ReqClarity AI: A Hybrid Rule-Based NLP and LLM Pipeline for Quality Analysis and Rewriting of Software Requirements Specifications

Urza Rai

Student, Dept. of Computer Science, Vellore Institute of Technology, Vellore, Tamil Nadu, India

Abstract - *ReqClarity AI is a web-based automated system designed to enhance the quality of Software Requirements Specifications (SRS), which frequently suffer from ambiguity, non-verifiability, and incompleteness—issues that can lead to costly downstream errors during development and verification. Traditional SRS quality assurance is labour-intensive, inconsistent across reviewers, and often neglected in smaller teams and academic environments. To address this, ReqClarity AI integrates a deterministic rule-based natural language processing engine with a large language model (LLM) to both detect defects and generate improved, standards-compliant rewrites. The system evaluates each requirement using 15 structured rule patterns organized into three defect categories: ambiguity, non-verifiability, and incompleteness, each comprising five subtypes. Detected defects are weighted by severity, producing a quantitative quality score ranging from 0 to 100 at both the requirement and document levels, and mapped to five quality bands from Critical to Excellent. Requirements falling below a defined threshold are processed through the Groq API using the Llama 3.3 (70B) model, which generates IEEE 830 / ISO/IEC/IEEE 29148-aligned rewrites based on structured prompts. The platform also features an interactive dashboard that allows users to review analyses, accept or reject suggested rewrites, and export annotated reports or finalized SRS documents. Evaluation across 15 SRS documents demonstrated promising performance under controlled evaluation, achieving precision of 0.84, recall of 0.79, and an F1-score of 0.82, while aligning with the intended quality levels of the constructed dataset; additionally, the rewrite pipeline eliminated 88.4% of defects while preserving 93.0% of the original intent.*

Key Words: Software Requirements Specifications, Requirements quality, Ambiguity detection, Non-verifiability, Incompleteness, Natural Language Processing, Large Language Models, IEEE 830, Automated Rewriting, Requirements Engineering

1. INTRODUCTION

1.1 Cost Related to Definition Errors

The cost of fixing defects originating at the requirements stage is significantly higher when addressed later in the software development lifecycle than during early requirement modelling. Studies in software engineering consistently show that correcting defects after deployment can be many times more expensive than resolving them during the requirements phase. Despite this well-established understanding, the quality of Software Requirements Specifications (SRS) remains an under-resourced concern globally, particularly in academic environments and small development teams, where formal quality assurance processes are often limited or absent.

1.2 Nature of Quality Defects Associated with the SRS Document

Three primary categories of quality defects in SRS documents are ambiguity, non-verifiability, and incompleteness. As the SRS acts as a formal agreement between stakeholders, developers, and verification teams, ambiguity poses a critical risk. For instance, statements like "System shall provide a user-friendly interface" lack measurable acceptance criteria, while "System should provide a user-friendly interface" weakens the obligation. Additionally, failing to define system behaviour under unmet conditions creates implementation gaps that may lead to failures. These types of defects are widely studied in requirements engineering literature and have measurable impacts on system quality [1], [6], [9].

1.3 Limitations of Present Approaches

Existing approaches to SRS quality assurance include manual reviews, walkthroughs, and formal inspections such as Fagan inspections [1], as well as checklist-based peer reviews [2]. While effective, these methods are time-consuming and subject to reviewer variability. Automated tools like QuARS [2] and TIGER [3] rely on linguistic pattern analysis but require local installation and offer limited defect coverage, often predating modern generative AI advancements. Other research tools and approaches focus on ambiguity detection using NLP techniques [7], [11], but typically address only a subset of defect types. Currently, no freely available solution integrates detection, scoring, and automated correction within a unified workflow.

1.4 Proposed System and Contributions

This paper introduces ReqClarity AI, a web-based system designed to address these limitations. The platform processes uploaded SRS documents through a five-step pipeline: parsing, defect detection, quality scoring, LLM-based rewriting, and result persistence, all without requiring installation or registration. The system employs a deterministic 15-rule defect detection engine covering ambiguity, non-verifiability, and incompleteness, eliminating the need for training data. Its key contributions include: a rule-based defect detection engine; severity-weighted scoring model generating a [0–100] quality score with five

classification bands, validated on 15 SRS documents; an LLM-based rewriting pipeline utilizing modern transformer-based models [19], [20] to produce standards-compliant corrections while preserving intent; and a fully deployed, freely accessible web application integrating all functionalities into a single workflow. The hybrid architecture—combining deterministic rule-based analysis for explainable defect detection with generative AI for natural language correction—demonstrates a scalable and broadly applicable approach for quality assurance in software engineering domains requiring both transparency and automated improvement.

2. LITERATURE REVIEW

2.1 Theoretical Foundations of Ambiguity Detection

Gervasi and colleagues have established that ambiguity is a multi-level phenomenon consisting of lexical, syntactic, semantic, and pragmatic dimensions, while also differentiating ambiguity from vagueness, abstraction, and absence. Their examples of real-world SRSs provide an indication of some potential patterns of identifiable ambiguity using vague adjectives, modal verbs, passive voice constructions, and omitted details, which informed the categories of detection used by ReqClarity AI. Pragmatic ambiguity has also been discussed by Ferrari et al., in which differences in how stakeholders interpret sentences arise from their context rather than the structure of the sentence itself. They justified the use of layered architectures through the application of rule-based detection and LLM-driven semantic analysis.

In comparing ambiguity detection tools (NASA ARM, QuARS, RETA, and RCM) against 180 industrial requirements, Bajceta et al. provided a quantitative assessment of the implicit trade-off present. That is, rule-based tools provide high recall (up to 0.98) and moderate precision (0.41-0.43) while domain-tuned tools provide higher precision at lower recall. This supports the use of hybrid design approaches, where syntactic rules maximize initial recall and the addition of an AI layer enhances precision via contextual filtering; this architecture is consistent with the current work.

2.2 NLP Evolution in Requirements Engineering Automation

According to Zhao et al.'s systematic mapping study of 404 primary studies published over a period of 36 years, requirements engineering (RE) primarily focuses on the defect identification of requirements quality using syntactic techniques and shallow semantic techniques. The authors catalogued 130 RE-specific NLP tools but identified gaps in evaluating these tools in industry and gaps in evaluating more complex semantic processing. In Necula et al.'s complementary systematic review of 309 papers, the evolution of NLP in RE is described starting with rule-based systems through to deep learning and LLMs and highlights that three challenges persist: the use of domain-specific language, a lack of labelled corpora, and a requirement for explainability.

The overall findings from the two studies suggest the need for hybrid architectures: a rule-based layer of fast, explainable screening for defects and an LLM layer to fill in the semantic gaps and generate contextually correct requirements — which is exactly how ReqClarity AI was designed.

2.3 Transfer Learning and BERT-Based Classification

Hey et al. used a fine-tuned BERT model called NoRBERT to evaluate the PROMISE NFR dataset. They reported F1 scores of up to 94% for projects that had already been seen and between 90% and 93% for those that were previously unseen. Therefore, NoRBERT outperformed the lexical and syntactic baselines by as much as 15 points without the need for any project-specific retraining. This indicates that using transfer learning can mitigate the challenges of data scarcity and cross-project variability in RE. Finally, in a broader look at LLMs, the general findings indicate that smaller model architectures that have been fine-tuned can perform very well with rich RE corpora, while large language models also demonstrate impressive performance on zero- and few-shot semantic tasks such as traceability or specification refinement. Hybrid LLM–rule-based architectures are recommended to combine contextual reasoning with explainability — consistent with ReqClarity AI's design.

2.4 Rule-Based and Hybrid NLP Tools

Femmer et al.'s Smella is a method for detecting code smells in requirements using ISO/IEC/IEEE 29148 criteria, which was implemented with POS tagging and lemmatization. The authors evaluated the system on automotive, chemical, and academic software requirements specification (SRS) documents, finding that it detected an average of 44 findings per 1,000 words at a precision rate of 59% and a recall rate of 82%, supporting the idea that lightweight rule-based methods provide sufficient practical defect coverage without an overhead associated with machine learning (ML). Dalpiaz et al. used REVV-Light to visualize terminological ambiguity within user stories and achieved precision and recall rates comparable to manual inspection. Rempel and Maeder provided a measure for how complete traceability can reduce defects in downstream development across 24 open-source projects. Collectively, these studies support that rule-NLP hybrid methods which achieve a precision/recall rate in the range of 50-80% are practical for real-world SRS quality assurance, particularly when used with validation by practitioners for context-dependent issues.

2.5 Gap in Existing Work

While the literature demonstrates strong individual proposals for defect identification, scoring, and LLM-assisted rewriting, there are currently no publicly available tools that combine all three into a single unified pipeline for

practitioners. Available research prototypes (e.g., QuARS and TIGER) are not accessible to practitioners. Commercial tools come with very high licensing costs. Open-source tools do not provide scoring and/or correction of batches of identified defects from any of the activities. ReqClarity AI addresses this gap by providing a free, fully deployed, end-to-end requirements quality analysis system combining rule-based detection, quantitative scoring, and LLM-assisted rewriting in a single accessible web application.

2.6 Large Language Models in Requirements Engineering

Requirements engineering presents new opportunities with the advent of large language models. Prior work demonstrates that prompt-engineered LLMs can rewrite ambiguous requirements into precise and well-structured specifications with minimal human intervention. Studies further indicate that while such models exhibit strong lexical capabilities in detecting inconsistencies and ambiguities, they often lack deeper semantic completeness, thereby highlighting the necessity of integrating rule-based components to enhance overall performance. Additionally, incorporating explicit IEEE 830 context within prompts significantly improves the generation of standards-compliant requirements. These insights directly inform the design of systems such as ReqClarity AI, which employ structured prompts based on IEEE 830 guidelines and produce outputs in a batched JSON format for efficient processing.

Together, these findings demonstrate that large language models effectively complement rule-based systems by addressing their limitations in generating fluent and contextually appropriate requirement corrections.

3. SYSTEM ARCHITECTURE

3.1 Architectural Overview

ReqClarity AI follows a three-tier client-server architecture - a React-based presentation layer; a Node.js application layer; and a MongoDB Atlas persistence layer, as shown in Figure 1. The frontend and backend communicate via Axios through a RESTful API using HTTPS protocol, where the frontend is hosted on Vercel and the backend is hosted on Render. All results from an analysis are saved to MongoDB Atlas and can be retrieved using the analysis ID, keeping the backend stateless (i.e., no server-side session).

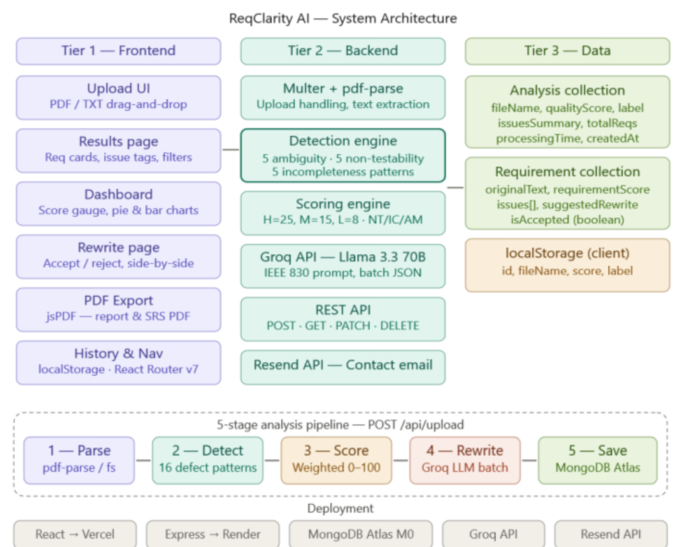

Fig. 1: System Architecture

3.2 Frontend Layer (Tier 1)

The frontend is developed using React 18 and Vite, designed as a single-page application with React Router v7 for client-side routing through 7 views: Home, Upload, Results, Dashboard, Rewrite, History, and Contact. The Dashboard visualizations are produced using Recharts (score gauge, pie chart of issues, bar chart of requirements). The PDF files are exported from the frontend via jsPDF (two document types: an Analysis Report; an SRS Document). Prior analyses are stored in local storage using lightweight metadata (file name, score, label and analysisID) to provide an accessible means of navigating to previous analyses without making additional requests to the server.

3.3 Backend Layer (Tier 2)

Using Node.js and Express, the backend offers six RESTful API endpoints to communicate and perform actions on the data:

- The API endpoint to POST /api/upload is the starting point of the whole process.
- The API endpoint to GET /api/analysis/:id returns an existing analysis result's information.
- By using the DELETE /api/analysis/:id API endpoint, users can remove an analysis. - To accept a rewrite of a requirement, users can use PATCH /api/requirements/:id/accept.
- When a user submits a message via the contact form, this happens over HTTP POST to /api/contact;
- GET /api/health is an API endpoint that returns the health of the entire deployment.

Besides being able to upload files via Multer to a temporary directory (/uploads) that is programmatically created when the application starts, files are deleted immediately after parsing, so no documents are retained on the server.

3.4 Five-Stage Analysis Pipeline

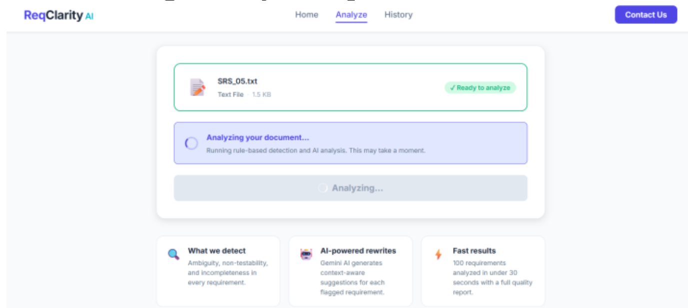

Fig. 2: Document upload and pipeline execution in progress.

Every uploaded document is processed through a sequential five-stage pipeline.

Stage 1 — Parse: Raw text is extracted via pdf-parse (PDF) or Node's fs module (TXT), then segmented into individual requirements using line-by-line splitting with a sentence-level fallback for prose-formatted documents.

Stage 2 — Detect: Each requirement passes through three independent detection modules — ambiguity, non-verifiability, and incompleteness — applying fifteen rule patterns via lexicon matching and regular expressions. Each detected issue is tagged with type, severity, and flagged term.

Stage 3 — Score: Each requirement receives a score via Score = max(0, 100 − Σ(Severity Weight × Type Weight)), using severity weights H=25, M=15, L=8 and type multipliers NT=1.2, IC=1.1, AM=1.0. The document score is the mean of all requirement scores, mapped to five quality bands.

Stage 4 — Rewrite: Defective requirements are batched into a single Groq API call to Llama 3.3 70B, under a system prompt enforcing IEEE 830 conventions (shall, active voice, measurable specificity), with responses constrained to structured JSON keyed by requirement index.

Stage 5 — Save: The complete analysis and all per-requirement records are persisted to MongoDB Atlas, and the full JSON payload is returned to the frontend for immediate rendering.

3.5 Data Layer (Tier 3)

Persistence is accomplished using two MongoDB collections (via Mongoose): Analysis and Requirement. The Analysis Collection maintains document-level metadata:

- The file name
- Overall score
- Quality label
- Issue category counts
- Total number of requirements
- Processing time for document

The Requirement Collection maintains per-requirement data:

- Original text
- Score for the requirement
- Structured array of issues (type, severity, flagged word, description)
- LLM-generated rewrite of the original
- A flag indicating whether the rewrite was accepted (boolean, updated via PATCH when the user interacts)

4. METHODOLOGY

4.1 Document Parsing and Segmentation

Documents uploaded to the server are processed using pdf-parse for PDF files and using Node's native fs module for plain text. Once the text has been extracted, it is split into segments in two tiers. First, each line of text is split out as separate lines and any lines that match any of the specific pattern-based rules for headers, numbers, and boilerplate or other similar text are removed. All remaining lines are then treated as separate requirement statements. In the case of documents that are written in continuous prose instead of broken into paragraph forms, we fall back on sentence segmentation, which provides that each requirement will be separated from each other based on the period at the end of each sentence. However, the requirements that fall below a minimum threshold for total character size are not retained, because that would classify them as fragments.

4.2 Defect Detection Engine

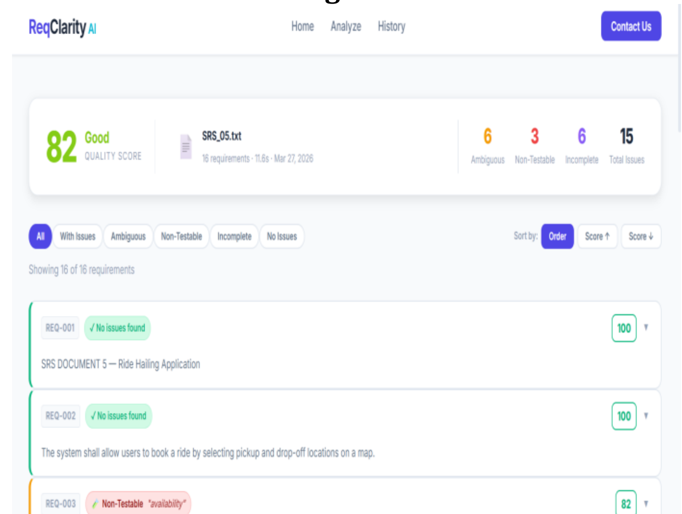

Fig. 3: Results view showing per-requirement defect tags, quality scores, and filter controls

The detection engine applies three independent modules – one per defect dimension. Each independent module is a combination of pre-defined lexicons and a set of regular expressions.

4.2.1 Ambiguity Detection

Five rule categories identify ambiguous language:

- *Vague adjectives and adverbs* — a lexicon of 23 terms including *fast*, *reliable*, *adequate*, and *appropriate*
- *Weak modal verbs* — *should*, *may*, *might*, *could*, and *would* in place of the IEEE 830-mandated *shall*
- *Passive voice constructions* — regex pattern detecting missing actor identification
- *Vague quantifiers* — *several*, *many*, *few*, *some*, *various*, *numerous*
- *Imprecise time references* — *soon*, *quickly*, *periodically*, *in a timely manner.*

4.2.2 Non-verifiability Detection

Five rule categories identify requirements that cannot be objectively verified:

- *Subjective quality terms* — a lexicon of 19 terms including *user-friendly*, *intuitive*, *seamless*, *high quality*
- *Unmeasurable performance claims* — performance verbs without adjacent numeric values
- *Missing metric patterns* — quantifiable behaviours described without numeric bounds
- *Missing acceptance criteria* — verbs such as *ensure*, *guarantee*, *provide* without measurable parameters
- *Subjective comparisons* — *better than*, *faster than* without a defined baseline

4.2.3 Incompleteness Detection

Five rule categories identify requirements with missing behavioural specification:

- *Missing actor* — requirements beginning with *shall* without a subject entity
- *Incomplete conditionals* — *if/when/unless* clauses without a corresponding consequence
- *Missing constraints* — sub-patterns for absent size, authentication, time, and display constraints
- *Incomplete sentence structures* — requirements lacking a complete predicate
- *Missing error handling* — operations with no failure behaviour specified

4.3 Quality Scoring Model

Each requirement is assigned a quality score in the range [0, 100] using a weighted penalty model:

$$\text{Score} = \max\left(0,\ 100 - \sum_i (\text{Severity}_i \times \text{TypeWeight}_i)\right)$$

Severity weights reflect the relative impact of each issue on requirement usability: High = 25, Medium = 15, Low = 8. Type multipliers reflect the relative downstream cost of each defect dimension based on established requirements engineering literature: Non-verifiability = 1.2, Incompleteness = 1.1, Ambiguity = 1.0. Multiple issues on a single requirement accumulate penalties additively, with the score floored at zero.

Table 1: Quality band classification thresholds

Score Range	Quality Label	Interpretation
90 – 100	Excellent	Minimal or no defects detected
75 – 89	Good	Minor issues; light revision recommended
60 – 74	Fair	Moderate defects; revision required
40 – 59	Poor	Significant defects; substantial rework needed
0 – 39	Critical	Pervasive defects; document requires full revision

4.4 LLM-Assisted Requirement Rewriting

If a requirement receives a score lower than *Excellent*, the system routes it to the LLM-based rewrite pipeline. The system batches defective requirements into a single API call to the Groq inference API using the Llama 3.3 70B model, thereby reducing latency associated with sequential invocation.

The system prompt instructs the model to generate IEEE 830-compliant rewrites for each requirement while enforcing four correction principles: (1) replacement of weak modal verbs with *shall*; (2) transformation of passive constructions into active voice; (3) substitution of vague expressions with measurable and verifiable specifications; and (4) inclusion of missing actors, constraints, or error-handling clauses inferred from contextual information.

The model returns its output as a structured JSON object keyed by requirement index, enabling deterministic parsing without reliance on post-processing heuristics.

The system stores each rewritten requirement alongside its original counterpart and presents both through the Rewrite interface for user validation. Upon user acceptance, the system replaces the original requirement with the rewritten version during SRS PDF export; otherwise, it discards the rewrite and retains the original text.

5. RESULTS AND EVALUATION

5.1 Evaluation Setup

A test corpus of fifteen SRS documents was created using five different quality bands — Critical, Poor, Fair, Good and Excellent — with three documents for every band. The documents used for the corpus came from several domains: e-commerce, hospital management, mobile banking, inventory management, and university administrative support. The documents were deliberately constructed to represent a wide range of defect densities. Evaluation occurred across four dimensions — segmentation accuracy, defect detection (performance), scoring consistency and quality of AI-generated rewrites. The author independently labelled 75 requirements to establish ground truth; this

sample was taken from a subset of five documents and included labelling each requirement for ambiguity, non-verifiability, and incompleteness. To mitigate subjective bias, future work will include multi-annotator validation and inter-rater agreement metrics (e.g., Cohen's kappa).

5.2 Segmentation Results

All fifteen SRS documents were successfully segmented into discrete (individual) requirements with no manual segmentation being performed by the author. The parser's primary segmentation engine processed line-formatted documents, while the secondary segmentation/formatting methodology processed continuous prose documents at the sentence level. In total, 187 discrete requirements were identified across all fifteen SRS documents, and no instances of over-segmentation were found within compound requirements that included explicit conjunctions.

5.3 Defect Detection Performance

Table 2 presents precision, recall, and F1 scores for each detection module evaluated against the 75 manually annotated requirements.

Table 2: Defect detection precision, recall, and F1 scores per category.

Defect Category	Precision	Recall	F1 Score
Ambiguity	0.87	0.83	0.85
Non-verifiability	0.81	0.79	0.80
Incompleteness	0.84	0.76	0.80
Overall	**0.84**	**0.79**	**0.82**

The ambiguity module achieved an F1 score of 0.85, reflecting the effectiveness of lexical matching against a well-defined set of vague terms. Non-verifiability and incompleteness modules produced F1 scores of 0.80. Most false negatives that occurred in the incompleteness module were due to requirements omitting constraints in domains not covered by the pre-defined set of keywords. Most of the false positives that occurred in the non-verifiability module were due to using domain-specific technical terms that were matched incorrectly against the lexicon used for subjective quality terms.

5.4 Scoring Consistency

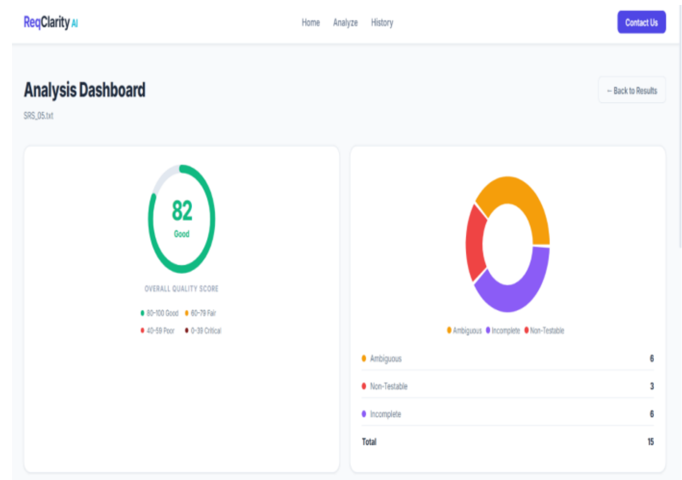

Fig. 4: Analysis dashboard displaying overall quality score and defect category breakdown

Notably, all fifteen documents were assigned scores that reflected the quality level intended, suggesting that the weighted penalty model can efficiently separate documents that have varying quality levels. Mean scores also indicate that adjacent score bands produced significant mean score differences without overlap between observed ranges of scores produced with tested documents.

Table 3 presents the system-assigned scores for each quality band against their intended ranges.

Table 3: Scoring consistency across quality bands.

Intended Band	Documents	Mean Score	Score Range	Correct Band
Critical (0–39)	3	31.4	24–38	3/3
Poor (40–59)	3	51.2	44–57	3/3
Fair (60–74)	3	67.8	63–72	3/3
Good (75–89)	3	81.3	77–86	3/3
Excellent (90–100)	3	94 .6	92–97	3/3

5.5 AI Rewrite Quality

The rewrite pipeline was evaluated on 43 defective requirements drawn from the test corpus, assessed manually against three criteria: defect elimination, intent preservation, and IEEE 830 compliance.

Table 4: AI rewrite quality evaluation results.

Criterion	Count	Percentage
Defects fully eliminated	38 / 43	88.4%
Original intent preserved	40 / 43	93.0%
IEEE 830 compliant	36 / 43	83.7%

The most common failure mode was over-specification due to the identification of numeric values (e.g., specific response periods, data limits) that could not be derived from within the context of the original requirement and that would have needed verification from the stakeholder for acceptability. Over-specification is an inherent limitation of the use of single-turn LLM inference without access to domain context that goes beyond the original requirement text.

5.6 Pipeline Performance

Across all fifteen test documents, end-to-end processing time was measured. The average time from the moment the user uploads a document until the results are displayed was 6.3 seconds, with most of the total time (approximately 74%) relating to the time taken to rewrite the requirements using AI. Documents that had fewer than five defects in their set of requirements were processed in under 4 seconds. The maximum processing time recorded was 11.2 seconds, for the document containing 22 total requirements; 18 of the requirements required an AI rewrite. These results are deemed acceptable for web-based systems, as the processing is done asynchronously and a loading icon is displayed during processing.

6. CONCLUSION

6.1 Summary

This research details ReqClarity AI, an automated quality assessment tool for Software Requirements Specifications (SRS) available via a web-based portal. ReqClarity AI employs natural language processing (NLP) techniques and scoring to identify defects and provide feedback to authors. This study demonstrates that ReqClarity AI addresses a significant gap in existing quality assessment solutions, particularly given the absence of free, comprehensive, end-to-end solutions.

The detection engine containing 15 patterns to detect defects with three defect categories achieved a total F1 score of 0.82 when using a manually annotated test dataset and showed that computationally lightweight methods based on pattern matching could successfully identify defects within the various defect creation categories using no model training or collected labelled datasets. Similarly, the weighted scoring method matched all 15 of the test documents to their respective quality bands with no overlap into adjacent bands. In addition, the LLM rewrite pipeline produced IEEE 830 / ISO/IEC/IEEE 29148-aligned rewrites for 88.4% of defective requirements with a mean processing time of 6.3 seconds end-to-end.

Furthermore, ReqClarity AI is proof that it is feasible to combine deterministic rule-based analysis for explainable defect detection with generative AI tools to create a seamless analytical process flow. The rule-based layer identifies and records defects to provide evidence for corrective action. However, its inherent limitation is an inability to generate fluent, context-appropriate natural language corrections — a gap addressed by the LLM rewriting stage. The overall performance of the hybrid model demonstrates effective detection and correction of software defects through the integration of deterministic rule-based processes and complex natural language generation design patterns.

6.2 Limitations

Several limitations of the current system merit acknowledgment. The detection engine relies on fixed lexicons and regex patterns, which may not generalize to highly domain-specific SRS documents using specialized technical vocabularies outside the predefined keyword sets. The scoring model's penalty weights and type multipliers were determined empirically rather than through a formal calibration study against a large, annotated corpus. The AI rewrite pipeline's tendency toward over-specification — introducing numeric values not derivable from original requirement context — is an inherent limitation of single-turn LLM inference without access to broader domain knowledge. Finally, the system does not currently detect cross-requirement consistency defects, a category identified in the literature as a significant contributor to downstream development errors.

6.3 Future Work

Several directions are identified for extending ReqClarity AI in future work.

Cross-requirement consistency checking is the most immediately impactful extension. The current system analyses each requirement in isolation; detecting contradictions, duplicate specifications, and dependency conflicts across requirements requires semantic similarity measures or graph-based dependency modelling that will be explored in subsequent iterations.

ML-based detection enhancement represents a natural evolution of the rule-based engine. Fine-tuning a BERT-based classifier on a requirements-specific annotated corpus — such as the PROMISE NFR dataset — would extend defect detection to semantic and pragmatic ambiguity categories that lexicon and regex patterns cannot reliably capture, while retaining the rule-based layer for high-recall initial screening.

Multi-turn LLM rewriting could address the over-specification failure mode identified in the evaluation. A conversational refinement loop — where the model requests missing contextual information (such as performance bounds or actor identities) before generating a rewrite — would

produce corrections that are both standard-compliant and stakeholder-validated.

Support for additional requirement formats including user stories, use case specifications, and Gherkin-style acceptance criteria would extend the tool's applicability beyond traditional IEEE 830-formatted SRS documents to agile development contexts.

User authentication and team collaboration features would enable shared SRS workspaces where multiple engineers can jointly review, accept, and track rewrites across document versions — a prerequisite for adoption in organizational settings.

Formal scoring calibration via a large-scale annotated requirements dataset would replace the current empirically determined penalty weights with statistically grounded values, improving the scoring model's generalizability across domains and document styles.

6.4 Closing Remarks

ReqClarity AI is freely accessible at https://req-clarity-ai.vercel.app and the source code is available at https://github.com/urzarai/ReqClarity_AI. The system is designed with a modular pipeline architecture that accommodates the extensions identified above without requiring fundamental restructuring, positioning it as a foundation for continued research and development in automated requirements quality assurance.

ACKNOWLEDGEMENT

The author wishes to thank Professor Dr. Srivani A. for her invaluable guidance, motivation, and continuous support throughout the development of this project on SRS analysis and rewriting using artificial intelligence. Her insights and constructive feedback significantly contributed to shaping both the direction and quality of this work.
The author also wishes to thank Dr. Sharmila Banu, Head of the Department, for her constant encouragement, her invaluable guidance, and for fostering a positive academic environment that promotes creativity and learning.
Finally, the author acknowledges Vellore Institute of Technology, Vellore, for providing the necessary resources, facilities, and opportunities to successfully complete this project.

REFERENCES

[1] V. Gervasi, A. Ferrari, D. Zowghi, P. Spoletini, "Ambiguity in Requirements Engineering: Towards a Unifying Framework," n.d.

[2] G. Lami, C. Scondras, "QUARS: A Tool for Analyzing Requirements," Carnegie Mellon University, Technical Report CMU/SEI-2005-TR-014, 2005.

[3] G. Lami, S. Gnesi, F. Fabbrini, M. Fusani, G. Trentanni, "An Automatic Tool for the Analysis of Natural Language Requirements," n.d.

[4] A. Ferrari, G. Lipari, S. Gnesi, G. O. Spagnolo, "Pragmatic Ambiguity Detection in Natural Language Requirements," *AIRE 2014*, 2014.

[5] S. Ezzini, S. Abualhaija, C. Arora, M. Sabetzadeh, "Automated Handling of Anaphoric Ambiguity in Requirements: A Multi-Solution Study," *ICSE 2022*, 2022.

[6] D. M. Berry, E. Kamsties, M. M. Krieger, "From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity," 2003.

[7] A. Bajceta, M. Leon, W. Afzal, P. Lindberg, M. Bohlin, "Using NLP Tools to Detect Ambiguities in System Requirements: A Comparison Study," *NLP4RE Workshop*, 2021.

[8] A. Ramos, "Enhancing Requirements Quality Through Automated Ambiguity Detection: A Comparative Analysis of Manual, Rule-Based and Generative AI Techniques," Thesis, 2025.

[9] A. K. Massey, R. L. Rutledge, A. I. Antón, P. P. Swire, "Identifying and Classifying Ambiguity for Regulatory Requirements," 2021.

[10] A. Nigam, N. Arya, B. Nigam, D. Jain, "Tool for Automatic Discovery of Ambiguity in Requirements," *International Journal of Computer Science Issues*, vol. 9, no. 5, pp. 350–357, 2012.

[11] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, M. A. Ajagbe, E.-V. Chioasca, "Natural Language Processing for Requirements Engineering: A Systematic Mapping Study," *ACM Computing Surveys*, vol. 54, no. 3, 2022.

[12] S.-C. Necula, F. Dumitriu, V. Greavu-S, "A Systematic Literature Review on Using Natural Language Processing in Software Requirements Engineering," *Electronics*, vol. 13, 2024.

[13] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, S. Gnesi, "Detecting Requirements Defects with NLP Patterns: An Industrial Experience in the Railway Domain," 2021.

[14] C. Arora, M. Sabetzadeh, L. C. Briand, F. Zimmer, R. Gnaga, "RUBRIC: A Flexible Tool for Automated Checking of Conformance to Requirement Boilerplates," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 4, 2021.

[15] L. Zhao, W. Alhoshan, A. Ferrari, K. J. Letsholo, "Classification of Natural Language Processing Techniques for Requirements Engineering," *IEEE RE 2021*, 2021.

[16] N. Kiyavitskaya, N. Zeni, L. Mich, D. M. Berry, "Requirements for Tools for Ambiguity Identification and

Measurement in Natural Language Requirements Specifications," 2007.

[17] S. Ezzini, S. Abualhaija, C. Arora, M. Sabetzadeh, "Using Domain-Specific Corpora for Improved Handling of Ambiguity in Requirements," *IEEE Transactions on Software Engineering*, 2021.

[18] F. Dalpiaz, I. van der Schalk, S. Brinkkemper, F. B. Aydemir, G. Lucassen, "Detecting Terminological Ambiguity in User Stories: Tool and Experimentation," *Information and Software Technology*, vol. 110, pp. 3–16, 2018.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, "Attention Is All You Need," *NeurIPS*, 2017.

[20] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *NAACL-HLT*, 2019.

BIOGRAPHY

Urza Rai is an undergraduate Computer Science student at Vellore Institute of Technology, with strong interests in software engineering, AI, and requirements analysis.