International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

Optimization of Regression Testing Using Graph-Based Dependency Models: A Theoretical Review

Pranay Raj Kanakala*

Worksoft Automation Consultant at MyTekX Inc,Oklahoma, United States
Email: pranayrajkanakalaa@gmail.com

Abstract

The paper provides a theoretical justification for optimizing regression testing using graph-based dependency models under conditions of frequent releases and high test execution costs. The relevance of the study is that the growth of software systems and the load on continuous integration infrastructure make full regression testing a chronic bottleneck, requiring formalized approaches to selecting, ordering, and reducing test suites at a controlled level of risk. The aim of the review is to synthesize the results of eight peer-reviewed studies and to construct a unified conceptual framework in which regression testing is interpreted as a risk-management problem over a dependency structure rather than as a mechanical re-execution of everything. The scientific novelty lies in interpreting the dependency graph as a carrier of a risk field, where edge and node weights reflect the strength of influence, interaction frequency, and node criticality, and an affected test is defined through reachability within an influence subgraph, subject to depth constraints and significance thresholds. The review systematizes three main classes of graph-based models (source-code level, component/service level, and bipartite tests–code graphs) and relates them to the operations of selection, prioritization, minimization, and time-budgeting, supplementing them with requirements for interpretability, reproducibility, and accounting for the total cost of ownership of the model. As a promising trajectory, the paper outlines a transition towards adaptive multimodal code–test–requirement–incident graphs enriched with observability data and learnable estimates of regression probability. The paper is intended for researchers and practicing software quality engineers who design or deploy methods for optimizing regression testing in scalable CI/CD processes.

Keywords: regression testing; graph-based dependency models; change impact analysis; test selection; test prioritization.

Received: *4/2/2026*
Accepted: *6/2/2026*
Published: *6/12/2026*

---

* *Corresponding author*.

1. Introduction

Regression testing is intended to repeatedly verify that, after changes are introduced, the correctness of already implemented functionality is preserved; however, this very repeatability makes it systemically expensive. As the system grows, the volume of checks increases, as does the number of possible combinations of inputs, states, and interactions. In engineering practice, the cost is expressed not only in machine time but also in the human effort required to maintain tests, analyze failures, and resolve false positives. The literature emphasizes that testing as a whole can account for 50–60% of the total project cost, whereas regression testing can account for up to 80% of the testing cost [1].

The situation is exacerbated under frequent delivery: changes are introduced in small increments and must be verified more often, so a full run quickly becomes a bottleneck in the build-and-verification pipeline. At the organizational budget level, this manifests as a significant cost item: empirical work on continuous integration infrastructures notes that large companies assess the cost of their continuous integration systems in the millions of dollars, and a substantial portion of this burden is associated precisely with repeated builds and test runs [2]. As a result, the trade-off appears stark: either spend more resources for greater confidence, or economize and accept the risk that a defect will surface later in subsequent life-cycle stages.

The key reason for such a high cost lies not only in the number of tests but also in the system's connection structure. A change rarely remains local; it spreads along call chains, module dependencies, and interaction contracts, giving rise to side effects that are difficult to predict from the modified location alone. Consequently, modern approaches increasingly rely on change impact analysis, in which dependent elements are identified through call graphs and other graph-based representations of dependencies. These models enable estimating potentially affected areas and thereby justify decisions about which checks to execute first and which to postpone without sacrificing a controlled level of risk [3].

2. Materials and Methods

This theoretical review is based on a targeted analysis of eight peer-reviewed sources that cover both applied regression-testing frameworks and contemporary directions in optimization via change-impact analysis and graph-based dependency representations. As the skeleton of the subject domain, the review uses works that fix the economic and engineering motivation for regression under continuous integration and for test selection/prioritization [1, 2], as well as studies that treat change impact analysis as an independent field of methods and terminology for describing the cascading propagation of change consequences [5]. Theoretical foundations of the graph-based approach were refined using sources in which regression optimization is linked to semantically meaningful changes and stricter definitions of test affectedness [4], relation-based models of prioritization in which not only nodes but also paths/call sequences become informative [6], traceability as a mechanism for stitching together artefacts (requirements–code–tests) for controlled selection [7], and the specifics of a microservice environment, where impact is evaluated as probabilistic propagation along inter-service links Reference [8]. Such a corpus keeps the review within a theoretical plane while preserving its connection to operational constraints (test-run cost, CI processes, change frequency) [2].

Methodologically, the study implements a synthetic review with elements of comparative conceptual analysis. From each source the following were extracted: definitions of optimization targets, affected test, impact region, dependency relation; assumptions underlying the construction of the dependency model, static code structures, dynamic execution/coverage data, traces, and operational logs; and types of optimization operations, selection, prioritization, minimization, and time-budgeting of the run [4–8]. Subsequently, a comparison was conducted against a unified set of parameters of the graph-based model (representation level, types of nodes and edges, extraction method, interpretation of weights/probabilities) in order to demonstrate how the same intention, reduce regression, gives rise to different formalisms: from reachability in a call graph to weighted/probabilistic schemes of impact propagation and coverage structures in bipartite graphs [5, 6, 8]. Finally, the comparison results were consolidated into a theoretical framework in which the graph is treated not as a binary mask of affectedness but as a carrier of risk tension: connection strength, transitivity, and execution context together define the compromise between selection completeness and its cost-effectiveness, while the requirement of explainability follows from the very nature of the graph-based solution, it must be reconstructable via influence paths and observable linking of tests to artefacts [6–8].

3. Results and Discussion

The accumulated body of research outlines several complementary trajectories that frame the present review. Demircioğlu and Kalipsiz develop an API message-driven regression testing framework and document the proportion of project budget consumed by repeated verification activity, establishing the economic baseline for any optimization argument [1]. Jin and Servant introduce CIBench as a dataset and methodological reference point for build and test selection in continuous integration pipelines and quantify the infrastructural burden carried by large-scale CI systems [2]. Göçmen and his colleagues extend change impact analysis into the domain of pull request reviews and demonstrate how dependency reasoning supports targeted verification decisions [3]. Liu and his colleagues refine the notion of test affectedness through reasoning about semantics-modifying changes and tighten the formal definition of when a regression test must be re-executed [4].

Kretsou and his colleagues provide a systematic mapping of change impact analysis and consolidate the terminology of cascading propagation across artefacts [5]. Chi and his colleagues propose relation-based test case prioritization in which call sequences carry information beyond node-level reachability [6]. Moldovan synthesizes the role of traceability in regression testing and connects requirement, code, and test layers into a navigable structure for selection [7]. Chen and his colleagues address microservice environments through belief propagation over inter-service dependencies and treat impact as a probabilistic quantity flowing along directed edges [8]. The convergence of these works around dependency structure, propagation semantics, and traceability motivates the unified graph-based perspective adopted in this review.

In the context of software system maintenance and evolution, regression testing is the deliberate re-execution of existing tests after a change in source code, configuration, or environment to confirm the preservation of observable behaviour and interface contracts. Its goal is not merely to repeat everything but to control risk: for each change, it is necessary to determine which previously confirmed properties may have become invalid, with usage scenarios ranging from defect fixing and refactoring to feature addition and dependency modification. At

the theoretical level, the central concept is that of an affected test: a test whose behaviour may change due to a transitive dependency on a modified fragment of the program, and precisely the identification of such affectedness underlies most rigorous approaches to reducing the cost of regression [4].

Optimization of regression testing usually decomposes into three interrelated tasks: test selection, prioritization, and test-set minimization. Selection seeks to exclude checks that, with high confidence, cannot respond to the change; prioritization reorders the remaining checks so that the first minutes of execution yield maximum informativeness and probability of early regression detection; minimization, in turn, attempts to remove redundancy while preserving a specified adequacy criterion. Graph-based models introduce a rigorous language for these operations: nodes represent artefacts at different levels (functions, classes, modules, services, requirements, tests), edges capture dependency or potential influence relations, direction encodes asymmetry of change propagation, and weights distinguish strong and weak links by probability or error cost. This formalism is naturally aligned with change impact analysis as a separate area that emphasizes the idea that the value of a model is determined not only by the completeness of coverage of dependencies but also by how controllably it allows evaluation of the risk of cascading effects during maintenance [5].

The data sources used to construct the graph fundamentally determine its accuracy and practical utility. Static analysis extracts dependencies from the program structure and build configurations: imports, module relationships, inheritance, calls, and field accesses form a static call and dependency graph that scales well but tends to inflate the set of possible impact paths. Dynamic analysis is built on actual runtime behavior during test execution: execution traces and coverage data link tests to the regions they actually visit; in distributed systems, this layer is supplemented by distributed tracing, which reconstructs inter-service chains of causality. Consequently, methods that use call sequences as a relational representation of behavior treat the program as a network in which not only nodes but also paths are informative [6].

Hybrid models aim to combine the skeleton of static structure with dynamic weights to reduce false positives in selection while retaining hidden dependencies. Additionally, when artefact traceability is introduced into the process, the graph is extended by requirement–component–test links, providing a foundation for optimization that accounts not only for technical proximity to the change but also for the semantic significance of the properties being verified [7]. When a graph-based model is constructed at the source-code level, nodes typically correspond to functions, methods, classes, or files, and edges encode calls and structural dependencies among them; under this formulation, regression optimization becomes the problem of localizing the impact zone of a change within the program's execution flow. A crucial nuance is that the same set of affected entities can give rise to different observable execution trajectories; therefore, the graph is often extended by explicitly representing paths rather than merely reachability. This allows behavior to be described as a network of calls in which sequences of transitions between nodes are informative [6]. On this basis, change-impact graphs naturally emerge: a change is treated as a source, and the set of potentially affected artefacts and their associated tests as the region of propagation. Theoretically, this is convenient because it provides a unified language for discussing selection completeness and inevitable trade-offs between not missing and not overloading [5]. If the system is described at the component and service level, the dependency graph starts to reflect not so much module structure as interaction structure: who requests data from whom, who depends on whom via a contract, and which call chains cross process

and deployment boundaries. In a monolithic architecture, many links remain within a single address space and the graph can be more easily aligned with the source code; in a microservice architecture, the links manifest as network calls and protocols, so the graph is more often extracted from operational traces, where inter-service traces and their fragments become the unit of observation. Practically significant is the fact that such graphs can be constructed even when development artefacts are distributed across teams and are not available as a single whole: dependencies are reconstructed from gateway logs and aggregated observability data, after which the impact of a change is evaluated as a quantity that spreads along directed edges [8]. Bipartite tests–code elements graphs occupy a distinct niche because they deliberately narrow the picture to the coverage relationship: one partition contains tests, the other program entities, and edges denote execution or an observed dependency. This form is convenient for minimization and for rigorous comparison of alternative test sets, because it reduces the problem to structured coverage of a set. However, bipartiteness almost always requires an additional layer. Otherwise, the model conflates reachability with risk. Hence, in contemporary theoretical reviews, weighted and probabilistic graphs become a natural extension: an edge ceases to be just a link and becomes a carrier of influence strength, interaction frequency, or node criticality. In this logic, the change-impact graph ceases to be a binary affected/not affected mask and turns into a field of tensions in which different paths compete, reinforce one another, or, conversely, cancel out; precisely this opens the way to more economical regression without the illusion that all dependencies are equally dangerous [8]. Table 1 shows the types of graph-based dependency models.

Table 1: Types of graph dependency models

Parameter	**Source code level**	**Component/service level**	**Bipartite tests-code**
Model level	source code	components and services	tests ↔ code elements
Nodes	functions/methods/ classes/files	components/services	1) tests 2) code elements
Edges	calls + structural dependencies	data requests, contract dependencies, cross-boundary call chains	execution/coverage fact (or observed dependency)
How it's built/extracted	static code analysis	architecture (monolith) or operational traces (tracing, logs, gateway)	coverage/execution data
Regression testing objective	localize the change impact zone in the program's execution flow	estimate impact as spreading along directed edges	minimize the test set and compare alternatives via structured set coverage
Key nuance/ extension	The same affected entities can yield different execution trajectories → model **paths** (call sequences), not just reachability	In microservices, links appear as network calls → graphs are often reconstructed from traces/logs	reachability ≠ risk → add **weights/probabilities** (influence strength, frequency, criticality), making impact a field rather than a binary mask

The theoretical value of a graph-based model in regression testing manifests itself when optimization is formulated as a rigorous problem over the structure of dependencies rather than as an intuitive choice of checks. Within this logic, a change is regarded as a source of perturbation that propagates along directed dependencies and forms a region of potential impact. Test selection is then reduced to reachability: a test is considered a candidate if it lies within the reachable region from the modified artefacts along influence edges. To manage the trade-off between completeness and redundancy, constraints on propagation depth are introduced, along with thresholds on weights or probabilities that cut off weak paths carrying negligible practical risk.

Prioritization is naturally interpreted as ranking test nodes according to their position within the influence structure. The most direct criterion is associated with distance to the source of change: the shorter the path, the fewer intermediate assumptions and the higher the probability that the test will respond to a regression. However, the same path length may traverse nodes of different importance, so structural characteristics of the graph that reflect a node's role in propagating effects are incorporated. As a result, the test order emerges from the combination of proximity to the change, importance of affected nodes, and an integral risk estimate that aggregates weights along paths.

Test-set minimization arises when the impact region has already been determined, but a minimal set of checks must be chosen that still covers the key elements of that region. In a strict formulation, each test corresponds to a set of affected entities it observes, and the selection task becomes a set-cover problem: to find a test subset that covers the entire required region at minimal cost, or to guarantee that each critical entity is intercepted by at least one test. Since exact solutions are computationally expensive, practical approaches adopt approximate schemes in which selection proceeds stepwise, adding the test with the largest gain in coverage relative to its cost or duration at each step.

Time-budget-constrained optimization brings the formulation even closer to real limitations of the delivery pipeline: only as many tests can be run as fit into the available time window. Here, the objective changes: instead of full coverage, the goal is to maximize expected utility under a given total duration. Utility may be defined as weighted coverage of the impact region, as expected risk reduction, or as the probability of early defect detection. Within such a model, some tests may be deliberately excluded even if they are reachable, provided that their contribution to risk reduction is small relative to their cost.

Algorithmically, selection and construction of the influence subgraph rely on classical graph-traversal procedures that sequentially expand the reachability frontier from the set of modified nodes. Such traversals are easily extended with constraints: propagation may stop upon reaching a given depth, edges below a weight threshold may be ignored, and a branch may be terminated once its accumulated influence becomes negligible. At the same time, not only traversal speed but also the shape of the result is important: the subgraph must preserve explainable paths so that decisions to execute or skip a test do not appear arbitrary.

Prioritization and importance estimation use metrics that measure the centrality of nodes in the dependency network and distinguish the role of tests as observers of critical influence routes. Some metrics are interpreted as the ability of a node to accumulate influence via multiple incoming edges; others as participation in shortest paths

and thus potential coverage of critical regression-propagation channels. For set-cover problems, greedy and heuristic methods are more common, as they offer predictable quality at reasonable cost and interact well with weighting schemes. Finally, since the graph changes along with the code and environment, incremental update techniques become essential: instead of full reconstruction, only the affected parts of the model are adjusted, which enables optimization to function as an ongoing process rather than a one-off analytical procedure. Figure 1 illustrates the Graph Model in Regression Testing.

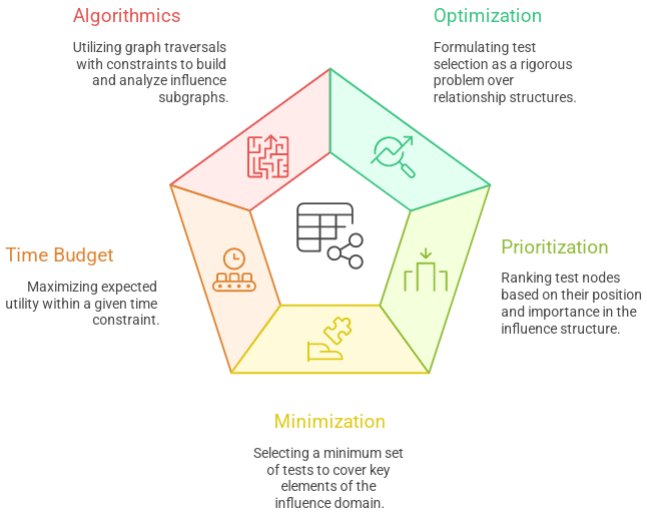

Figure 1: Graph Model in Regression Testing

Evaluation of the effectiveness of regression testing optimization based on a dependency graph begins with treating the result of test selection as a classification: some checks are deemed relevant to the change, others not. Precision and recall thus become natural criteria. Precision measures how often the tool is wrong when it reports that a test is an omission, while recall measures how often the tool fails to report a test when it is a potential regression. These two metrics are usually in conflict. For example, high precision may fail to detect rare but important influence paths, whereas high recall may yield a large test set, losing the ideal time savings.In graph-based models, this dilemma is particularly acute because of edge thresholds, propagation-depth limits, and choices about which dependencies to treat as causal.

For prioritization, the central question is not how many tests were executed but how quickly a signal about a problem was obtained. Accordingly, metrics that relate the execution order to the dynamics of defect detection become important. One metric aggregates how early failures appear in the sequence; another records the actual time until the first failure. These indicators complement each other: the aggregated measure characterizes the overall quality of the ranking, while the time to first failure captures the practical effect in continuous verification, where the main goal is to stop the pipeline quickly and return feedback to the developer. In the graph-based setting, the meaning of such metrics is heightened because the test order is determined not only by historical data but also by the structure of influence paths. Evaluation must therefore confirm that structural features genuinely accelerate defect discovery rather than merely supporting an elegant theory.

Finally, any optimization makes sense only when the total cost is accounted for: the gain in test-run time and the

cost of constructing and maintaining the model. It is important to measure not only the reduction in test execution duration but also the costs of data collection, graph update, and priority computation; in conditions of frequent changes, even moderate computational overhead can become a new bottleneck. Interpretability constitutes a separate criterion: decisions must be explainable so that engineers can trust why certain tests are excluded, and others promoted; otherwise, the system is likely to be gradually overridden manually. Reproducibility complements this requirement: the same change applied to the same data should yield the same selection. Otherwise, optimization becomes a source of instability that complicates failure analysis and comparison of results across runs. Regression Testing Optimization Evaluation principles are shown in Figure 2.

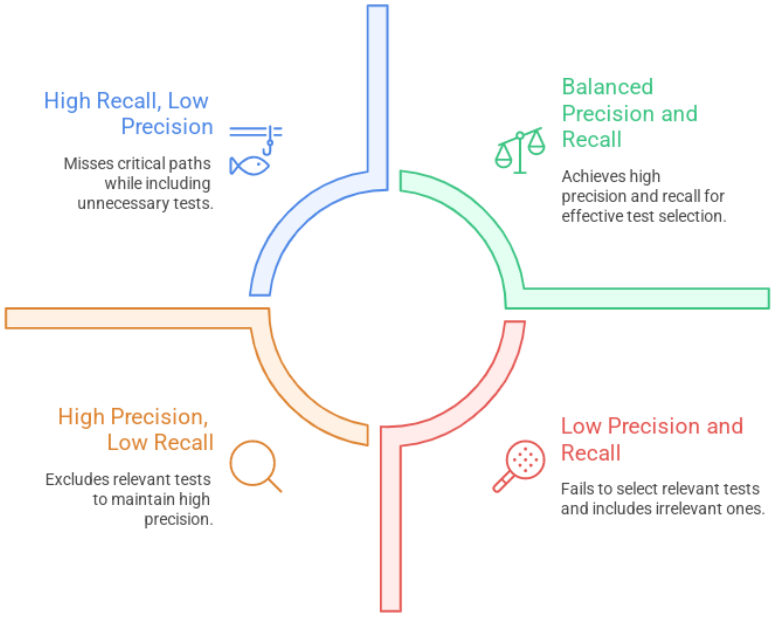

Figure 2: Regression Testing Optimization Evaluation

Future directions involve turning the graph from a rigid reachability scheme into an adaptive risk model. Edge and node weights can be oriented towards regression probability, taking into account change intensity, defect history, and structural complexity, so that test selection is determined not only by proximity to the change but also by expected error cost. For service-oriented systems, tight integration with observability is particularly important: traces and logs provide an empirical picture of inter-component paths, enabling refinement of the graph and discovery of dependencies that are invisible at the code level.

A further step is the integration of heterogeneous knowledge into a multimodal graph in which code, tests, requirements, and incidents form a single fabric of causes and effects; optimization then can account not only for technical influence but also for the semantic criticality of behavior. To preserve trust, explainable decisions are required: the system must be able to show the influence path and the contribution of each selected test. Otherwise, rigorous mathematics will be perceived as an opaque heuristic. Finally, the graph can be combined with learnable models that estimate the probability of a test failure based on change features and dependency structure. In such a combination, the graph provides a causal backbone, and statistical prediction fills gaps and smooths noise without replacing engineering interpretation.

The synthesis of the reviewed material reveals a coherent gradient running through the three classes of graph-based models. Source-code graphs anchor the analysis in fine-grained call structure and yield high resolution at the cost of inflated impact regions, component and service graphs trade granularity for the ability to capture cross-boundary causality reconstructed from operational traces, and bipartite tests-code graphs concentrate analytical power on coverage relations at the price of conflating reachability with risk unless weighted edges are introduced. The interpretation of these models as instruments of selection, prioritization, minimization, and time-budgeting demonstrates that the same formal apparatus serves divergent engineering objectives once the optimization target is fixed.

Reachability constrained by depth and weight thresholds governs selection, centrality and proximity govern prioritization, set-cover formulations govern minimization, and expected utility under duration constraints governs budgeted runs. The cross-cutting result is that graph-based optimization gains traction when classification quality, signal acquisition speed, and total cost of ownership are evaluated as a single triad rather than treated as isolated indicators. The discussion further indicates that the explanatory power of the graph emerges from the preservation of influence paths during traversal. This property differentiates principled regression reduction from opportunistic heuristics and links the formal model with engineering trust in automated selection decisions across continuous integration cycles.

4. Conclusion

The theoretical review presented here frames regression testing not as mechanical repetition, but as a discipline of risk management in contexts where repeatability inevitably becomes a systemic cost. As the software system grows, so does the test suite, the cost of running it, and the human cost of maintaining the tests, debugging them when they fail, and eliminating false positives. In short release cycles, an expensive test run can become a chronic bottleneck in the pipeline, forcing developers to choose between running expensive tests and discovering defects late in the process. The primary cause of such tension lies in the topology of dependencies: changes rarely remain local and propagate along call chains, module dependencies, and interaction contracts, so the key theoretical object becomes the affected test, whose behavior can change due to a transitive dependency on a modification.

The central contribution of graph-based dependency models is that they cast regression optimization as a rigorous problem over a structure of influence: a change is interpreted as a source of perturbation, and the system as a directed network along which this perturbation spreads, forming a region of potential impact. In this logic, test selection reduces to reachability within the influence subgraph, prioritization to ranking by proximity to the source and the role of nodes in propagation routes, and minimization to a set-cover problem over affected entities under minimal cost. The review emphasizes that the practical soundness of the model is determined by the provenance of data: static graphs scale well but tend to inflate the space of possible paths; dynamic coverage and tracing data capture actual behavior, which is particularly valuable in distributed systems; the hybrid approach attempts to maintain a structural skeleton while simultaneously reducing noise via weights and probabilities. In doing so, a conceptual shift is achieved from a binary affected/not affected mask to a risk field, where connection strength, interaction frequency, and node criticality enable more economical test selection without the illusion that all dependencies are equally hazardous.

The final conclusion of the review is that the value of graph-based optimization manifests only when three feedback loops are considered simultaneously: the quality of selection as classification (precision and recall), the quality of prioritization as speed of signal acquisition (earliness of detection and time to first failure), and the total cost of ownership of the model (data collection, graph update, decision computation), supplemented by requirements for interpretability and reproducibility.

The scope of the present review is shaped by several boundary conditions that situate the obtained conclusions within a defined analytical perimeter. The argument rests on eight peer-reviewed sources spanning regression testing frameworks, change impact analysis, traceability, and microservice-oriented selection, which establishes a coherent thematic field and opens a natural extension toward industrial reports, open-source case archives, and longitudinal observations of CI pipelines in production settings. The synthesis operates at the conceptual level and treats graph-based dependency models as a unified formal apparatus, while the calibration of edge weights, propagation thresholds, and centrality estimators against measured defect propagation traces marks a productive direction for subsequent inquiry.

The comparative interpretation of selection, prioritization, minimization, and time-budgeting relies on qualitative criteria distilled from the literature, and the construction of quantitative metrics tied to precision-recall trade-offs, time to first failure, and total cost of ownership represents a further analytical horizon. The conceptual framework treats classification quality, signal acquisition speed, and model maintenance cost as a single triad, and the disaggregation of these dimensions through dedicated empirical instruments offers a path toward more granular decision support for regression testing in scalable CI/CD environments.

The prospective trajectory of development is described as transforming the graph from a rigid reachability scheme into an adaptive risk model: weights are oriented towards regression probability and error cost, observability in service systems refines actual inter-component paths, and multimodal code–test–requirement–incident links make it possible to account not only for technical proximity to the change but also for the semantic criticality of behavior. At the same time, the robustness of the approach is tied to explainability: the system must be capable of presenting the influence path, and the contribution of each selected test, otherwise even a rigorous formalization will be perceived as an opaque heuristic.

References

[1] E. D. Demircioğlu and O. Kalipsiz, “API Message-Driven Regression Testing Framework,” Electronics, vol. 11, no. 17, p. 2671, Aug. 2022, doi: https://doi.org/10.3390/electronics11172671.

[2] X. Jin and F. Servant, “CIBench: A Dataset and Collection of Techniques for Build and Test Selection and Prioritization in Continuous Integration,” 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 166–167, May 2021, doi: https://doi.org/10.1109/icse-companion52605.2021.00070.

[3] I. S. Göçmen, A. S. Cezayir, and E. Tüzün, “Enhanced code reviews using pull request based change impact analysis,” Empirical Software Engineering, vol. 30, no. 3, Feb. 2025, doi:

https://doi.org/10.1007/s10664-024-10600-2.

[4] Y. Liu, J. Zhang, P. Nie, M. Gligoric, and O. Legunsen, “More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes,” Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Jul. 2023, doi: https://doi.org/10.1145/3597926.3598086.

[5] M. Kretsou, E.-M. Arvanitou, A. Ampatzoglou, I. Deligiannis, and V. C. Gerogiannis, “Change impact analysis: A systematic mapping study,” Journal of Systems and Software, vol. 174, p. 110892, Apr. 2021, doi: https://doi.org/10.1016/j.jss.2020.110892.

[6] J. Chi et al., “Relation-based test case prioritization for regression testing,” Journal of Systems and Software, vol. 163, p. 110539, May 2020, doi: https://doi.org/10.1016/j.jss.2020.110539.

[7] A.-M.-N. Moldovan, “Regression Testing via Traceability: A Systematic Literature Review,” Lecture Notes in Computer Science, vol. 16082, pp. 219–234, Sep. 2025, doi: https://doi.org/10.1007/978-3-032-04200-2_15.

[8] L. Chen, J. Wu, H. Yang, and K. Zhang, “A microservice regression testing selection approach based on belief propagation,” Journal of Cloud Computing, vol. 12, p. 20, Feb. 2023, doi: https://doi.org/10.1186/s13677-023-00398-7.