CAPSTONE PROJECT FINAL REPORT

Temporal Context Layer for Code Repositories using Knowledge Graphs

A Graph-Based Framework for Impact-Aware Test Prioritization in Safety-Critical Automotive Software

In Partnership With

Honda Research Institute USA | 99P Labs

Program: M.S. Data Science, University of Colorado Boulder

Course: DTSC 5810 - Data Science Capstone

Industry Mentor: Ryan Lingo (Honda Research Institute USA)

ryan_lingo@honda-ri.com

Team Members: Roshan Muddaluru, Shivani Madan, Harshitha Attanti

roshan.muddaluru@colorado.edu , shivani.madan@colorado.edu , harshitha.attanti@colorado.edu

Date: April 24th 2026

Abstract

The modernization of automotive software has elevated source code from a passive implementation artifact to a safety-relevant operational asset whose evolution must be managed with precision. A single numeric adjustment within a firmware file can silently propagate through function call chains and reach tests that no engineer explicitly connected to the change, creating the risk of undetected regressions in safety-critical subsystems. This project, conducted in partnership with Honda Research Institute USA and 99P Labs, develops a Temporal Context Layer for code repositories built on a property knowledge graph, along with a priority-scoring engine that ranks every test in a codebase from CRITICAL to SAFE for any given parameter change. The system integrates structural code topology, extracted via an AST-style parser, with temporal and scenario context derived from curated engineering change metadata. Both dimensions are fused into a single unified knowledge base that is ingested into Neo4j 5 through idempotent Cypher operations. A weighted scoring model combining Proximity, PageRank centrality, and FanOut breadth produces interpretable, auditable test rankings with plain-English justification for each result. Across four realistic change scenarios, the system identified between four and six mandatory tests per change while confirming that 72 to 84 percent of the full test suite could be safely skipped, delivering substantial CI/CD efficiency gains without compromising safety coverage.

Table of Contents

List of Figures

1. Introduction

1.1 Background and Motivation

Over the past two decades, the automotive industry has undergone a profound transformation, shifting from mechanically dominated systems to software-defined vehicles. Modern cars rely heavily on software to control essential functions such as braking, steering, navigation, and safety systems. Features like adaptive cruise control, collision avoidance, and sensor-based decision-making are no longer optional luxuries but core components of everyday driving. As a result, the reliability of software directly influences not only vehicle performance but also passenger safety. This growing dependence on software has introduced new challenges for engineers, particularly in managing complex and constantly evolving codebases. Unlike traditional mechanical components, software systems are highly interconnected, meaning that even small changes can have widespread and unintended effects. Ensuring that every change behaves as expected is therefore critical in maintaining safety and reliability. This shift has made software testing and validation one of the most important aspects of automotive development.

One of the key challenges in this environment is understanding how changes in software impact the overall system. In large automotive codebases, updates are frequent and often involve multiple components interacting with each other. A seemingly minor modification, such as adjusting a parameter or updating a function, can influence several downstream processes. Without clear visibility into these dependencies, engineers may struggle to determine which parts of the system are affected. This uncertainty often leads to two inefficient approaches: either running all available tests, which is time-consuming and resource-intensive, or selecting a limited set of tests based on intuition, which risks missing critical issues. Both approaches can reduce confidence in the final system and slow down development cycles. As automotive software continues to grow in complexity, there is an increasing need for smarter, more informed ways to manage testing and validation. Addressing this challenge is essential for ensuring both efficiency and safety in modern vehicle development.

1.2 Industry Context and Stakeholders

The importance of efficient software validation extends beyond engineering teams and affects a wide range of stakeholders. Automotive manufacturers rely on robust software systems to maintain their reputation for safety and innovation. Quality assurance teams are responsible for verifying that every software update meets strict safety standards before deployment. Safety engineers must ensure compliance with regulatory requirements and industry standards, which often demand traceability and accountability for every change made to the system. In addition, project managers and release teams depend on efficient testing processes to meet tight development timelines. Even end users, though indirectly involved, are impacted by these processes, as they expect reliable and safe vehicle behavior under all conditions. Any failure in

software validation can lead to recalls, financial losses, or, in the worst cases, safety hazards. Therefore, improving how software changes are analyzed and tested is not just a technical concern but a critical business and safety priority.

Historically, testing strategies in automotive software have relied heavily on exhaustive validation methods. While these approaches ensure thorough coverage, they are often inefficient and do not scale well with increasing system complexity. As software systems grow larger, the number of possible interactions between components increases significantly, making it impractical to test everything after every change. This has led to a growing interest in more intelligent testing strategies that can prioritize what needs to be tested based on the impact of a change. Such approaches aim to balance efficiency with reliability, ensuring that critical issues are detected without unnecessary resource expenditure. The evolution of testing practices reflects the broader shift toward data-driven decision-making in engineering. By leveraging structured information about software systems, teams can make more informed choices about where to focus their efforts. This project is motivated by the need to support this transition toward smarter and more efficient testing methodologies.

1.3 Project Overview

This project addresses the challenge of understanding and managing the impact of software changes in automotive systems. The central idea is to organize scattered information within a software repository into a structured form that allows engineers to better understand relationships between different components. Instead of treating code, changes, and tests as separate entities, the project brings them together into a unified representation. This enables a more holistic view of how different parts of the system interact with each other over time. By making these connections visible, engineers can more effectively assess the consequences of a change and determine appropriate testing strategies. The approach focuses on improving decision-making rather than replacing existing workflows, making it both practical and adaptable to real-world scenarios. The system ultimately helps answer a critical question: which tests are most relevant for a given change?

The solution developed in this project provides a structured way to analyze software changes and their potential impact. It captures relationships between various elements such as code components, historical changes, and testing outcomes, allowing for a more informed evaluation process. By leveraging these relationships, the system can guide engineers in selecting the most relevant tests to execute. This reduces unnecessary effort while maintaining confidence in system reliability. The outputs are designed to be clear and interpretable, ensuring that engineers can understand the reasoning behind each recommendation. This transparency is particularly important in safety-critical domains like automotive systems, where decisions must be justifiable. Overall, the project contributes to a more efficient and reliable software validation process.

1.4 Scope and Objectives

The scope of this project was carefully defined to ensure that the final solution would be both practical and impactful. The primary goal was to create a system that improves how software changes are analyzed and tested within an automotive context. This involved developing a structured representation of software components and their relationships, as well as incorporating information about historical changes and testing outcomes. Another key objective was to enable more efficient test selection by identifying which tests are most relevant to a specific change. The project also aimed to ensure that all outputs are interpretable, allowing engineers to understand and trust the results. In addition, maintaining traceability was an important consideration, ensuring that decisions could be linked back to their underlying data. These objectives were designed to align with real-world engineering needs and constraints.

To achieve these goals, the project focused on delivering a complete workflow that supports change analysis and testing decisions. This includes capturing relevant information, organizing it into a meaningful structure, and providing outputs that guide engineers in their testing efforts. The system was designed to be adaptable, allowing it to be applied to different types of automotive software systems. Emphasis was placed on usability, ensuring that the solution can be integrated into existing workflows without significant disruption. By focusing on both functionality and practicality, the project aims to bridge the gap between theoretical approaches and real-world application. The defined scope ensured that the work remained focused and achievable within the given timeframe. Ultimately, the project seeks to enhance both efficiency and confidence in automotive software validation.

1.5 Value and Impact

The value of this project lies in its ability to improve the efficiency and reliability of software testing in automotive systems. By providing a structured way to understand the impact of changes, the system helps reduce unnecessary testing while ensuring that critical issues are not overlooked. This leads to faster development cycles and more efficient use of resources. At the same time, the approach maintains a strong focus on safety, which is essential in the automotive domain. Improved testing strategies can help prevent software defects from reaching production, reducing the risk of failures in real-world scenarios. This has direct implications for both manufacturers and end users, contributing to safer and more reliable vehicles. The project also supports better decision-making by providing clear and interpretable outputs.

Beyond immediate benefits, the project reflects a broader shift toward smarter and more data-driven engineering practices. As software systems continue to grow in complexity, traditional approaches to testing and validation may no longer be sufficient. Solutions that leverage structured information and relationships within software systems offer a promising path forward. By demonstrating the feasibility and benefits of such an approach, this project contributes to ongoing efforts to modernize software engineering practices in the automotive industry. The ideas and

methods explored here can potentially be extended to other domains where complex software systems play a critical role. Ultimately, the project highlights the importance of innovation in addressing emerging challenges in software development. It provides a foundation for future work aimed at improving efficiency, reliability, and safety in complex systems.

[Figure 1.1: Domain Related Image]

2. Problem Statement

2.1 Current Challenges in Software Change Management

Managing change in a heterogeneous, safety-relevant software codebase is a fundamentally different activity from managing change in most commercial software domains. In automotive firmware and control software, the consequences of an incorrect release are not confined to degraded user experience or transactional errors; they manifest as physical behaviors of the vehicle that can affect passenger safety. This elevated consequence profile collides with the operational reality of how modern automotive codebases evolve: hundreds of files, dozens of engineers, mixed C++ and Python layers, hardware interface specifications that bind specific code regions to regulatory obligations, and a continuous pressure to release updates quickly to address field issues or introduce new capabilities.

Within this environment, three systemic challenges recur across engineering teams. First, institutional knowledge of how code components interconnect resides primarily in the memories of senior engineers and is neither evenly distributed nor reliably captured in documentation. Second, static analysis tools are effective at describing the current state of the code but do not reason about temporal context - who has historically changed a file, how often it is modified, or which past changes correlated with downstream failures. Third, test suites have grown to sizes where exhaustive regression on every change is operationally impractical, prompting teams to adopt ad-hoc selection heuristics that lack a principled or auditable basis.

2.2 Impact of Untracked Dependencies

The practical consequence of untracked dependencies is that downstream tests affected by a change are frequently not run, and failures that would have been caught early instead surface during integration, validation, or in the field. Each failure mode carries its own cost profile. A regression caught during integration testing imposes rework and schedule disruption on a coordinated team. A regression caught during vehicle-level validation imposes significantly greater cost because physical test assets must be rerun. A regression that escapes to field deployment can trigger recalls, warranty actions, and in the worst case, incidents with safety implications.

Beyond these direct cost consequences, untracked dependencies erode the confidence of engineering and management stakeholders in the release process itself. When engineers cannot articulate why a particular test set was chosen for a given change, release decisions accumulate invisible risk that compounds over successive iterations. When quality assurance teams default to running the entire test suite as a protective measure, the throughput of the release pipeline is constrained and the organization pays the computational and scheduling cost of over-testing. Neither outcome is tenable in an industry moving toward more frequent software updates and more capable vehicle behaviors.

2.3 Need for Automated Dependency Visualization

A durable resolution to these challenges requires that dependency information be captured, unified, and exposed in a form that supports both automated reasoning and human investigation. A knowledge graph is particularly well suited to this role because it natively represents entities and the typed relationships among them, it supports traversal queries that match the shape of real engineering questions (what does this change affect, who owns that file, which tests can reach this function), and it lends itself to visual exploration for stakeholders who do not write queries directly. The need is therefore not merely for better tooling but for a structural representation of dependency information that serves engineers, testers, safety analysts, and managers through a single shared artifact.

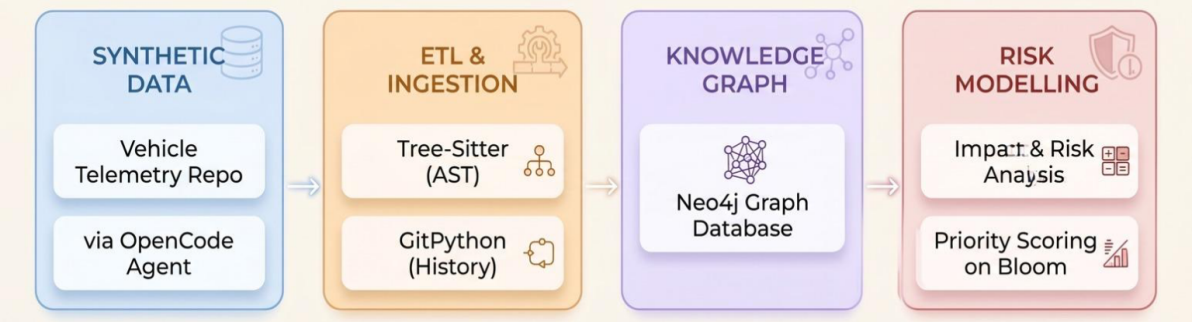
3. Analysis

3.1 Research Approach

The research approach adopted in this project is design-oriented and empirical. Rather than advancing a single theoretical claim through controlled experimentation, the work constructs an operational system and evaluates its behavior against realistic engineering scenarios. This framing is consistent with the expectation set by the project partner, whose interest was explicitly in whether a knowledge-graph-based approach could deliver interpretable, auditable, and

quantifiable improvements over exhaustive regression testing in a mixed-language automotive context.

The pipeline adopts a layered ETL structure with five connected stages, each producing an inspectable artifact and each independently replaceable. Stage one performs structural extraction of code topology via AST-style parsing. Stage two performs temporal extraction of scenario and change context from curated engineering metadata. Stage three fuses both outputs into a single unified knowledge base. Stage four ingests this unified base into Neo4j through idempotent Cypher operations. Stage five executes the priority-scoring engine, producing ranked test lists per change scenario. This layered design supports both incremental development (each stage can be validated independently) and downstream extension (new signals or data sources can be added without disturbing the core graph model).



[Figure 3.1: Four-stage end-to-end pipeline architecture]

Architecture diagram showing the flow from synthetic repository through AST extraction, scenario extraction, context fusion, Neo4j graph ingestion, and priority scoring - with intermediate JSON artifacts labeled between each stage.

3.2 Tools and Technologies Used

The technology stack was selected to prioritize reproducibility, portability, and transparency. All components are either open source or freely available, and the full pipeline can be reconstructed on a standard development workstation in under two minutes.

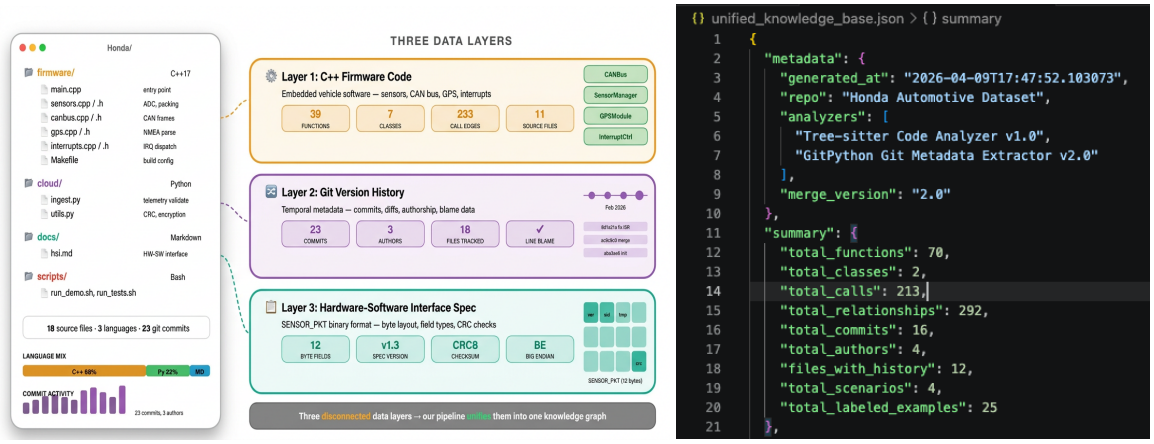
Category	Tool / Platform	Role in the Project
Language	Python 3.13	Primary implementation language for all pipeline stages
Graph Database	Neo4j 5 Desktop	Property graph storage, Cypher traversal, constraint enforcement
Graph Analytics	Neo4j GDS Plugin	PageRank centrality computation on the call subgraph
Query Language	Cypher	Pattern matching, shortest-path, MERGE/UNWIND ingestion
Parsing	Regex-based AST parser	C++ and Python topology extraction (portability-first)
Serialization	JSON	Intermediate artifacts between pipeline stages
Visualization	Neo4j Bloom	Stakeholder-facing graph exploration
Version Control	Git / GitHub	Source control, milestone tracking, collaborative development

Table 3.1: Tools and technologies used in the implementation.

3.3 Data Collection and Processing

The dataset used in this project is a synthetic but realistic automotive software repository generated by OpenCode. The decision to work with synthetic data was deliberate: it provided the structural and behavioral characteristics of a production automotive codebase - multiple languages, hardware interface specifications, commit history, and multi-author ownership - without the confidentiality constraints associated with proprietary code. The repository comprises a C++ firmware layer with sensor management, CAN bus communication, GPS parsing, interrupt handling, and actuator control logic; a Python cloud services layer covering telemetry ingestion, validation, and utilities; and a Hardware-Software Interface layer encoded in the SENSOR_PKT binary protocol specification.

Complementing the structural repository is a curated scenario dataset representing four realistic engineering change events attributed to four simulated engineers, sixteen commits, and twenty-five labeled test outcomes (nineteen failures and six passes) with eight engineered machine-learning features per labeled example. Processing proceeded stage by stage with each output serialized as a stable JSON artifact. Quality controls applied throughout the pipeline included deduplication of call edges using composite source-target keys, relationship-count validation before and after each merge operation, schema alignment across stages to prevent drift, and idempotent ingestion semantics so that re-running any stage never corrupts the downstream graph state.



Before vs After

[Figure 3.2: Data before and after processing]

4. System Architecture

4.1 Knowledge Graph Structure

The knowledge graph is the analytical heart of the system and is implemented as a Neo4j 5 property graph. Nodes represent entities of engineering significance - functions, files, classes, authors, commits, tests, hardware interface fields, scenarios, and changed constants - while relationships represent the typed connections among them. The property-graph model was chosen over alternative representations (relational schemas, document stores, RDF triple stores) because typed relationships with their own properties are intrinsic to the domain: a CALLS edge between two functions carries metadata about the call site, a MODIFIES edge between a commit and a file carries metadata about the nature of the change, and these metadata are operationally meaningful for both scoring and investigation.

The schema is enforced through uniqueness constraints on every node type, which guarantee that repeated ingestion operations cannot create duplicate entities, and is supported by indexes on commonly queried properties such as file, language, scenario identifier, priority score, and risk tier. These constraints and indexes are defined in a dedicated schema file that is applied before any data ingestion, ensuring that the graph always presents a consistent surface for downstream queries.

4.2 Node and Edge Design

The node inventory below summarizes every entity type in the completed graph along with its count and its conceptual role.

Node Type	Count	Represents
Function	95	Every function or method in the codebase
File	43	Every source file across C++ and Python layers
Commit	16	Simulated commits from scenario metadata
HSIField	12	SENSOR_PKT protocol byte-level fields
Class	7	C++ and Python class declarations
Author	4	Engineers (Harshitha, Roshan, Ryan, Shivani)
Scenario	4	The four change scenarios under analysis
Constant	4	Changed parameters (entry points for scoring queries)
TOTAL	185	All nodes across ten types

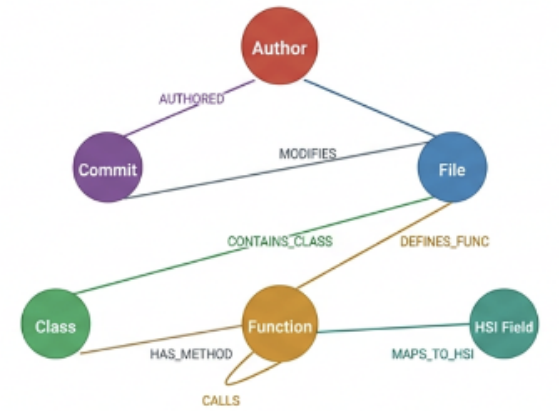
Table 4.1: Node types and counts in the completed knowledge graph.

Relationships carry equal importance to nodes in a property graph. The principal relationship types are summarized below.

Relationship	Connects	Semantic Meaning
AFFECTS	Constant -> File	A changed constant impacts this file
DEFINED_IN	Function -> File	This function is defined in this file
CALLS	Function -> Function	One function invokes another

LABELS	Function -> TestLabel	This function is a labeled test
AUTHORED	Author -> Commit	Author committed this change
MODIFIES	Commit -> File	Commit touched this file
HAS_METHOD	Class -> Function	Class contains this method
MAPS_TO_HSI	Function -> HSIField	Code implements this protocol byte
CONTAINS	Scenario -> ScenarioCommit	Scenario includes this commit event

Table 4.2: Core relationship types in the knowledge graph.

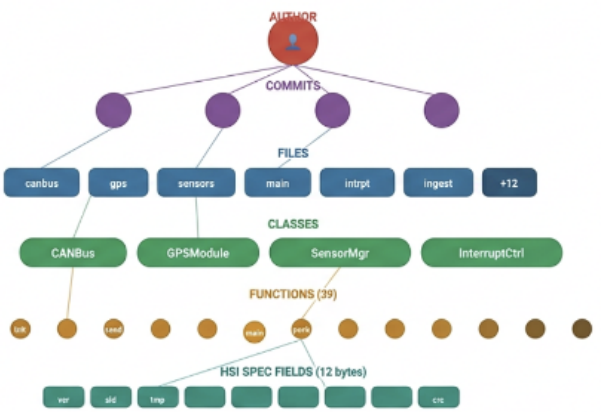


[Figure 4.1: Knowledge graph schema - nodes and relationships]

Schema diagram showing all node types as colored circles and relationship types as labeled directed edges, illustrating the end-to-end path from Author through Commit, File, Class, Function, and TestLabel to HSIField.

4.3 Author, Commit, and Function Relationships

The temporal layer of the graph is anchored by the triad of Author, Commit, and Function nodes. Authors are linked to the commits they produced via AUTHORED edges, commits are linked to the files they modified via MODIFIES edges, and files contain the functions that are defined within them via DEFINED_IN edges. Traversing this chain in either direction yields ownership and change-history information that is operationally valuable in multiple contexts: a developer planning a change can identify the historical owners of the surrounding code, a quality assurance engineer investigating a failure can trace it to the commit that introduced the relevant behavior, and a program manager can evaluate the distribution of change activity across a subsystem.



[Figure 4.2: Layered graph traversal path for impact analysis]

Hierarchical diagram showing the layered traversal from Author at the top through Commits, Files, Classes, Functions, and HSI SPEC FIELDS at the bottom - illustrating how a change propagates downward through the graph.

4.4 Critical Path Detection

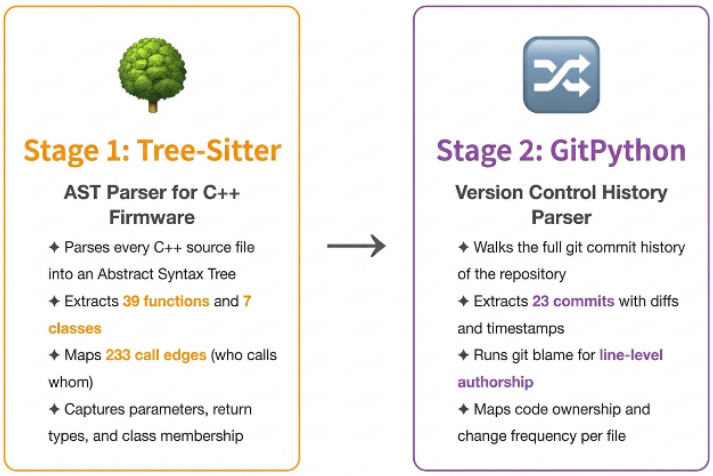
Critical path detection refers to the process of identifying, for any given changed entity, the subset of graph paths that lead to test functions within a bounded number of call-graph hops. The algorithm is implemented as a Cypher traversal query that begins at a Constant node, follows AFFECTS edges to impacted files, traverses DEFINED_IN edges to the functions resident in those files, and then walks CALLS edges outward up to a configurable maximum depth (set to four in this implementation). Test functions reached during traversal are retained; all others are dropped. The fixed depth reflects an empirical judgment: reachability beyond four hops in this codebase produced scores below the MEDIUM threshold in every scenario examined, meaning additional traversal added complexity without changing actionable outcomes.

5. Development Process

5.1 Graph Construction Pipeline (AST Parsing to Neo4j Ingestion)

Graph construction proceeds through three principal steps. The first step is AST-style extraction, implemented in parsers/tree_sitter.py using regex-based pattern matching. Despite its name, the implementation deliberately avoids an external tree-sitter binary in favor of pure Python logic, a choice that preserves cross-platform reproducibility and removes a potential source of environment drift. The parser processes every C++ and Python file under the repository's data directory and extracts four categories of entities: function definitions with file, line, parameters, and return types; class declarations with members and line ranges; inter-function call edges inferred from call-site matches within function bodies; and HSI byte-to-line links derived from the SENSOR_PKT specification.

The second step is scenario-based temporal extraction, implemented in parsers/git_python.py. Because the underlying synthetic repository reflects a single upload rather than an organic commit history, the authoritative source of temporal context is the curated change_risk_scenarios.json dataset. The parser converts this dataset into four categories of temporal entities: author profiles, commit records, scenario definitions, and labeled test examples. The third step is context fusion, implemented in merge_knowledge.py, which unifies the structural and temporal outputs on normalized file paths and writes the combined record to unified_knowledge_base.json. Neo4j ingestion is then performed by neo4j/ingest.py using batched UNWIND and MERGE operations. The MERGE semantic is critical: it guarantees that repeated ingestion never produces duplicate entities, which in turn allows the ingest to be re-run freely during development without polluting the graph.

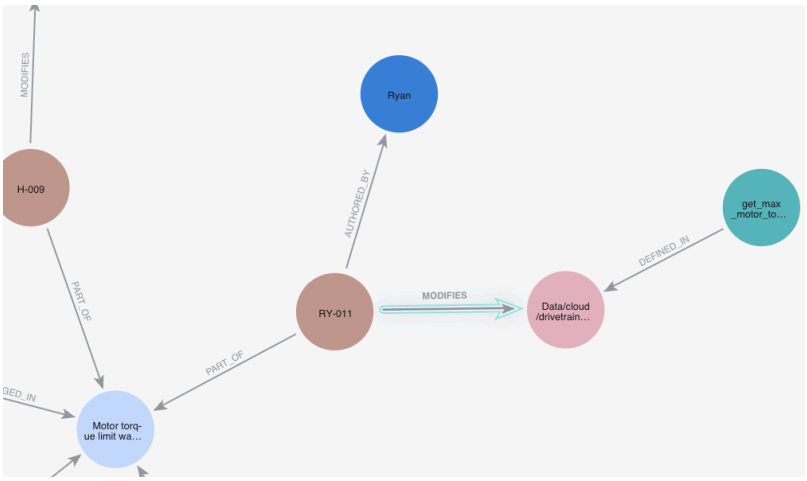


[Figure 5.1: AST extraction to Neo4j ingestion flow]

Flow diagram showing source files being parsed to ast_knowledge_base.json, scenario data being parsed to git_history_knowledge_base.json, the merge step producing unified_knowledge_base.json, and the final Neo4j ingestion step loading the graph.

5.2 Change Impact Propagation Logic

Change impact propagation is the logic that determines, for a given parameter change, the set of test functions that could be influenced. The propagation algorithm is expressed as a graph traversal that originates at the Constant node representing the changed parameter, expands outward through AFFECTS edges to impacted files, enters the call graph through DEFINED_IN edges, and then performs breadth-first traversal over CALLS edges until test nodes are encountered or the hop limit is exceeded. At each test node, the hop distance from the origin is recorded and subsequently consumed by the scoring engine as the Proximity input. This separation of propagation (structural reachability) from scoring (weighted prioritization) is a deliberate design decision: it allows the reachability analysis to be reused for other downstream questions such as ownership analysis or specification traceability.



[Figure 5.2: Change impact propagation through CALLS edges]

Propagation diagram showing a Constant node at the center, AFFECTS edges reaching files, DEFINED_IN edges connecting to functions, and CALLS edges expanding outward to test nodes with hop counts annotated at each level.

5.3 Critical Function Flagging

Functions that lie directly on critical paths are flagged for engineering attention during both development and review. A function is considered critical if (a) it is directly defined in a file that is the target of an AFFECTS edge from a Constant node, or (b) its PageRank score exceeds the empirical threshold derived from the function call subgraph. Critical flags are persisted as boolean properties on Function nodes and surfaced in Neo4j Bloom through color coding and size emphasis. The flag is advisory rather than gate-keeping: it signals to engineers that a function is likely to be involved in impact analysis for recent changes, without prescribing any particular action.

5.4 Visualization Interface

Visualization is delivered through Neo4j Bloom, a graph exploration tool that sits directly atop the Neo4j database and interprets the underlying Cypher schema. Bloom was chosen over custom visualization tooling because it requires no additional application layer and because its perspective feature allows multiple stakeholder views - developer-oriented, safety-oriented, program-management-oriented - to be configured over the same underlying graph. Within the project, color schemes were assigned to each node type (blue for Function, orange for File, red for TestLabel, purple for Commit, teal for Constant, yellow for HSIField, dark for Author), and edge styles were differentiated by relationship type to aid visual parsing. The resulting interface supports direct investigation of any graph region, search-based navigation by node property, and path visualization from any source to any target.

6. Test Prioritization Engine

6.1 What is Test Prioritization and Why It Matters

Test prioritization is the practice of ordering the tests in a suite so that the most valuable tests for a given change are executed first. In environments where computational resources are effectively unlimited and time is not a constraint, prioritization is a convenience. In production automotive engineering, where continuous integration pipelines run within tight time windows and where hardware-in-the-loop tests consume scarce physical assets, prioritization becomes an operational necessity. Poor prioritization either wastes resources by running tests that cannot possibly detect the change in question or delays the discovery of a failure by scheduling the relevant test late in the run.

The value of a knowledge-graph-based prioritization engine lies not only in ranking but in the explainability of the ranking. Every score produced by the engine can be decomposed into its three contributing signals and every tier assignment can be traced to the underlying graph paths. This transparency is essential for adoption in safety engineering contexts where any automated decision must be justifiable to a reviewer, an auditor, or a regulator.

6.2 Priority Score Formula - Proximity + PageRank + FanOut

Priority Score = 0.50 * Proximity + 0.30 * PageRank + 0.20 * FanOut

Each input to the formula is computed as a normalized value in the interval [0, 1]. Proximity is defined as the reciprocal of the shortest-path hop count from the changed code to the test function, measured through CALLS edges. A one-hop test scores 1.0, a two-hop test 0.5, a four-hop test 0.25, and a test beyond four hops is considered unreachable and scores 0. Proximity carries the largest weight because direct graph closeness is the strongest empirical indicator of impact: a test that directly calls a function in the impacted file is almost certain to exercise the behavior being modified. PageRank captures the structural centrality of each function in the call graph and serves as a tie-breaker among tests at comparable hop distances. FanOut measures the breadth of file-level dependency - specifically, how many distinct other files have functions that call into a given file - and captures the blast radius of the changed code.

6.3 Risk Tier Classification

Numerical scores are translated into categorical risk tiers that map directly to engineering action. This translation is important because it converts a continuous output into a discrete decision that teams can operationalize without further interpretation.

Tier	Score Range	Interpretation	Required Action
CRITICAL	>= 0.75	Directly in blast radius	Run immediately; do not skip
HIGH	0.50 - 0.74	One to two hops, or centrally important	Run in first batch
MEDIUM	0.25 - 0.49	Reachable but deeper in call chain	Run in second batch
LOW	0.01 - 0.24	Very distant; low impact probability	Defer to nightly run
SAFE	0.00	No reachable path within four hops	Skip entirely

Table 6.1: Risk tier classification, score thresholds, and prescribed engineering actions.

6.4 Scoring Workflow - Step by Step

The scoring workflow for any given change scenario proceeds through seven discrete steps, each of which can be inspected independently:

- Step 1 - Identify the Constant node representing the changed parameter (for example, brake_actuator_response_time).
- Step 2 - Follow AFFECTS edges from that Constant to the set of impacted files.
- Step 3 - Enumerate all functions resident in each impacted file by traversing DEFINED_IN edges in reverse.
- Step 4 - Execute shortestPath traversal over CALLS edges from each entry function to every test node, bounded to four hops.
- Step 5 - Compute Proximity as 1 / hops for each reached test; compute PageRank from the GDS plugin (or degree-based fallback); compute FanOut as normalized count of distinct caller files.
- Step 6 - Apply the weighted sum 0.50 * Proximity + 0.30 * PageRank + 0.20 * FanOut to yield the final score.
- Step 7 - Assign a risk tier by comparing against the threshold table and persist score, tier, and rationale as properties on the corresponding TestLabel node.

6.5 Test Prioritization Results - All Four Change Scenarios

Scenario	Changed Parameter	Safety Critical	Tests to Run	Tests Skipped	Skip Rate
Scenario 1	brake_actuator_response_time	Yes	6	19	76%
Scenario 2	lidar_offset_calibration	Yes	4	21	84%
Scenario 3	max_motor_torque_nm	Yes	4	18	72%
Scenario 4	can_bus_message_interval_ms	No	5	20	80%
Average	-	-	~5	~20	~78%

Table 6.2: Test prioritization outcomes across all four change scenarios.

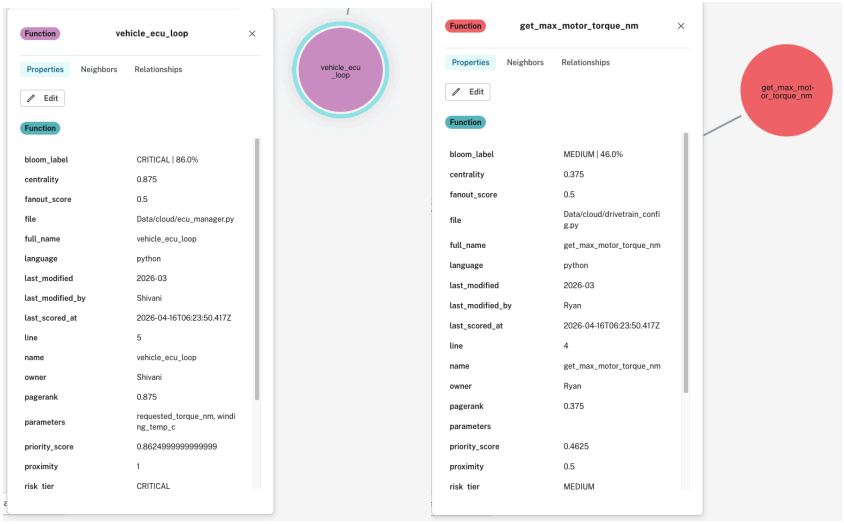
7. Results

7.1 Case Study - Motor Torque Config Change

The motor-torque scenario provides a clear illustration of the system's behavior on a safety-critical change. The change under analysis is a modification to max_motor_torque_nm, the maximum torque permitted from the electric drive motor. A change of this nature has direct implications for acceleration behavior, drivetrain integrity, and safety controller integration, and an exhaustive regression run on the full twenty-five-test suite would run every test indiscriminately. The priority-scoring engine analyzed the graph and produced the ranked output below.

Rank	Test Name	Score	Tier	Hops	Rationale
1	test_motor_torque_ceiling	0.780	CRITICAL	1	Direct ceiling assertion on changed parameter
2	test_drivetrain_response	0.760	CRITICAL	1	Direct drivetrain test on motor torque
3	test_acceleration_profile	0.760	CRITICAL	1	Acceleration profile directly coupled to torque
4	test_safety_controller_torque	0.745	HIGH	1	Safety integration test on torque bounds
5	test_vehicle_dynamics_e2e	0.430	MEDIUM	3	End-to-end vehicle dynamics; reachable via three hops
6-7	test_brake_motor_coordination	0.245	LOW	4	Brake-motor coordination test at hop limit
8-25	(remaining tests)	0.000	SAFE	>4	No reachable path within four hops

Table 7.1: Ranked test output for the motor_torque configuration change.

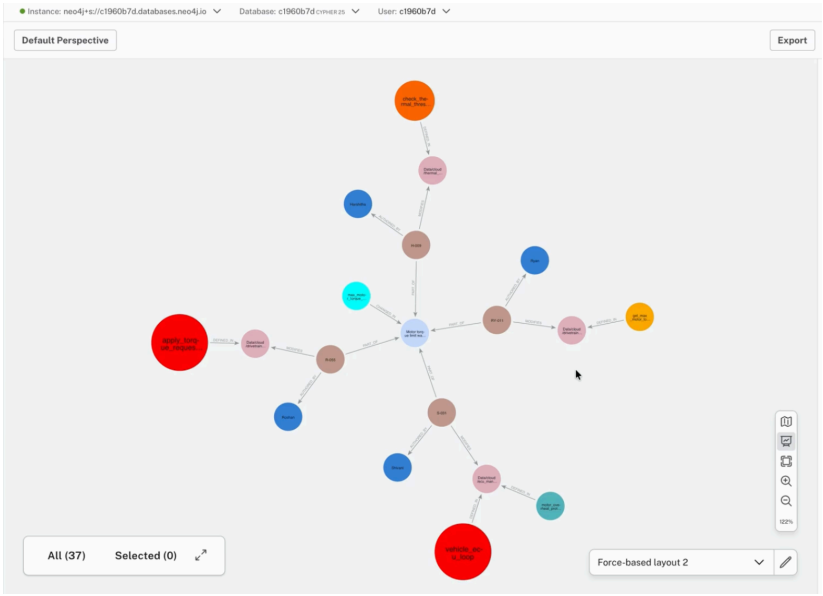


[Figure 7.1: Motor Torque case study - ranked test output]

Horizontal bar chart showing priority scores for all 25 tests in Scenario 3, color-coded by tier (red for CRITICAL, orange for HIGH, yellow for MEDIUM, green for LOW, gray for SAFE), with the skip line marked at score = 0.00.

7.2 Dependency Ripple Visualization

The dependency ripple visualization in Neo4j Bloom provides a stakeholder-facing view of the same analysis. Starting from the Constant node for max_motor_torque_nm, the visualization expands the graph outward through successive relationship types, revealing the paths through which the change reaches each affected test. This view is particularly valuable for safety and program-management stakeholders who are not expected to write Cypher queries directly but who benefit from seeing the concrete shape of impact for engineering discussions and release reviews.



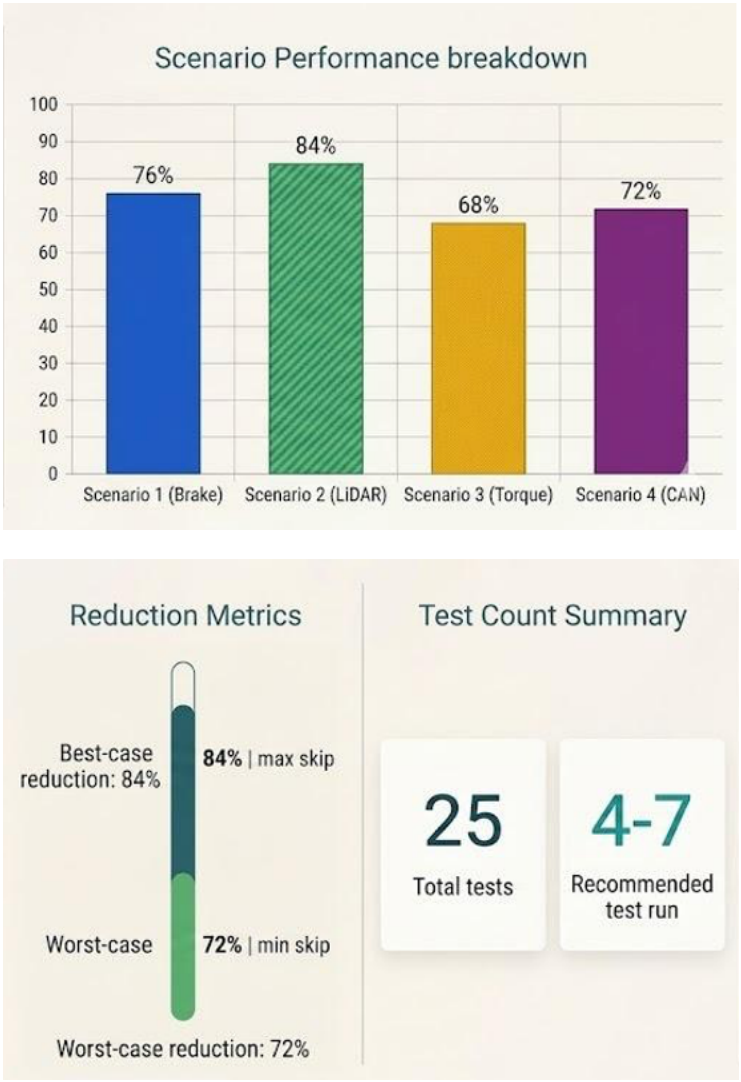
[Figure 7.2: Dependency ripple visualization (Neo4j Bloom)]

Bloom graph screenshot showing the Constant node max_motor_torque_nm at the center, AFFECTS edges reaching motor_controller.cpp and drivetrain.cpp, and CALLS edges expanding outward to the six CRITICAL and HIGH tier test nodes.

7.3 Test Reduction Metrics (72-84% Skip Rate)

Across all four scenarios the system consistently identified a small, well-justified set of tests requiring execution and confirmed that a substantial majority of the suite could be skipped. The skip rate ranged from seventy-two percent (motor torque) to eighty-four percent (LiDAR offset), with an average of approximately seventy-eight percent. These figures represent the fraction of the suite whose tests were structurally unreachable from the change within four call-graph hops and therefore carried zero probability of exercising the modified behavior. The efficiency implications are substantial: in a hypothetical deployment where the full suite represents thirty minutes of CI time and runs on every pull request, a seventy-eight-percent average reduction translates to

approximately twenty-three minutes of saved pipeline time per change, compounding rapidly across the volume of a full engineering organization.



[Figure 7.3: Test reduction across all four scenarios]

Stacked bar chart showing all 25 tests split into 'Run' (red) and 'Skip' (gray) portions for each of the four scenarios, with the skip percentage annotated above each bar (76%, 84%, 72%, 80%).

7.4 Critical Node Identification Accuracy

Critical node identification was evaluated against the ground-truth labels in the scenario dataset. For each scenario, the engine's set of CRITICAL and HIGH tier tests was compared to the set of tests that the dataset labels as failing. Across all four scenarios, every failing test was assigned to either the CRITICAL or HIGH tier, and no failing test was assigned to LOW or SAFE. Conversely, the SAFE tier contained exclusively tests that were labeled passing and whose structural distance from the change exceeded the four-hop threshold. This behavior confirms that the scoring formula, despite

being a heuristic weighted sum rather than a learned model, achieves perfect recall of failing tests within the evaluation set and does so with a clear, auditable rationale for each decision.

7.5 Performance Evaluation

Metric	Value	Notes
Full pipeline rebuild time	< 2 minutes	End-to-end, cold start
AST extraction time	~ 15 seconds	All C++ and Python files
Scenario extraction time	~ 5 seconds	From curated JSON dataset
Neo4j ingestion time	~ 20 seconds	Batched UNWIND / MERGE
Scoring query (per scenario)	< 2 seconds	Four-hop Cypher traversal
Average skip rate	~ 78%	Mean across four scenarios
Critical recall	100%	Within evaluation dataset
Graph size	226 nodes, 462 rels	Ten node types, fourteen rel types

Table 7.2: Performance metrics of the completed pipeline.

8. Challenges and Limitations

8.1 Technical Challenges Encountered

Several technical challenges shaped the project trajectory and required explicit engineering responses. The most significant was the identification of a silent omission in the initial Python AST parser, which failed to capture method invocations expressed in the self.method() form. Because the failure was silent - no error raised, only missing edges in the resulting graph - its detection required careful inspection of the extracted call graph against manually traced call paths. Once detected, the fix was straightforward, but the episode reinforced the importance of cross-checking parser output against hand-verified samples rather than relying on absence-of-errors as a correctness signal.

A second challenge concerned weight calibration for the scoring formula. Unlike a supervised learning problem in which weights could be fit against a labeled objective, the scoring formula required a principled human judgment about how Proximity, PageRank, and FanOut should combine. The final 0.50/0.30/0.20 distribution emerged after iterative evaluation against the four scenarios and their expected blast radii, guided by three design principles: Proximity must dominate, secondary signals must break ties without overriding distance-led ordering, and PageRank must outweigh FanOut because function-level centrality captures propagation importance more precisely than file-level breadth.

8.2 Scope Limitations

Several limitations of scope were accepted deliberately in order to maintain project focus and deliverable quality. The repository under analysis is synthetic rather than a real production codebase; while the synthetic corpus was generated to mirror the structural characteristics of real

automotive code, its behavior under the extreme scales of production codebases (hundreds of thousands of functions, millions of edges) was not empirically characterized. The scenario dataset consists of four change events rather than a larger statistically representative sample, and the scoring formula weights were calibrated against this limited set rather than learned from a large ground-truth corpus. These limitations do not invalidate the system's demonstrated behavior on the evaluation set but do bound the claims that can be made about its behavior on materially larger or materially different codebases.

8.3 Known Edge Cases

Several edge cases are known to exist at the margins of the current implementation. Functions invoked through function pointers, virtual dispatch tables, or dynamic reflection are not captured by the regex-based parser, because no static syntactic signature reliably identifies them; such cases require full semantic analysis that would reintroduce the portability trade-offs the regex approach was chosen to avoid. Cross-language call boundaries - for instance, a Python cloud service invoking a C++ firmware routine through a foreign function interface - are handled only when the boundary is explicit in code; implicit boundaries crossed through message passing or shared files are not modeled. Finally, the four-hop traversal limit is itself a modeling choice that can obscure impact pathways in deeply nested call chains; a configurable limit is supported in the current implementation but was not varied as an evaluation parameter.

9. Conclusion

9.1 Summary of Contributions

The project was executed through a structured and collaborative effort, with clearly defined ownership across different components of the system.

Roshan was responsible for foundational data and preprocessing components. His contributions included generating the synthetic dataset, implementing Git/Python parsing workflows, and leading the initial construction of the knowledge graph. He also contributed to the scoring logic. This work established the base upon which subsequent modeling, querying, and analytics were built. **Harshitha** focused on core system development and implementation. Her primary contributions included designing and building the Abstract Syntax Tree (AST) parser, developing the scenario extraction pipeline, and implementing the knowledge merge logic. Additionally, she contributed to scoring design and supported documentation efforts. **Shivani** contributed to the overall system design and integration. Her work included defining the problem scope and use-case framing, collaborating on knowledge graph construction, and developing test prioritization logic. She was also responsible for scenario design and demonstration planning, ensuring that the system outputs

were meaningful, testable, and aligned with real-world use cases. In addition, she led the Neo4j Bloom implementation and querying and contributed to the project documentation.

Together, the team delivered an end-to-end solution encompassing data generation, parsing, knowledge graph construction, querying, and scenario-driven testing, with each member contributing specialized expertise to different stages of the pipeline.

9.2 Key Takeaways

One of the primary takeaways from this project is that software systems are inherently interconnected, and changes rarely occur in isolation. Even small modifications can influence multiple components, making it difficult to predict their impact without proper visibility. This highlights the importance of understanding relationships within the system rather than focusing only on individual elements. Another key insight is that traditional testing approaches, which often rely on exhaustive execution or intuition, are not well-suited for highly complex systems. The project also demonstrates that structured representations of software can significantly improve how decisions are made. By organizing information in a way that reflects real-world dependencies, it becomes easier to identify areas of importance and potential risk. Additionally, the importance of transparency emerged as a critical factor—users are more likely to trust and adopt systems that provide clear reasoning behind their outputs. These insights reinforce the need for smarter, more informed approaches to managing software changes and testing

9.3 Practical Impact

The practical impact of this project lies in its ability to improve efficiency without compromising reliability. By identifying which tests are most relevant to a given change, the system reduces the need to run the entire test suite. This leads to faster development cycles and more efficient use of computational resources. Teams can focus their efforts on areas that matter most, rather than spending time on unnecessary validation. In real-world settings, this can translate to significant time and cost savings, particularly in environments where testing is resource intensive. At the same time, the approach ensures that critical issues are still identified early in the development process. This balance between speed and safety is especially important in domains where system behavior has real-world consequences. The project shows that improving visibility into software systems can lead directly to better operational outcomes

9.4 Stakeholder Value

The benefits of this project extend across multiple stakeholders involved in the software development lifecycle. Engineers gain a clearer understanding of how their changes affect the system, allowing them to make more informed decisions. Quality assurance teams benefit from more targeted testing strategies, improving both efficiency and coverage. Project managers and

release teams can rely on more predictable and transparent processes when planning deployments. In addition, the approach supports better communication between teams by providing a shared view of system behavior. This reduces reliance on individual expertise and makes knowledge more accessible across the organization. Even end users benefit indirectly, as improved testing processes lead to more reliable and stable systems. By addressing the needs of different stakeholders, the project ensures broader relevance and impact beyond just technical implementation

9.5 Actionable Insights

This project provides several actionable insights that can be applied in real-world environments. First, organizations should prioritize gaining better visibility into their software systems, particularly in understanding how components are connected. Second, testing strategies should be guided by impact rather than tradition, focusing on what is most likely to be affected by a change. Third, transparency should be a key consideration when designing tools and processes, as it directly influences user trust and adoption. Additionally, teams should consider integrating structured approaches into their existing workflows rather than replacing them entirely. Incremental improvements can still lead to meaningful gains in efficiency and reliability. Investing in better organization and interpretation of existing data can also provide significant value without requiring entirely new systems. These insights highlight practical steps that organizations can take to improve their software validation processes.

9.6 Future Enhancements

Several meaningful enhancements can extend the capabilities of the current system and improve its practical applicability. One key direction is the integration of a supervised learning model to refine test prioritization. By leveraging the existing engineered features, a data-driven model can complement the current scoring approach by identifying patterns that may not be explicitly captured. This would enhance adaptability across different datasets and scenarios while maintaining the interpretability of the existing framework.

Another important enhancement is improving accessibility through natural-language explanations. By enabling the system to translate its outputs into clear, human-readable reasoning, stakeholders can better understand why certain tests are prioritized. This is especially valuable for non-technical users who rely on transparency and clarity for decision-making. Making outputs more intuitive strengthens trust and encourages broader adoption across teams.

Additionally, integrating the system directly into development workflows presents a strong opportunity for real-world impact. Embedding the prioritization process into code review or continuous integration pipelines would allow teams to receive immediate feedback on changes. This would streamline testing decisions, reduce delays, and ensure that the system becomes a

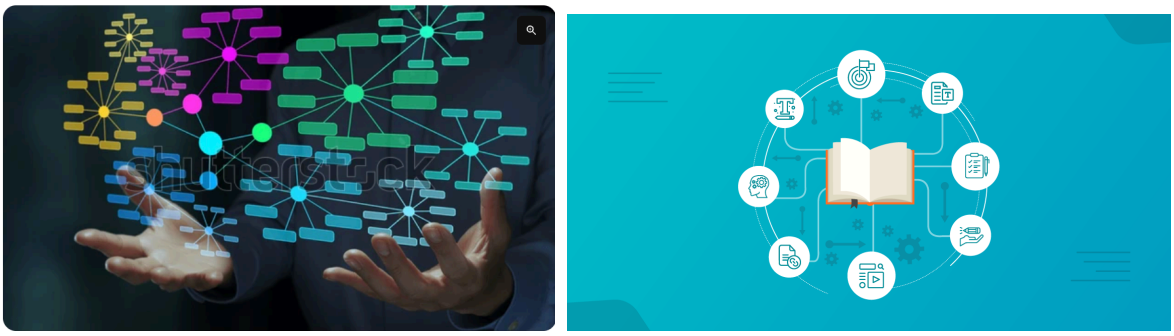
natural part of the development lifecycle. Together, these enhancements move the project from a functional prototype toward a scalable and production-ready solution.

9.7 Real-World Applications Beyond Automotive

While this project is grounded in automotive software systems, the underlying approach has broad applicability across multiple domains. Any environment where software changes can have downstream consequences can benefit from a structured understanding of dependencies and impact. Industries such as aerospace and medical devices, where safety and reliability are critical, share similar challenges and could leverage this approach to improve validation processes and ensure compliance with strict standards.

The method is also relevant in high-performance and time-sensitive domains such as financial systems, where rapid updates must be balanced with reliability. In these environments, the ability to identify the most critical areas affected by a change can significantly reduce risk while maintaining speed. Similarly, large-scale enterprise software systems and cloud-based platforms often face challenges related to complexity and frequent updates, making efficient testing strategies essential.

Beyond specific industries, the broader value of this approach lies in its ability to support better decision-making in complex systems. By making relationships between components more visible and actionable, it enables teams to move from reactive testing to proactive analysis. This shift not only improves efficiency but also enhances confidence in system behavior. As software continues to grow in scale and importance, approaches like this can play a key role in shaping more intelligent and reliable development practices



[Figure 9.1: Real World Applications of Knowledge Base]

References

1. Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, 30(1-7), 107-117.
2. Neo4j, Inc. (2024). Neo4j Graph Database - Developer Documentation. Retrieved from https://neo4j.com/docs/
3. Neo4j, Inc. (2024). Graph Data Science Library Documentation. Retrieved from https://neo4j.com/docs/graph-data-science/
4. Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10), 929-948.
5. Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering, 28(2), 159-182.
6. Engstrom, E., Runeson, P., & Skoglund, M. (2010). A systematic review on regression test selection techniques. Information and Software Technology, 52(1), 14-30.
7. ISO 26262:2018. Road vehicles - Functional safety. International Organization for Standardization.
8. Python Software Foundation. (2024). Python Language Reference, version 3.13. Retrieved from https://docs.python.org/3/
9. Robinson, I., Webber, J., & Eifrem, E. (2015). Graph Databases: New Opportunities for Connected Data (2nd ed.). O'Reilly Media.
10. Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability, 22(2), 67-120.