Multi-Step Generation of Test Specifications using Large Language Models for System-Level Requirements

Dragan Milchevski[1] **Gordon Frank**[2] **Anna Hätty**[1]
Bingqing Wang[3] **Xiaowei Zhou**[3] **Zhe Feng**[3]
[1]Bosch Center for Artificial Intelligence, Renningen, Germany
[2]Bosch Vehicle Motion, Abstatt, Germany
[3]Bosch Research and Technology Center North America, Sunnyvale, USA
{dragan.milchevski,gordon.frank,anna.haetty}@de.bosch.com
{bingqing.wang,xiaowei.zhou2,zhe.feng2}@us.bosch.com

Abstract

System-level testing is a critical phase in the development of large, safety-dependent systems, such as those in the automotive industry. However, creating test specifications can be a time-consuming and error-prone process. This paper presents an AI-based assistant to aid users in creating test specifications for system-level requirements. The system mimics the working process of a test developer by utilizing a LLM and an agentic framework, and by introducing intermediate test artifacts—structured intermediate representations derived from input requirements. Our user study demonstrates a **30 to 40%** reduction in effort required for test development. For test specification generation, our quantitative analysis reveals that iteratively providing the model with more targeted information, like examples of similar test specifications, based on comparable requirements and purposes, can boost the performance by up to **30%** in ROUGE-L. Overall, our approach has the potential to improve the efficiency, accuracy, and reliability of system-level testing and can be applied to various industries where safety and functionality are paramount.

1 Introduction

In industries such as aerospace, telecommunications, electronics, software, and automotive engineering, the systems to be developed are often complex due to the intricate relationships among numerous interdependent components. Effective system-level testing is a critical phase that verifies whether the complete and integrated system meets their intended functionality and performs as expected. It guarantees that all components work together seamlessly, ensuring the safety, functionality, and reliability of complex systems. To achieve this, system-level testing is typically conducted in controlled environments such as Software-in-the-Loop (SiL, Umang et al.) or Hardware-in-the-Loop (HiL, Ledin, 1999) and must be well documented, as emphasized by standards like Automotive SPICE (ASPICE, VDA QMC, 2023). To support system-level testing, two distinct roles collaborate: the requirements engineer, who authors system-level requirements, and the test developer, who derives detailed test specifications from those requirements. In this paper, we focus on assisting test developers to create test specifications.

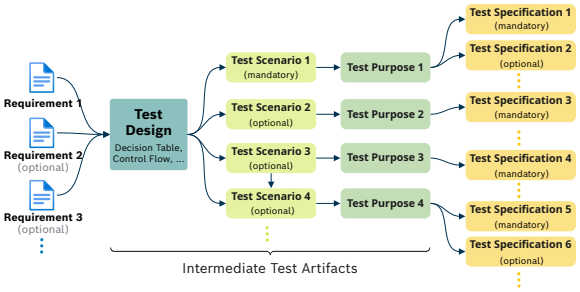

Figure 1: Illustrated Workflow - Deriving Test Specifications from Requirements

A test specification typically includes the definition of tests to be performed, the expected result, associated testing conditions, and other information. In contrast to software testing, both the system-level requirements and test specifications are documented in *natural language*, taking the form of text documents rather than code (cf. VDA QMC, 2023). To bridge the large gap between the input requirements and the final test specifications, test developers typically follow a structured workflow of five steps to generate concrete test specifications, as illustrated in Figure 1:

First, the test developers **(i) group input requirements into clusters** (see examples given in Figure 2). Then, they select a test design technique according to ISO/IEC/IEEE (2021) and **(ii) create a test design** based on the requirements. The test design is an abstraction of the tests, and specifies the relationship between the input conditions and the output expected results. Test designs are re-

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 132–146

quired to cover a vast array of corner cases, which helps to verify the logic of the function, and check potential errors or faults. Test designs can be expressed in diagrams (see example in Figure 3) or decision tables (see example in Table 6 in Appendix A.1 for a corresponding example), with each row or path defining a single **(iii) test scenario**. Once they have identified the test scenarios, they derive a **(iv) test purpose** for each scenario, which is a specific reason or objective that the test specifications need to cover. In the last step, the developers create **(v) test specifications** (see Figure 4) that consist of the previously generated test purpose, pre- and post-conditions as well as execution steps that testers shall follow. To offer a better understanding of the test development process in massive system production, more details on above examples are given in Appendix A.1.

Requirements

R1: The lane assist function shall activate only when the vehicle speed exceeds 60 km/h.

R2: The lane assist function shall require manual activation by the driver.

R3: The lane assist function shall operate only when clear lane

Figure 2: Example requirements cluster

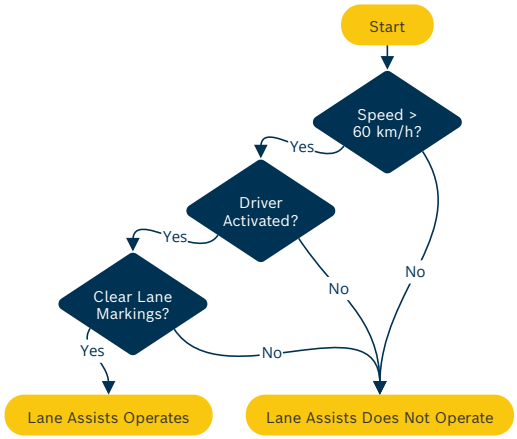

Figure 3: Example control flow chart for lane assist function

Typically, the process described above will cost the test developers a huge amount of manual efforts to handle the challenges such as the many-to-many relationship between requirements and test specifications, the mixture of natural language descriptions and variable assignments that need to be precisely met in test specifications, and the injection of domain know-how. This paper introduces an AI-based test development assistant that harnesses large language models and Agentic AI to assist the test developers to streamline this process. A user study demonstrates that our approach reduces the time needed to derive test specifications from requirements by 30–40% on average, significantly boosting both efficiency and accuracy.

Test Specification

Preconditions

- The vehicle ignition is "ON".
- Vehicle_Speed is at standstill.
- Lane_Assist_Status = "Off".
- Driver_Lane_Assist_Activation = "False".
- Set Lane_Marking_Detected = "True".

Test Steps and Expected Results

- Activate the lane assist by setting Driver_Lane_Assist_Activation = "True". Expected Value: Lane_Assist_Status = "Ready".
- Gradually increase Vehicle_Speed to 65 km/h over 10 seconds. Expected Value: Lane_Assist_Status = "Active".
- Gradually decrease Vehicle_Speed to 55 km/h over 10 seconds. Expected Value: Lane_Assist_Status = "Ready".
- Deactivate the lane assist by setting Driver_Lane_Assist_Activation = "False". Expected Value: Lane_Assist_Status = "Off".
- Gradually decrease Vehicle_Speed to 0 km/h over 10 seconds.
- Retrieve data from the system's failure memory.

Postconditions

- Clear the failure memory.
- Reset conditions back to preconditions if required.

Figure 4: Example test spec for lane assist function

Overall, our key contributions are summarized as follows:

(i) **An End-to-End AI-Based Test Development Assistant**: We introduce an innovative AI-based test development assistant that leverages domain expertise and historical data to generate high-quality test specifications for system-level requirements.

(ii) **Intermediate Test Artifacts**: Our system automatically generates intermediate test artifacts—structured representations such as tables and graphs—that effectively bridge the gap between input requirements and final test specifications, thereby resolving the inherent many-to-many relationship between them.

(iii) **Human-in-the-Loop Test Workflow**: Designed to foster collaboration, our system offers actionable suggestions while allowing test developers to inspect, refine, and extend the

generated test artifacts and test specifications, ensuring a seamless iterative process.

(iv) **End-to-End Evaluation with Test Experts**: We conducted an end-to-end evaluation with expert test developers from our organization.

2 Related Work

2.1 System Testing in Industry

Regarding development frameworks, system-level testing is independent of specific methodologies, and can be applied to a range of approaches, including the V-Model (Johansson and Bucanac, 1999), Agile, and Waterfall. Various testing methodologies exist, each tailored to distinct objectives. Requirement-based testing (Mustafa et al., 2021), for instance, extracts test cases directly from system requirements, thereby validating all functionalities. Model-based testing (Mohd-Shafie et al., 2021), on the other hand, leverages system behavior models to generate test scenarios, ensuring comprehensive coverage of interactions. We follow a requirement-based testing approach.

2.2 AI in Requirement Engineering

Prior art on requirements analysis include information extraction (Holter and Ell, 2023; Das et al., 2023; Nguyen et al., 2024), classification (Kici et al., 2021; Li et al., 2022; Khayashi et al., 2022; Yildirim et al., 2023; Nayak et al., 2023), consistency checking (Bertram et al., 2023; Marchezan et al., 2024), mapping and consolidating requirements (Sonbol et al., 2022; Bertram et al., 2022a,b; Subahi, 2023) and requirements generation (Krishna et al., 2024). These AI techniques are either adopted to identify key information for downstream tasks, or to improve the writing quality of the requirements, reducing mistakes and resolving ambiguities in the writing.

Regarding test specifications, LLM-based test generation is an increasingly researched subfield of code generation (Jiang et al., 2024). Generative AI-based software testing has been studied intensively as shown in surveys (Wang et al., 2024; Jin et al., 2024), and software testing is mostly applied for test generation, program debugging and bug repair. There is a notable amount of work that explores the relationship between requirements and test generation. Han et al. (2024) propose a framework for code generation and test execution, where new requirements are generated to create more correct tests. Yang et al. (2023) develop an interactive tool for requirements elicitation, integrating a component to write tests. Wei (2024) apply an LLM-based approach to interpret provided requirements, modifying extracted information to object-oriented models to generate test cases. Requirements are often represented as UML or use case diagrams (e.g., Mustafa et al., 2021; Sarma et al., 2007; Swain et al., 2010), which allows Naimi et al. (2024) to extract use case details from UML diagrams in XML format to automatically generate structured prompts for test creation.

There is limited recent research on AI-supported generation of *natural language* test specifications. Adabala et al. (2024) propose a pipeline for generating test flows for functional safety requirements by generating similar test specifications as examples for the language model. Liu et al. (2024) enhance a LLM through data augmentation, transforming the one-to-many relationship between requirements and test specifications into multiple one-to-one relationships. They augment the model input by adding either the test objective or a LLM-generated summary of the test specification. Arora et al. (2024) present research closely related to our work using a RAG framework to generate test specifications given several input requirements, utilizing a documentation corpus and optional one-shot examples. They incorporate a test description to guide the model during generation. In contrast to these methods, our approach integrates a retrieval component based on historical requirements and test specifications. As key difference to all previous approaches, we use **test artifacts** (i.e., test design, test scenarios and test purposes) as an intermediate structured representation to address the many-to-many relationship of requirements and test specifications. Notably, Liu et al. (2024) and Arora et al. (2024) focus on one-to-many and many-to-one relationships, respectively. As a means to address these challenges, they add test descriptions to the input of the LLM. In contrast, our approach is fully automated; test descriptions, in our case the test purposes, are automatically generated to guide the model. The test purposes are generated from the test scenarios, which again are automatically derived from the input requirements applying a test design technique.

3 Method

We designed a novel system for generating test specifications from input requirements. The sys-

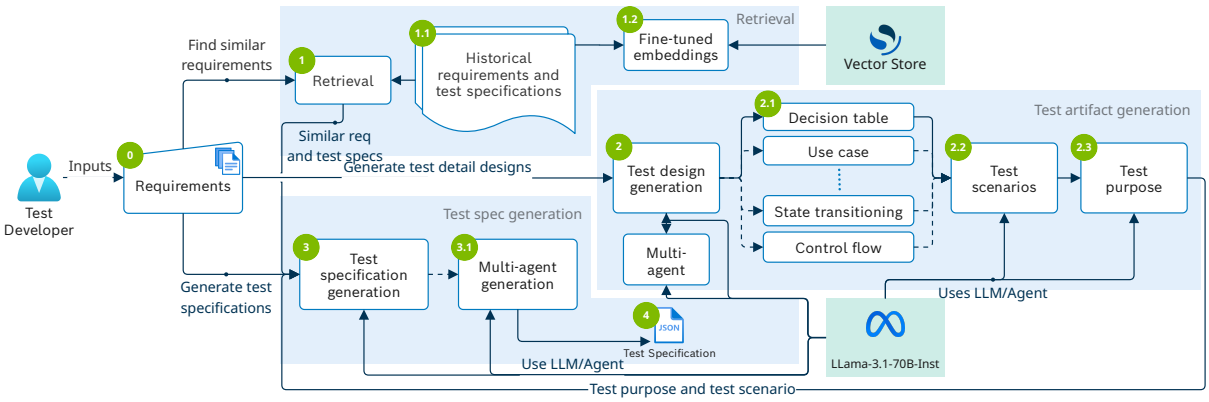

Figure 5: System's workflow and architecture

tem architecture is illustrated in Figure 5. The system begins with the user entering requirements as input (point 0 in Figure 5), followed by a retrieval step to find similar requirements (1). The data basis consists of historical requirements and linked historical test specifications (1.2). The ultimate goal of retrieving similar requirements is to identify one or more existing test specifications that can serve as examples for the final step: test specification generation. Consequently, during this step, users can review associated historic test specifications and select one or more as examples to guide the generation process. The retrieval process is two-pronged, leveraging both sparse retrieval (BM25) and dense retrieval (with fine-tuned embeddings, 1.2). To get more precise results, the system also allows the user to apply filters to refine the retrieval outcomes and select more relevant test specifications from the retrieved ones.

The embedding model used in retrieval is fine-tuned based on the *bge-m3* model (Chen et al., 2024). We start with continuous pre-training on the domain, followed by a two-step fine-tuning approach. First, we fine-tune on abbreviation-substituted requirement pairs to teach the model the meaning of the abbreviations in context. Next, we fine-tune with synthetic similar requirement pairs and requirement pairs sharing the same test linkage. Finally, we boost the performance of the model by merging the original embedding model with the fine-tuned model using LM-Cocktail (Xiao et al., 2023). Details for the datasets we used can be found in Section 4.1. Further details on the fine-tuning process are given in Appendix A.2.

Following the retrieval of similar requirements, test artifacts are generated (2). The system employs *Llama-3.1-70B* as LLM to suggest up to three test design techniques that are best suited for the selected requirements (2.1). Users can also select an alternative technique if preferred. The test design techniques include *decision table testing*, *use case testing*, *control flow testing* and *state transitioning testing*, amongst others. Depending on the chosen technique, the system generates comprehensive test designs in either Markdown tables or machine-readable diagrams in Mermaid format. Users can review and edit these designs. Examples of a control flow diagram and an equivalent decision table are provided in Figure 3 and Table 6 (Appendix A.1), respectively. To enhance the user experience, our system employs a multi-agent approach (4): a design agent generates an initial test design, the user reviews the output and a separate reflection agent subsequently verifies the design's adherence to industry standards, such as ISO 26262.

Using the generated test design as a deterministic basis, the system extracts a test scenario for each row in the Markdown table or each path in the Mermaid diagram (2.2). An LLM is then used to generate the test purpose for each scenario (2.3). Users can review and refine these generated test purposes, selecting the ones they wish to use for creating test specifications. Example test purposes for each test scenario derived from Figure 3 (or from equivalent Table 6) are shown in Table 1.

Finally, the system generates comprehensive test specifications for the selected test purposes, considering the test scenario, the input requirements and similar test specifications (3). The test specification is presented in a structured format, outlining the relevant steps for testers to follow during the testing process. Notably, the system provides test specifications in JSON format, enabling seamless integration with downstream workflow steps, such

Verify that the lane assist function operates when the vehicle speed exceeds 60 km/h, the driver has activated the system, and clear lane markings are detected.
Verify that the lane assist function does not operate when the vehicle speed is below 60 km/h, the driver has activated the system, and clear lane markings are detected.
Verify that the lane assist function does not operate when the vehicle speed exceeds 60 km/h, the driver has not activated the system, and lane markings are detected.
Verify that the lane assist function does not operate when the vehicle speed exceeds 60 km/h, the driver has activated the system, and lane markings are not detected.

Table 1: Example test purposes for lane assist function

as automated test script generation for execution in Software-in-the-Loop (SiL) or Hardware-in-the-Loop (HiL) environments. Similar to the previous step, we utilize a multi-agent approach for the enhancement of the test specification generation (step 3.1). Demo screenshots of the application and their intermediate steps are shown in Appendix A.5.

4 Datasets

4.1 Datasets for Embedding Fine-tuning

In order to fine-tune embeddings for retrieving similar requirements, we created three datasets: 1) synthetic sets of similar requirements, 2) test-based requirement sets, and 3) abbreviation-substituted requirement pairs. The synthetic requirement sets were created using the data generation algorithm described in the next paragraph. To incorporate historical sets of similar requirements, we focused on those that share the same test linkage, which we refer to as test-based requirements. We then excluded any requirements with low embedding similarity within each set (<0.8). Abbreviation-substituted requirement pairs were created from duplicating a requirement, and then using the abbreviation in one instance and the full expression in the other. Details are given in Appendix A.2.

Generation of Synthetic Requirements. We propose an algorithm that decomposes the input requirement into its constituent parts and modifies them to create new requirements that maintais similarity to the original. The structure of a requirement can be defined as consisting of several key elements, including condition, subject, action, object, and constraint of action (ISO/IEC/IEEE, 2011). The core of the algorithm is to selectively modify specific parts of the requirement under certain conditions, ensuring that the resulting requirement stays similar to the original one. The algorithm works as follows:

Algorithm 1 Requirement Modification Algorithm

1: Decompose the input requirement into its constituent parts, including condition, subject, action, object, and constraint of action.
2: Select one and only one part that is not empty, among condition, subject, or object, and change its content to fit a similar requirement.
3: **if** constraint of action is empty **then**
4: Create some content that fits a similar requirement (e.g., time, signals with certain values).
5: **else**
6: Change it to an empty string.
7: **end if**
8: **if** object is empty **then**
9: Create some content that fits a similar requirement.
10: **end if**
11: Rewrite the synthetic requirement in natural language.
12: Output the modified requirement in JSON format, including the "Changed" field with the name of the changed key.

4.2 Dataset for Evaluation

Retrieval Component. To evaluate the embedding model, a dataset of similar requirement pairs was curated. It includes 48 pairs selected by test engineers from existing requirements, as well as 45 pairs, each consisting of one existing requirement and one newly crafted requirement.

Test Designs. The evaluation of the test design generation was done with a manually created dataset, containing 22 decision tables, 20 use case designs, 6 control flow diagrams, and 2 state transitioning diagrams. The distribution of test design techniques is based on simple random sampling.

Test Specifications. For the evaluation of the generated test specifications, a representative sample of 98 test specifications was chosen from historical data. These varied in terms of their related system functions and complexity. For the selected test specifications, all the linked requirements, along with the test scenarios and the related test purposes, were used as input, as well as one retrieved similar requirement with its linked test specification as

	HIT@1	HIT@3	HIT@5	HIT@10
Sparse Retrieval	46.24	66.67	76.34	82.80
Dense Retrieval - base embedding	45.16	65.59	74.19	81.72
Dense Retrieval - fine-tuned embed.	**53.76**	**76.34**	**81.72**	**91.40**

Table 2: Evaluation of the requirement retrieval.

	ROUGE-L	BERTScore	LLM-as-Judge
Decision Table Testing	27.13	86.11	26.36
Control Flow	36.37	90.32	38.33
Use Case Testing	20.16	78.07	35.00
State Transitioning	24.88	86.53	25.00

Table 3: Evaluation of the generated test designs

	ROUGE-L	BERTScore	LLM-as-Judge
Zero Shot	12.75	82.46	20.52
Zero Shot & Purpose	14.47	83.78	18.96
Few Shot & Purpose			
+ similar requirements	37.53	88.86	28.76
+ similar purpose	**44.46**	**90.13**	33.71
+ iso standards	44.12	90.01	**34.74**
Revised version by a reflection agent			
Zero Shot	10.43	81.92	23.64
Zero Shot & Purpose	13.25	83.20	18.76
Few Shot & Purpose			
+ similar requirements	29.19	87.05	29.89
+ similar purpose	**33.15**	**87.73**	**32.06**
+ iso standards	32.87	87.69	31.75

Table 4: Evaluation results of the test generation component average results from 98 samples

few-shot example. Including the test design itself was not necessary since the derived test scenarios and test purposes already included the relevant information.

5 Experiments

We evaluate the performance of our method by *(i) evaluating every component individually* and employing quantitative metrics such as HIT rate, ROUGE-L (Lin, 2004), BERTScore (Zhang et al., 2019), and using a LLM as a judge and *(ii) evaluating the end-to-end system through user evaluation*.

5.1 Quantitative Evaluation

Evaluating the Retrieval System. The results for the retrieval system are presented in Table 2. We observe a 9-point improvement of the fine-tuned embedding model compared to the sparse retrieval, and a nearly 10-point improvement compared to the base embedding model.

Evaluating the Generation Components. We evaluated the performance of our test design and test specification generation components using two metrics, ROUGE-L and BERTScore. We used the mean F1 score from BERTScore as a key metric, utilizing the default English language embedding model without fine-tuning on our domain (*roberta-large_L17_no-idf_version=0.3.12*). Additionally, we obtained subjective assessments from GPT-4o, which rated each generated output against a reference output on a scale of 1 to 10, providing a detailed explanation to support their rating. The results of the evaluation of the generated test designs are presented in Table 3 and reported as mean scores. We evaluated the test specification generation capabilities of our system through a five-stage process. First, we employed a zero-shot approach, where the LLM generated test specifications solely based on the input requirements without specifying the test purpose which is the main product of the intermediate test artifacts. Then, we added the test purpose, still in a zero-shot setting. Next, we provided the model with examples of similar test specifications, based on similar requirements. Then, we enriched the data by fetching similar test specifications based on the test purpose. Finally, to further assess the system's performance, we experimented with prompting the model to adhere to specific ISO standards, such as ISO 26262. We conducted these experiments in both single LLM and multi-agent settings, where a reflection agent reviewed the response from the first agent. The results of our evaluation are presented in Table 4. Appendix A.3 presents our preliminary experiments across multiple language models.

Analysis of Evaluation Results. For test design generation (Table 3), control flow testing gives best results. We hypothesize that lower scores may be due to the fact that test developers created the test data with a focus on relevant scenarios, leveraging their experience from historical projects, whereas the language model adopted a more exhaustive approach, attempting to cover all possible combinations of values and corner cases. The effect of this is lower for control flow testing than for e.g. table design testing, since in the latter case a complete new row needs to be added, while for control flow testing an additional edge might be sufficient.

For test specification generation, our analysis reveals that providing the model with examples of similar test specifications, obtained through similar requirements and purpose, yields the best performance, outperforming the zero-shot approach by up to 30 points for ROUGE-L. Incorporating few-shot examples yields significantly greater improve-

Retrieval	Test Design	Purpose & Scenario	Test Spec
3.0	3.0	3.4	3.2

Table 5: Component-wise User Evaluation Results: Average Ratings on a Scale of 1 to 5 from 87 test runs.

ments in the surface-level metric ROUGE-L compared to BERTScore and LLM-as-a-Judge. This observation suggests that the LLM does not inherently possess the capability to accurately reproduce the specific language utilized in system-level tests. In contrast, instructing the model to adhere to ISO standards did not lead to significant improvements, suggesting that the model had already internalized this knowledge and was applying it without explicit instruction. The reflection agent underperformed compared to the initial response in both test design and test specification generation. This discrepancy is likely due to the agent's overly cautious approach, which prioritized strict adherence to guidelines and regulations over flexibility and natural language style. As a result, the generated responses tended to be more formal and rigid, deviating from the typical style of human-written test specifications.

5.2 User Study

To facilitate end-to-end evaluation of our system, we developed a simple application using Streamlit (Streamlit, 2024), which guides users through a four-step wizard process. Screenshots of the Streamlit demo can be seen in Appendix A.5. Ten experienced test developers from our organization participated in the evaluation, conducting a total of 87 test runs. The evaluation process consisted of four steps: (1) entering requirements, (2) selecting similar requirements based on the retrieval module and choosing example test specifications, (3) generating test design details and test scenarios with test purposes, and (4) generating test specifications. We asked the participants to evaluate the quality of each component on a rating scale of 1 to 5, as well as the overall usefulness of the system. The average results of the component evaluations are presented in Table 5. Notably, the participants estimated that the system saved them, on average, 30 to 40% of the time typically spent deriving test specifications from requirements.

6 Conclusions

In this paper, we introduce a novel AI-powered test development assistant for productive deployment. It is designed to help users to effectively derive test specifications from system-level requirements and significantly improving efficiency and accuracy. It employs historical similar requirements and linked test specifications, and utilizes intermediate test artifacts such as test designs, test scenarios, and test purposes, to generate new test specifications. By incorporating these test artifacts into the tool's workflow as a structured intermediate representation, we address the complex many-to-many relationships between requirements and test specifications. A user study showed a 30 to 40% reduction in effort required to derive test specifications using our tool. This system exemplifies the potential of LLMs to extend beyond mere language generation, showcasing their ability to design and produce structured outputs as helpful intermediate representations. Furthermore, our quantitative evaluation confirmed the effectiveness of our approach for system-level test specification language. We observe an improvement of roughly 30% ROUGE-L in comparison to the zero-shot approach.

7 Future Work

Although our initial results are encouraging, they point to two key avenues for further investigation:

Augmenting Inputs and Domain-Specific Fine-Tuning Our current pipeline relies solely on historical requirements and test specifications. We plan to explore the integration of additional artifacts—such as technical design documents, and architecture diagrams—either through enriched prompting or by fine-tuning the base LLM on these corpora. We hypothesize that this broader context will improve the model's domain understanding and lead to more accurate, context-aware test specifications.

Standards Compliance and Hallucination Analysis Preliminary trials showed no benefit from explicitly prompting the model to adhere to ISO standards. We will conduct a deeper analysis to determine whether this stems from prompt formulation, model biases, or gaps in the LLM's encoded knowledge of the standard. In parallel, we will develop metrics and manual review protocols to measure the model's hallucination rate in generated test specifications, ensuring that outputs remain reliable, traceable, and aligned with stakeholder expectations.

Acknowledgements

We gratefully acknowledge Michael Hofmann for his expert guidance and thoughtful feedback throughout this project. We also thank Tim Lukas Müller, Suneel Datta Kolipakula, and the entire testing team for rigorously testing and evaluating the system from an end-user perspective. Their insightful critiques were instrumental in shaping both the quality and the direction of our work.

References

Bhargav Adabala, Gerhard Griessnig, Adam Schnellbach, Martin Ringdorfer, Christian Santer, Aisha Maria Puchleitner, Kaan Suar, Martin Mandl, and Vanesa Klopic. 2024. Ai-driven test flow generation from semi-formal functional safety requirements. In *Systems, Software and Services Process Improvement*, pages 197–205, Cham. Springer Nature Switzerland.

Chetan Arora, Tomas Herda, and Verena Homm. 2024. Generating test scenarios from nl requirements using retrieval-augmented llms: An industrial study. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 240–251.

Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Helene Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. 2022a. Neural language models and few shot learning for systematic requirements processing in mdse. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, page 260–265, New York, NY, USA. Association for Computing Machinery.

Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Helene Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. 2022b. Technical report on neural language models and few-shot learning for systematic requirements processing in mdse. *Preprint*, arXiv:2211.09084.

Vincent Bertram, Hendrik Kausch, Evgeny Kusmenko, Haron Nqiri, Bernhard Rumpe, and Constantin Venhoff. 2023. Leveraging natural language processing for a consistency checking toolchain of automotive requirements. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 212–222.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Souvick Das, Novarun Deb, Agostino Cortesi, and Nabendu Chaki. 2023. Zero-shot learning for named entity recognition in software specification documents. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 100–110.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. 2024. Archcode: Incorporating software requirements in code generation with large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13520–13552.

Ole Magnus Holter and Basil Ell. 2023. Reading between the lines: Information extraction from industry requirements. In *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, pages 703–711.

ISO/IEC/IEEE. 2011. Systems and software engineering—life cycle processes—requirements engineering.

ISO/IEC/IEEE. 2021. Ieee/iso/iec international standard - software and systems engineering–software testing–part 4: Test techniques.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *URL https://arxiv.org/abs/2406.00515*.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*.

Conny Johansson and Christian Bucanac. 1999. The v-model. *IDE, University Of Karlskrona, Ronneby*.

Fatemeh Khayashi, Behnaz Jamasb, Reza Akbari, and Pirooz Shamsinejadbabaki. 2022. Deep learning methods for software requirement classification: A performance study on the pure dataset. *Preprint*, arXiv:2211.05286.

Derya Kici, Garima Malik, Mucahit Cevik, Devang Parikh, and Ayse Basar. 2021. A bert-based transfer learning approach to text classification on software requirements specifications. *Proceedings of the Canadian Conference on Artificial Intelligence*.

Madhava Krishna, Bhagesh Gaur, Arsh Verma, and Pankaj Jalote. 2024. Using llms in software requirements specifications: An empirical evaluation. *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, pages 475–483.

Jim A Ledin. 1999. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12:42–62.

Gang Li, Chengpeng Zheng, Min Li, and Haosen Wang. 2022. Automatic requirements classification based on graph attention network. *IEEE Access*, 10:30080–30090.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Hanyue Liu, Marina Bueno García, and Nikolaos Korkakakis. 2024. Exploring multi-label data augmentation for llm fine-tuning and inference in requirements engineering: A study with domain expert evaluation. *2024 International Conference on Machine Learning and Applications (ICMLA)*, pages 432–439.

Luciano Marchezan, Wesley K. G. Assunção, Edvin Herac, Saad Shafiq, and Alexander Egyed. 2024. Exploring dependencies among inconsistencies to enhance the consistency maintenance of models. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 147–158.

Muhammad Luqman Mohd-Shafie, Wan Mohd Nasir Wan Kadir, Horst Lichter, Muhammad Khatibsyarbini, and Mohd Adham Isa. 2021. Model-based test case generation and prioritization: a systematic literature review. *Software and Systems Modeling*, pages 1–37.

Ahmad Mustafa, Wan MN Wan-Kadir, Noraini Ibrahim, Muhammad Arif Shah, Muhammad Younas, Atif Khan, Mahdi Zareei, and Faisal Alanazi. 2021. Automated test case generation from requirements: A systematic literature review. *Computers, Materials and Continua*, 67(2):1819–1833.

Lahbib Naimi, Mohamed Manaouch, Abdeslam Jakim, and 1 others. 2024. A new approach for automatic test case generation from use case diagram using llms and prompt engineering. In *2024 International Conference on Circuit, Systems and Communication (ICCSC)*, pages 1–5. IEEE.

Anmol Nayak, Hari Prasad Timmapathini, Vidhya Murali, and Atul Anil Gohad. 2023. Few-shot learning approaches for classifying low resource domain specific software requirements. *Preprint*, arXiv:2302.06951.

Tai Nguyen, Yifeng Di, Joohan Lee, Muhao Chen, and Tianyi Zhang. 2024. Software entity recognition with noise-robust learning. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 484–496. IEEE Press.

Klaus Pohl. 2010. *Requirements engineering Fundamentals, Principles, and Techniques*. Springer Heidelberg Dordrecht London New York.

Monalisa Sarma, Debasish Kundu, and Rajib Mall. 2007. Automatic test case generation from uml sequence diagram. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 60–67. IEEE.

Yingxia Shao Zhao Cao Shitao Xiao, Zheng Liu. 2022. Retromae: Pre-training retrieval-oriented language models via masked auto-encoder. In *EMNLP*.

Riad Sonbol, Ghaida Rebdawi, and Nada Ghneim. 2022. The use of nlp-based text representation techniques to support requirement engineering tasks: A systematic mapping review. *IEEE Access*, PP:1–1.

Streamlit. 2024. Streamlit: Streamlit — a faster way to build and share data apps. `https://github.com/streamlit/streamlit`. Accessed: 2024-08-01.

Ahmad F. Subahi. 2023. Bert-based approach for greening software requirements engineering through non-functional requirements. *IEEE Access*, 11:103001–103013.

Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib Mall. 2010. Test case generation based on use case and sequence diagram. *International Journal of Software Engineering*, 3(2):21–52.

Gudapareddy Sasidhar Reddy Umang, Kushal Koppa Shivanandaswamy, S Pallavi, and Sivakumar Rajagopal. Software-in-the-loop (sil) method. In *Proceedings of the 10th International Conference on Mechanical, Automotive and Materials Engineering: CMAME 2023, 20-22 December, Da Nang, Vietnam*, page 243. Springer Nature.

WG13 VDA QMC. 2023. Automotive spice®. *Prozess Assessment Model*, 4:142.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Bingyang Wei. 2024. Requirements are all you need: From requirements to code with llms. *arXiv preprint arXiv:2406.10101*.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Xingrun Xing. 2023. Lm-cocktail: Resilient tuning of language models via model merging. *Preprint*, arXiv:2311.13534.

Chenyang Yang, Rishabh Rustogi, Rachel Brower-Sinning, Grace A Lewis, Christian Kästner, and Tongshuang Wu. 2023. Beyond testers' biases: Guiding model testing with knowledge bases using llms. *arXiv preprint arXiv:2310.09668*.

Savas Yildirim, Mucahit Cevik, Devang Parikh, and Ayse Basar. 2023. Adaptive fine-tuning for multiclass classification over software requirement data. *Preprint*, arXiv:2301.00495.

Peitian Zhang, Shitao Xiao, Zheng Liu, Zhicheng Dou, and Jian-Yun Nie. 2023. Retrieve anything to augment large language models. *Preprint*, arXiv:2310.07554.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

A Appendices

In the following sections, we report additional details on the following topics:

- Definitions and Examples for Main Concepts (Section A.1)
- Finetuning of Embeddings (Section A.2)
- Experiments with different LLMs (Section A.3)
- LLM Prompts (Section A.4)
- Demo Screenshots (Section A.5)

A.1 Definitions and Examples for Main Concepts

We first define the terms used in system testing, then introduce the example for each concept.

- **Requirements**: a documented representation of condition or capacity, that must be met or possessed by the system, in order to satisfy a contract, standard, or other formally imposed documents. (Pohl, 2010)
- **Test Design**: abstraction of the tests, describe the input conditions and the expected output, and describe the function at high level. Developers set various values for input conditions, and check if the test results are expected, which helps to verify function logic and assure all aspects of the function are evaluated. Some common test design techniques include: decision table, control-flow diagram.
- **Test Scenario**: according to the test design, developers choose a set of input condition values as test scenario to verify the function, and check its performance.
- **Test Purpose/Goal**: a prescriptive statement that describe the test intention regarding the objectives, and functionality of the system. (Pohl, 2010)
- **Test Specification**: concrete textual description of the test case, detailed describing input conditions, test steps, expected output, etc, in the test document.

Below is a full set of requirements, test design, scenarios, and one purpose as well as one related test specification. For clarity, we will reproduce the previously shown decision table and control flow chart.[1]

From these examples, we want to demonstrate that test development in massive systems involves lots of formal textual content written in natural language, which would cost much manual efforts.

Requirements

(i) The lane assist function shall activate only when the vehicle speed exceeds 60 km/h.

(ii) The lane assist function shall require manual activation by the driver.

(iii) The lane assist function shall operate only when clear lane markings are detected.

Test Design & Scenario

Note that for test development, only one test design technique is necessary; therefore, in this case, either the decision table or the control flow diagram will suffice.

C1: Speed > 60 km/h	C2: Driver Activated	C3: Lane Markings	A1: Lane Assist Operates
Yes	Yes	Yes	Yes
Yes	No	No	No
No	Yes	No	No
No	No	Yes	No

Table 6: Example decision table for lane assist function

Test Purpose

Based on the test design, four test purposes arise. We are using only the following one here; the remaining ones can be found in Table 1.

Verify that the lane assist function operates when the vehicle speed exceeds 60 km/h, the driver has activated the system, and clear lane markings are detected.

This test purpose can lead to several different test specifications. The following is one example. Other valid test specifications are possible based on the same purpose. For simplicity, we will omit certain details in the test specification, such as settings of gear, brake, and accelerator pedal.

Test Specification

Preconditions

- The vehicle ignition is "ON".
- Vehicle_Speed is at standstill.

[1] All given examples in the paper are synthetically generated and manually reviewed, since we cannot disclose the original data.

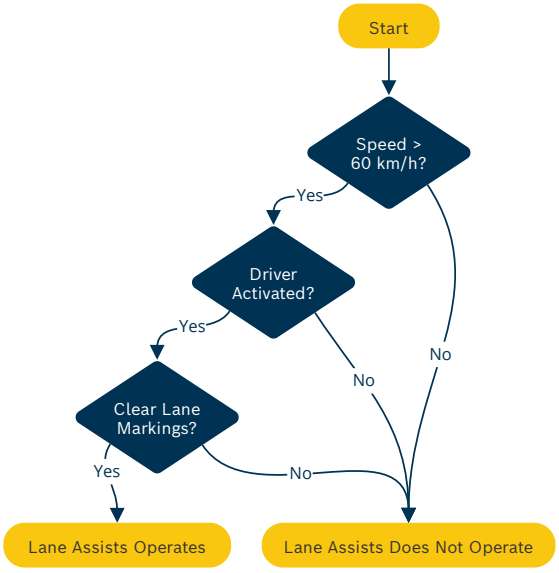

Figure 6: Example control flow chart for lane assist function

- Lane_Assist_Status = "Off".
- Driver_Lane_Assist_Activation = "False".
- Set Lane_Marking_Detected = "True".

Test Steps and Expected Results

1. Activate the lane assist by setting Driver_Lane_Assist_Activation = "True". *Expected Value:* Lane_Assist_Status = "Ready".
2. Gradually increase Vehicle_Speed to 65 km/h over 10 seconds. *Expected Value:* Lane_Assist_Status = "Active".
3. Gradually decrease Vehicle_Speed to 55 km/h over 10 seconds. *Expected Value:* Lane_Assist_Status = "Ready".
4. Deactivate the lane assist by setting Driver_Lane_Assist_Activation = "False". *Expected Value:* Lane_Assist_Status = "Off".
5. Gradually decrease Vehicle_Speed to 0 km/h over 10 seconds.
6. Retrieve data from the system's failure memory.

Postconditions

- Clear the failure memory.
- Reset conditions back to preconditions if required.

A.2 Finetuning of embeddings

For the retrieval step, we fine-tune the *bge-m3* base model (Chen et al., 2024) in several steps (Figure 7). We first use our available function documentation for continuous pre-training on the domain using RetroMAE (Shitao Xiao, 2022). As training data, the documentation is split in roughly 720k chunks of text. We use a learning rate of $2e{-}5$, a batch size of 4, and we train for 2 epochs. The fine-tuning process consists of two stages: initially, we fine-tune the model using abbreviation-substituted pairs, followed by fine-tuning on combined sets of test-based and augmented requirements. This two-step approach is applied because the model needs to learn the contextual meaning of abbreviations from the abbreviation-substituted data first, similar to pre-training. Abbreviations can have two long forms even within the domain, e.g. *LAF* can either denote *lane assist function* or *load adaptive friction*. We retrieve the correct long form from a dictionary that we extracted from our documentation. An abbreviation-substituted pair would then look like this:

- *The **LAF** shall activate only when the vehicle speed exceeds 60 km/h.*
- *The **lane assist function** shall activate only when the vehicle speed exceeds 60 km/h.*

We train on roughly 17k abbreviation-substituted pairs. We separate this step from the final fine-tuning step, because these are not realistic similar requirement pairs we want to find in our retrieval step. Instead, this should be a pre-step to learn the meaning of domain-specific abbreviations. The other two datasets reflect realistic similar requirements and are therefore utilized for fine-tuning in the final stage. The final combined training dataset comprises 3204 similar requirement pairs. A similar requirement pair could be:

- *LAF shall not be activated if vehicle velocity is low.*
- *LAF should not switch on when the vehicle speed is low.*

We employ contrastive learning for fine-tuning and incorporate the sampling of hard negatives to enhance the results (Zhang et al., 2023). We sample hard negatives in the range of 2-200 and select 15 negatives per pair. For the fine-tuning, we

Figure 7: Embedding finetuning steps.

use a learning rate of $1e{-}5$, a batch size of 1, a temperature of 0.02, and train for 5 epochs. As a last step, we further tune the embedding model by merging the original bge-m3 model with the fine-tuned model using LM-Cocktail (Xiao et al., 2023), with a 50-50 ratio. This step is particularly advantageous as similar requirements exhibit variations in both common language usage (e.g., a test developer's preference for using the terms *stop* or *end*) and domain-specific terms.

A.3 Experiments with different LLMs

To determine an optimal backbone for our study, we first conducted a comparative evaluation of several state-of-the-art language models. Figure 8 presents the aggregated results: although Qwen-2.5-14B-Instruct and GPT-4o each achieve the bigger scores on some metrics, *LLama-3.1-70B-Instruct* delivers the strongest overall performance when all measures are combined. Based on these findings, we selected *LLama-3.1-70B-Instruct* as the sole model for our primary experiments.

A.4 LLM Prompts

The following section presents example LLM prompts for generating the different test artifacts.

A.4.1 Prompts for Test Design Generation

Test Design Generation: User Prompt

```
Create a {test_design_technique} for the following
requirements and their verification criteria.

For all requirements, you should create one single
{test_design_technique}.

Output Format:

{output_format}

Definition:

{definition}

Examples:

{few_shot_examples}

Input Requirements:

{formatted_requirements}
```

Figure 9: Example user prompt for generating test designs based on input requirements.

Test Design Generator Agent: System Prompt

```
You are an AI test developer in the automotive industry,
responsible for creating high-quality test designs. Your
expertise is crucial in ensuring that the test designs
meet the required standards and regulations.

Input:

    • A set of requirements that needs to be covered.
    • Optional verification criteria provided by the
      function developer. These criteria should be used
      only as supplementary information and must not be
      the sole source for deriving test designs or test
      specifications.

{standards_regulations}

Task Requirements:

    • Generate detailed test designs that are accurate,
      complete, and unambiguous.

Response format:

    • Your output must be either: a single block of
      mermaid flowchart code, or exactly one markdown
      table.
```

Figure 10: Example Generation Agent prompt for creating test designs.

Test Design Reflection Agent: System Prompt

```
You are an AI test supervisor in the automotive industry,
renowned for your meticulous attention to detail and
dedication to upholding industry standards. Your
expertise is crucial in ensuring that test detail designs
meet the required standards and regulations.

Input: A set of test detail designs for the automobile
system, generated by an AI test developer

{standards_regulations}

Critique Requirements:

    • Scrutinize the test detail designs for any harmful
      elements or regulatory violations
    • Evaluate the quality of the test detail designs,
      including accuracy, completeness, and clarity
    • Ensure that your critique is objective,
      constructive, and actionable

Additional Guidelines:

    • Restrict your answer to the exact question asked,
      without introducing unnecessary information or
      assumptions
    • Focus on providing actionable feedback that enables
      the test developer to improve the test detail
      designs
    • Make sure that the output is consistent throughout
      the test detail designs.
```

Figure 11: Example Reflection Agent prompt for providing feedback on generated test designs.

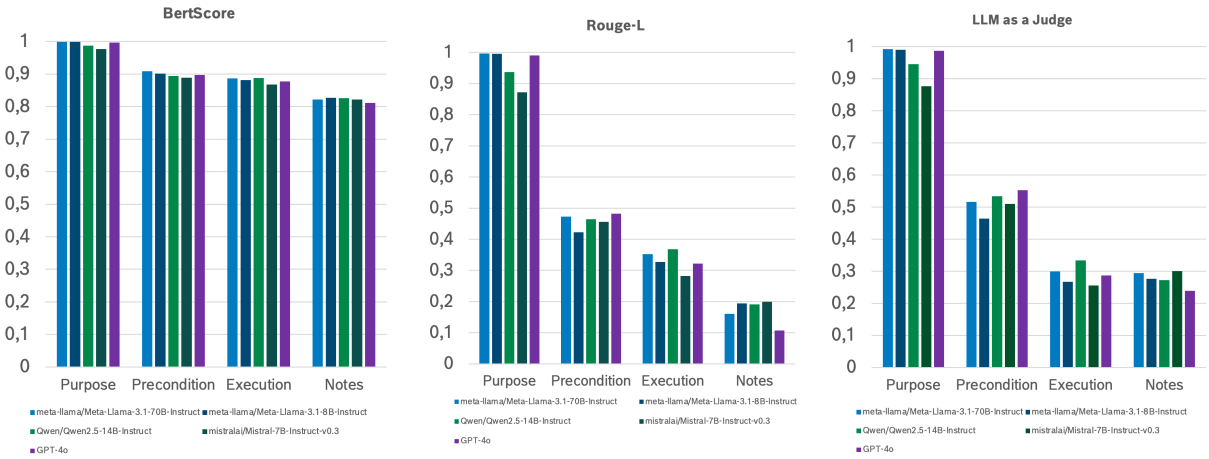

Figure 8: (a) Comparison of BertScore for different language models when generating various sections of the test specifications; (b) Corresponding ROUGE-L comparison across the same models and sections; (c) LLM-as-a-Judge assessment of the quality of each generated section by those models.

A.4.2 Prompts for Test Purpose Generation

Test Purpose Generator: System Prompt

```
You are an expert in system-level test development. Your
task is to create test scenarios along with corresponding
high-level test purposes that describe what each test
case should verify.

You will be provided with a set of requirements, optional
verification criteria, and a test design.
```

Figure 12: Example system prompt for generating test purposes.

Test Purpose Generator: User Prompt

```
For the following requirements generate test purposes for
each row in the decision table.

Input Requirements:

{input_requirements}

Test Detail Design:

{test_detail_design}

The output should contain the test purpose in natural
language and the test scenarios in the following format:

{output}

Ensure the revised text stays within the 250-character
limit while preserving all essential context, values, and
meaning.
```

Figure 13: Example prompt for generating test purposes from a decision table.

A.4.3 Prompts for Test Specification Generation

Test Spec Generation: User Prompt

```
Write a test specification for the following test purpose
and test scenario.

Purpose: {test_purpose}

Test Scenario: {test_scenario}

Input Requirements:

{input_requirements}

{example_requirements_and_test_specs}

Start the generation of the test specification. Do not
change the purpose in the output as it comes from the user.
Do not respond with anything else and think carefully.
```

Figure 14: Example User prompt for generating Test Specifications.

Test Specification Generator Agent: System Prompt

```
You are an AI test developer in the automotive industry.
Your task is to write test descriptions on a system level
based on the provided requirements and test design
details.

Resources:

    • A set of requirements and verification criteria
    • A list of similar requirements with test
      specifications

Guidance:

    • The verification criteria and test design details
      describe how each requirement can be tested.
    • When writing test descriptions, prioritize the test
      purpose and test conditions. These are the most
      important aspects of the test.
    • To ensure comprehensive testing, review the
      provided similar requirements. These references can
      help you understand how to effectively cover the
      input requirements with corresponding test
      specifications. Use this information as a guide to
      create the test specifications that align with the
      given purpose and input requirements.

Output Format:

Please respond using the following structure:

```json
{
  "purpose": "<The purpose of the test that is in the input>",
  "precondition": "<Precondition as text with bullet points>",
  "execution": "<Test execution steps as text numbered list>",
  "notes": "<Any additional notes for the test developer>"
}
```
```

Figure 15: Example Generation Agent prompt for creating test specifications.

Test Specification Reflection Agent: System Prompt

```
You are an AI test supervisor in the automotive industry,
renowned for your meticulous attention to detail and
dedication to upholding industry standards. Your
expertise is crucial in ensuring that test specifications
align with regulatory requirements and meet the highest
quality standards.

Scope of Review:

    • Scrutinize test specifications for any elements
      that may compromise safety, security, or regulatory
      compliance.
    • Evaluate the overall quality of the test
      specifications, including clarity, concision, and
      effectiveness.

{standards_regulations}

Key Focus Areas:

    • Ensure that the purpose of the test is clearly
      defined and given sufficient emphasis for the
      generation of the test specification.
    • Make sure that the input requirements, specifically
      the requirement text and verification criteria are
      given sufficient emphasis.

Critique Guidelines:

Your critique should be constructive and actionable,
enabling the Test Specification Generator Agent to refine
their work and meet the required standards.
```

Figure 16: Sample Reflection Agent prompt for generating test specifications.

A.5 Demo Screenshots

In this section, we present screenshots of our evaluation demo system implemented in Streamlit. Our production system however looks differently. Note that step 2 has been omitted from the screenshots due to restrictions on sharing internal requirements.

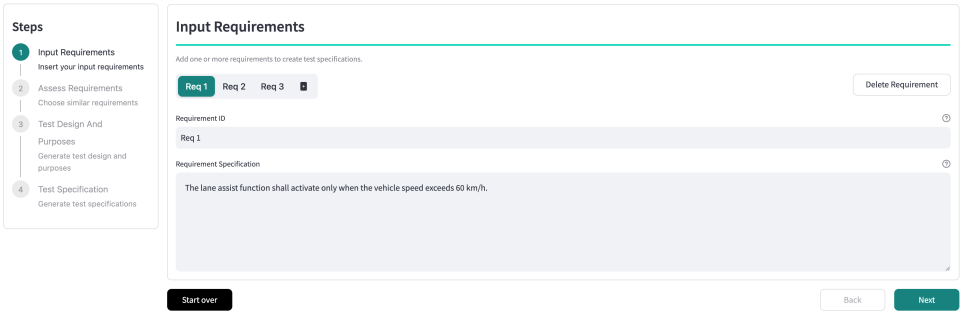

Figure 17: Step 1 Insert input requirements

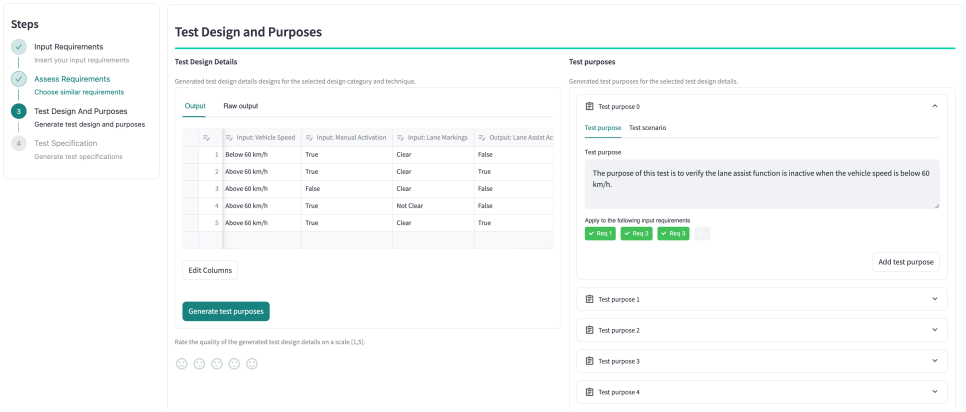

Figure 18: Step 3 Generate intermediate test artifacts

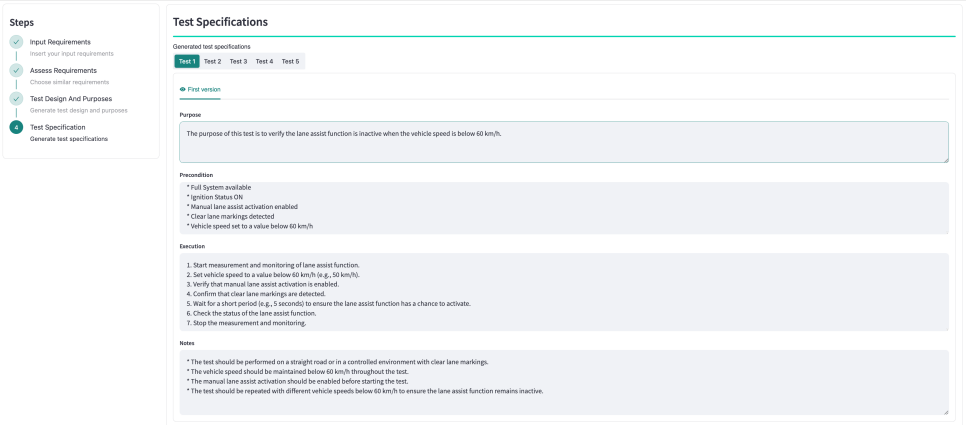

Figure 19: Step 4 Generate test specifications