captures the analytical semantics of a generated program at an abstraction level suitable for equivalence comparison while remaining independent of the syntactic idioms through which those semantics are expressed. Formally, an Analytical Dataflow Graph is a directed acyclic graph $G = (V, E, \tau, \sigma)$ in which each vertex v in V represents a discrete analytical operation, each directed edge (u, v) in E represents data flow from operation u to operation v, $\tau: V \rightarrow T$ is a type function assigning each vertex one of ten canonical operation types, and $\sigma: V \rightarrow \Sigma$ is a signature function encoding normalized operation parameters.

The ten operation types—Load, Filter, Project, Group By, Aggregate, Sort, Derive, Join, Pivot, and Window—constitute a vocabulary covering the predominant operations performed on tabular data in analytical programs. The extraction procedure parses a generated Python program into an abstract syntax tree, applies pattern matching against a library of approximately fifty canonical operation templates, and reconstructs the dataflow graph by tracing variable assignments and method chain dependencies. Large language model-assisted dataflow analysis has demonstrated that decomposing interprocedural analysis into local, structured reasoning steps achieves both scalability and precision exceeding several classical static analysis baselines on Python codebases [12], and the ADG extraction procedure operationalizes this finding by restricting the model's role to resolving template-matching ambiguities while delegating the structural reconstruction to deterministic procedures.

A normalization stage is applied after template matching to ensure that syntactically distinct but semantically identical operations converge to the same graph vertex. The normalization transformations include variable name canonicalization, commutative operand sorting, algebraic simplification of equivalent expressions, and resolution of syntactic sugar to canonical operator forms. Semantic-preserving

10.48047/jocaaa.2026.35.05.09

transformation research has established that such normalization steps are necessary preconditions for reliable program equivalence analysis, as unresolved syntactic variation in intermediate representations propagates into false equivalence distinctions at the comparison stage [13]. For operations that do not match any template, the framework falls back to opaque Custom vertices, retaining the full abstract syntax tree subtree, compared by structural AST equivalence. This fallback is sound; it never incorrectly classifies non-equivalent operations as equivalent but is conservative, and the template registry is designed to be extensible by domain teams without modification to the core framework.

Operation type	Example Python code patterns (after normalization)
Load	pd.read_csv(), pd.read_parquet(), pd.read_excel(), DataFrame constructor
Filter	Boolean indexing: df[df.col > x], .query("col > x"), .loc[] with condition, filter() with lambda
Project	Column subset df[["a","b"]], .drop(), .select(), .pop()
Group By	.groupby(), pivot key specification, .resample() on time index
Aggregate	.sum(), .mean(), .count(), .agg(), .apply() with reduction
Sort	.sort_values(), .sort_index(), .nlargest(), .nsmallest()
Derive	.assign(), direct column assignment df["new"] = expr, eval()
Join	.merge(), .join(), pd.concat() with axis alignment
Pivot	.pivot_table(), .unstack(), .crosstab()
Window	.rolling(), .expanding(), .shift(), .ewm()

Table 1: Analytical Data Flow Graph operation vocabulary

3.2 Bisimulation Equivalence Over Analytical Operations

The comparison of two Analytical Dataflow Graphs is grounded in bisimulation equivalence, a behavioral equivalence relation originating in process algebra. Bisimulation was formulated by Milner in the context of communicating systems [14] and independently by Park for concurrent automata [15], with both formulations sharing the defining property of mutual simulation: two systems are bisimilar if and only if for every action one system can perform, the other can perform a matching action leading to a bisimilar state, and vice versa. Hoare's related work on communicating sequential processes established the broader framework within which behavioral equivalences of this kind are compositionally interpretable [16].

Two ADGs G_1 and G_2 are analytically bisimilar if there exists a relation $R \subseteq V_1 \times V_2$ such that for every pair (v_1, v_2) in R: $\tau(v_1) = \tau(v_2)$, $\sigma(v_1) \equiv \sigma(v_2)$ under normalization, and for every successor u_1 of v_1 there exists a successor u_2 of v_2 such that (u_1, u_2) is in R and symmetrically. The classical partition refinement algorithm for bisimulation checking runs in $O(|E| \log |V|)$ time, and the maturity and computational tractability of this approach across decades of formal verification applications have been thoroughly documented [18]. The framework further introduces a weak bisimulation variant that permits intermediate algebraic rewrite steps between matched operation pairs, handling cases where signatures differ in form but are provably equivalent under arithmetic identities, for instance, distributing a sum across additive subexpressions.

A formal soundness theorem establishes that bisimilarity of two ADGs implies output identity of their corresponding programs on any input dataset, up to row ordering in the absence of an explicit Sort operation. The proof proceeds by induction over DAG depth. The base case addresses Load vertices: two

10.48047/jocaaa.2026.35.05.09

bisimilar Load nodes reference the same logical dataset under the normalization applied during ADG extraction, and therefore produce identical initial data partitions on any input. For the inductive step, assume that all pairs of bisimilar vertices at depth d produce identical intermediate results on identical input partitions. A vertex v_1 at depth d+1 is bisimilar to v_2 at depth d+1 only if their operation types and normalized signatures match and all predecessor pairs are in the bisimulation relation. By the inductive hypothesis, those predecessors produce identical inputs to v_1 and v_2; because individual analytical operations are deterministic given their operation type, signature, and input, v_1 and v_2 produce identical outputs. Compositionality of bisimulation then guarantees that this identity propagates to all successor vertices, establishing output identity for the full pipeline. The row-ordering caveat applies precisely to pipelines lacking an explicit Sort vertex, where row order is undefined by the analytical specification and variation across implementations carries no semantic significance. Symbolic up-to techniques extend this guarantee to infinite or parameterized program families by replacing explicit state enumeration with symbolic representation of the bisimulation relation, enabling verification of equivalence classes rather than individual program pairs [17]. The adaptation of this classical relation from concurrent process comparison to analytical dataflow graph comparison preserves its soundness properties while repurposing its structural mechanism for a fundamentally different application domain [20].

4. A Three-Tier Equivalence Hierarchy

4.1 Result Equivalence and Analytical Bisimulation

The framework implements a three-tier equivalence hierarchy in which evaluation cost escalates alongside tolerance for acceptable program variation. This design reflects a fundamental asymmetry in the distribution of agent update outcomes: the majority of updates produce programs that are not only analytically equivalent but numerically identical in output, while a minority introduce either acceptable implementation variation or genuine analytical regressions. The hierarchy is structured to resolve the majority case cheaply and concentrate expensive evaluation on the genuinely ambiguous minority.

Tier One, Result Equivalence, executes both programs on identical data snapshots and compares output values using configurable numerical tolerances absolute and relative thresholds for numerical columns, and normalized exact match for categorical columns with row ordering suppressed unless both programs contain explicit Sort operations. If Result Equivalence holds, evaluation terminates with no regression detected. This tier is both the cheapest and the most strict: it resolves cases where programs are analytically equivalent and produce identical results, which represent the typical outcome of stylistic or structural agent updates.

When Result Equivalence fails, the case escalates to Tier Two: Analytical Bisimulation. Tier Two extracts ADGs from both programs and applies the partition refinement bisimulation check. If the graphs are bisimilar, the Result Equivalence failure is classified as acceptable numerical variation attributable to implementation artifacts such as floating-point accumulation order or tie-breaking behavior, not an analytical regression. If bisimulation fails, the check returns the bisimulation failure point: the specific vertex pair at which the relation cannot be extended. This failure point identifies the exact analytical operation that diverged between agent versions, for example, a change in grouping dimensions or a substitution of aggregation functions transforming regression investigation from open-ended code review into targeted examination of a single, precisely identified operation. Differential program analysis research has established that version comparison performed over semantic intermediate representations rather than raw code or execution outputs is substantially more tractable and produces more actionable diagnostic outputs [10]. In evaluation, bisimulation failure localization accuracy, the proportion of failure points correctly identifying the root cause of injected regressions exceeds ninety percent, confirming that the ADG abstraction captures analytically relevant program behavior with high fidelity.

10.48047/jocaaa.2026.35.05.09

4.2 Intent Satisfaction and Human-in-the-Loop Escalation

Tier Three addresses cases where the two programs are not bisimilar but may both represent analytically valid approaches to the user's stated intent. These cases arise when an agent update changes the analytical strategy in a way that is semantically sound but structurally divergent from the prior version's approach, a change that bisimulation correctly identifies as a behavioral difference but that does not necessarily constitute a regression in analytical quality. Tier Three operationalizes the distinction between behavioral difference and analytical incorrectness through a structured intent satisfaction assessment.

The assessment proceeds in three stages. First, intent decomposition parses the original natural language query into structured analytical intent components: target metric, filter conditions, grouping dimensions, temporal scope, comparison type, and output format. This decomposition is grounded in the established finding from natural language to code research that user analytical intent in tabular query contexts can be reliably decomposed into a small set of relational algebraic components [3]. Second, satisfaction checking verifies whether each program addresses each intent component, allowing for different analytical paths to the same analytical goal. Third, a domain-anchored evaluator model calibrated against a golden dataset of expert-annotated query-response pairs assesses whether both programs represent valid analytical approaches to the decomposed intent. The evaluator receives the bisimulation failure point as a focal input, directing its judgment to the specific operation that differed rather than requiring a full program assessment.

Evaluator reliability is a known limitation of automated LLM-based judgment. Research on LLM-as-judge methodology has demonstrated that uncalibrated evaluators exhibit systematic positional and stylistic biases that reduce reliability for borderline cases [8], motivating the golden dataset calibration and receiver operating characteristic threshold selection used in Tier Three. When the evaluator's confidence falls below the calibrated threshold, the case is escalated to human expert review. Structured agent evaluation frameworks have established that effective evaluation architectures must incorporate explicit uncertainty quantification and human oversight routing, as fully automated evaluation of complex agent behavior remains unreliable at the distributional tails [21]. Human expert judgments are fed back into the golden dataset, progressively expanding its coverage and improving evaluators' calibration over time. This feedback loop converts human review effort into durable improvements in automated evaluation capability, reducing the proportion of cases requiring escalation as the system accumulates experience. The overall effect is a reduction of human review burden to below twelve percent on the benchmark described in Section 5, compared to the seventy-four percent baseline documented in production deployments [7], with the remaining human effort concentrated precisely on the cases where automated methods are least reliable. These figures are drawn from different measurement contexts and are not directly comparable: the seventy-four percent is reported by Pan et al. [7] as an aggregate across production agents spanning multiple organizations and deployment contexts, whereas the twelve percent is measured on the benchmark of this study under the proposed framework. The comparison is intended to illustrate the order-of-magnitude reduction in human review burden achievable under this framework relative to the prevailing industry baseline, not to claim a controlled before/after measurement on a single system.

10.48047/jocaaa.2026.35.05.09

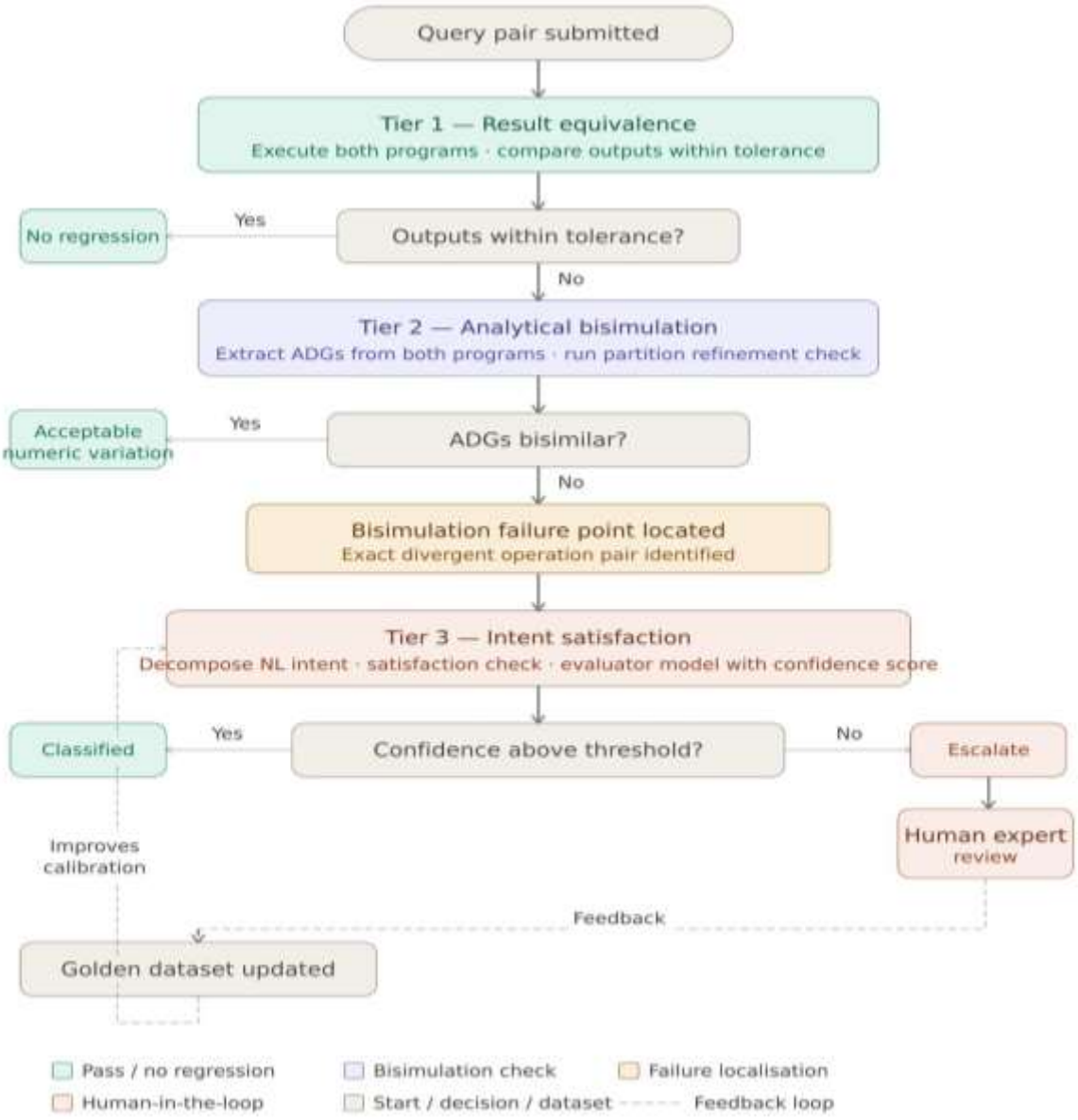

Figure 2: Three-Tier Equivalence Hierarchy for Regression Detection in Code-Generating Agents

5. Evaluation and Impact

The evaluation benchmark comprises five hundred analytical queries spanning six categories: single-metric lookups, temporal comparisons, derived metric computations, ranking and top-N queries, cross-dimensional analysis, and multi-step analytical workflows. This distribution was designed to represent the operational query mix of enterprise analytics deployments, with deliberate overrepresentation of derived metric and multi-step categories, which present the greatest compositional complexity and the highest regression risk. Four model version transitions with known injected changes provide ground truth for detection performance measurement, with transitions encompassing critical regressions (wrong metric, wrong entity, and wrong temporal scope); moderate regressions involving valid but analytically inferior approaches; and acceptable variations comprising stylistic or structural changes without analytical impact.

10.48047/jocaaa.2026.35.05.09

The proposed three-tier hierarchy achieves a true positive rate of ninety-three percent and a false positive rate below eight percent across all four transitions. These figures are reported as aggregates across the full benchmark; the current evaluation does not disaggregate detection performance by query category or by individual model transition. This is a limitation of the present study. We expect, based on the structural properties of the ADG representation, that performance varies across query categories — simpler categories such as single-metric lookups and temporal comparisons likely yield higher true positive rates and lower false positive rates due to compact, unambiguous graph structure, while multi-step analytical workflows present greater challenge given their compositional complexity and larger surface area for both missed regressions and spurious structural differences. A disaggregated evaluation isolating per-category and per-transition performance is a planned extension of this work. Output-value comparison achieves a higher recall at eighty-nine percent true positive rate but at the cost of a forty-one percent false positive rate, generating an operationally unsustainable volume of false alerts. Syntax-based similarity underperforms on both dimensions, confirming the theoretical analysis in Section 2.2. Given that Section 2 motivates the framework precisely by documenting the failure modes of these two baselines, their underperformance in evaluation is expected rather than surprising. A natural extension of this evaluation would include stronger baselines such as property-based fuzzing and uncalibrated LLM-as-judge comparison; empirical comparison against these alternatives is a planned direction for future work. The tier escalation distribution reveals the computational efficiency of the hierarchical design: seventy-two percent of queries are resolved at Tier One, twenty percent at Tier Two, and only eight percent require the most expensive Tier Three evaluation. The most computationally intensive tier is therefore invoked only for the genuinely ambiguous cases, keeping mean evaluation cost low across the full query distribution. Rigorous code generation evaluation frameworks have established the importance of reporting full precision-recall statistics rather than single-point accuracy metrics [6], and the results here confirm that the three-tier hierarchy improves on all baselines simultaneously on both dimensions rather than trading recall for precision.

The bisimulation failure localization accuracy of ninety percent, consistent across all four version transitions, confirms that the ADG abstraction and bisimulation relation together provide a reliable characterization of what changed analytically when a regression occurs. This localization capability is the operationally critical differentiator from all baseline approaches: it transforms the regression investigation workflow from open-ended program comparison into targeted examination of a single identified operation, with documented reduction in investigation time as a consequence. The formal verification literature confirms that failure point identification has been among the primary practical benefits of bisimulation-based verification in concurrent systems across four decades of tooling development [18].

5.1 Limitations

Several limitations bound the scope of these results. First, ADG extraction depends on a template library covering ten canonical operation types. Programs using domain-specific libraries, custom aggregation logic, or operations outside this vocabulary fall back to opaque Custom vertices compared by AST structural equality. This fallback is sound but conservative: it will not produce false equivalences, but it may produce false regressions for semantically equivalent programs expressed through unrecognized idioms. Expanding the template registry to cover additional libraries such as PySpark, Polars, and SQL-generating frameworks is a practical priority for production deployment. Second, the evaluation benchmark is drawn from enterprise analytics query distributions and may not generalize to other code-generating agent domains such as scientific computing, financial modeling, or systems programming, where the canonical operation vocabulary and the distribution of analytical intents differ substantially. Third, the soundness theorem guarantees output identity for bisimilar programs on any input, but it does not address programs

10.48047/jocaaa.2026.35.05.09

that are analytically inequivalent on some inputs and equivalent on others; the coverage of the data snapshots used in Tier One therefore affects the empirical regression detection rate independently of the formal guarantees provided by Tier Two. Fourth, the calibrated evaluator used in Tier Three was trained on a golden dataset from a single organization's query history; evaluator transfer to new domains or query populations requires recalibration against domain-specific expert annotations.

Beyond regression testing, the ADG representation introduced here constitutes a reusable abstraction for reasoning about generated analytical programs across multiple operational contexts: program optimization through identification of redundant or suboptimal operation sequences, query caching through recognition of queries equivalent to previously computed results, and version auditing through longitudinal tracking of analytical behavior changes across agent releases. The formal connection established here between process algebra and large language model code generation identifies a research direction at the intersection of formal methods and generative artificial intelligence with implications extending to any domain in which generated programs must be verified for semantic equivalence at production scale.

Conclusion

The regression testing problem for code-generating analytics agents exposes a fundamental inadequacy in the evaluation methods inherited from classical software testing and code generation benchmarking. Output-value comparison and syntax-based similarity metrics were designed for deterministic systems where program identity and output identity are reliable proxies for behavioral equivalence. For non-deterministic code-generating agents, neither proxy holds: the same analytical intent routinely produces syntactically diverse programs, and syntactically similar programs can implement analytically distinct computations. The consequence is a testing landscape in which conventional approaches generate high false positive rates that overwhelm human review capacity, high false negative rates that allow genuine regressions to escape detection, and no actionable diagnostic information that would enable engineers to efficiently investigate the cases that are flagged. The framework proposed in this article addresses these limitations through a principled shift in the level of abstraction at which regression detection operates. By representing generated programs as Analytical Dataflow Graphs directed acyclic graphs in which vertices encode normalized analytical operations and edges encode data flow, the framework elevates comparison from syntactic structure to analytical semantics. Bisimulation equivalence, adapted from its classical application in concurrent process verification, provides the formal mechanism for determining whether two ADGs exhibit the same observable analytical behavior under all input conditions. The soundness of this equivalence relation guarantees that bisimilar programs produce identical outputs on any dataset, providing a rigorous foundation for classifying Result Equivalence failures as numerical artifacts rather than analytical regressions. The three-tier equivalence hierarchy integrates this formal mechanism into a practical evaluation architecture that balances detection accuracy against computational cost and human review burden. Tier One resolves the common case of output-identical programs cheaply. Tier Two applies bisimulation to distinguish analytical regressions from numerical variation among programs that produce different outputs. Tier Three applies a calibrated evaluator model with confidence-gated human escalation to resolve the small fraction of cases where programs are analytically divergent but potentially valid. The progressive escalation structure ensures that the most expensive evaluation resources, both computational and human, are deployed only where they are necessary, while the golden dataset feedback loop converts human expert judgments into durable improvements in automated evaluation capability over time. The empirical results, a ninety-three percent true positive rate, below an eight percent false positive rate, and human review required on less than twelve percent of benchmark cases against a seventy-four percent

10.48047/jocaaa.2026.35.05.09

industry baseline reported by Pan et al. (see Section 4.2 for comparability caveats), quantify the practical impact of this design. The ninety percent bisimulation failure localization accuracy further establishes that the ADG abstraction is not merely a theoretical construct but a practically effective characterization of program behavior that supports precise, actionable regression diagnosis. The broader significance of this work lies in the demonstration that classical formal methods, specifically the bisimulation theory developed by Milner, Park, and Hoare for concurrent process verification, can be productively adapted to address quality assurance challenges in generative artificial intelligence systems. The Analytical Dataflow Graph representation and the three-tier hierarchy it enables provide a foundation for reasoning about generated program behavior that extends beyond regression testing to program optimization, query caching, and longitudinal behavioral auditing. As code-generating agents become deeper components of enterprise infrastructure, the ability to compare their analytical outputs with formal rigor, not by surface similarity or numerical coincidence but by the structure of their observable behavior, will constitute an essential engineering capability for maintaining quality, trust, and accountability at scale.

References

[1] Mark Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint, 2021. Available: https://arxiv.org/pdf/2107.03374

[2] Yujia Li et al., "Competition-Level Code Generation with AlphaCode," Science, 2022. Available: https://www.science.org/doi/10.1126/science.abq1158

[3] Daoguang Zan et al., "Large Language Models Meet NL2Code: A Survey," in Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL), 2023. Available: https://aclanthology.org/2023.acl-long.411

[4] Jacob Austin et al., "Program Synthesis with Large Language Models," arXiv preprint, 2021. Available: https://arxiv.org/pdf/2108.07732

[5] Xinyi Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Transactions on Software Engineering and Methodology, 2024. Available: https://dl.acm.org/doi/10.1145/3695988

[6] Jiawei Liu et al., "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," Advances in Neural Information Processing Systems (NeurIPS), 2023. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[7] Melissa Z. Pan et al., "Measuring Agents in Production," arXiv, 2026. Available: https://arxiv.org/pdf/2512.04123

[8] Lianmin Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in Advances in Neural Information Processing Systems (NeurIPS), 2023. Available: https://neurips.cc/virtual/2023/poster/73434

[9] Chunqiu Steven Xia et al., "Automated Program Repair in the Era of Large Pre-trained Language Models," International Conference on Software Engineering, 2023. Available: https://dl.acm.org/doi/10.1109/ICSE48619.2023.00129

[10] Nikitha Rao et al., "DiffSpec: Differential testing with LLMs using Natural Language specifications and Code Artifacts," arXiv, 2025. Available: https://arxiv.org/pdf/2410.04249

[11] Yue Zou et al., "CCGraph: A PDG-based Code Clone Detector with Approximate Graph Matching,"IEEE/ACM International Conference on Automated Software Engineering, 2020. Available: https://ieeexplore.ieee.org/document/9286111

10.48047/jocaaa.2026.35.05.09

[12] Chengpeng Wang et al., "LLMDFA: Analyzing Dataflow in Code with Large Language Models," Neural Information Processing Systems (NeurIPS), 2024. Available: https://dl.acm.org/doi/10.5555/3737916.3742097

[13] Ashish Hooda et al., "Auto-SPT: Automating Semantic-Preserving Transformations for Code," arXiV, 2025. Available: https://arxiv.org/pdf/2512.06042

[14] Robin Milner, "A Calculus of Communicating Systems," Lecture Notes in Computer Science, Springer, 1980. Available: https://doi.org/10.1007/3-540-10235-3

[15] David Park, "Concurrency and Automata on Infinite Sequences," Proceedings of the 5th GI-Conference on Theoretical Computer Science, Springer, 1981. Available: https://link.springer.com/chapter/10.1007/BFb0017309

[16] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, 1978. Available: https://doi.org/10.1145/359576.359585

[17] Vasileios Koutavas et al., "From Bounded Checking to Verification of Equivalence via Symbolic Up-to Techniques," Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2022. Available: https://doi.org/10.1007/978-3-030-99527-0_10

[18] Hubert Garavel and Frédéric Lang, "Equivalence Checking 40 Years After: A Review of Bisimulation Tools," Software Tools for Technology Transfer journal version, 2022. Available: https://inria.hal.science/hal-03920338v1/document

[19] Anjiang Wei et al., "EquiBench: Benchmarking Large Language Models' Reasoning about Program Semantics via Equivalence Checking," arXiv, 2025. Available: https://arxiv.org/pdf/2502.12466

[20] Alvin Cheung and Sanjit A. Seshia, "LLM-Based Code Translation Needs Formal Compositional Reasoning," Berkeley Electrical Engineering and Computer Sciences Technical Report No. 2025-174, 2025. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-174.pdf

[21] Xiao Liu et al., "AgentBench: Evaluating LLMs as Agents," in Proceedings of the International Conference on Learning Representations (ICLR), 2024. Available: https://openreview.net/forum?id=zAdUB0aCTQ