Detecting Diagnostic Trouble Code Requirement–Implementation Inconsistencies Using LLMs

An Experience Report

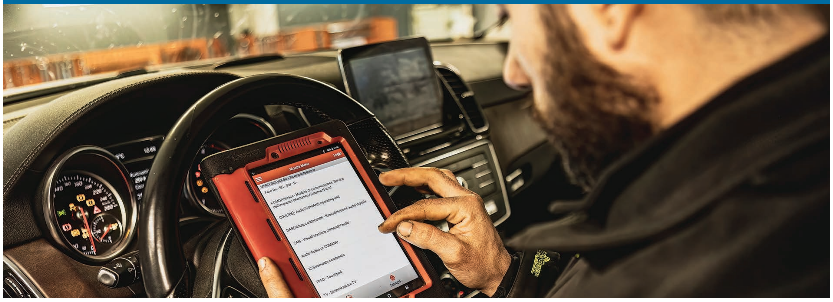
Tongwei Zhang and **Kundi Yao**, University of Waterloo

Hanyang Hu, Wind River Systems

Pengyu Nie, University of Waterloo

Krishna Koravadi, Aptiv

Weiyi Shang, University of Waterloo



©SHUTTERSTOCK.COM/PPORIPHOTOS

// *Ensuring alignment between requirements and implementation is a critical challenge in safety-critical automotive systems. In this industrial study of 89 diagnostic trouble code methods, detected inconsistencies were confirmed by domain experts and challenges and lessons learned were documented for future practitioners.* //

AS THE AUTOMOTIVE industry advances toward increasingly software-defined systems, requirements are becoming more intricate and multilayered. This complexity creates coordination challenges among stakeholders with diverse backgrounds. Throughout the lifecycle, stakeholders often use varied terminologies to express similar requirements, leading to inconsistencies. These issues are especially evident when requirements are translated into code, increasing the risk of misalignment and inadequate test coverage.

A prominent example arises with diagnostic trouble codes (DTCs), which are generated by onboard diagnostics to signal malfunctions in vehicle systems. Accurate DTC behavior is critical for vehicle maintenance. Typically, DTC requirements are described in semistructured natural language and implemented in procedural C code. This implementation process often leads to inconsistencies since behavior is scattered across files, enriched with defensive checks,

Digital Object Identifier 10.1109/MS.2025.3631386
Date of current version: 26 February 2026

0740-7459 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies.

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

and expressed using abbreviations that may drift from glossary terms. Addressing these inconsistencies is essential for system robustness.

Large language models (LLMs) offer opportunities to automate inconsistency detection. However, direct requirement-to-code comparison is unreliable due to missing domain knowledge, terminology gaps, and an abstraction gap: requirements are high-level, while code is low-level and optimized for execution. In practice, comparison can proceed either bottom-up by abstracting code into requirement-like form, or top-down by generating code from requirements. Because many implementations can satisfy the same requirement, we adopt the bottom-up approach, which requires only simplifying code to a comparable abstraction.

This article reports our experience building an automated pipeline for detecting inconsistencies in DTC implementations. We transform source code into semistructured requirements (Code → Req), enrich both sides with domain knowledge, and apply an LLM to compare the resulting requirements with documented ones. Applied to 89 DTC methods, the pipeline achieved 100% precision for severe mismatches and over 75% precision for moderate cases, enabling engineers to triage low-scoring pairs and locate inconsistencies efficiently. We also document encountered challenges and lessons learned to support future adoption of LLMs in requirements quality assurance. Because no public gold-standard labels exist for this industrial setting, we report precision with *expert validation as the ground truth*, using similarity scores solely for triage rather than automatic acceptance or rejection.

Background

System requirements for modern automotive systems define functional and nonfunctional behaviors of hardware and software components. DTC requirements govern the generation, encoding, and lifecycle management of DTCs within standards, such as OBD-II, UDS, and ISO 14229. DTCs detect, log, and communicate faults [e.g., U0121 indicates a loss of communication with the anti-locking braking system (ABS) module] and trigger the malfunction indicator lamp. Accuracy and completeness are vital, as technicians and drivers rely on them for safety and maintenance. Traceability to tests and safety goals is essential for ISO 26262 compliance.

Recent research explores natural language processing (NLP) and LLM techniques to bridge the gap between natural-language requirements and code.[1] Graph-based representations and ontologies capture domain semantics but often require manual curation.[2] Our work builds on these by combining automated graph extraction with LLM-powered JavaScript Object Notation (JSON) transformation for fine-grained inconsistency detection in an industrial setting.

While prior approaches advanced the state of the art, they typically used small-scale data sets and required extensive manual effort. Our pipeline automates extraction of requirements and code, enriches artifacts with expert-provided definitions, and leverages LLM-powered transformation for field-level inconsistency detection.

Challenges

Traditional approaches often rely on rule-based or keyword-based traceability, comparing requirements and code directly using shared identifiers or manually crafted links.[3],[4]. More recent methods use NLP or retrieval-augmented generation (RAG) to retrieve relevant requirements or code fragments as grounded context for LLMs, improving reasoning, reducing hallucinations, and enhancing accuracy.[5],[6],[7],[8] Prior work shows that grounding LLMs in structured knowledge enhances retrieval precision and reduces hallucination in safety-critical domains.[9]

Approach Overview

Our method extracts requirements and implementation artifacts from industrial systems, enriches them with domain knowledge, and models their relationships in a property graph. We apply static code analysis and LLM-powered transformation to convert source code into requirement-like JSON, enabling field-level comparison with documented requirements. This surfaces semantic inconsistencies that traditional traceability or rule-based matching cannot detect alone.

Applying this pipeline to real-world automotive projects revealed challenges in data preparation, mapping accuracy, and comparison reliability. We detail these challenges and solutions next.

Challenge 1: Implicit Domain Knowledge in Code and Requirements

Inconsistencies are difficult to detect due to implicit domain knowledge in both code and requirements. Source code often uses acronyms or abbreviations for brevity and maintainability, introducing ambiguity when establishing traceability since meanings are rarely defined in code. Requirements may also reference external items not identifiable in the codebase. For example, a requirement may specify `WID23 == TRUE`, where `WID23` refers to an external work item.

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

Large systems often maintain a glossary to capture domain terminology. In practice, these glossaries are frequently incomplete, evolve more slowly than the system, or are stored in formats that are not suitable for automated use. Our setting reflected this common situation, with scattered documents and metadata further complicating automated processing.

Solution. We built an expansion table from Polarion metadata, curated by domain experts. During preprocessing, tokens (e.g., `WID23`) are replaced with full definitions (e.g., "Engine temperature threshold exceeded"), and code identifiers like `ITEM_12_CHK` are mapped to requirement terms, such as `NotDisabledByItm12`. This ensures the LLM sees fully qualified terminology, improving interpretation.

Challenge 2: Ambiguity in Code-to-Requirement Alignment

Mapping source code to requirements is nontrivial due to structural and semantic gaps. Deeply nested calls obscure context, while requirements are structured with attributes and conditions. Logic in code is scattered across locations, often without explicit linkage. Inconsistent identifiers add complexity—for example, a requirement might use `DTC$672001`, while code uses `Rtn_Event_DTC_0x672001_SetStatus()`.

Solution. We normalize identifiers with regex-based rules and the mapping table from Challenge 1. For example:

- Rtn_Event_DTC_0x672001_SetStatus() → DTC_672001
- ITEM_12_CHK → NotDisabledByItm12

Normalized tokens serve as reliable anchors in the graph for linking code fragments to requirements.

Challenge 3: Complexity and Interdependencies in Real-World Requirements

Requirements exist in an interrelated network of specifications, conditions, and artifacts. They span multiple levels (stakeholder, system, software), reference each other through constraints, and are linked to domain definitions, interface specs, design docs, test cases, and verification results.

> Requirements and test suites evolve continuously but often at different paces under different teams, leading to misalignment between what is specified and what is verified.

Solution. We model requirements, code artifacts, and their interrelationships as a property graph in ArangoDB. This supports efficient querying, traversal, and extraction of relevant subgraphs for LLM comparison. Providing a grounded subgraph instead of raw text improves retrieval, reduces hallucinations, and supports better reasoning.

Challenge 4: Misalignment of Evolving Requirements and Test Suites

Requirements and test suites evolve continuously but often at different paces under different teams, leading to misalignment between what is specified and what is verified.

Solution. We annotate each requirement and test node with version metadata (timestamp, author, rationale). In our prototype, only the latest versions are linked to simplify analysis. We note the tradeoff between completeness (tracking full history) and performance (single-version graph). Full version alignment is planned as future work.

Implementation Details

The proposed framework detects misalignments between DTCs in an industry-provided requirements database managed in ArangoDB[a] and their implementation in C-based source code. Because full top-down ontology or knowledge graph construction was infeasible in the constrained industrial environment, we designed enrichment to be lightweight and flexible, incrementally incorporating expert knowledge via prompting on top of existing infrastructure. Figure 1 shows the three-stage pipeline, aligned with challenges C1–C4.

Manage Artifacts

Our industry partner utilizes Siemens Polarion[b] to manage artifacts, such as source code, test cases, stakeholder

[a] https://arangodb.com/
[b] https://polarion.plm.automation.siemens.com/

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

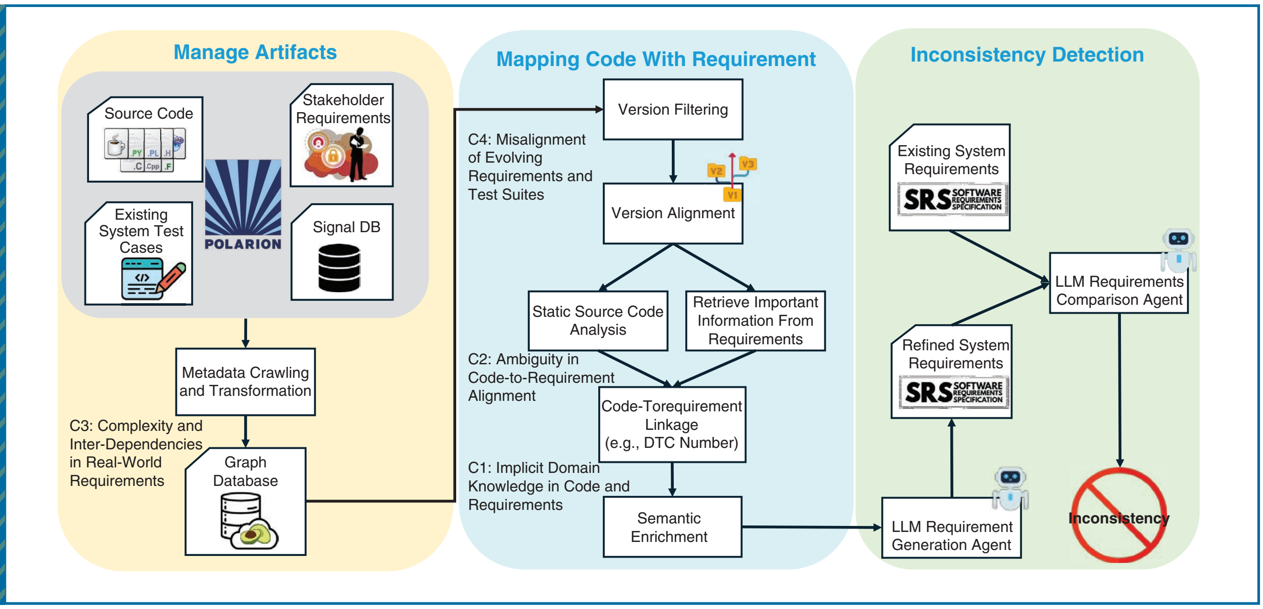

Manage Artifacts
Source Code
PY
PL
H
.C
Cpp
.F
Stakeholder Requirements
POLARION
Existing System Test Cases
Signal DB
Metadata Crawling and Transformation
C3: Complexity and Inter-Dependencies in Real-World Requirements
Graph Database
Mapping Code With Requirement
Version Filtering
C4: Misalignment of Evolving Requirements and Test Suites
V2
V3
V1
Version Alignment
Static Source Code Analysis
Retrieve Important Information From Requirements
C2: Ambiguity in Code-to-Requirement Alignment
Code-Torequirement Linkage (e.g., DTC Number)
C1: Implicit Domain Knowledge in Code and Requirements
Semantic Enrichment
Inconsistency Detection
Existing System Requirements
SRS SOFTWARE REQUIREMENTS SPECIFICATION
LLM Requirements Comparison Agent
Refined System Requirements
SRS SOFTWARE REQUIREMENTS SPECIFICATION
LLM Requirement Generation Agent
Inconsistency

FIGURE 1. An overview of the inconsistency detection pipeline.

requirements, and signal DB elements. Although Polarion models artifacts and their relationships in a graph-like structure, the implementation relies on relational tables. For our study, which required frequent and complex graph traversal queries, this tabular representation was inefficient for query performance and data modeling.

To address these limitations, we developed a crawler that extracts relevant data from Polarion and migrates it to ArangoDB, a native multimodel database with graph capabilities. The crawler converts relational records into graph-compatible JSON documents. Each artifact is modeled as a vertex, while inter-artifact relationships (e.g., a requirement verified by a test) are edges with metadata. During transformation, the crawler normalizes identifiers, flattens nested structures, and maps Polarion's link roles to defined edge types to preserve semantics. Documents are then ingested into ArangoDB via its HTTP application programming interface (API) with updates performed through ArangoDB Query Language.

To ensure consistency between Polarion and ArangoDB, we extended the crawler with incremental synchronization to continuously track and export updated artifact information. In our current implementation, we focus on extracting metadata for requirements and test artifacts along with their interrelationships. The incremental crawler is executed daily, ensuring the graph database remains current with minimal overhead.

Mapping Code With Requirement

We follow a multistep approach to map source code implementations with their corresponding requirements and address the aforementioned challenges.

We first filter source code, requirements, and related artifacts by version tags to ensure temporal consistency across all components. Next, we perform static code analysis using *srcML*,[c] converting C code into XML-based abstract syntax tree (AST) representation. This tool is widely adopted in prior studies[10],[11] for analyzing large codebases. Using the AST, we identify method boundaries and extract diagnostic functions related to DTCs. Concurrently, we parse the requirements from the graph database, focusing on key fields identified by practitioners as high priority for consistency verification. We then establish traceability links between implementations and requirements through multiple anchors: naming patterns (e.g., function names containing DTC identifiers), semantic anchors (DTC numbers), and external references. For example, a diagnostic method named **DTC_X_Sensor_Down_Test** is linked to the requirement for **X_Sensor_Down** in the database through consistent naming conventions. We spotchecked all 89 methods to verify conformance before proceeding to LLMpowered comparison. These anchors serve only to propose candidate links; inconsistencies are ultimately surfaced

[c] https://www.srcml.org

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

by field-level JSON comparison and expert validation, not by anchors alone. With the goal of inconsistency detection, establishing traceability is the preliminary step of our approach.

Finally, as domain-specific terms (e.g., acronyms or indexed terms) exist in both code and requirements, which impede LLM comprehension, we infuse enriched knowledge into both. We perform inline replacement of acronyms in code (e.g., converting `if (!LRCF_MOD_STR)` to `if (!LeftRadarConfigModuleStarted)`) and requirements (expanding pattern-based tokens like `WID{$ID$}` to full definitions). This enrichment ensures the LLM interprets domain-specific terminology that lacks explicit definition in the original artifacts.

Inconsistency Detection

We then detect inconsistencies between source code and requirements with LLMs.

Generate Refined System Requirements From Source Code. We generate refined requirements from enriched source code. Rather than directly comparing source code to requirements, we instruct the LLM to transform code into requirement-like representations. Specifically, we provide the LLM with a crafted prompt containing domain knowledge and context about DTC diagnostics, guiding it to extract key fields, such as method name, DTC number, enable conditions, and actions for mature and demature cases. The LLM output is structured in JSON format to facilitate field-level comparisons. The prompt text is as follows.

Detect Inconsistencies Between Refined and Existing Requirements. After generating refined requirements from code and extracting key fields from existing requirements, we compare each field pair using the LLM-as-judge paradigm.[12,13] For every pair, the LLM outputs both a numeric similarity score between 0 and 1 and a justification, enabling field-level inconsistency identification. For instance, suppose the existing requirement JSON has

`“Enable Condition”: “EngineTemp > 90”`

while the LLM-generated JSON from code contains

`“Enable Condition”: “EngineTemp > 95”`

The mismatch flags a deviation in conditional logic. Such field-specific deviations map directly to implementation gaps relative to the documented requirement.

For each DTC method, we calculate an aggregated similarity score by taking the mean of similarity scores across all fields:

$$\text{Average Score} = \frac{\sum(\text{Similarity Score of Each Field})}{\text{Nmber of Fields}}$$

The threshold of 0.6 for flagging critical inconsistencies was determined empirically: we plotted the 89 average similarity scores against expert-validated outcomes and selected the elbow point in the precision curve that maximized separation between true mismatches and consistent mappings. Scores between 0.6 and 0.75 are analyzed further for field-specific discrepancies, while scores above 0.75 typically indicate high consistency.

Experiment and Results

Experiment Settings

We evaluate the pipeline's effectiveness in ensuring consistency between source code and database requirements on an industrial data set.

System prompt:
Analyze the attached source code that checks DTC conditions, with each method implementing one requirement.

Extract the following information:

- **Method Name:** Name of the method
- **DTC Number:** Found between `DTC_0x` and the next underscore in function calls
- **Enable Condition:** All conditions in variable `$DTC_Example $` (joined by “Or” relationships)
- **Mature Actions:** Actions performed when both enable and mature conditions are met
- **Demature Actions:** Actions performed when the enable condition is met but mature condition fails

Code structure: Enable conditions are checked first, then mature conditions are checked in a nested if-statement, followed by either mature or demature actions.

Return the information in JSON format with these keys: “Method Name,” “DTC Number,” “Enable Condition,” “Mature Actions,” and “Demature Actions.”

User prompt:
Here is the C code: `$code_str $`. Generate only the JSON output as requested.

Data Set. The data set includes 89 DTC-related methods paired with their corresponding requirements extracted from Polarion.

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

Semantic Analysis and Detection. We used Claude 3.5 Sonnet[d] to transform code into JSON requirements and generate similarity scores with justifications, due to its reasoning capability and contextual understanding.

Validation Process. All inconsistencies flagged by the pipeline were forwarded to domain experts, including senior test engineers and requirements analysts, for confirmation. Their manual inspection of LLM-generated requirements and similarity scores served as the final arbiter of correctness for reported precision. In practice, similarity scores were used only as a triage mechanism to prioritize potential inconsistencies, while expert inspection provided definitive evaluation. As no independent gold-standard data set exists for this task, we report precision at two severity levels (critical versus moderate) rather than recall.

Results

We present similarity scores correlated with actual inconsistencies.

Low Similarity Range (<0.6). All three nodes with scores below 0.6 exhibited clear inconsistencies across multiple fields, demonstrating 100% precision for severe mismatches after expert review. These included mismatched *Enable Conditions* and incorrect or missing *Mature* and *Demature Actions*. Results validate the pipeline's reliability in identifying critical errors, suggesting scores below 0.6 can be treated as a "hard inconsistency threshold."

Moderate Similarity Range (0.6–0.75). This range presented the most complex challenges. Of 31 nodes, 14 (45%) were confirmed inconsistent, yielding precision greater than 75%. These inconsistencies were typically isolated to specific fields, while others remained accurate. For instance, some cases showed discrepancies in the *Enable Condition*, where code used more granular expressions than database requirements; others differed in action sequences due to naming variations. These highlight the need for targeted analysis rather than relying on aggregate scores, as averaging similarity across fields can obscure critical mismatches.

High Similarity Range (>0.75). All 55 nodes with scores above 0.75 were verified consistent with their requirements, with no field-level inconsistencies detected. Although syntactic and structural differences (e.g., naming conventions, abbreviations, or procedural versus declarative styles) reduced some similarity scores, these did not impact semantics. Findings underscore the need for similarity measures that account for stylistic variations.

Types of Inconsistencies Identified

Through expert validation, we classified confirmed inconsistencies into three categories:

- *Enable condition mismatches*: Code checks additional or missing flags compared to requirements.
- *Mature/demature action errors*: missing or extra actions in nested branches.
- *Identifier/terminology drift*: discrepancies in term usage between code and database descriptions.

Overall, our analysis shows the approach effectively identifies requirement–implementation inconsistencies in an industrial setting, achieving high precision on critical and moderate cases while revealing opportunities to refine handling of field-specific mismatches.

Limitation of Prompting

Despite using a constrained JSON-schema prompt and expert-provided domain definitions, our approach cannot ensure LLM outputs are always accurate. Hallucinations can still occur, particularly for complex conditions or unfamiliar terms. Users should treat LLM results as provisional and validate critical fields through manual review or complementary checks.

Lessons Learned

The following lessons are distilled from iterative experiments and ablation studies exploring enrichment, alignment normalization, and LLM comparison strategies. We share the key insights gained from developing the requirement quality assurance pipeline with our industrial partner.

L1: Traceability is crucial and requires industrial specifications. In our initial attempt, we mapped requirements to code without domain knowledge. This failed because source code identifiers and natural language requirements differed too much in style and structure, making direct matching unreliable. Our industrial partner later provided a formal naming convention in which each C function (e.g., Diag_DTC_LostComm_EPS_Test) corresponded to its associated Polarion requirement (e.g., LostComm_EPS). Consistent identifiers across artifacts eliminated ambiguity and provided a solid foundation for inconsistency detection. In environments lacking such specifications, advanced code search and

[d] https://www.anthropic.com/claude/sonnet

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

identifier-mapping techniques are essential before applying LLM-based verification.

L2: Domain knowledge trumps model selection. LLMs cannot be used as a black box for requirements engineering. While we experimented with newer LLM versions, performance gains were minimal compared to improvements from domain knowledge integration. Most publicly available LLMs lack specialized automotive knowledge critical for accurate analysis. The breakthrough came from close collaboration with practitioners to extract domain expertise and systematically inject it through prompting. For example, experts clarified the logic of key components and the meaning of "mature action" and "demature action," which are not obvious without domain guidance. We encoded this understanding into prompts, guiding the LLM to structurally parse code and reverse-engineer requirements consistent with our database. Domain-enriched context proved far more valuable than model upgrades. For future work, RAG could further enhance domain-specific reasoning.

L3: Domain knowledge should guide inconsistency quantification. Similarity scores were effective for our structured DTC requirements, but complex cases demand field-specific strategies. Instead of relying on embeddings, we instructed the LLM to act as a judge: for each requirement pair, it outputs a similarity score and justification, later validated by experts. This follows the LLM-as-judge design,[13] where the model provides judgment plus reasoning. In practice, discrepancies in the *Enable Condition* carry higher risk than mismatches in action sequences, so we weighted fields by domain impact when computing aggregate scores. Quantification should ultimately be guided by expert knowledge to sharpen thresholds. Future work could implement field-specific comparison strategies by incorporating domain expertise about which differences are most significant.

Hybrid tools combine relational databases and graph queries for traceability but still rely on rule-based matching rather than semantic understanding.

L4: Inconsistency detection delivers concrete business value. Our findings were shared with practitioners across development, testing, and management. Both the approach and detected inconsistencies received positive feedback. The partner recognized immediate value in detected cases, which helped identify potential issues before production. The LLM prompting did not replace human inspection but amplified it: after extracting requirement-relevant behavior from execution scripts and comparing it to the (potentially outdated) database, the pipeline ranked candidates so experts could prioritize reviewing the lowest-scoring pairs (e.g., below 0.6 before those between 0.7–0.75). This reduced the discrepancy between the database and code with less effort. The core challenge lay not in traceability but in aligning natural-language requirements with implementation; our three inconsistency types arise independently of traceability quality. The LLM helped surface semantic mismatches difficult to verify automatically. Validation led to plans to integrate our approach into future quality assurance processes and extend it to other product lines. Initial adoption will treat formal traceability as the core, with LLM scoring used only to triage mismatches for expert review. This shows that LLM-based requirements analysis can deliver tangible business value when designed to complement existing processes.

Related Work

Research on requirement–implementation consistency spans traditional rule-based and emerging AI-driven approaches. Spanoudakis and Zisman[3] used clustering and logic-based transformations to detect conflicts in requirements documents, but such techniques struggle with the semantic gap between natural-language specifications and code, especially in embedded systems.

In the automotive domain, Bengtner and Smilevski analyzed real-world DTC occurrences in Volvo Group vehicles.[14] They collected fault data via diagnostic tools and wireless transmission, studied distributions and trigger conditions, and evaluated compliance with ISO/SAE standards. Their work characterizes

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

DTC patterns, while our pipeline leverages LLMs to generate requirement-like representations from C code and perform field-level comparisons against database requirements.

Contract-based oracles are another line of research. Araujo et al.[15] evaluated design-by-contract in large-scale concurrent systems using the Java Modeling Language and runtime assertion checking. Manually authored pre/postconditions detected about 76% of injected faults and reduced diagnosis effort. In contrast, our approach automates requirement extraction and uses LLMs for structured JSON comparison, avoiding manual contract writing.

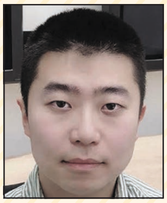
LLMs have recently been applied to requirements engineering. Fantechi et al. studied self-correction to mitigate hallucinations,[9] Pan et al. proposed interactive prompting for summaries,[5] Wen et al. integrated LLMs with static analysis for embedded verification,[6] and Widyasari et al. showed consensus-based ensembles enhance robustness when matching code to specifications.[16]

Beyond AI, graph-based approaches model requirements and their interdependencies.[2,3,4] Hybrid tools combine relational databases and graph queries for traceability but still rely on rule-based matching rather than semantic understanding. Traceability from natural-language requirements to code remains a major and evolving research topic,

ABOUT THE AUTHORS

TONGWEI ZHANG is a direct-entry Ph.D. student at the University of Waterloo, Waterloo, ON N2L 3G1, Canada. Direct-entry Her research interests include software analytics, knowledge modeling, and LLM-based software engineering in safety-critical systems. Contact her at t98zhang@uwaterloo.ca.

KUNDI YAO is a postdoctoral fellow at the University of Waterloo, Waterloo, ON N2L 3G1, Canada. His research interests include software log compression, software log analytics, and mining software repositories Yao received his master's in computer science engineering from Concordia University. Contact him at kundi.yao@uwaterloo.ca.



HANYANG HU is a tech lead at Wind River Systems, Ottawa, ON K2K 2W2, Canada. Contact this author at phenom.hu@windriver.com.

PENGYU NIE is an assistant professor at the University of Waterloo, Waterloo, ON N2L 3G1, Canada. His research interests include machine learning and natural language processing for software engineering. He received his Ph.D. from the University of Texas at Austin. He is a Member of IEEE. Contact him at pynie@uwaterloo.ca.

KRISHNA KORAVADI is a senior manager at Aptiv, Troy, MI 48098 USA. Contact this author at krishna.koravadi@aptiv.com.

WEIYI SHANG is an associate professor at the University of Waterloo, Waterloo, ON N2L 3G1, Canada. His research interests include big data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. Shang received his Ph.D. in computer science from Queens University. He is a Senior Member of IEEE. Contact him at wshang@uwaterloo.ca.

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.

particularly with the rise of LLM-based methods.[17]

Our novelty lies in an industrial experience report with practitioner feedback and expert-validated precision for DTC requirement–implementation consistency. To our knowledge, this is the first study to apply LLM-based semantic comparison specifically to DTC requirement–implementation consistency, with expert-validated precision metrics and lessons from real-world deployment.

This article reports our experiences using LLMs to detect inconsistencies between DTC requirements and their implementations in C source code within an industrial automotive system. Our results demonstrate that LLM-as-judge similarity scores, combined with expert validation, effectively detect real-world inconsistencies, which received positive practitioner feedback. We document our experiences and lessons to assist practitioners seeking to leverage LLMs in requirements quality assurance.

References

1. A. Ferrari and G. Ginde, Eds., *Handbook on Natural Language Processing for Requirements Engineering*, 1st ed. Cham, Switzerland: Springer-Verlag, 2025.
2. J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained BERT models," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 324–335, doi: 10.1109/ICSE43902.2021.00040.
3. G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," in *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, S. K. Chang, Ed., Singapore: World Scientific, 2001, pp. 329–380.
4. G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause, "Rule-based generation of requirements traceability relations," *J. Syst. Softw.*, vol. 72, no. 2, pp. 105–127, 2004.
5. L. Pan, M. Saxon, W. Xu, D. Nathani, X. Wang, and W. Y. Wang, "Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies," *ACM Trans. Mach. Learn.*, vol. 20, no. 3, pp. 1234–1258, 2023.
6. C. Wen et al., "Enchanting program specification synthesis by large language models using static analysis and program verification," in *Proc. Int. Conf. Comput. Aided Verification*, 2024, pp. 302–328.
7. D. Fuchs et al., "LiSSA: Toward generic traceability link recovery through retrieval-augmented generation," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, 2025, pp. 1396–1408.
8. T. Hey, D. Fuchs, J. Keim, and A. Koziolek, "Requirements traceability link recovery via retrieval-augmented generation," in *Proc. Int. Working Conf. Requirements Eng.: Found. Softw. Qual. (REFSQ)*, 2025, pp. 381–397.
9. A. Fantechi, S. Gnesi, L. Passaro, and L. Semini, "Inconsistency detection in natural language requirements using ChatGPT: A preliminary evaluation," in *Proc. 31st Int. Requirements Eng. Conf. (RE)*, Piscataway, NJ, USA: IEEE Press, 2023.
10. Y. Zeng, J. Chen, W. Shang, and T.-H. (Peter) Chen, "Studying the characteristics of logging practices in mobile apps: A case study on F-Droid," *Empirical Softw. Eng.*, vol. 24, no. 6, pp. 3394–3434, 2019, doi: 10.1007/s10664-019-09687-9.
11. Z. Ding, Y. Tang, X. Cheng, H. Li, and W. Shang, "*LoGenText-Plus*: Improving neural machine translation based logging texts generation with syntactic templates," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 1–45, 2024.
12. N. Wu, M. Gong, L. Shou, S. Liang, and D. Jiang, "Large language models are diverse role-players for summarization evaluation," in *Proc. Conf. Natural Lang. Process. Chin. Comput. (NLPCC)*, 2023, pp. 695–707.
13. X. Wang, H. Kim, S. Rahman, K. Mitra, and Z. Miao, "Human-LLM collaborative annotation through effective verification of LLM labels," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2024.
14. J. Bengtner and A. Smilevski, "Diagnostic trouble code analysis: A statistical approach, surrounding fault codes within Volvo group electromobility," Bachelor's thesis, Dept. of Elect. Eng., Chalmers Univ. of Technol., Gothenburg, Sweden, 2022.
15. W. Araujo, L. C. Briand, and Y. Labiche, "On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 971–992, Oct. 2014, doi: 10.1109/TSE.2014.2339829.
16. R. Widyasari, D. Lo, and L. Liao, "Beyond ChatGPT: Enhancing software quality assurance tasks with diverse LLMs and validation techniques," *ACM Trans. Softw. Eng.*, vol. 1, no. 1, pp. 1–21, 2024.
17. J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Recovering trace links between software documentation and code," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2024, pp. 1–13.

Authorized licensed use limited to: University of Waterloo. Downloaded on March 17,2026 at 15:28:07 UTC from IEEE Xplore. Restrictions apply.