



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

---

# **AutoComply: Automating Requirement Compliance in Automotive Integration Testing**

Master's thesis in Computer science and engineering

Divyam Sood Moritz Sprenger

---

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2024



MASTER'S THESIS 2024

# AutoComply: Automating Requirement Compliance in Automotive Integration Testing

Divyam Sood  
Moritz Sprenger



UNIVERSITY OF  
GOTHENBURG

---



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
UNIVERSITY OF GOTHENBURG  
Gothenburg, Sweden 2024

AutoComply: Automating Requirement Compliance in Automotive Integration  
Testing

Divyam Sood

Moritz Sprenger

© Divyam Sood, 2024.

© Moritz Sprenger, 2024.

Supervisor: Yinan Yu, Department of Computer Science and Engineering

Advisor: Dhasarathy Parthasarathy, Volvo Trucks

Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2024

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

SE-412 96 Gothenburg

Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X

Gothenburg, Sweden 2024

AutoComply: Automating Requirement Compliance in Automotive Integration Testing

Moritz Sprenger & Divyam Sood

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

## Abstract

This Master’s thesis explores the application of Large Language Models (LLMs) for automating the process of ensuring requirement compliance in automotive software integration testing. As the automotive industry increasingly incorporates intelligent technologies, the complexity of testing for safety and functionality has grown, making traditional manual compliance methods time-consuming and prone to errors. This study aims at addressing these challenges by leveraging the capabilities of LLMs to interpret and verify the compliance of test scripts with given natural language requirements. The research follows a structured approach, starting with thoroughly examining the current landscape in software development, testing practices, and the specific challenges faced in the automotive sector. It then delves into the theoretical underpinnings of LLMs, their application in software engineering tasks, and the potential for automating compliance checks. Through a methodical process involving dataset construction, perturbation, and evaluation, this study assesses the performance of different LLM-based approaches to requirement compliance.

The results indicate that the available open-source models are not yet suited to fully solve a domain-specific task requiring strong reasoning over long contexts. However, they can potentially be employed as assistants, helping developers by providing initial compliance suggestions. Additionally, the results show that these systems are generally sensitive to small changes in the prompt and show different behavior for distinct input perturbations. In contrast to the literature, our experiments do not show improvements when using in-context learning or including external knowledge, stressing that these techniques are not always beneficial in specific domains. Further experiments with synthetic data reveal that the length of the test scripts is an influential factor for the system performance as the performance degrades considerably, even for extremely simplified test sequences and requirements, when noise in form of unrelated test code is introduced. Building on the prompting techniques, more agent-like LLM-based systems, that produce a compliance decision in multiple steps, are explored. The results for the agent systems do generally not improve upon the simpler prompting techniques and, although showing some promising research avenues, suffer from accumulating errors in the intermediate steps.

Keywords: Automotive Software Testing, Requirement Compliance, Large Language Models (LLMs), Perturbations, Natural Language Processing (NLP), Automotive Safety Standards, Generative AI, Classification, Deep Learning.



# Acknowledgements

We want to express our deepest gratitude to all those who have supported us throughout the completion of this thesis. The journey has been enriching and informative, and we could only have navigated it with the invaluable guidance and assistance of many.

Firstly, we are immensely grateful to Dhasarathy Parthasarathy, our supervisor at Volvo Group, for his expert guidance, encouragement, and insights throughout our research. His expertise and thorough understanding of the automotive sector, deep learning, and software engineering have been instrumental in shaping our approach and methodology.

We also extend our heartfelt thanks to Yinan Yu, our academic supervisor at Chalmers University of Technology. Her profound knowledge and academic rigour have greatly enhanced the quality of our work. Her constructive feedback and unwavering support have been crucial in refining our thesis.

Special thanks go to Carl-Johan Seger, our examiner, for his critical insights and rigorous evaluation of our work. His constant feedback has been vital in pushing our thesis to its highest standard.

We thank Daniel Karlsson, our manager at Volvo Group, for allowing us to engage in this project. His support has been fundamental to this thesis.

We are also thankful to Anton, who provided the initial problem statement and offered continued support and guidance throughout the research process. His practical insights into the problem area were invaluable.

Furthermore, we would like to acknowledge our peers and colleagues at the Volvo Group and the Department of Computer Science and Engineering for their collaborative spirit and helpful discussions. We are also grateful to the administrative and technical staff at Volvo Group for helping us set up the proper setup to pursue this thesis.

Lastly, we thank our families and friends, whose endless encouragement and support have sustained us through this intense academic endeavour.

Special thanks goes to my (Moritz) girlfriend Maria, whose support and encouragement helped me stay motivated and positive throughout the thesis. Your light always illuminates a path forward.

Divyam Sood, Moritz Sprenger  
Gothenburg, 2024-05-29



# Contents

<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Related Work . . . . .	2
1.3 Problem definition . . . . .	3
1.4 Research question . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 Software Development & Testing . . . . .	5
2.1.1 Ideation . . . . .	5
2.1.2 Requirement Gathering Analysis . . . . .	5
2.1.3 Design . . . . .	6
2.1.4 Implementation . . . . .	6
2.2 Requirement engineering . . . . .	6
2.3 Automotive context . . . . .	7
2.3.1 Software Design . . . . .	8
2.3.2 Tools . . . . .	8
<b>3 Theory</b>	<b>11</b>
3.1 Large Language Models . . . . .	11
3.1.1 Models . . . . .	12
3.1.1.1 CodeLlama34 . . . . .	12
3.1.1.2 Mixtral 8x7B . . . . .	14
3.1.2 Prompting . . . . .	15
3.1.3 RAG . . . . .	16
3.1.4 Self-Refine . . . . .	18
3.1.5 Self-Consistency . . . . .	18
3.2 Language agents . . . . .	19
3.2.1 Memory . . . . .	19
3.2.2 Actions . . . . .	20
3.2.3 Decision procedure . . . . .	22
3.3 Model explainability . . . . .	22

<b>4</b>	<b>Methods</b>	<b>25</b>
4.1	System overview . . . . .	25
4.2	Data . . . . .	25
4.2.1	Dataset . . . . .	25
4.2.2	Data preprocessing . . . . .	27
4.2.3	Data perturbations . . . . .	28
4.2.4	Data generation . . . . .	31
4.2.5	Dataset construction . . . . .	33
4.3	Evaluation procedure . . . . .	34
4.4	Prompting techniques . . . . .	36
4.5	Agent approaches . . . . .	39
4.5.1	Function evaluation . . . . .	39
4.5.2	Requirement reformulation . . . . .	40
4.5.3	Self-Consistency . . . . .	40
4.6	Baseline . . . . .	41
4.7	Failed Approaches . . . . .	41
<b>5</b>	<b>Results</b>	<b>43</b>
5.1	Experimental setup . . . . .	43
5.2	Prompting techniques . . . . .	44
5.3	Model understanding . . . . .	46
5.4	Requirement quality . . . . .	48
5.5	Synthetic data . . . . .	48
5.6	Function evaluation . . . . .	52
5.7	Self-Consistency . . . . .	53
5.8	Natural language explanations . . . . .	54
<b>6</b>	<b>Conclusion</b>	<b>57</b>
6.1	Discussion . . . . .	57
6.2	Summary . . . . .	58
6.3	Limitations . . . . .	59
6.4	Contribution . . . . .	60
6.5	Future research . . . . .	60
	<b>Bibliography</b>	<b>63</b>
<b>A</b>	<b>Appendix</b>	<b>I</b>
A.1	Prompts . . . . .	I

# List of Figures

3.1	The Code Llama specialization pipeline . . . . .	13
3.2	Mixture of Experts Layer . . . . .	14
3.3	Overview of the Cognitive architectures for language agents framework	19
3.4	Action space of cognitive language agents . . . . .	21
4.1	Overview of the dataset construction process. . . . .	33
4.2	Overview of the counterfactual-based evaluation scheme. . . . .	35
4.3	Overview of the performance evaluation scheme. . . . .	35
4.4	Overview of the label flip evaluation scheme. . . . .	36



# List of Tables

5.1	Prompting technique performance results . . . . .	44
5.2	Impact of input perturbations on system classification . . . . .	46
5.3	Impact of requirement data quality . . . . .	49
5.4	Results for synthetic data experiments . . . . .	49
5.5	Impact of input perturbations on system classification for synthetic data	51
5.6	Results comparing the function evaluation agent with simple prompt- ing techniques . . . . .	53
5.7	Results for the Mixtral_Zero-Shot system using self-consistency and stratified by the minimum agreement for the compliance decision. . .	53



# 1

## Introduction

### 1.1 Motivation

Within the domain of automotive engineering, testing plays a vital role in ensuring the integrity, safety, and performance of vehicles. The automotive domain, characterized by complex systems, evolving technologies, and safety requirements, necessitates comprehensive testing procedures to enable optimal functionality [1].

The trajectory of automotive development witnesses a fundamental shift towards intelligent technologies, advanced driver-assistance systems, and the emergence of autonomous vehicles. Consequently, the complexity of testing has increased considerably. Validation of individual functions and integration of these functions into the vehicle architecture becomes essential to ensure a safe and functional system [1].

Moreover, the consequences of inadequate testing within the automotive sector extend beyond functional deficits. They encompass potential safety hazards, economic implications, and decrease in consumer trust; this is further underscored by guidelines like ISO 26262 [2] and Automotive SPICE [3] requiring rigorous testing processes. System failures or oversights in identifying flaws can result in substantial recalls, negatively impacting brand reputation and posing risks to drivers, passengers, and road users.

When developing new features, parts, or capabilities in automotive engineering, their correct and safe functionality has to be ensured. This can mean that engineers formulate functional requirements on a Controller Area Network (CAN) bus signal processing level in natural language. Software-based tests must meet these requirements to ensure correct functionality. Since the number of tests is enormous, the test responsible and requirement author are usually different persons, and the test scripts cover a collection of different scenarios, determining if existing tests cover the requirements is time-consuming and expensive. This determination is necessary to decide whether additional tests are needed to cover the requirement and ensure system functionality.

Large language models (LLMs) have shown impressive performance in language understanding and generation tasks, in general [4]. Additionally, they have been successfully applied to software engineering tasks such as code generation [5]. We aim to use an LLM to decide if a requirement formulated in natural language is covered by existing test code.

### 1.2 Related Work

The proposed project is on the intersection between requirements engineering, software testing, and natural language processing (NLP).

The literature on requirement engineering is characterized by the trade-off between the accessibility of natural language requirements and the ease-of-use of structured representations of requirements [6]. Natural language requirements make it easy for non-technical users, designers, and engineers to formulate requirements quickly; however, due to the ambiguous and context-sensitive nature of natural language, they are challenging to use in downstream tasks [7]. Some earlier works manually translated requirements from natural language to formal language and used these to automatically determine the requirement coverage of test code [8]. Other works try to parse natural language requirements and iteratively refine them to generate formal representations that can be used in downstream tasks; however, manual intervention was still required in this process [6].

In recent years, model-based software engineering has decreased the manual labor of deriving test cases from textual requirements. Here, requirements are formulated directly or translated to a systems modeling language supported by graphical tools, and the resulting model can be used to generate test cases or perform other analysis tasks [7]. Some works in test suite reduction, where the goal is to reduce the size of the test code, also consider requirement coverage as an optimization criterion. Marchetto et al. [9] establish a connection between textual requirements and code artifacts using lexical similarity techniques. However, in their approach, they have access to the application, test codes, and use structures in object-oriented programming languages.

The difficulty in processing natural language is a common denominator in all the described approaches. In recent years, LLMs have transformed the natural language processing (NLP) field, achieving state-of-the-art results on multiple benchmarks and tasks like machine translation, text summarization, and question answering [4]. They have also gained public attention as conversational agents, with ChatGPT [10] being the most famous example. In addition to processing natural language, a growing research field is applying LLMs to software engineering tasks, such as code generation and code completion [5]. These models are either based on existing LLMs and fine-tuned on code repositories or include code repositories in their initial training phase. A majority of these works make use of closed-source very large LLMs such as GPT-4 [11], but some authors show that smaller open-source models, such as Code Llama [12] can also be effective at programming tasks.

Many of these models have also been applied to software testing. In software testing, the most common uses of LLMs are test generation, test oracle generation, where the desired output is generated for a test, or test input generation [13]. However, code is mainly used as the model input for these tasks. An example that uses natural language inputs is the work from Fakhoury et al. [14]. They explore the usage of natural language issue descriptions to fix bugs in source code.

Another work by Fischbach et al. proposes a tool called CiRA (Causality detection in Requirement Artifacts) [15], automatically extracting the causal relationships

from natural language requirements, which will solve two problems: the manual labor required by practitioners to extract these relations in order to determine the combinations of cause and effect that need to be covered by test cases. The second is automatic dependency detection between requirements. As modern systems become more complex, the requirements and their relations constantly increase. Practitioners need to keep an overview of the relationships between the requirements. This may lead to undetected redundancies and inconsistencies within the requirements and, consequently, to faults in the system design [7]. An automatic causality extraction from requirements can help compare semantics by analyzing the different embedded causal relations. As a result, relations between requirements can be identified automatically (e.g., contradictory and redundant requirements).

Although there exists much work regarding the application of LLMs to testing, most of the literature is focused on generating or altering code, and there is a lack of work regarding early testing life-cycle processes like test requirements or the analysis of existing code going further than creating comments or natural language explanations [13]. An example of work in code analysis is Tufano et al. [16], who predict the code coverage of test cases given both the test code and the application code.

To the best of our knowledge, no prior work has explored the compliance of natural language requirements within the existing test code, given the textual requirements formulated in natural language using an LLM. However, this is a specific use case of applying LLMs to software engineering tasks that benefit from insights gained with similar outlined problems.

### 1.3 Problem definition

The problem can generally be described as determining if an existing test case code complies with a requirement given in natural language. Here, we define that a test covers/is in compliance with a requirement when the test script tests the necessary system functionality to ensure that the tested system fulfils the requirement. In this project, the requirements refer to signals sent over the vehicle’s CAN bus, which encode the state of vehicle components and functions. The requirement usually states the desired behaviour, dependent on the component’s state or changes in state. The following statement shows an example of such a requirement:

If *Signalname* goes missing, then the previous valid value (Active or Deactive) shall be assumed. If no previous value exists, then the value Deactive shall be assumed.

Given this formulated requirement and a collection of test scripts, the task is to automatically decide if the requirement is covered by a test included in the test scripts.

In this work, the decision is solely based on the text representation of the tests and requirements, and no access to the application code, tested system or execution environment is available. This restricts the available information and feedback for the

system, making the problem more challenging to solve. However, a solely text-based decision represents a fast and resource-efficient approach since executing test cases takes additional time and can also require a specific execution environment, which can be expensive to provide since it needs to model the tested system.

### 1.4 Research question

The overarching goal of this project is to investigate the application of Large Language Models (LLMs) in determining the compliance of a set of test scripts with natural language requirements in the automotive domain. The primary focus is exploring methods that leverage LLMs without access to the application code. The primary research question guiding this exploration is the following.

*How can LLMs be used to determine if a test script complies with a requirement given in natural language?*

Since the proposed question is rather broad, we will split it up into 2 distinct subquestions:

1. *How can the performance of an LLM-based system in the context of requirement compliance be evaluated regarding performance and model understanding?*
2. *How do different LLM-based approaches perform according to objectives determined in (1)?*

Understanding the performance of models is essential for assessing their usefulness in achieving the task. Additionally, investigating possible limitations is essential in developing an understanding of model behavior and can be a requirement for the adoption of the model by users and other stakeholders. Thus, subquestion (1) seeks to identify and implement a suitable evaluation scheme that considers the model's performance and gives further insights into the limitations of the model.

To comprehensively address (2), the project will explore various LLM-based approaches and evaluate their effectiveness using the evaluation scheme identified in (1).

# 2

## Background

### 2.1 Software Development & Testing

Software testing refers to verifying whether the software meets the technical requirements as guided by its design and development [17]. It involves efficient and effective testing that checks whether the software meets user requirements by handling all exceptional and edge cases. Software testing aims to identify errors, gaps, or anything missing in the system compared to the actual requirements. Testing is an integral part of the Software Development Life Cycle (SDLC), a systematic approach for creating and deploying any software [18]. The SDLC includes a series of steps: ideation, project initiation, requirement gathering analysis, design, implementation, development, testing, deployment, and maintenance. Although these steps are not strictly sequential, different levels of iterations occur between them [19].

#### 2.1.1 Ideation

The ideation phase includes analyzing and brainstorming various ideas to find a solution to a specific problem. These ideas may originate from users who articulate their challenges or stakeholders and developers who seek solutions to existing problems or aim to develop a new product.

#### 2.1.2 Requirement Gathering Analysis

This phase involves communication among various stakeholders and the accumulation and documentation of requirements from both the design and end-user perspectives [20]. Requirements are broadly categorized into two parts: functional and non-functional requirements [21]. Functional requirements describe what the software and associated features should do, focusing more on user requirements. Non-functional requirements describe how the software should work, define the corresponding properties, and focus more on user expectations. While they are desirable, non-functional requirements are not mandatory.

After gathering the above, the software requirement specification (SRS) document [22] is obtained, which is critical for the next steps.

### 2.1.3 Design

This phase uses the SRS document produced in the previous step as input. It involves identifying the testing techniques, test scenarios, test cases, test data, and expected test results. At the end of this stage, the testing team should have an outlook of the test cases needed to provide appropriate coverage for the software.

### 2.1.4 Implementation

After completing the design phase, developers write test cases planned for the design phase. This phase aims to take the low-level design created from the previous phase and convert it into the programming language of choice. At the end of this stage, the testing team should have a set of comprehensive and accurate test cases that provide adequate coverage of the software or application.

## 2.2 Requirement engineering

Requirement Engineering is a systematic and disciplined approach to discovering, documenting, and maintaining the requirements of a software system [23]. It is a critical process that aims to ensure that the system meets the needs of its stakeholders. The process involves various activities such as requirements elicitation, analysis, validation, and management.

The first step in requirement engineering is to gather the requirements from stakeholders. Stakeholders can be end-users, customers, managers, developers, or any other party involved in the development and implementation of the system. The gathered requirements are then analyzed to ensure they are complete, consistent, and unambiguous [24]. This involves identifying any conflicts or ambiguities in the requirements and resolving them. Once the requirements are analyzed, they are validated to ensure they are accurate, feasible, and achievable. This involves confirming that the requirements meet all stakeholders' needs and comply with relevant regulations or standards. During the development process, the requirements are managed to ensure that changes to the requirements are tracked and controlled. This helps to ensure that the system is developed in a controlled and organized manner. Since this process of generating requirements is iterative and oscillates between the development and the testing phase [25], there are many people involved in the process, such as the customer, the sponsor, the users with different areas of expertise, the development and the testing team and those responsible for system architecture. This can result in the requirements being written in text form, making them relatively informal—the ambiguity and diversity of human language causing this informal nature.

In order to handle the growing complexity and variability of software systems, model-based design, and testing methods have been developed [26]. Especially for safety-critical systems such as those in the automotive, railway, and aerospace domains, extensive testing based on the requirements is necessary. However, the manual creation of requirements models from natural language requirements is time-consuming, error-prone, and requires extensive expert knowledge. This is especially

true in agile software development, which involves continuous improvements and many changes to requirements. To assess the quality of textual requirements, different criteria have been identified in the literature [27]–[29]. Génova et al. [30] give a condensed overview of these criteria. The criteria can be split into two groups, criteria that measure the quality of the requirement in how it helps to define the product and criteria that judge the usability of the requirement. Examples for the first group include *validability*, which states that the requirements effectively describe the system that fulfill the users' needs and *modifiability*, which states how adaptable requirements are. Criteria from the second group include, *verifiability*, which states that the engineer must be able to check (verify) that the produced system meets the specified system, *understandability*, which states that the requirement is correctly understood without difficulty, *unambiguity*, which states that there exists only one interpretation for the requirement, and *precision*, which states that all used terms are concrete and well defined [30].

## 2.3 Automotive context

The automotive industry is undergoing a significant transformation, incorporating advanced features and functions into automobiles to meet the ever-growing market demand. As a result, the development of automotive embedded systems has changed from an electrical and mechanical engineering discipline to a combination of software and electrical/mechanical engineering [31]. This transition has led to a paradigm shift in development processes, methods, tools, and the required engineering skills. The effects of this shift have been particularly significant in software testing, which plays a critical role in ensuring the safety and reliability of automotive software. Advanced testing equipment and test generation techniques are required to test the complex software running on hundreds of interconnected electronic control units (ECUs) distributed throughout the vehicle [32].

In terms of the software testing lifecycle in the automotive industry, the process involves several testing stages to ensure the software's safety and reliability. Several SDLC methodologies, including Waterfall, V-model, and Agile, are utilized in software development, each with its advantages and disadvantages [25], [33]. The Waterfall model follows a sequential development approach, facilitating timely completion of each phase but lacking flexibility for changes in earlier stages. Conversely, the V-model, a modified version of Waterfall, allows for iterative testing and the involvement of testers during the requirement phase. Agile methodology, characterized by its rapid and adaptive approach to development, is also prevalent in the automotive industry. Agile emphasizes collaboration, flexibility, and incremental delivery of software features to meet evolving customer needs.

The testing process starts with developing test cases based on the system requirements and specifications. It then proceeds to Unit Testing, where individual software components are tested to ensure they work as expected. It is followed by Integration Testing, where the different software components are integrated and tested.

System Testing is the next stage, where the entire system is tested for functionality

and performance. Finally, the software is tested in the natural environment, known as Acceptance Testing. In addition to these testing stages, automotive software testing involves various types of testing, such as functional, performance, security, and regression [34].

In addition to regular software testing paradigms, adhering to high quality and safety standards is a challenge within the automotive industry, as software defects can cause malfunctions, failures, or accidents. Moreover, software testing must consider the high diversity and variability of software products, user preferences, driving behaviors, and scenarios. Additionally, software testing must consider software systems' high complexity and interdependence, as they involve multiple layers, components, and interfaces that communicate and coordinate. Furthermore, interactions and dependencies between software and hardware must be considered between software and external factors like traffic, weather, or network conditions [35].

### 2.3.1 Software Design

AUTOSAR (AUTomotive Open System ARchitecture) is the basis for developing new modules inside automotive systems [36]. Before the adoption of this system, a significant problem in the automotive software design was poor re-usability within the development of software and hardware modules. This architecture supports the development of standardized electronic systems that improve quality, performance, safety, and environmental friendliness [37]. It provides a standardized architecture enabling developers to create reusable software modules that can be easily integrated into different automotive systems. This standardization also facilitated the development of interoperable systems, allowing for more accessible communication and cooperation between components and suppliers.

### 2.3.2 Tools

Within the Original Equipment Manufacturer (OEM) domain, specialized tools such as CANoe and Simulink/MATLAB are used for software testing during different stages. These tools enable the efficient and practical testing of complex automotive software, ensuring that they function as intended and meet the industry's stringent safety and reliability requirements. These tools are integral to integration testing as they enable the simulating and corresponding testing of vehicle behaviors based on different conditions and changes made within the system. And, when a new feature is to be added, an older one is to be removed, or both, the system as a whole needs to be tested to maintain the requirement coverage. These tests are written using tools such as CANoe, and the simulation environment is created within Simulink/MATLAB [38].

## CANoe

The Controller Area Network (CAN) protocol, a widely adopted communication protocol, enables different electronic devices in the automotive industry to communicate with each other. CAN connects different ECUs in a vehicle and facilitates the exchange of information and control signals [39].

The CANoe is widely used for developing, testing, and analyzing entire ECU networks and individual ECUs [40]. It supports network designers, development, and test engineers at OEMs and suppliers [41] on the whole development process – from planning to the startup of entire distributed systems or individual ECUs. At the beginning of the development process, CANoe is used to create simulation models that simulate the behavior of ECUs. These models are the basis for analyzing, testing, and integrating bus systems and ECUs throughout ECU development. This makes it possible to detect problems early and correct them. Graphic and text-based analysis windows are provided to evaluate the results. CANoe contains the Test Feature Set for easy and automated test execution. It is used to model and execute sequential test sequences and automatically generate a test report. The Diagnostic Feature Set is also available within CANoe for diagnostic communications with the ECU [41]



# 3

## Theory

In the following section, Large Language Models (LLMs) are introduced and fundamental aspects of their training and usage are discussed, before language agents are introduced. The chapter ends with a description of model explainability.

### 3.1 Large Language Models

In this work, we follow the definition of Zhao et al. [42] of LLMs, which states that LLMs are Transformer-based [43] language models with billions of parameters, which are trained on massive text data [44]. Examples for LLMs are GPT-4 [45], LLama2 [46], GPT-3 [47] or Flan-T5 [48]. These models exhibit strong natural language understanding abilities, usually pre-trained with an auto-regressive left-to-right language modeling task. Given a sequence of tokens  $\mathbf{x} = \{x_1, \dots, x_n\}$  the LLM is tasked to predict the next token  $x_i$  conditioned on the previous tokens  $x_{<i}$  and optimize the objective given in Equation 3.1.:

$$\mathcal{L}_{LM}(\mathbf{x}) = \sum_{i=1}^n \log P(x_i | \mathbf{x}_{x<i}) \quad (3.1)$$

Due to the computational complexity of calculating this probability, the considered tokens are usually restricted to a specific maximum count of considered tokens, the context length. Using this training objective enables the usage of massive amounts of unlabeled texts, usually crawled from internet sources, and is referred to as self-supervised pre-training. Standard corpora used in this step are books (BookCorpus [49]), web sources (CommonCrawl [50], Wikipedia [51]) or open-source licensed code repositories (e.g. GitHub) or code-related discussion forums (e.g. StackOverflow). Before being used in the pre-training step, these resources are usually pre-processed to increase data quality.

After pre-training, LLMs can further be trained to increase their abilities in following instructions to solve tasks and align their answers with human expectations. Instruction tuning denotes fine-tuning a pre-trained LLM with a supervised learning task, where the model is presented with a task description (*instruction*) and optionally some additional input and examples. The model is then expected to produce the associated ground truth output and trained with the loss given by the produced and

expected output [52]. Models, additionally trained in this fashion, tend to be able to adapt to task descriptions not seen during this process [48].

The third step in the training process of LLMs is alignment tuning. This step aims at aligning the model outputs with human preferences regarding criteria such as helpfulness, honesty, or harmlessness. Alignment tuning is usually conducted by gathering a dataset of human feedback, where human annotators rank a selection of model-generated responses [53], or provide responses themselves [54], adhering to previously outlined criteria. Then, a reinforcement learning-based approach is used to align the model with the given human preferences [55], [56].

When using LLMs, tokens are generated iteratively, one at a time. For each token, the LLM outputs a probability distribution of all the possible output tokens. There are multiple different parameters governing how the distribution is generated and used.

The temperature is a parameter that controls the entropy of the distribution, with high values of temperature moving the distribution towards a uniform distribution and low values of the temperature accentuating high probability outcomes [57].

Decoding is the process of choosing which token to choose from this probability distribution. Multiple different decoding strategies effect the probabilistic behavior of LLMs. The simplest decoding strategy is greedy decoding, where the token with the highest probability is selected at each step. This enables a deterministic behavior of an LLM given the same input and a fixed seed. However, due to the low search depth, greedy decoding can miss more probable sequences of tokens. Beam search addresses the issue of greedy decoding by keeping track of the current best  $n$  token sequences at each generation step according to their probability. It selects the final generation sequence with the highest probability [58]. Due to this method being biased towards shorter sequences, a length penalty is introduced to normalize the sequence probability regarding its length [59].

#### 3.1.1 Models

This work focuses on two prominent language model architectures: CodeLlama34 [12] and Mixtral8x7B [60]. Initially, our investigation began with CodeLlama34 due to its reported ability to comprehend code better than its predecessor, Llama2 [46]. Subsequently, we explored various models and found Mixtral8x7B to be particularly noteworthy. Below, we provide a concise overview of each architecture based on publicly available information.

##### 3.1.1.1 CodeLlama34

CodeLlama34 comes from a family of large language models released by Meta. Along with this model, Meta also released many other differently-sized and specialized models. In terms of sizes, there were 7B, 13B, 34B, and 70B, and in terms of their specializations, the instruction-tuned models that were used for chatting and question-answering capabilities, and Python specialization models, which are called Code Llama – Python. Each type has 7B, 13B, and 34B params. Meta has followed a gradual approach to specialize and increase the capabilities of Llama2 models,

ranging from infilling to extended input contexts and instruction finetuning. Different variations and combinations of these three approaches have created Code Llama, Code Llama-Python, and Code Llama-Instruct. Roziere et al. [12] explains these combinations using the Figure 3.1.

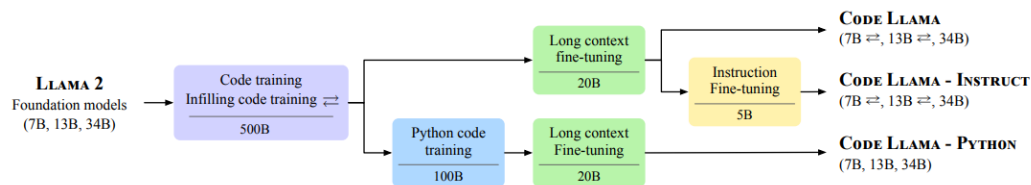


Figure 3.1: The Code Llama specialization pipeline.

The basic recipe for training includes code training and infilling code training, where the base model Llama 2 is finetuned on a code dataset of 500B tokens. The dataset comprised 85% code, another 8% of natural language related to code, and the last 7% are natural language to help the model keep its natural language understanding skills. For the Python-specific model, an additional dataset of 100B tokens consisting of 75% Python code, 10% of another code, and 10% natural language was used for finetuning. For the next step, to make these models support context length longer than the native context length of LLaMA 2 (4096), the models are finetuned on 16k length sequences. Then, they can be used via extrapolation for sequence lengths up to 100k tokens with some degradation in the output quality. This longer context length would allow for repository-level reasoning, which is often impossible to fit within the context of 4096.

Instruction finetuning is used for further training on Code Llama - Instruct. An example of such a data point used as shown in [12] is:

”” **Prompt:**

In Bash, how do I list all text files in the current directory (excluding subdirectories) that have been modified in the last month

**Response:**

You can use the 'find' command in Bash to list all text files in the current directory (excluding subdirectories) that have been modified in the last month. The command would be:

```
find . -maxdepth 1 -type f -mtime -30 -name "*.txt"
```

Here's a breakdown of the command:

- `.` specifies the current directory.
- `-maxdepth 1` tells 'find' to only search the current directory and not any subdirectories.
- `-type f` tells 'find' to only search for files (not directories or other types of files).

- `-mtime -30` tells 'find' to only search for files that have been modified in the last 30 days.
- `-name "*.txt"` tells 'find' to only search for files with the '.txt' extension.

Note that the `-mtime` option uses the number of days since the file was last modified, so `-mtime -30` means "modified in the last 30 days." If you want to search for files modified in the last month, use the `-mtime -3030` option. ””

Phind has done further fine-tuning of this instruction-tuned model [61], and we choose to go with this fine-tuned version because of its better positioning on relevant evaluation benchmarks. The model is named Phind-CodeLlama-34B-v2.

### 3.1.1.2 Mixtral 8x7B

Mixtral 8x7B [60] is a sparse mixture of expert language models with the same architecture as Mistral 7B [62], with the difference that each layer is composed of 8 feedforward blocks termed experts. Jiang et al. claim to have the model perform better or equivalent to bigger models like Llama2 70B and GPT-3.5 on most evaluation benchmarks, e.g., it performs better than Llama2 70B on tasks that require multilingual understanding, code generational capabilities, and mathematical abilities. The architecture allows only a certain number of parameters during inference for every token, which allows faster inference speeds [60]. The Mixture Of Experts (MoE) block, it introduces, contains multiple experts (networks/Feed Forward Network (FFN)) preceded by a gating function that decides which expert an incoming token must be routed to, as shown in Figure 3.2. Therefore, each token has access to 47B parameters. However, only 13B parameters are used during actual inference [60].

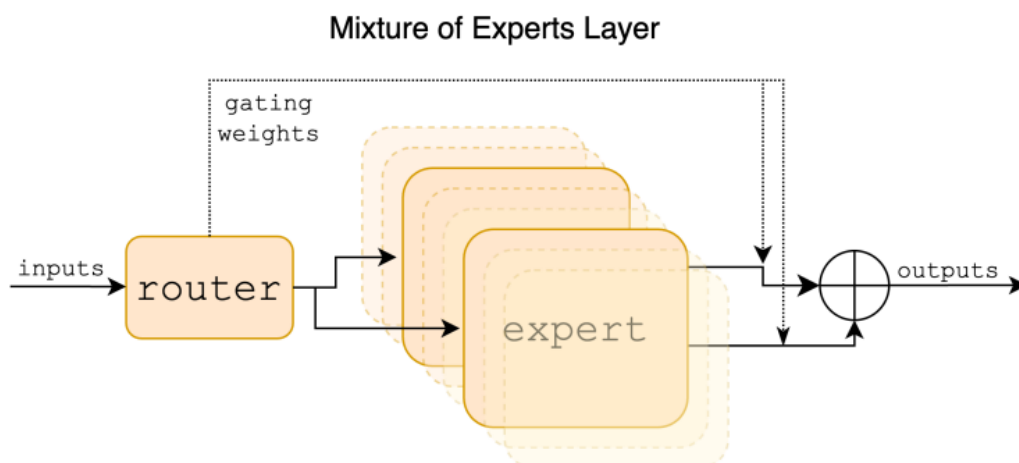


Figure 3.2: Mixture of Experts Layer [60].

MoE modules for a given input  $x$  are determined by the weighted sum of the output

of the expert networks, where the gating network gives the weights. i.e., given  $n$  experts  $E_0, E_1, \dots, E_{n-1}$ , the output of the expert layer is given by:

$$\sum_{i=0}^{n-1} G(x)_i \cdot E_i(x)$$

Compared with the available state-of-the-art LLM Llama 2 70B, Mixtral outperformed or matched it on all the evaluation benchmarks except the reading comprehension benchmarks, using 5x fewer active parameters during inference.

### 3.1.2 Prompting

Since LLMs are designed to generate words conditioned on previous words, the primary way of interacting with them is by providing a seed text that is to be completed by the LLM, a so-called (prefix) prompt [63]. Prompts can take many forms, like questions, instructions, or unfinished text fragments. Due to the importance of prompting for the performance of LLMs, there is an extensive body of work outlining how prompts elicit the desired behavior of an LLM [63]. This section introduces a selection of prompting techniques and terminology.

Prompt engineering denotes creating prompts for an LLM to achieve a specified task [64]. Although prompts can have a plethora of different forms, four main elements differentiate prompts: The *task description*, which typically contains the task and specific instructions the LLM is supposed to follow; the *input data*, which is the instance data necessary to achieve the task, *contextual information*, which is task and input data specific context that can help the LLM in completing the task and the *prompt style*, which describes how the other elements are adapted for different LLMs as well as formulated to elicit the desired behavior [42].

Some empirically discovered design principles can help engineer effective prompts [42], [65]. An example of such a principle is the unambiguous formulation of task descriptions. Additionally, utilizing model-specific formats or giving a role description, like "*You are an expert code-reviewer*", can increase the performance of these models [42], [65].

#### In-context learning

First proposed by Brown et al. [47], in-context learning is a paradigm that utilizes the observation that LLMs can generalize better to unknown tasks by presenting a few examples as demonstrations. Critically, in-context learning requires no update to the model parameters, which can be prohibitively expensive, but the demonstrations are given as part of the prompt.

Depending on the number of given demonstrations, in-context learning differentiates between zero, one, and few-shot prompts. In a zero-shot setting, no examples are presented; just the task description and optional input data and contextual information are provided to the model. For one- and few-shot approaches, the contextual information includes demonstrations of the required tasks with task descriptions and the expected answers [66]. Following are example prompts for zero, one, and few-shot learning techniques for the given task description:

Task Description: "Translate the following sentence into Swedish: 'Hello, how are you?'"

- **Zero-shot Learning:**

- Prompt: "Translate the given English sentence into Swedish. Sentence: 'Hello, how are you?' Translation: "

- **One-shot Learning:**

- Prompt: 'Translate the given English sentence into Swedish'. Sentence: 'How is your day going?' Translation: 'Hur går din dag?' Sentence: 'Hello, how are you?' Translation: "

- **Few-shot Learning:**

- Prompt: 'Translate the given English sentence into Swedish'. Sentence: 'How is your day going?' Translation: 'Hur går din dag?' Sentence: 'Where do you live?' Translation: 'Var bor du?' Sentence: 'Hello, how are you?' Translation: "

For one- and few-shot approaches, the selection of demonstrations and their structure influences the effectiveness. Random sampling can be employed for the selection, or more advanced approaches use the similarity between the current task and the demonstration to select an appropriate one [67]. For tasks that require more complex reasoning, Wei et al. [68] showed that including intermediate reasoning steps in the explanations, referred to as *Chain-of-thought (CoT) prompting*, can elicit better performance and reasoning of the model e.g

Task: If a box contains 3 apples and 4 oranges, how many fruits are there in total?

**Chain of thought prompt:**

- If a box contains 3 apples and 4 oranges, how many fruits are there in total? Start by counting the number of apples in the box. Then, count the number of oranges in the box. Add the number of apples and oranges to get the total number of fruits.  
Answer: 3 apples plus 4 oranges equals 7 fruits in the box.

### 3.1.3 RAG

Recent advancements in LLMs have showcased remarkable abilities in generating highly fluent and human-like texts while possessing extensive world knowledge. However, a notable challenge arises from their susceptibility to producing hallucinations or factually incorrect information. This issue stems from the models' limitations in accurately memorizing all information, compounded by a training procedure that needs more incentives for expressing uncertainty. Auto-regressive models used in LLMs treat their previous statements as evidence, leading to these "hallucinations". Moreover, the constant need to keep these models updated with the latest facts poses a challenge due to the significant computational resources required for training [69].

Retrieval Augmented Generation (RAG) has emerged as a promising approach to address this challenge [70]. This methodology involves coupling user input with additional information to assist the model in providing factually correct responses, even for information it may not have been initially trained on. An example of how this approach is used is augmenting the model’s prompt with proprietary company information, enabling it to accurately address questions that a model trained solely on open-source data might struggle with or provide erroneous responses to [71]. This framework involves retrieving external information and passing it to the model through prompts.

RAG operates by initially retrieving information from an external data source based on a query and then using this information to inform subsequent generations using the LLM. By grounding responses in retrieved evidence, RAG increases accuracy and mitigates issues such as hallucinations or factually incorrect information. Initially, RAG was designed to improve existing pre-training strategies by training LLM and retriever components in an end-to-end system [72]. Today, RAG is commonly associated with inference tasks only.

A naive RAG system involves three essential steps: pre-processing, indexing & retrieval, and generation [70]. Pre-processing refers to converting raw data into a standard text format. The standardized text is then divided into smaller chunks suitable for the context length of the LLM’s and the embedding model. The smaller chunks are transformed into vector representations using an embedding model and stored in a vector database post-indexing using appropriate indexing structures. These embeddings enable similarity search during the retrieval process. Finally, an index is created based on these chunks and their corresponding vectors. Indexes are generally grouped into two categories: flat and approximate nearest neighbor [73]. The user can choose either of these categories based on their preference for accuracy needs versus retrieval speed.

Upon receipt of a user query, the system employs the same embedding model to convert the query into vector representations. Then, it performs a similarity search between the encoded query and the vector representations inside the indexed vector store created earlier. This gives retrieved chunks, which are then used in the next generation step wherein the chunks, along with the user query constructed with a carefully selected prompt template, are fed to the LLM. The approach to designing the prompt template may vary based on the task, allowing it to either draw upon its inherent parametric knowledge or restrict its responses to the information in the provided documents.

The RAG framework is not just limited to semantic searches within vector databases [74] but also extends to lexical ones based on pre-defined criteria, e.g. if to answer a query, a model needs external information about the customers who made purchases between specific date ranges, it would be more efficient to search and filter for those date ranges and pass the information to the LLMs instead of encoding the data to vectors. Task-specific modelling problems may require information from specific databases, which may only sometimes need the keys to be represented as vectors to find the corresponding values. The keys may be directly filterable from within the

query, and the corresponding values should be retrieved from a simpler relational or non-relational database instead.

#### 3.1.4 Self-Refine

Like humans, LLMs do not always produce the best answer on their first try; this could happen for various reasons, ranging from generation configurations to slight changes in prompt tokens. Some recent works have been done to address this problem, one of them being self-refine [75]. This technique proposes refining an initial LLM output through iterative feedback and refinement. Unlike other techniques primarily within the training paradigm, such as additional training, reinforcement learning, and supervised finetuning, this technique relies on just one LLM: the generator, refiner, and feedback provider. The loop discussed in the work entails alternating between two generative steps, i.e., the feedback and the refiner. It works by initially generating an output, which is then sent back to the same model to ask for feedback. The feedback and the initial output are sent back to the model to refine the initially generated output. This loop continues to be executed until no further refinement is needed; this stopping criterion can either be a score that the feedback model provides, or we can have a pre-defined number of iterations, i.e., this stopping criterion is task-based. The technique has been shown to improve over the base models across multiple tasks and benchmarks, but the gains on math-based reasoning tasks are only about 5%. This modest improvement stems from the inability of LLMs to accurately identify whether there is any error, as the errors here can be much more nuanced than text-based tasks. This, however, can be mitigated if there is an external source of feedback that can identify if the answer is incorrect.

#### 3.1.5 Self-Consistency

Self-consistency [76] seeks to enhance the decision-making process of large language models by moving away from conventional greedy decoding methods [77]. The approach involves generating multiple, varied reasoning paths by adjusting generation parameters and utilizing these diverse outputs to determine the most consistent answer. This technique draws on cognitive and social psychology principles, which propose that humans naturally strive for consistency in their beliefs, attitudes, and actions. By applying this concept to language models, self-consistency ensures that their outputs remain logical and consistent throughout different sections of a response or across multiple iterations [76].

This method, like self-refinement, does not require additional training and is used during the model’s inference phase. The technique uses majority voting among various responses, generating or choosing the most consistent response. Multiple responses are generated from different model configurations that can include changes in model parameters such as the temperature or decoding techniques. Initially, it generates a variety of reasoning paths rather than just following the direct, greedy approach. Then, it identifies the most consistent response by evaluating and integrating these varied paths. It is based on the intuition that a complex reasoning problem typically admits multiple ways of thinking, leading to its unique correct answer.

## 3.2 Language agents

In recent years, more complex systems based on LLMs have been proposed to solve complex tasks and interact with the world. These language agents combine the language understanding capabilities of LLMs with the field of agent design [78]. We follow the conceptual framework of *Cognitive Architectures for Language Agents* (CoALA) [79] to describe the structure, components, and terminology of language agents in this work. Summers et al. [79] ground their framework in prior research into production systems [80], which generate outcomes by iteratively applying rules, and cognitive architectures, which were systems designed to mimic human decision-making by explicitly modeling processes such as perception, memory, and planning [81].

Due to its abstract nature, this framework is also suited to describe prompting techniques introduced in Section 3.1.2, although these techniques by themselves don't constitute a (cognitive) language agent.

In this framework, language agents are structured into three main components: the information storage or memory, the action space, and the decision-making procedure [79]. Figure 3.3 shows an overview of the framework, whose components will be broken down further in the following sections. The memory stores and provides information, and the decision procedure represents the agents' source code, which includes a decision cycle loop that determines the following action that defines the interactions between components.

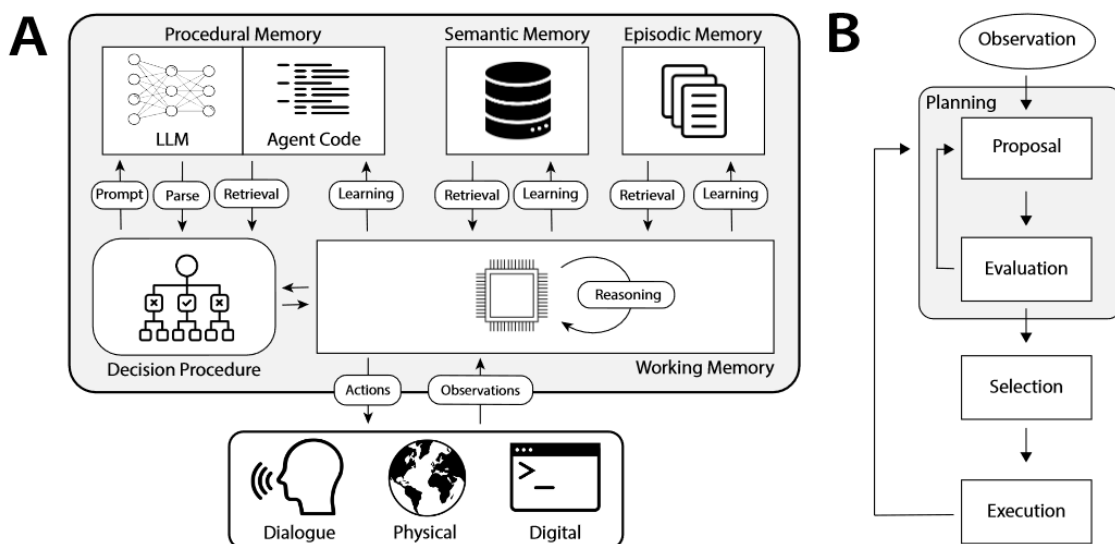


Figure 3.3: Overview of the CoALA framework. **A**: Components and processes of the framework. **B**: A decision cycle in the decision procedure [79].

### 3.2.1 Memory

In contrast to LLMs, which do not maintain information between calls and, thus, all information has to be included in the prompt, language agents store information

in different types of memory, enabling iterative problem solving and information storage between interactions [79].

#### **Working memory**

The working memory is the agent’s core memory component, which holds information available for the ongoing decision cycle. This data structure persists through LLM calls but is reset between agent calls, which have some analogies to the working memory of a computer. When calling the LLM, inputs are derived from the information in the working memory, and outputs can be parsed back into it. Additionally, long-term memory information and grounding action results observations can populate the working memory [79].

#### **Episodic memory**

Episodic memory is a form of long-term memory that persists through agent calls and contains experiences from past decision cycles or executions. These can be retrieved to assist in planning. Writing to this memory can be considered a form of learning [79].

#### **Semantic memory**

The semantic memory is the long-term memory that holds knowledge about the world and can be initialized with knowledge sources relevant to the agents’ tasks. This memory can also be accessed to retrieve information and populate the working memory or write knowledge from the working memory, which enables the agent to learn new semantic knowledge from experience [79].

#### **Procedural memory**

The procedural memory can be split into the implicit knowledge in the LLM weights and the agent’s source code, which represents explicit knowledge. The agent’s code contains the defined actions and the decision-making procedure. Compared to other types of long-term memory, which can be considered optional or initially empty, procedural memory is strictly required to establish the agent. Writing to explicit knowledge can also be considered a learning action, but it is riskier since it can threaten the syntactic functionality of the agent.

### **3.2.2 Actions**

Actions define the interactions and information flow between different components of cognitive language agents. The CoALA framework distinguishes between four types of actions, which are displayed in Figure 3.4.

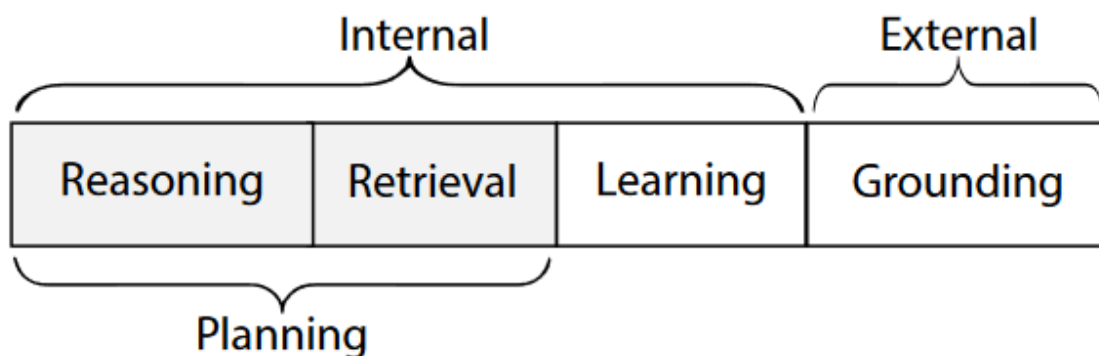


Figure 3.4: Action space of cognitive language agents [79].

### Grounding actions

Grounding actions define the interaction of agents with external environments. Since the primary processing unit of a language agent is an LLM, the action results must be represented in textual form and stored in the working memory. Language agents usually interact with three main types of external environments. Physical environments, where actions can alter the physical environment through some physical embodiment (robotic arms) and perceptual inputs, must be converted to text. Dialogue interactions with humans or other language agents that can provide instructions, feedback, or other forms of guidance. Digital environments like games, APIs, or code execution, which is the most common and available kind of external environment [79].

### Retrieval actions

Retrieval actions are considered internal actions that read information from the various forms of long-term memory into the working memory. Examples of retrieval actions are database and document lookups, or dense-retrieval methods [79].

### Reasoning actions

Reasoning actions read and write to working memory. Their purpose is to allow the agent to process and give context to memory items, and results can either be written to long-term memory as a kind of learning or support decision-making and future actions of the agent [79].

### Learning actions

Learning actions alter parts of the long-term memory of the agent to affect the available information or behavior between agent calls. The CoALA framework differentiates between four different kinds of learning actions. First, it involves writing experiences, like action sequences, to episodic memory. Second, it updates knowledge of the semantic memory. Third, it updates LLM parameters using fine-tuning or reinforcement learning techniques. Lastly, it involves updating parts of the agent’s code. This can include altering or adding actions or altering the decision procedure itself [79].

### 3.2.3 Decision procedure

The decision procedure defines the agent’s control flow and represents the agent’s top-level function. This procedure is usually a loop of decision cycles displayed in Figure 3.3B, which results in a grounding or learning action. A decision cycle comprises two main stages: the planning stage, which decides which action to take, and the execution stage, which executes the determined action and terminates or continues the decision cycle [79].

#### Planning

Planning consists of three steps and is supported by flexibly using reasoning or retrieval actions. The first step is to propose a set of candidate actions. Depending on the complexity of the agent and action space, this can be the complete set of actions or a subset. Additionally, this process can be governed by predefined stages or if-else conditions set by the agent developers. When multiple actions are proposed, the actions can be evaluated by assigning a value to each. This can, for example, be realized with heuristic rules or LLM reasoning. Based on the evaluation results, in the selection stage, action is chosen, or the planning stage is re-executed if evaluation results are unsatisfactory [79].

#### Execution

After selecting an action in the planning stage, the action is executed by calling the associated code. Depending on the agent implementation, this action might be a grounding or a learning action. After the execution, either the decision cycle starts again and an observation from the environment is made, or the decision cycle is terminated, and the agent execution ends [79].

## 3.3 Model explainability

Although LLMs have shown strong performance in a range of NLP tasks, their usage in production systems depends on the reliability of their outputs and the trust users can have in these systems. For both of these factors, model explainability plays a central role. Following Doshi et al., we understand explainability as the ability to present the behavior of a model in a human-understandable manner and use it interchangeably with interpretability [82].

Model explainability techniques can be categorized into local and global explanations. Local explanations try to increase the understanding of how a decision is made for a single instance. In contrast, global interpretability methods try to provide insights into the general behavior of models [83].

Different local explanation methods are used for LLMs. Feature-attribution-based methods aim to discover the relevance or importance of individual features on the model’s output. These methods include perturbation-based methods [84], where inputs are perturbed by deletion, masking, or altering, and gradient-based approaches [85], where the partial derivate of the output concerning the input is calculated to measure the sensitivity of the output to changes in the input [83].

In contrast, example-based methods aim to explore model behavior from the perspective of individual samples, so they probe how the output changes with different inputs. Here, two main perturbation-based techniques exist: adversarial examples and counterfactuals [83]. Adversarial examples exploit that neural-based models are highly sensitive to small changes in the input, which are typically insignificant to humans. They are constructed by perturbing the input’s less important components to expose language model weaknesses [83].

Counterfactuals aim to provide causal explanations and assume the input to be the cause of the prediction under the Granger causality [86]. More specifically, given an input  $x$  and the perturbed input  $\hat{x}$ , the output changes from  $y$  to  $\hat{y}$ . This probes the influence of changes to the input on potential changes in the predictions [83]. Counterfactuals are mostly constructed by humans or perturbation techniques. For perturbation techniques, usually, a two-step process is employed. First, important tokens are identified. Then, the tokens are edited or deleted to produce a perturbed sample [87]. Ross et al. [88] use gradient-based attribution to identify necessary tokens in their system MICE and minimize the number of edits using binary search.

For LLMs, natural language explanations are another approach to generating and presenting explanations for users. These approaches differ when explanations concerning a prediction task are generated. The most common approaches are predict-and-explain, explain-and-predict, and generating predictions and explanations jointly. Additionally, different models can be employed for the explanation and prediction task [83]. Although natural language explanations are very user-friendly, due to their easy interpretability for humans, they are harder to quantify, and the relation between the given prediction and the provided explanation is an open research field [89].

Next to getting insights into the model reasoning and explanations, estimating uncertainty is another approach to increasing user trust [90]. Commonly, logits have been used to derive the uncertainty [91]. However, these are not always accessible, especially with closed-source models, which is why multiple non-logit-based (known as confidence elicitation) approaches have been proposed. Consistency-based approaches generate multiple predictions by introducing randomness in the generation process and checking the consistency between them [83]. When the consistency between answers is high, a high confidence is assumed [83]. LLMs can also be prompted to provide textual certainty in their answers. This can either be implicit by them using direct and specific language when they are confident, more indirect and ambiguous language when the uncertainty is higher, or explicit by prompting them to generate a percentage number for how certain they are. However, calibrating models to provide accurate uncertainty estimates has proven to be difficult, and models are often overconfident in their answers [92].



# 4

## Methods

This chapter describes the employed methods and lays out the performed steps and design choices.

### 4.1 System overview

Although we propose different systems that solve the described task of deciding if a given test script covers a requirement given in natural language, the systems share some common structure described in this section. Even though some proposed systems can be classified as prompting techniques, we frame all of our systems using the framework of language agents as introduced in Section 3.2. This is done for two main reasons: firstly, it eases the comparisons between different systems by using the same terminology for components, and secondly, it simplifies the identification of possible extensions in future research.

Each proposed system has one or multiple LLMs as its main components. Additionally, all systems take a test script and a requirement as textual inputs and produce a compliance decision in combination with natural language reasoning as their final grounding action. With the task of checking the compliance of a test script with a requirement being very specific and uniform in how it is solved between different samples, we identified that having a dynamic planning process is of lower priority and use static plans, determined by the developers, for our systems.

### 4.2 Data

This section first describes the used dataset, including the data sources and structure. Then, the preprocessing steps are introduced, followed by the applied data transformations and the generation/synthesis of new data.

#### 4.2.1 Dataset

The data in this work concerns automotive embedded systems, specifically the functions and interactions of electronic control units (ECUs). These are usually controlled by setting signals over the CAN bus. The initial dataset consists of 201 requirements and 237 test scripts.

The requirements are formulated in natural language and are the basis on which vehicle functions are developed per the V-model. The requirements cover various functions, from washing wipers to the light machine. Additionally, they differ in complexity, clarity, and self-containment, i.e., their reliance on external information to be understandable. Some requirements describe the interactions of a few selected signals, while others define global policies that must be met for groups of signals and ECUs.

The test scripts are written using Volvo’s internal test specification framework. The framework produces a structurally simple language that differentiates between three main building blocks. Functions are the main elements, and a test script is described and executed as a sequence of functions. The available functions are pre-defined and have a capitalized name followed by a colon (":"). Functions consist of (multiple) potentially nested parameters that specify, change, or check the states of the ECUs. These parameters take arguments as their input, which can be numerical values, signal names, or signal states. Additionally, comments can be defined inside functions as comment functions. When comments are embedded inside a function, they are prefixed with a ";" and continue until the next parameter is given. An example of a function in a test script is:

Code Listing 4.1: Example of functions in test scripts

```
StateChange: ;Go to GB Stacked Set(Type(CAN) Node(ECU1) Message
(**) Signal(A) Value(1)) Wait(200 ms)
StateCheck: ;Verify that output signal is NotAvailable Wait(0
ms) Check(Type(CAN) Node(ECU_A) Message(***) Signal(B)
Comparer(=) Value(7))
```

Here, the two functions *StateChange* and *StateCheck* are called consecutively. In the *StateChange* function, first a descriptive comment is given, before *Signal A* is set to the *Value 1* and a *Wait* of *200 ms* is enforced. Then, the *StateCheck* function starts with a descriptive comment and a waiting time of *0 ms*, before *Signal B* is compared to the *Value 7*.

The dataset annotates the relationship between test scripts and requirements with human supervision. The annotated relationship is the compliance of a test with a given requirement. That means that the associated test script is understood to cover the requirement. However, there is no one-to-one relationship between requirement and test script. There can be multiple associated test scripts for each requirement, and each test script can cover multiple requirements. Even when a test script is only associated with one requirement in the dataset, the code in the test script can cover more functionality than described in the requirement.

In addition to requirements and test scripts, two external data sources are considered in this work. The first data source is a snapshot of the CanDB, which contains information about the signals in the car, like their encoding, datatype, and possible values. We precisely extract the value information, which defines a value range by stating minimum and maximum values or enumerating possible states and their

names. This information is stored in a JSON file with the signal name as the key and the associated information as a dictionary.

The second data source is function descriptions from the internal knowledge-sharing platform, referred to as PnTool information. These descriptions are manually created and presented on an HTML page. They contain the function's name, a natural language description of its purpose, a selection of parameters used in the function, and an example of syntactically correct usage. We scrape these web pages and extract the information for each described function. Then, the information is stored in a JSON file, with the function name as the key and the function information as a dictionary.

## 4.2.2 Data preprocessing

For the internal data sources, i.e., test scripts and requirements, several data cleaning and pre-processing measures are done to format them correctly for further analysis. We received these datasets in delimiter-separated file format with "α" as the delimiter. The test scripts and the requirements are provided in separate files. After renaming the column to more explicit and descriptive names, e.g., renaming the description within the requirement dataset to the name requirement, we fill the null values with empty strings within both datasets.

Within the test script dataset, for a certain number of rows, the annotated ground truth *req\_ground\_truth* was mentioned as a string value '[]'; we remove such rows. Additionally, the *req\_ground\_truth* had to be cleaned using a regex as it was sometimes present between '[]' and sometimes as is. Additionally, two possible columns within the test dataset could contain the actual test script, namely *test script* and *purpose*. Having looked at the dataset comprehensively, we concluded that the column 'test script' would often contain the test script. However, the string value had to begin with either the string *ScriptTool* or *TestScriptStart*; if neither is found within the test script, we do the same check within the column 'purpose' to identify a potential test script. If neither column satisfied this condition, we remove the row. Now, we rename this column to 'test'. Some test scripts reference other test scripts using an *Include* function; since we did not have access to the referenced test scripts, we remove all test scripts that reference other test scripts.

After manually reviewing the provided requirement dataset, we split the requirements into three quality categories (1, 2, 3) in decreasing quality order. The criteria used to classify the requirements are the complexity, defined by the length and number of described steps in the requirement; the clarity, meaning how unambiguous the requirement is formulated; and the self- containment, i.e., how much external knowledge not contained in the requirement is needed to understand the requirement. These criteria are derived and slightly adapted from the criteria described in Section 2.2. After classification, the requirement dataset contains 89 of quality 1, 65 of quality 2 and 46 of quality 3.

Finally, the pre-processed datasets are written into JSONL files and ready to be utilized for subsequent analysis or integration into other systems for further exploratory or

development work.

We create an abstract lookup table class for the external data sources, which offers the method lookup. Building on top of it, we create *CanDirLUT*, which takes a directory path containing the .dbc files (files containing data about can signals), *use\_presaved\_dict* as arguments. The class contains several methods to create a signal table to find information about particular signals later. The *use\_presaved\_dict* arg was introduced later as the initialization and creating of the signal table took quite a lot of time depending on the number of .dbc files, so we create and save the table into a JSON, as stated earlier, to use since the dbc files do not change too often and are static in the info they give per signal.

The second lookup class we create is named *PnDocumentLUT*, which allows for looking into the available documentation of functions within the test scripts. To create this lookup table, we scrape the SharePoint page using Beautiful Soup using the Sharepy API, wherein we look at the relevant HTML tags that hold the necessary information to store inside the table and extract them from each function's page. As stated earlier, this is stored inside a JSON, and any particular function description can be searched by calling the class method lookup.

### 4.2.3 Data perturbations

Several data perturbations are conducted to construct a more diverse dataset and probe our system's behavior in different circumstances. The applied perturbations are mainly based on the structure of the test scripts described in Section 4.2.

Due to the content of the test scripts being a programming language instead of natural language, the effect of applying specific perturbations is more predictable and results in the construction of counterfactuals. For example, if the requirement asks for a specific behavior of signal x and all mentions of signal x in the test script are replaced with different signals, one can assume that the label changes and the transformed test script no longer covers the requirement. However, natural language requirements often do not mention specific signals and can cover more general aspects. Still, we argue that the broad application of perturbations on these structured inputs leads to a change in the label. This change can be caused by the syntactic impact of changes or the semantic impact of content changes like replacing signal names. Although the change in label is not guaranteed, its probability is high enough to gain insights into how the system reacts to the perturbation. In the following, the perturbation used is described, as well as the assumed impact the perturbation likely has on the label.

Following are the possible perturbations for test scripts with examples highlighting the effects of different perturbations:

Code Listing 4.2: Example of a short test script before perturbation

```
[Preparation Sequence] [Main Sequence]
Comment: ;>>>> INITIALIZATION <<<<<<
CaplTestCase.Common_Init: ;Initializes all common signals to get the test system up and running.
Comment: ;>>>> START TEST <<<<<<
StateChange: ;Request fan activation and verify that outputs are active. Set(Type(CAN) Node(
    ECU1) Message(ECU1) Signal(A) Value(InactiveActive_Active (1))) Wait(200 ms) Check(Type(
```

```

    Internal) Node(ECU2) Message(-) Signal(B) Comparer(=) Value(1)) Check(Type(Internal) Node(
    ECU2) Message(-) Signal(C) Comparer(=) Value(1))
Comment: ;>>>> RESTORATION <<<<<
CaplTestCase.Common_Exit: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]

```

### Remove comments:

This perturbation identifies and removes comments from the test script as displayed in Listing 4.3. Comments are identified in the test script by being prefixed with ";" and ending when a different component, functions, or parameters start. Additionally, the *Comment* function is removed. When the comments of a test script are removed, the label should not change, and the test script should be assumed to be the same as the original. This perturbation probes the system's sensitivity and reliance on the content or presence of comments.

Code Listing 4.3: Remove comments perturbation example

```

[Preparation Sequence] [Main Sequence]
CaplTestCase.Common_Init:
StateChange: Set(Type(CAN) Node(ECU1) Message(ECU1) Signal(A) Value(InactiveActive_Active (1)))
              Wait(200 ms) Check(Type(Internal) Node(ECU2) Message(-) Signal(B) Comparer(=) Value(1))
              Check(Type(Internal) Node(ECU2) Message(-) Signal(C) Comparer(=) Value(1))
CaplTestCase.Common_Exit:
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]

```

### Remove/Replace components:

Test scripts are subjectively split into three phases by test developers: Initialization, Test, and Restoration. Phases are marked by a comment that states the name of the phase. This perturbation either removes the specified phases from the test script or replaces them by sampling a random phase from a different test script while ensuring that the phase is chosen from a test that is not compliant with the same requirement as displayed in Listing 4.4. Due to the inconsistent usage, shortness, and lack of relevance for the compliance, we choose only to replace the Test phase and assume a change of the label.

Code Listing 4.4: Replace components perturbation example

```

[Preparation Sequence] [Main Sequence]
Comment: ;>>>> INITIALIZATION <<<<<
CaplTestCase.Common_Init: ;Initializes all common signals to get the test system up and running.
Comment: ;>>>> START TEST <<<<<
Set: ;Request lock Set(Type(CAN) Node(ECU1) Message(ECU1) Signal(F) Value((1)))
AwaitValueMatch: ;Wait for command to be sent on CAN Timeout(200 ms) LogicalOperator(All)
                Check(Type(CAN) Node(ECU1) Message(ECU1) Signal(G) Comparer(=) Value((1)))
Comment: ;>>>> RESTORATION <<<<<
CaplTestCase.Common_Exit: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]

```

### Replace functions:

This perturbation replaces a subset or all of the functions in a test script as displayed in Listing 4.5. Functions are replaced by sampling a different function from all functions used in the available data set. Since there is external information regarding the available functions, the perturbation probes the system's sensitivity to proper function usage. This can either be in regards to semantics or syntax. Since these changes would lead to the test not being possible to compile, we assume a change of label.

Code Listing 4.5: Replace functions perturbation example

## 4. Methods

---

```
[Preparation Sequence] [Main Sequence]
StateChange: ;>>>> INITIALIZATION <<<<<<
StateCheck: ;Initializes all common signals to get the test system up and running.
Set: ;>>>> START TEST <<<<<<
Wait: ;Request fan activation and verify that outputs are active. Set(Type(CAN) Node(ECU1)
      Message(ECU1) Signal(A) Value(InactiveActive_Active (1))) Wait(200 ms) Check(Type(Internal)
      Node(ECU2) Message(-) Signal(B) Comparer(=) Value(1)) Check(Type(Internal) Node(ECU2)
      Message(-) Signal(C) Comparer(=) Value(1))
AwaitValueMatch: ;>>>> RESTORATION <<<<<<
StateCheck: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]
```

### Replace parameters:

This perturbation replaces a subset or all of the parameters in a test script as displayed in Listing 4.6. Parameters are replaced by sampling a different parameter from all parameters used in the available dataset. This perturbation probes the sensitivity towards changes in the parameters and we assume a change of the label. This could, for example, be a mismatch of the arguments used with a replaced parameter or the improper sequencing of nested parameters.

Code Listing 4.6: Replace parameters perturbation example

```
[Preparation Sequence] [Main Sequence]
Comment: ;>>>> INITIALIZATION <<<<<<
CaplTestCase.Common_Init: ;Initializes all common signals to get the test system up and running.
Comment: ;>>>> START TEST <<<<<<
StateChange: ;Request fan activation and verify that outputs are active. Check(Set(CAN)
      Compare(ECU1) Signal(ECU1) Check(A) Message(InactiveActive_Active (1))) Type(200 ms) Type(
      Check(Internal) Set(ECU2) Compare(-) Value(B) Signal(=) Message(1)) Set(Node(Internal)
      Message(ECU2) Signal(-) Set(C) Value(=) Check(1))
Comment: ;>>>> RESTORATION <<<<<<
CaplTestCase.Common_Exit: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]
```

### Replace arguments:

This perturbation replaces a subset or all of the arguments in a test script as displayed in Listing 4.7. The subsets are built by specifying which parameter the arguments should or should not be replaced. Arguments are either replaced by sampling a different argument from all arguments used in the available dataset or all arguments used for the related parameter. The case where arguments are sampled globally from all possible arguments generally leads to more severe perturbations since value types are not respected and can vary depending on the parameter. When sampling from arguments for the specific parameter, the resulting perturbations are less severe since the arguments are used in the parameter context. This perturbation probes the sensitivity towards changes in the arguments, which could be using a different signal name or setting an existing signal to a different value. We assume that replacing arguments in the test script changes the label.

Code Listing 4.7: Replace arguments perturbation example

```
[Preparation Sequence] [Main Sequence]
Comment: ;>>>> INITIALIZATION <<<<<<
CaplTestCase.Common_Init: ;Initializes all common signals to get the test system up and running.
Comment: ;>>>> START TEST <<<<<<
StateChange: ;Request fan activation and verify that outputs are active. Set(Type(Internal)
      Node(ECU2) Message(ECU2) Signal(F) Value(VehicleMode_Parked (1))) Wait(300 ms) Check(Type(
      CAN) Node(ECU2) Message(ECU1) Signal(G) Comparer(..) Value(5)) Check(Type(CAN) Node(ECU2)
      Message(ECU3) Signal(Z) Comparer(<>) Value(-6))
Comment: ;>>>> RESTORATION <<<<<<
CaplTestCase.Common_Exit: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]
```

### Compound perturbations:

This perturbation allows the combinations of the specific perturbations already

described. The perturbations are executed sequentially on the test script. The perturbations used are order sensitive, so the same result is not guaranteed when the same perturbations are applied in different order. The effect of the resulting compound perturbation depends on the included perturbations. An example that combines the removal of comments with the replacement of arguments is displayed in Listing 4.8. We assume a label change if one of the involved perturbations leads to a label change by themselves.

Code Listing 4.8: Compound perturbation example, which combines the removal of comments with the replacement of arguments

```
[Preparation Sequence] [Main Sequence]
CaplTestCase.Common_Init:
StateChange: Set(Type(Internal) Node(ECU2) Message(ECU2) Signal(F) Value(VehicleMode_Parked (1)
)) Wait(300 ms) Check(Type(CAN) Node(ECU2) Message(ECU1) Signal(G) Comparer(...) Value(5))
Check(Type(CAN) Node(ECU2) Message(ECU3) Signal(Z) Comparer(<>) Value(-6))
CaplTestCase.Common_Exit:
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]
```

#### 4.2.4 Data generation

Due to the lack of data quality and to probe the limitations of the systems, we further generate some manual and synthetic data.

To investigate the importance of requirement complexity and clarity, we generate 25 manually written requirements. These requirements are simplified and adapted versions of the original requirements classified as quality 1. The following shows an example of an original requirement and the simplified, manually written version.

##### Original:

requirement name: *SignalName*: Washer Operation  
requirement description: When *SignalName* changes to Active, the following actions shall be carried out: Put the Windscreen Washer Pump control pin to to +24V When *SignalName* changes to Inactive the following actions shall be carried out: Put the Windscreen Washer Pump control pin to *GND* or *Open circuit* For all other values of *SignalName*, the Windscreen Washer Pump control pin shall be set to *GND* or *Open circuit*.

##### Adapted:

requirement name: Washer Operation  
requirement description: When *SignalName* changes to Active the following actions shall be carried out: Put the Windscreen Washer Pump to on. When *SignalName* changes to Inactive the following actions shall be carried out: Put the Windscreen Washer Pump control pin off. For all other values of *SignalName* (Error, NotAvailable), the Windscreen Washer Pump should be off.

This simplifies the requirement in three ways. The values are described more clearly since in the test script, 24V is modeled with the value 1 and *GND/Open Circuit* with the value 0 and using *on/off* respectively should simplify the translation requirement

If `{set_signal}` is equal to `{set_signal_value}`, then `{check_signal}` should be `{comparison_operator}` `{check_signal_value}`.

Listing 1: Template for a simplified requirement

from the natural language values to the ones used in the test. The mention of *pins* is removed since they are not explicitly mentioned in the test scripts, and only signal names are used. The explicit enumeration for all other values (*Error*, *NotAvailable*) because the model would need to rely on external information or assumptions to know if all other states are tested.

To better understand possible limitations of the proposed system, we also generate a synthetic dataset of test scripts and associated requirements. To build the synthetic test scripts, we extract simple sequences of setting a signal value and then checking the values of a selection of other signals. Here, for every test script we first check for the presence of the *Set* and *Check* parameters in each function. If both are present in a single function, the function is saved as a candidate. If only a *Set* parameter is found the next function (skipping *Wait* functions) is searched for a *Check* parameter before any other *Set* parameter. Given this condition the sequence of both functions is saved as a candidate. For each test script we then sample one of these new simplified sequences to limit the size of the resulting dataset.

Given these sequences, we generate a testscript by adding the *INITIALIZATION*, *START TEST* and *RESTORATION* components, as well as common test scripts prefixes and suffixes. Listing 4.9 shows an example of a simplified test script.

Code Listing 4.9: Example of a simplified test script

```
[Preparation Sequence] [Main Sequence]
Comment: ;>>>> INITIALIZATION <<<<<<
CaplTestCase.Common_Init: ;Initializes all common signals to get the test system up and running.
Comment: ;>>>> START TEST <<<<<<
StateChange: ;Request fan activation and verify that outputs are active. Set(Type(CAN) Node(
    ECU1) Message(ECU1) Signal(A) Value(InactiveActive_Active(1))) Wait(200 ms) Check(Type(
    Internal) Node(ECU2) Message(-) Signal(B) Comparer(=) Value(1)) Check(Type(Internal) Node(
    ECU2) Message(-) Signal(C) Comparer(=) Value(1))
Comment: ;>>>> RESTORATION <<<<<<
CaplTestCase.Common_Exit: ;Clean up after execution (including setting vehicle mode to Parked).
[Completion Sequence] [Conditions Sequence] [Constraints Sequence]
```

Having extracted a simplified test sequence, we generate the associated response with a string template displayed in Listing 1. Here the signal names, signal values and comparison operators are extracted from the simplified test sequence. If multiple *Set* or *Check* parameters exist, they are concatenated to the requirement and connected with an "and". The resulting requirement explicitly names the all the signal information, making them substantially more precise than the requirements in the dataset.

Having generated a simplified dataset, we scale the difficulty of the dataset in two dimensions: length of the test script and complexity of the test sequence. To scale the length we inject the generated test sequence into the *START TEST* component of a randomly sampled test script that doesn't share a signal name with the test sequence. This is done to probe the effect of prompt length and noise on the LLM-based systems. The complexity is scaled by concatenating multiple generated test sequences and their associated requirements, with each complexity level being an

additional sequence added.

### 4.2.5 Dataset construction

As described in Section 4.2.1, the available dataset has the relationship between requirements and compliant test script as its ground truth labels. This establishes a clear set of instances where the system is tasked to output a positive classification decision, in the following referred to as positive samples. However, the choice of negative samples is not as clearly defined. We refer to the task of constructing a dataset for a specific experiment as dataset construction.

The construction of the dataset concerns only the choice of requirements and tests. It does not include the decision about using external information sources since we regard this as a matter of the compliance check system used.

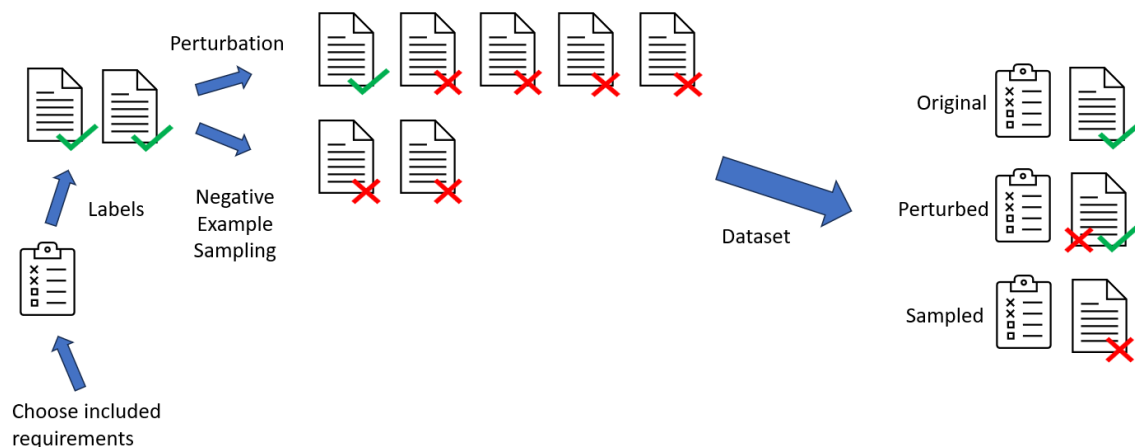


Figure 4.1: Overview of the dataset construction process.

Figure 4.1 outlines the process for constructing a dataset. The first decision point for dataset construction is the choice of which requirements are included in the experiment. This can include choosing manual requirements, a subset of the provided requirements decided by their quality classification, the complete set of provided requirements, or a combination.

After the included requirements are defined, we choose the test scripts with which they are matched in the classification decision. Generally, we identify the compliant tests for each requirement and include the tuple of requirements and compliant tests in the dataset. Additionally, perturbations of the test scripts can be constructed using the perturbations described in Section 4.2.3. The choice of perturbations used is elaborated further in the respective experiment sections.

To include negative samples not created by perturbations, all non-compliant test scripts are potential candidates. However, this poses several problems. The amount of negative samples is much more significant than the amount of positive samples for a requirement, which leads to a very imbalanced dataset. Additionally, the computational cost of testing each requirement against approximately 240 test scripts

is infeasible. Furthermore, the task’s difficulty might be low since test scripts usually have different purposes and thus do not include the same or similar signal names. To alleviate this problem, we sample one random test script as a negative sample and one similar test script, where we define similarity as sharing at least one signal name with the original test script.

Constructing a dataset has some inherent risks since it directly influences the difficulty of the task and, therefore, the model performance. For this reason, comparative experiments are always conducted with the same constructed dataset, making different approaches comparable to the same task.

### 4.3 Evaluation procedure

This section introduces the evaluation scheme addressing research question 1. Following the provided dataset and problem definition, we frame the problem as a classification task. Here, a system is required to provide a binary classification decision if a test script covers a requirement formulated in natural language, given a pair of test script and requirement.

For performance metrics, we focus on standard classification metrics, such as precision, specificity, sensitivity, accuracy, and F1-score. The type of error the classification system makes has different significance and costs in this particular use case of checking the compliance of a test with a given requirement. When the system makes a falsely positive classification, and the system’s decision is relied upon, it is falsely assumed that the test complies with the requirement. This can have severe consequences if the requirement describes safety or security-relevant aspects. On the other hand, making a falsely negative decision leads to possible redundancy in the test code or unnecessary work because a new test would have to be written or an existing test altered, which is more of a monetary risk and, therefore, lesser risk. Since the types of errors have different significance, we explicitly include precision and sensitivity in our classification metrics.

The evaluation scheme is primarily based on local example-based explanation methods, which, when combined, can give a global perspective on model behavior. In this work, we focus the evaluation mostly on structured input perturbations, described in Section 4.2.3, to gain insights into how the systems react to changes in the input. Data of different complexity and quality is generated or classified to further probe the influence of changes in the data on the classification performance.

Given a constructed dataset (see Section 4.2.5) for an experiment, we evaluate the performance of a system in the following ways. We stratify the results by the perturbation used, the sampling technique of negative examples, or if it is an original test. Then, each strata’s performance metric is calculated depending on the underlying label as illustrated in Figure 4.2. We report the model’s sensitivity for strata where the label is positive (the original test script, the remove comments perturbation). For strata where the label is negative (sample of random test script, sample of similar test script) or we make an assumption of a negative label as described in Section 4.2.3, we report the specificity. This allows us to evaluate the

system performance for different types of perturbed inputs and gain insights into which changes in the input the model is sensitive to. This part of the evaluation scheme relies on the reliable construction of counterfactuals, which is not guaranteed in all cases (see Section 4.2.3) and evaluates the transformed or sampled samples independently from the original test script and its classification.

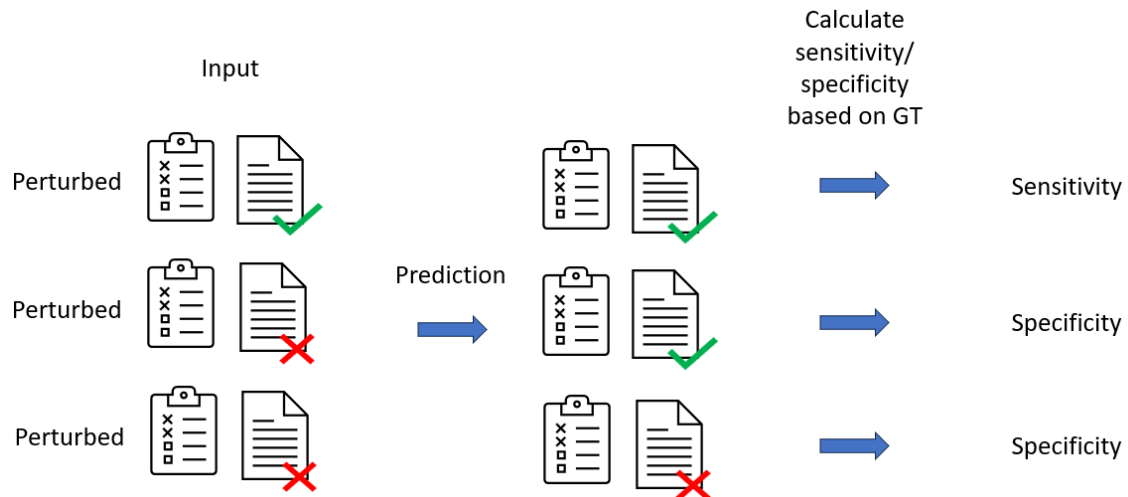


Figure 4.2: Overview of the counterfactual-based evaluation scheme.

Since models that either always make a positive or negative classification decision would perform very well in the respective strata, we also report metrics for larger slices of the dataset. Figure 4.3 illustrates the approach. The global performance includes the original tests and two sampled test scripts (random, similar), and accuracy, precision, sensitivity, and F1-score are reported since, here, the dataset consists of a mix of positive and negative samples. This evaluation does not include any perturbation and can, thus, be interpreted as an approximation of the performance if the system was tested against the whole provided dataset.

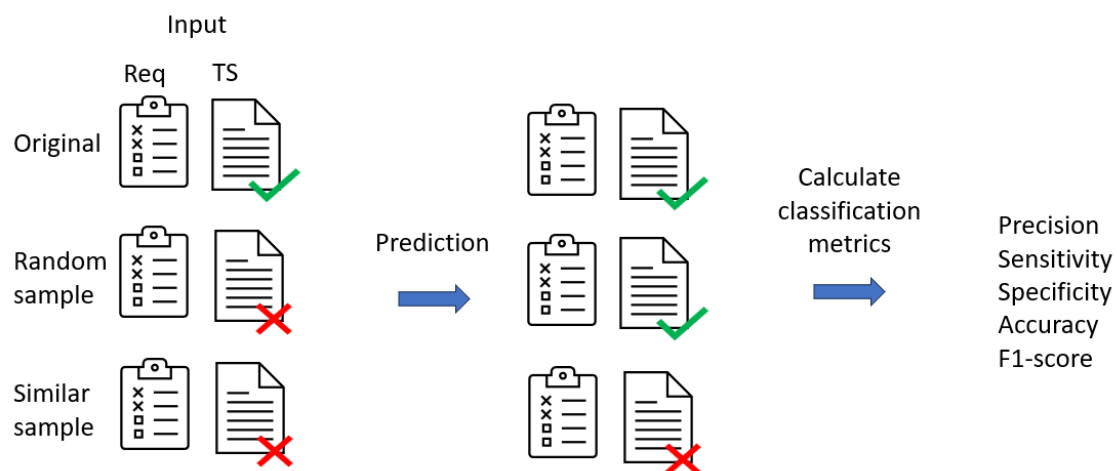


Figure 4.3: Overview of the performance evaluation scheme.

To further deepen the understanding of the impact of input perturbations, we consider the original test script and the associated system prediction jointly with all perturbations and negative samples and analyze label flips as outlined in Figure 4.4. Here, we first observe the prediction for the original test script and then record if a label flip, meaning a change of the prediction either from positive to negative or vice versa, occurs for a specific perturbation or negative sample. We then report the percentage of label flips independent of the initial prediction as the global flip fraction and stratified by the initial prediction.

This approach does not rely on any label assumptions for transformed samples. However, the correctness of the original prediction is not considered, and this evaluation does not independently constitute a performance measure of the underlying system. The results can give insights into what kind of perturbations the system is sensitive to and which parts of the input are essential for the classification.

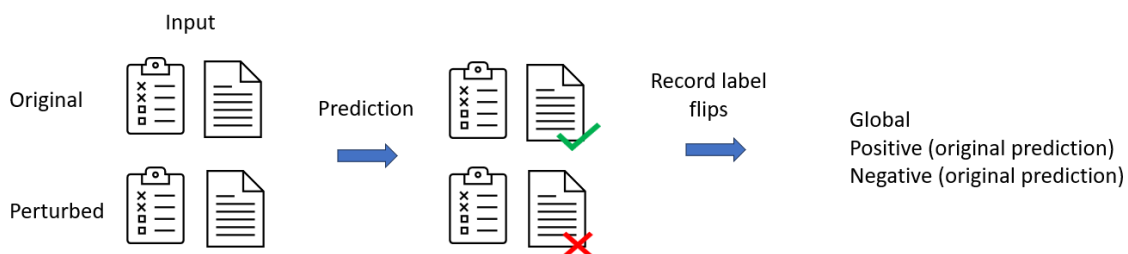


Figure 4.4: Overview of the label flip evaluation scheme.

## 4.4 Prompting techniques

This section talks about the methodology and technical aspects of our experimentation pipeline. Our system aims to do the review process by using different in-context learning techniques, external data sources, and parsing protocols.

### Component Overview

On a high level, our two main components that form the experimentation chain are the *Reviewer* and *ReviewerProtocol* classes. These components help with the interaction between the user sending in a pair of test scripts and a requirement and our reviewing logic.

### Reviewer & ReviewerProtocol

The *Reviewer* class manages dialogue between users and the LLM system, determining appropriate responses to user queries. It is initialized with arguments such as: the LLM, test script, requirements, and parameters related to external data sources and parsing protocols.

The *ReviewerProtocol* class works in conjunction with the *Reviewer* to formulate prompts made for specific review cases. By considering various combinations of input parameters, such as the inclusion of signal dictionaries and function descriptions, the protocol constructs prompt strings.

The protocol extracts relevant functions from the test script if we enable signal dictionaries. Additionally, if signal extraction is needed, it parses signals using regular expressions from the test script. It then dynamically combines this information with placeholders within a pre-formatted string, including placeholders for the test script, requirement, signal dictionary, and function documentation.

Subsequently, the *Reviewer's chat* method communicates with the LLM, sending the constructed prompt string as input and receiving output containing decision and reasoning. The protocol's *parse* method uses regular expressions to extract decisions and reasoning from the output string. In case of parsing errors, the system iteratively re-sends the output string to the LLM, allowing up to 10 iterations to ensure proper formatting for successful parsing.

Following is our current prompt configuration based on different in-context learning techniques and external information sources:

### Protocol Options

Displayed below are the protocol options defined for different scenarios with the key of the dictionary representing the tuple of the arguments `signal_dict_info`, `pn_document_info` and in-context learning technique, and the corresponding values are the prompts called for them:

```
protocol_options = {
    (True, True, "ZERO_SHOT"): BASE_PROMPT + INPUT + SIGNAL_INFO +
        FUNCTION_INFO +
        FINAL_INSTRUCTIONS,
    (True, False, "ZERO_SHOT"): BASE_PROMPT + INPUT + SIGNAL_INFO +
        FINAL_INSTRUCTIONS,
    (False, True, "ZERO_SHOT"): BASE_PROMPT + INPUT + FUNCTION_INFO
        + FINAL_INSTRUCTIONS,
    (False, False, "ZERO_SHOT"): BASE_PROMPT + INPUT+
        FINAL_INSTRUCTIONS,
    (True, True, "TWO_SHOT"): BASE_PROMPT + TWO_EXAMPLE_INSTRUCTION
        + EXAMPLE_1 +EXAMPLE_2 +
        INPUT + SIGNAL_INFO +
        FUNCTION_INFO +
        FINAL_INSTRUCTIONS,
    (True, False, "TWO_SHOT"): BASE_PROMPT +
        TWO_EXAMPLE_INSTRUCTION +
        EXAMPLE_1 +EXAMPLE_2 + INPUT
        + SIGNAL_INFO +
        FINAL_INSTRUCTIONS,
    (False, True, "TWO_SHOT"): BASE_PROMPT +
        TWO_EXAMPLE_INSTRUCTION +
        EXAMPLE_1 +EXAMPLE_2 + INPUT+
        FUNCTION_INFO +
        FINAL_INSTRUCTIONS,
    (False, False, "TWO_SHOT"): BASE_PROMPT +
        TWO_EXAMPLE_INSTRUCTION +
        EXAMPLE_1 +EXAMPLE_2 + INPUT+
        FINAL_INSTRUCTIONS,
}
```

Within the *BASE\_PROMPT* given in Appendix 3.2, we refer to [65] to give the model a role description by stating, "You are an expert code reviewer." . Through empirical evidence from various iterations using different prompts, we observe that this approach helps the model generate relatively better reasoning than when a persona is not provided. Additionally, we provide the model with a description of the task, stating, "You are given a test script written by a developer and a requirement that it should fulfill. Your task is to evaluate if the test script strictly fulfills the provided requirement." Additionally, we provide our definition of compliance as defined in Section 1.3. We then add a string describing how it should reason about the task before making a decision: "The decision should be Yes/No on whether the test script meets the requirements. As a reason, you should break down the requirements and point out which parts of the test script fulfill them. Try to interpret the requirement and reason if the tests meet the intention behind the requirement. Some things are not directly mentioned in the requirement but are implied, so you should also look for those in the test script. The Test Script is a sequence of functions, which are separate lines. Each function has a name followed by a ':'. Inside the function, one or more (nested) parameters are called, with their arguments in brackets '()'." Within the *BASE\_PROMPT*, we also instruct the model that it needs to "Return the output in a JSON format, i.e., with decision and reason as its keys."

We add the actual test script and the requirement in the *INPUT* (see Appendix 4). And, for providing the model with more context using external information, the *SIGNAL\_INFO* (see Appendix 6) and *FUNCTION\_INFO* (see Appendix 7) are included. This is where we place the signal definitions obtained from our CanDirLUT and the function definitions obtained from our PnDocumentLUT.

For the two-shot learning technique, we introduce two different examples that demonstrate the task and its completion, i.e., we show the model a test script, a requirement, the relevant compliance decision, and the reason for compliance. We use two such examples, making the prompt a two-shot prompt (see Appendix 8).

Within the reasoning, we follow the chain-of-thought reasoning trace approach outlined in [68], which includes the sequence of steps to make the decision. This is done by adding "Let us think step by step" at the end of our prompt and providing explanations in our examples (see Appendix 5).

Following the agent framework described in Section 3.2, these systems' decision procedure consist of only one decision cycle that results in making the compliance decision. Furthermore, the planning step in the system is fixed and not made by an LLM. The cycle starts by getting the requirement and test script as inputs, then, depending on the described system configuration, external information or examples are retrieved from semantic memory to the working memory. These are then included in the prompt as described above to elicit the LLM to make a compliance decision, which is the resulting grounding action.

### **LLM Protocol**

Underneath the functionality of inputting prompts as strings to the LLM, additional formatting is needed for each kind of model based on the tokens used for the start and end of the sequence to train it. For the two models we use, our protocols are

defined as follows:

The *MixtralProtocol* & *CodeLlama34Protocol* classes are designed to interact with the Mixtral 7x8B model and CodeLlama-34B, respectively, facilitating communication between the user and the models.

- **Initialization:** Upon initialization, the protocol accepts an optional argument *system\_instruction*, a system prompt or instruction to guide the model’s behavior during interaction. It sets up default roles for the user and AI and start and end tokens for string and instruction sequences.
- **Formatting:** The *format* method constructs a prompt string for the model based on the given user query (*prompt*), chat history (*history*), and primer (*primer*). If a *system\_instruction* is provided, it is incorporated into the prompt to guide the model’s behavior. The primer can provide additional context or constraints for the model’s response.
- **Parsing:** After generating a response, the *parse* method extracts the answer from the model’s generated text. It also formats the text with start and end tokens for string sequences to maintain coherence in the conversation history.

## 4.5 Agent approaches

We denote the following systems as agent approaches, since they include more than one iteration of the decision loop. As described in Section 4.1, all outlined approaches rely on static planning that doesn’t involve LLMs deciding on the next action directly.

### 4.5.1 Function evaluation

To simplify the test script, this approach tries to discard functions deemed useless for making the final compliance decision.

Compared to the systems described in Section 4.4, this approach aims at simplifying the problem before solving it, with the techniques described in Section 4.4. The decision procedure for this system can be divided into two phases. The first phase considers each function of the test script and decides if it is helpful for the final compliance decision, whereas the second phase consists of the final compliance decision as described in Section 4.4.

In the first phase,  $n$  decision loops are executed, where  $n$  is the number of functions in the considered test script. In each decision loop, the LLM is given the requirement and a function out of the test script in addition to the optional context around it. Here, we consider three different systems depending on how much context is included around the considered function. The first version considers the full test script and marks the considered function; however, due to the high computational cost of executing this for each function and such long prompts, we didn’t run experiments for this version. The second version only includes the considered function, which leads to the fastest inference but loses the most information, which might be necessary to make a relevant decision for an individual function. The third version considers a

window of one function before and after the considered function and includes those in the prompt while marking the function of interest. This version is a compromise between the two versions since it includes some context for the function of interest while limiting the prompt length. We run experiments for both version 2 and 3, which are further described in Section 5.6.

After processing the function and its context, the available external information for the considered function is retrieved from semantic memory and saved to working memory. Depending on the used version, prompt Appendix 9, 10 are then used to elicit the LLM to make a decision on whether the considered function can contribute to the final compliance decision. The function is deemed useful and saved to the working memory. The construction of the prompt follows the same principles as described in Section 4.4, with minor changes mainly to the task description.

After every function was evaluated for its usefulness, the second phase begins by reconstructing the test script by concatenating the remaining functions that were saved to the working memory. With this shortened test script, a final compliance decision is made using the same approach as described in Section 4.4.

### 4.5.2 Requirement reformulation

Next to modifying the test script before making the final compliance decision, the requirement is the second textual input that can be modified to impact the system’s performance. This approach follows a model-based approach to reformulate the requirement, given some quality criteria, without changing the semantic content.

Compared to the approaches outlined in Section 4.4, this system introduces an additional decision loop before making the final compliance decision. In this loop, the requirement is given to an LLM that is elicited to reformulate the requirement with the prompt displayed in Listing 11. The prompt follows the outlined prompt engineering principles and includes four criteria derived from the criteria described in Section 2.2. The system then makes the final compliance decision with the reformulated requirement following the process outlined in Section 4.4.

This approach aims at leveraging the language capabilities of LLMs to improve the language of formulated requirements. However, by using an LLM to change the requirement, it is possible that the LLM changes the requirement in ways that change the human-annotated ground truth relationship between the requirement and the test script.

### 4.5.3 Self-Consistency

In our agent framework, self-consistency is utilized as a selectable action to evaluate the consistency of the responses generated. This approach addresses the sensitivity of our models to specific prompt tokens, which led to the implementation of a method where an odd number of responses are generated from a set of slightly varied but semantically equivalent prompts. Having a consistency check as an action allows us to use majority voting to force the most consistent response along with a confidence score in the compliance decision.

---

This aims at improving the decision-making by evaluating and comparing multiple generated responses. This action leads to creation of several versions of a response using different generation configurations, such as changes in temperature and sampling settings, or by simply using semantically similar prompts. By verifying the consistency among these varied responses, the approach helps ensure that the model’s outputs are accurate and not too much dependent on the initial prompt we came up with. This method enhances the overall quality and dependability on the model’s decision.

## 4.6 Baseline

It is common practice to compare machine learning systems to a simple baseline to test if they can outperform a more straightforward approach and justify their increased computational cost and investment effort. We define our baseline as a regular expression-based string match. The baseline extracts all signal names from a test script and makes a positive classification if any of the extracted signals are mentioned in the test script. Otherwise, it makes a negative classification decision.

## 4.7 Failed Approaches

To address the research questions, we experimented with various methodologies and explored different techniques; however, not all were extensively pursued. Time constraints and preliminary assessments of the results with a small dataset led us to prioritize other approaches for our experiments. In the following section, we will enumerate these failed approaches:

- **Requirement Decomposition:** Since the requirements we had, contained multiple causal relationships, we followed the approach proposed by Frattini et al. [93], which aims at breaking down complex causal relationships into simpler ones. We utilized a LLM to decompose more extensive requirements into smaller pieces and attempted to measure the compliance of each sub-requirement. However, this approach encountered several problems. Firstly, verifying if all the sub-requirements generated were accurate representations of the original requirement was challenging. Additionally, qualitative analysis showed a general inability of the model to generate meaningful decompositions. Furthermore, evaluating the compliance of each sub-requirement separately did not necessarily guarantee compliance with the entire requirement.
- **Test Translation to Natural Language:** Checking the compliance of a test script written in a specialized language with a natural language requirement is a complex task involving various sub-tasks. These include understanding the functions within the test script, their sequencing, arguments, parameters, and more. Furthermore, understanding how these components interact within the test script is required. Additionally, one must analyze the requirement within the context of the system and cross-reference it with the understanding derived from the test script. Making decisions based on these tasks in a single prompt input can be challenging for the model. Therefore, we considered translating

our dataset of test scripts into natural language pseudocode-based descriptions to improve the model’s performance when comparing these descriptions against the natural language requirements. However, this approach did not yield satisfactory results as the translation process overlooked minor peculiarities from the test script that could be crucial for ensuring compliance.

- **Test Translation to Python:** Following the same concept as in the translation to natural language, we attempted to convert the test scripts into Python. This decision was based on the understanding that the model has been trained in Python and might be able to make better compliance decisions when presented with test scripts written in Python. However, we encountered issues similar to those with natural language translation. The translated Python scripts often overlooked certain checks, and the generated Python code’s syntax was inconsistent across different test scripts. This inconsistency made constructing prompts challenging for the next step of checking compliance.
- **Exact Signal Name Match:** Based on the assumption that if a signal name mentioned in the requirement is not even mentioned in the test script, we attempted to extract signals from both our test scripts and requirements. Signals from the test scripts were extracted using regular expressions, while signals from the requirements were extracted using a LLM. This decision was made because signals within requirements were not present in a consistent pattern. However, this approach failed due to how the LLM extracted signals from the requirements. The LLM tended to generate signal names based on the information provided in the requirement, even if these signals were not genuine. Consequently, we abandoned this approach.
- **Self-Refine:** This technique involved reflecting on an initial generation, creating feedback based on the initial input and output, and iterating this process to gradually enhance the responses. We explored two different methods for generating feedback. The first method allowed the model to review the initial reasoning and decision-making process, prompting it to identify and correct any issues within these areas over a fixed number of iterations (in our case, five iterations). The second method involved assigning a feedback score to each generation, with the model refining its previous answers until the feedback score reached a specified threshold (for us, 8 out of 10). However, the technique struggled due to the lack of an external feedback source, a limitation highlighted in [94]. Our conjecture suggests that better results could be achieved with access to an execution environment that could provide error tracebacks as feedback. This area remains a promising direction for further research.

# 5

## Results

This chapter describes, analyzes and discusses the conducted experiments in this work. First, the experiment for systems relying on prompting techniques are discussed.

### 5.1 Experimental setup

If not stated otherwise, the dataset used for all experiments is constructed according to the descriptions in Section 4.2.5. Following are the included perturbations:

- RemoveComments
- ReplaceComponent-2: Replaces the *START TEST* component of the test leaving *INITIALIZATION*, *RESTORATION* unchanged.
- ReplaceAllTestFunctions: Replaces every function in the test script.
- ReplaceAllTestParameters: Replaces every parameter in the test script.
- ReplaceAllTestParameters-Signal: Replaces every parameter in the test script except the *Signal* parameter.
- ReplaceTestParameters-Signal: Replaces every *Signal* parameter in the test script.
- ReplaceAllTestArguments: Replaces every argument in the test script.
- ReplaceAllTestArguments-Signal: Replaces every argument in the test script except the *Signal* argument.
- ReplaceAllTestArguments-Comparer: Replaces every argument except the *Comparer* argument in the test script.
- ReplaceAllTestArguments-Value: Replaces every argument except the *Value* argument in the test script.
- ReplaceTestArguments-Signal: Replaces every *Signal* argument in the test script.
- ReplaceTestArguments-Comparer: Replaces every *Comparer* argument in the test script; this changes the comparison operator, for example, from `==` to `<`.

- `ReplaceTestArguments-Value`: Replaces every *Value* argument in the test script, changing the values signals are compared against.

In addition to the 2 sampled negative samples, this leads to 13 perturbed test scripts being added to the dataset for each annotated tuple of test script and requirement. The choice of the transformations is mainly motivated by the observation that the basis of the testing procedure is setting values for specific signals and checking related signals for a specific value change in most test scripts. To probe if systems can detect small changes in this procedure, we replace the three elements involved: the signals, the comparison operator, and the values set or checked.

For the experiments, we fix the model temperature to 1 and use greedy decoding to reduce the search space of potential configurations and to follow the given requirement that the results should be deterministic.

## 5.2 Prompting techniques

This section describes the experiments conducted using systems mainly using prompting techniques and solving the classification task in one prompt. In these experiments, we consider system changes in 3 dimensions: Model type, in-context learning technique, and the usage of external information. With 2 different options for each of these dimensions, this leads to 8 run experiments and 1 experiment run with the baseline system.

The results for the described experimental setup are displayed in Table 5.1, where the metrics are reported for a subset of the dataset, including the original test script and the two sampled negative samples, leading to a 1 : 2 ratio of positive and negative classes. Generally, the results for systems that try to solve the problem in one prompt are poor, and these systems generally do not exceed and mostly have noticeably worse performance than the defined baseline. The only exception is the Mixtral model that uses a Zero-Shot prompt and does not use external information. This model setup outperformed the baseline closely in 3 of the 5 metrics. The generally poor performance of these approaches might indicate that the problem is too complex to be solved in one LLM prompt.

System	Precision	Sensitivity	Specificity	Accuracy	F1-score	Mean prompt length
Mixtral_Zero-Shot	<b>0.66</b>	0.38	<b>0.90</b>	<b>0.73</b>	0.49	2559
Mixtral_Zero-Shot*	0.53	0.31	0.86	0.68	0.39	3493
Mixtral_Two-Shot	0.43	0.43	0.72	0.62	0.43	4984
Mixtral_Two-Shot*	0.35	0.68	0.38	0.48	0.47	5919
CodeLlama_Zero-Shot	0.38	<b>0.96</b>	0.22	0.47	<b>0.55</b>	2669
CodeLlama_Zero-Shot*	0.36	<b>0.96</b>	0.15	0.42	0.53	3644
CodeLlama_Two-Shot	0.40	0.70	0.46	0.55	0.51	5143
CodeLlama_Two-Shot*	0.39	0.89	0.22	0.46	0.54	5809
Baseline	0.6	0.51	0.83	0.72	<b>0.55</b>	

Table 5.1: Results including original and sampled test scripts for different combinations of LLMs, prompting type, and the inclusion of external information. The "\*" indicates whether external information was made available to the model.

### **Influence of LLM Model**

There is a distinct difference between the two used models. Mixtral-based systems make more negative classifications and, thus, generally have higher specificity and precision, while CodeLlama-based systems have higher sensitivity due to generally making more positive classifications. This is the case even having adjusted the prompts to elicit more lenient behavior in the classification for Mixtral and more strict behavior for CodeLlama, as can be seen in the different base prompt for both models in Appendix 2, 3.

### **Influence of prompting type**

The results for the prompting type are less conclusive than for the influence of the model. For Mixtral-based systems, the accuracy degrades when using Two-Shot approaches compared to Zero-Shot approaches, going from approximately 0.73, 0.68 to 0.62, 0.48 respectively. However, the F1-score is slightly reduced for systems that do not use external data. For systems using external data, the F1-score increases, and the general model behavior changes since this is the only Mixtral-based system that gives more positive classifications than negative. For CodeLlama-based systems, the accuracy increases slightly, going from approximately 0.47, 0.42 to 0.55, 0.46, respectively. However, for F1-scores, although smaller and inconclusive, the trends look similar to Mixtral-based systems, where giving examples seems harmful for not using external information and neutral or slightly positive when using external information.

For Mixtral, including examples makes it more likely to give a positive classification, and for CodeLlama, the effect is the opposite, increasing the fraction of negative classification decisions.

Qualitatively reviewing the given reasoning for decisions reveals that the model often conflates the given examples with the actual requirement and test query. In its reasoning it then mixes the requirements together or refers to the example tests.

### **Influence of external information**

The results for including external information are consistent between the two model types. Including external information in a Zero-Shot prompt generally degrades the system’s performance in all metrics. For Mixtral-based systems, this effect is slightly more pronounced than for CodeLlama-based systems. For Two-Shot approaches, the inclusion of external information has a different effect. It increases the fraction of positive classifications for both system types, even reversing the general preference of negative classification for Mixtral-based systems. This leads to increases in sensitivity and decreases in specificity, ultimately leading to a drop in accuracy and slight gains in F1-score.

### **General observations**

Taking into account the mean prompt length for the different prompting techniques, we conjecture that the prompt length or more precisely the ratio of relevant information and irrelevant information/noise might be impacting the performance of these systems.

### 5.3 Model understanding

To better understand how the system makes its decision, we present the analysis of input perturbation impacts for the Mixtral-based Zero-Shot system without external information since it achieves the best precision out of all systems. Table 5.2 displays the results, where columns for *sensitivity* and *specificity* report these metrics for evaluation schemes relying on counterfactual assumption. The columns *global*, *positive*, *negative* report the fraction of label flips. *Positive* and *negative* describe the classification decision for the original test, whereas *global* is the fraction of label flips regardless of the initial decision.

Since the original and test scripts, which were transformed by removing the comments have a positive label, only sensitivity is reported for them. The label for the sampled test scripts is negative, and only specificity is reported. Similarly, for all input transformations except the removal of comments, we assume them to be counterfactuals as argued in Section 4.2.3 and, therefore, having a negative label.

Perturbation	Sensitivity	Specificity	Global	Positive	Negative
original	<b>0.38</b>				
RemoveComments	0.20		0.27	0.58	0.05
ReplaceComponent-2		0.86	0.38	0.80	0.10
ReplaceTestArguments-Signal		0.76	0.23	0.49	0.05
ReplaceAllTestArguments-Signal		0.68	0.19	0.34	0.09
ReplaceTestArguments-Comparer		0.60	0.09	0.13	0.06
ReplaceAllTestArguments-Comparer		0.74	0.27	0.52	0.10
ReplaceTestArguments-Value		0.60	0.16	0.21	<b>0.12</b>
ReplaceAllTestArguments-Value		0.72	0.26	0.49	0.11
ReplaceAllTestArguments		0.78	0.25	0.46	0.10
ReplaceTestParameters-Signal		0.63	0.16	0.24	0.10
ReplaceAllTestParameters-Signal		0.77	0.22	0.49	0.03
ReplaceAllTestParameters		0.79	0.25	0.55	0.04
ReplaceAllTestFunction		0.73	0.19	0.40	0.04
SampleRandomTests		<b>0.91</b>	<b>0.45</b>	<b>0.93</b>	0.11
SampleSimilarTests		0.90	0.39	0.86	0.07

Table 5.2: Results for Mixtral-based Zero-Shot system using no external information, for different test inputs. The columns for *sensitivity* and *specificity* report these metrics for the evaluation scheme relying on a counterfactual assumption. The columns *global*, *positive*, *negative* report the fraction of label flips. *Positive* and *negative* describe the classification decision for the original test, whereas *global* is the fraction of label flips regardless of the initial label.

Generally, the input-dependent results show that the system shows different performances depending on the type of test input provided, which indicates that the system considers particular variables or structures of the test more critical for its decision-making. The label flip results clearly show that label flips primarily occur from a positive initial classification to a negative classification when the test input is transformed or sampled. This strongly indicates that the system classification decisions are not purely random, and the system recognizes that these changes make

the test less likely to comply with the requirement. Furthermore, the results show that a decrease in sensitivity and an increase in specificity positively correlate with an increase in the flip ratio from originally positive to negative classification, indicating that both proposed evaluation methods measure similar effects.

In removing comments, the sensitivity decreases sharply to approximately 0.20 from 0.38 for the original test. This is further underscored by a flip ratio of 0.58 from a positive original classification. This shows that the system is less likely to give a positive classification when comments are removed, indicating that the model is basing part of its decision on the content of comments.

For input changes that result in a label change, the most robust performance is shown for the sampling of random and similar tests as well as the replacement of the *START TEST* component with sensitivities of 0.91, 0.90, 0.86 and flip ratios from positive to negative of 0.93, 0.86, 0.80 respectively. This is expected since these constitute the most sweeping changes compared to the original test script, where either the whole test script or the most important and most significant part of the test script is completely replaced. The performance of the sampling of similar tests, which have at least one signal in common with the original test, is only slightly behind the performance of randomly sampled tests.

Analyzing the three elements of the predominant testing procedure, *Signal*, *Comparer*, *Value*, shows that they have different impacts on the system decisions. To probe this, transformations that only replace the argument for a specific element or all other elements except the specified element are used. The results show that *Signal* is decidedly the element with the biggest impact. Replacing only *Signal* shows a label flip ratio of 0.49 compared to 0.13, 0.21 for *Comparer*, *Value* respectively, considering flips from positive to negative. Additionally, when replacing all arguments except *Signal*, the system shows the lowest positive flip ratio of 0.34 compared to 0.52, 0.49 for *Comparer*, *Value* respectively. This clearly indicates that from these three elements, the system is primarily focused on the presence of the Signal and, to a lesser extent, how and to what value it is compared to. Surprisingly, replacing all arguments in the test script has a slightly lesser effect than keeping the *Comparer* or *Value* argument.

To probe the impact of parameter changes in the test script, we mostly focus on the parameter *Signal*. Only replacing the *Signal* parameter shows a diminished impact, with a specificity of 0.63 and a positive flip ratio of 0.24, compared to replacing the actual signal name, which is the argument. This further indicates that the system is more focused on the presence of a specific signal name than its proper usage in the test. However, the proper usage of parameters, in general, has a sizable impact on the system decision with positive flip ratios of 0.49, 0.55 for all parameters except *Signal* and all parameters, respectively, which is higher than the replacement of all arguments with a positive flip ratio of 0.46.

The lowest impact out of all three main components: *arguments*, *parameters*, *functions*, seem to be the functions with the lowest positive flip ratio of 0.4. This can be interpreted as an indication that the system is not primarily focused on the language aspect of the test script and its proper usage.

Since the external knowledge available mainly covers valid signal values for the CanDB and function information for the PnTool documentation, using the stratified

analysis, it is possible to gain some insights into whether the system uses this information. Comparing the results of the Mixtral-based Zero-Shot system without external information to the one using external information, some differences can be identified. Although, as established, the system using external information generally performs worse, it shows improvements for transformations concerning the *Value* argument. Specifically, the specificity and positive flip ratio increase starkly from 0.60, 0.13 to 0.70, 0.45, respectively. However, the negative flip ratio also increases from 0.12 to 0.17, which can likely be attributed to a general increase in the negative flip ratio for this system. This can be interpreted as indicating that the external information positively changes the system’s behavior for the specific information it provides. For the function information in the PnTool documentation, no such notable change can be observed for the transformation concerning the replacement of all functions.

### 5.4 Requirement quality

Due to the deficiencies discovered in the data review process regarding requirements, we probe how the quality of requirements affects system performance. Table 5.3 shows the results for or a subset of the dataset, including the original test script and the two sampled negative samples. The results are stratified by the quality of the requirements in the dataset described in Section 4.2.2 for requirements from the entire dataset or given for the manually written requirements described in Section 4.2.4. The results clearly show that the higher the quality of the requirements, the better the system performs. For the quality levels in the given dataset, gradual improvements in all metrics are observable. An especially large improvement in sensitivity occurs for the manually written requirements, while the specificity decreases slightly. These results indicate that the quality of the requirements can significantly impact the system’s performance. Additionally, results are reported for the system described in Section 4.5.2, where requirements are first reformulated by the system given some quality criteria before the compliance decision is made. These results are marked as "model-based" in Table 5.3. The results for the model-based reformulation show a generally worse performance compared to their unaltered counterparts. This can be explained by the model not having access to additional information that would enable it to improve the requirement noticeably. Just providing a selection of requirement quality criteria is not sufficient to improve the requirement and largely results in a simple rephrasing of the requirement. Furthermore, although LLMs are generally well versed at rephrasing text passages, the human-annotated ground truth relationship between requirement and test script can get jeopardized by this process, posing a risk to the integrity of the analysis.

### 5.5 Synthetic data

To probe the developed conjecture that prompt length and requirement complexity impact the performance of the developed systems, we conduct experiments with synthetic data generated as described in Section 4.2.4. The experimental setup

Requirement Data	Precision	Sensitivity	Specificity	Accuracy	F1-score
Full	0.66	0.38	0.90	0.73	0.49
Full model-based	0.58	0.36	0.87	0.70	0.44
Quality 3	0.58	0.34	0.88	0.70	0.43
Quality 2	0.68	0.39	0.91	0.73	0.49
Quality 1	0.70	0.40	<b>0.91</b>	0.74	0.51
Manual	<b>0.71</b>	0.62	0.87	<b>0.79</b>	<b>0.67</b>
Manual model-based	0.63	<b>0.63</b>	0.81	0.75	0.63

Table 5.3: Results for Mixtral-based Zero-Shot system using no external information including original and sampled test scripts for different requirement data.

matches use the same perturbations with a minor adaption for the sampling of similar test scripts. Due to the shortness of the extracted sequences and the similarity criterion of sharing at least one signal name, it is possible for a similar test to cover the same requirement as the original test. To ensure that such overlaps do not occur, we check that the sampled similar test is different in at least one signal name or value. If that is not the case, a random signal name in the sampled test will be replaced. We generate datasets for complexity levels 1, 2, 4, where the complexity refers to the number of concatenated test sequences, and with and without noise, where noise refers to the placement of the generated test sequence into an unrelated test script. We probe Mixtral-based systems either with or without the inclusion of external information. This leads to 12 combinations of systems and datasets. Table 5.4 shows these experiments’ results.

External information	Noise	Complexity	Precision	Sensitivity	Specificity	Accuracy	F1-score	Mean prompt length
False	False	1	0.83	<b>1</b>	0.90	0.93	0.91	651
False	True	1	0.71	0.66	0.86	0.79	0.68	2458
True	False	1	0.79	<b>1</b>	0.87	0.91	0.89	1003
True	True	1	0.71	0.82	0.83	0.82	0.76	3224
False	False	2	0.88	0.99	0.93	0.95	0.93	871
False	True	2	0.75	0.79	0.87	0.84	0.77	2775
True	False	2	0.84	<b>1</b>	0.90	0.93	0.91	1277
True	True	2	0.69	0.87	0.80	0.83	0.77	3766
False	False	4	<b>0.93</b>	0.99	<b>0.96</b>	<b>0.97</b>	<b>0.96</b>	1326
False	True	4	0.79	0.94	0.87	0.90	0.86	3143
True	False	4	0.85	<b>1</b>	0.91	0.94	0.92	1929
True	True	4	0.75	0.92	0.85	0.87	0.83	4399

Table 5.4: Performance results for Mixtral-based systems applied to synthetically generated data. The noise column indicates whether the generated test was injected into a longer random test script and the complexity column indicates the number of consecutive test sequences as described in Section 4.2.4.

Generally, the systems used on synthetic data vastly outperform those used on the original and manually created datasets. This is unsurprising since the requirements are vastly simplified and explicitly name every relevant signal and the associated value. Additionally, the test scripts are vastly simplified for some synthetic datasets, containing only a single test sequence for setting and checking a signal value in extreme cases. The best-performing system didn’t make use of external information and achieved an F1-score of approximately 0.961 on a dataset that did not include

noise and had a test sequence complexity of 4.

### **Influence of noise**

The results show that noise is the most influential factor in this experiment, with the F1-score degrading by at least 0.10 when adding noise independent of the other probed factor. Additionally, the mean prompt length increases substantially and at least doubles when applying the noise procedure. This shows that the test script is the most influential factor in the prompt length. Additionally, these results support the conjecture that the prompt length impacts the systems' performance. We conjecture that the noise around the relevant test sequence makes it harder for the model to identify the relevant information to make good decisions. Additionally, the impact of the noise decreases slightly the longer the initial test sequence is. This could indicate that the ratio between relevant information and noise might also be an influencing factor in addition to the absolute length increase.

### **Influence of complexity**

The results suggest that an increase in complexity also increases performance. This surprising observation can partly be explained by how similar tests are sampled (described above) since, for low complexities, the probability of a similar test being very close to an original test is higher and often only differs in one signal name. However, the trend of increasing complexity resulting in increased performance is also shown for datasets with added noise. The ratio of essential and noisy data might be an influencing factor. This could be further encouraged by injecting all test sequences in a consecutive block into the noise test script, and further experiments are required to confirm if this phenomenon also occurs when the test sequences are spread throughout the noise. Regardless of these factors, the results suggest that appending unrelated requirements does not majorly effect the performance of the systems, and other types of complexity need to be explored to further the understanding of what makes a requirement complex for an LLM. Additionally, these experiments highlight the limitations of the chosen performance evaluation scheme, which is problematic for very short test script and leads to different difficulties in the task.

### **Influence of external information**

The results for the inclusion of external information are not conclusive. For non-noisy data, the inclusion of external information degrades performance and, additionally, increases the prompt length by approximately 50%. For noisy test scripts, including external information improves performance for the least complex test sequences but degrades performance for the most complex explored test sequences. Generally, external information increases the sensitivity of the system.

### **Model understanding**

To further deepen the understanding of the model decision making, we analyze the model using the defined input perturbations similarly to Section 5.3. In contrast to the performed analysis, and mainly for brevity purpose, we focus on datasets without noise and complexities 1, 4 as well as restrict the analysis to the sensitivity

and specificity. Table 5.5 displays the results.

Perturbation	Complexity 1		Complexity 4	
	Sensitivity	Specificity	Sensitivity	Specificity
original	<b>1</b>		<b>0.99</b>	
RemoveComments	<b>1</b>		<b>0.99</b>	
ReplaceComponent-2		0.98		0.95
ReplaceTestArguments-['Signal']		0.97		<b>1</b>
ReplaceAllTestArguments-['Signal']		0.99		0.90
ReplaceTestArguments-['Comparer']		0.34		0.05
ReplaceAllTestArguments-['Comparer']		<b>1</b>		<b>1</b>
ReplaceTestArguments-['Value']		0.52		0.16
ReplaceAllTestArguments-['Value']		<b>1</b>		<b>1</b>
ReplaceAllTestArguments-[]		<b>1</b>		<b>1</b>
ReplaceTestParameters-['Signal']		0.01		0
ReplaceAllTestParameters-['Signal']		0.14		0.11
ReplaceAllTestParameters-[]		0.16		0.10
ReplaceAllTestFunction		0		0
SampleRandomTests		0.96		0.97
SampleSimilarTests		0.84		0.94

Table 5.5: Results for the Mixtral-based Zero-Shot system using no external information, for different test inputs. The columns for *sensitivity* and *specificity* report these metrics for the evaluation scheme relying on a counterfactual assumption.

Compared to the results for the real dataset analyzed in Section 5.3, we generally see more extreme results for the synthetic data. The real dataset specificity values are mainly between 0.60 and 0.90, whereas specificity ranges from 0 to 1 for the synthetic dataset. The sensitivity is also significantly higher and approaches 1 for some perturbations. Additionally, the system does not seem to rely on information in comments for the simplified synthetic data since the sensitivity does not show a decrease for the *RemoveComments* perturbation in contrast to the real dataset, which showed a drop of sensitivity of nearly half for this perturbation. This can be explained by the far more precise and complete requirements for the synthetic dataset, making it easier for the system to extract all the necessary information from the test code.

The trends identified in Section 5.3 are generally mirrored and amplified for the synthetic data. The system is most sensitive to changes in the signal names, showing specificities approaching 1 for perturbations that replace signal names. For *Comparer* and *Value*, the specificity is significantly lower than *Signal*, supporting the conjecture that the system is primarily sensitive to the correct signal names being present. Here, similarly to Section 5.3, the impact of the value compared with seems to be higher than the comparison operator with specificities of 0.52 and 0.34, respectively. More language-based perturbations, like replacing parameters and functions, have a significantly smaller impact on the synthetic data than the real data. The specificities for these perturbations range between 0.16 and 0, indicating that the system does not consider these changes when making a compliance decision. This starkly contrasts the

real data, which showed comparable values between the main components: *arguments*, *parameters*, *functions*. We argue that the synthetic results give a more realistic indication of the system’s capabilities since the system results in these simplified scenarios are more reliable and less noisy than the real data, where the systems did not outperform the defined baseline. Additionally, and in contrast to the conjecture and observation made in Section 5.3, the inclusion of external information does not improve the results for the related perturbation, i.e., the replacement of values and the replacement of function names. This indicates that the general length of the prompt is not the primary reason for the ineffective inclusion of external information, and other factors might impact it.

Comparing the two complexity levels for synthetic data without noise shows two distinct trends. The lower complexity data set performs worse for sampling similar tests with a specificity of 0.84 compared to 0.94 for complexity 4. This can be explained by the sampling procedure as explained above. Additionally, this also supports the explanation for the counter-intuitive performance differences between lower and higher complexities for noiseless data. Considering the perturbation for *Signal*, *Comparer*, *Value*, it can be observed that the complexity does not strongly impact the performance regarding the signal names. However, a significant change can be observed for the comparison operator and the value argument. The replacement of comparison operator and value shows considerably higher specificity for complexity 1 of 0.34, 0.52, respectively, compared to 0.05, 0.16 for complexity 4. A similar albeit lesser trend can be observed for language-based perturbations altering parameters and functions. This indicates that in contrast to what the performance analysis suggests, the system performance is more sensitive to the generally less impactful perturbations and, thus, performs more reliably for lower complexities.

## 5.6 Function evaluation

Following the observation that test length impacts the system performance, we report results for the function evaluation agent described in Section 4.5.1. Table 5.6 displays the results for the function evaluation agents compared to the simpler prompting techniques discussed in Section 5.2. Here, the results for original requirement dataset and the manually created requirements are both reported. Due to the third version of the function evaluation agent, which considers a slice of functions, consistently outperforming the second version, which only considers a singular function, the reported results refer to the slice version.

Generally, the function evaluation agent is effective at reducing the length of the test script and consequently the prompt length for both datasets. The mean prompt length is approximately halved for the final decision with prompt lengths of 2559, 2104 for original and manual requirement datasets respectively, and 1114, 1074 for the respective prompts shortened by the function evaluation.

For the original dataset, the function evaluation shows a drop of sensitivity by approximately half from 0.38 to 0.19, leading to a substantial drop in F1-score from 0.49 to 0.28. Additionally, the precision and accuracy drop slightly, while the specificity increases slightly. This indicates that for this dataset the agent is

not successful at identifying which functions are important and relevant for the final compliance decision and too much information is lost by discarding relevant functions.

In contrast, for the manual dataset the function evaluation only results in a slight decrease in sensitivity from 0.62 to 0.59, while all other metrics increase slightly. This indicates that the approach of simplifying the test script by discarding irrelevant information shows some promise although it comes at a significant computational cost, due to prompting the LLM for each function in addition to the final compliance decision prompt.

System	Data	Precision	Sensitivity	Specificity	Accuracy	F1-score	Mean prompt length
Mixtral_Zero-Shot	Original	0.66	0.38	0.90	0.73	0.49	2559
Mixtral_Zero-Shot_FuncEval	Original	0.6	0.19	<b>0.94</b>	0.69	0.28	1114
Mixtral_Zero-Shot	Manual	0.71	<b>0.62</b>	0.87	0.79	0.67	2104
Mixtral_Zero-Shot_FuncEval	Manual	<b>0.8</b>	0.59	0.93	<b>0.81</b>	<b>0.68</b>	1074

Table 5.6: Results comparing the function evaluation agent with the simple prompting technique.

## 5.7 Self-Consistency

To achieve more reliable results, we conduct experiments employing self-consistency and determining the final compliance decision with a majority vote. Due, to the experiments with varied generation configurations not leading to disagreements in the results, we report the results obtained by using 5 different base prompts to elicit a compliance decision as described in Section 4.5.3. Table 5.7 shows the results for the Mixtral zero-shot system using no external information and making use of self-consistency. The *Confidence* column indicates the minimum amount of agreement when conducting the majority vote.

Confidence	Precision	Sensitivity	Specificity	Accuracy	F1-score
3/5	0.39	0.09	0.93	0.65	0.14
4/5	0.37	0.09	0.94	0.68	0.14
5/5	0.36	0.07	0.95	0.70	0.12

Table 5.7: Results for the Mixtral\_Zero-Shot system using self-consistency and stratified by the minimum agreement for the compliance decision.

The results show a significant degradation of most metrics compared to the simple prompting system. Especially the sensitivity decreases from 0.38 to approximately 0.09. This also affects the F1-score, which decreases from 0.49 to around 0.14. This could be explained by a selection of the prompts being formulated in a way that elicits stricter behavior by the model, underscoring how sensitive the models are to prompts and, thus, how important proper prompt calibration is. Although the self-consistency agent shows stricter behavior, meaning that it generally produces more negative decisions, the precision of the agent does not increase but decreases significantly as well. This trend continues when the required confidence for a decision

increases, with continual decreases in sensitivity, F1-score, and precision.

However, self-consistency improves the specificity compared to the simple prompting and with higher confidence thresholds. This indicates that the stricter base prompts are leveraged to make negative decisions more likely, with a higher proportion of negative samples also increasing accuracy.

These experiments show that the choice of changes in parameters and prompts has to be well-calibrated to be successful and that the inferred confidence from the voting proportions is not a reliable indicator of uncertainty in this case.

### 5.8 Natural language explanations

Next to the quantitative analysis, we also qualitatively analyze the reasoning given by the system every time a final compliance decision is provided. Due to the amount of experiments and results we selectively review a subset of the provided explanations and try to draw conclusions and identify common failure modes from them.

In almost all cases, the models are producing coherent and plausible explanations, which in all reviewed cases are consistent with the compliance decision. In some cases the explanations end in the middle of the explanation leading to unfinished sentences and incoherence. However, in most cases the explanations are highly convincing on first sight seemingly disagreeing with the observed quantitatively weak performance of the probed systems. Further analysis reveals the weaknesses and problems of the systems and provided explanations though.

The models struggle to match occurring signal names to the requirements if they are not explicitly named. An example for this behavior is the conflation of the "Bi-Xenon Light Shutter", which is named in the requirement, with a signal referring to the direction indicator lights. In this example the model assumes them to be equivalent, although the required signal explicitly contains "Bi-Xenon" in its name. This behavior underscores the difficulty of the problem, especially when lacking domain specific expertise that can resolve such nuances, since it can be seen as reasonable to assume by the model that the indicator lights is a more generally phrased signal that could also cover Bi-Xenon lights.

Next to the name matching, the explanations also show reasoning errors. These reasoning errors are mostly related to correctly interpreting timings and sequence steps. Examples for these are tests, where changes shall occur after a specified time and the relevant signal is tested shortly before the time is reached to see if the original value is still active and shortly after the time to see if a change occurred. In those scenarios the models look for an exact matching wait time and don't consider the timing of adding the intermediate step. Other common failure examples are occurring when the model has to keep track of previous signal states and conditionally perform actions based on those. These examples show two major challenges. Firstly, the models struggle with reasoning and secondly, the models can't resolve the ambiguity of natural language, that always relies on interpreting the writers intention.

Additionally, the models hallucinate signal names in some cases, where they give plausible explanations regarding a signal mentioned in the requirement, which is not present in the test script at all.

Comparing the generated explanations between different input perturbations shows

that the reached reasoning traces are wildly different unrelated to the actual applied perturbation. For example when replacing all signal values one would expect for the explanation to take into account this change and refer to it, but instead commonly completely different explanations are generated. This underscores how sensitive the models are to small changes in the input and that these changes can lead to unpredictable results.

Regarding the effect of different prompting type, we can observe that the models tend to conflate the given examples with the actual query and solely refer to contents in the examples in their explanations or combine information from the examples with the actual query.

For the inclusion of external information, the explanations often explicitly refer to the external information, however the proper usage of the external information is lacking, which is reflected in the quantitative performance analysis.

All in all the qualitative analysis highlights and empirically supports the described characteristics of LLM-based natural language explanations. They are convincing and appear to be plausible, but when investigating further they tend to be incorrect or miss some critical nuance regularly.



# 6

## Conclusion

### 6.1 Discussion

This section discusses the extracted experimental results and infers potential reasons for the results.

Generally, the results from our technique of trying to solve the problem in one prompt tend to yield less robust results than those of a more straightforward approach, such as our baseline of matching signal names.

We conjecture that two main reasons exist for the limited performance of one-prompt systems. First, the developed systems lack knowledge about the vehicle as a system of interconnected ECUs and signals and their relationships. For making reliably correct classification decisions, the system must understand how different signals interact to fully grasp what a requirement entails and how it is modeled in a test script. For example, when a requirement asks for a specific behavior when *signal A* is activated, it usually does not explicitly state which signals an activation effect, which has to be tested inside the test script. Due to not having access to external information that models these relationships, the implemented systems have to rely on semantic name similarities or assumptions based on their pretraining to decide if the correct signals are checked. However, it is doubtful that the pretraining data included this kind of domain- and even company-specific knowledge.

Next to the conjectured domain-specific knowledge deficiencies, we argue that the reasoning required to make correct classification decisions exceeds the reasoning difficulties in standard reasoning benchmarks, making this task substantially more complex than the reasoning task against which these LLMs are tested. This is mainly due to the following factors. The questions in reasoning benchmarks are commonly short [95], require commonsense reasoning [95]–[97], or are from a general domain such as mathematics, law or physics [98]. Additionally, specific benchmarks are aimed at long context understanding [99], [100]. However, the tasks in these benchmarks require less reasoning capabilities and are mainly focused on retrieving information from different parts of the input. The problem of deciding if a test script covers a requirement requires reasoning in multiple steps and along different dimensions; for example, temporal reasoning is required to understand the order of sequence steps, mathematical reasoning is required to, for instance, correctly add wait times, and semantic reasoning is required to connect related signals and how functions interact with them [101]. This is combined with long inputs, and tests are formulated in

a language not represented in the pretraining of the used LLM. Furthermore, as outlined above, the reasoning has to be performed in an unknown domain.

The experiments with synthetic data support the conjecture that the length of the test script and, thus, prompt seems to have a significant impact on the system performance even for the simplified case. Addressing the prompt length with more agent-like iterative approaches, revealed that in most cases the iterative solutions suffer from accumulating errors making them perform worse than the simpler techniques.

To alleviate some of these problems, in-context learning techniques or RAG are commonly employed to support the model in reasoning tasks and adapt to unknown domains. Contrary to other works, the results do not improve in this work when using these approaches. For the applied in-context learning approach, i.e., providing two examples with reasoning traces, we conjecture that the system overfits to the provided examples since the task is not well represented in the pretraining and the examples add substantial length to the input, which generally degrades model performance [99]. Although adding external information suggests improvements in specific areas, it generally also degrades performance. The experiments with synthetic data suggest, that the prompt length alone is not responsible for the ineffective usage of the external information, calling into question the general usefulness of the provided information for the model. Additionally, the formatting and selection of external information presented to the model can be influential factors that need further exploration.

The experiments also showed that different LLMs show vastly different behavior regarding their classification and use, more specifically some model types are more prone to producing positive classifications, while others show the opposite behavior and produce generally more negative classifications. Furthermore, the explored LLMs proved to be sensitive to small changes in the input prompts, this is especially visible in the self-consistency experiments with different prompts. The systems also showed varying responsiveness to different input perturbations. These aspects suggest that the robustness of the probed LLMs is still not comparable to human-level robustness, making the proper usage of these systems an engineering challenge. However, the different behaviors of models and their selectively superior performance compared to the baseline also offer a chance in their usage. The systems have applications in being a preliminary step in a more layered and interactive validation process. By using the systems as an initial filtering step or providing compliance suggestions, they can assist developers in effectively speeding up and guiding the solely manual process. With its high specificity, precision and accuracy, the Mixtral Zero-Shot system can for example be leveraged to provide an initial decision before the final decision is being made by a developer.

## 6.2 Summary

This work explores the application of LLMs to automate the process of ensuring requirement compliance in automotive software integration testing. As the automotive industry increasingly incorporates intelligent technologies, the complexity of testing for safety and functionality has grown, making traditional manual compliance methods

time-consuming and prone to errors. This study aims at addressing these challenges by leveraging the capabilities of LLMs to interpret and verify the compliance of test scripts with given natural language requirements. Guided by the literature review, we propose an evaluation scheme that gives insights into the general performance of systems and explores limitations by leveraging targeted input perturbations of the test language. Using the proposed evaluation scheme, standard prompting techniques that utilize in-context learning and RAG and different open-source models are experimented with. The results indicate that the available open-source models are not yet suited to fully solve a domain-specific task requiring strong reasoning over long contexts. However, they can potentially be employed as assistants, helping developers by providing initial compliance suggestions.

Additionally, we show that these systems are generally sensitive to small changes in the prompt and exhibit different behavior for distinct input perturbations. In contrast to the literature, our experiments do not show improvements when using in-context learning or including external knowledge, stressing that these techniques are not always beneficial in specific domains. In further experiments with synthetic data, we show that the length of the test scripts is an influential factor for the system performance as the performance degrades considerably, even for extremely simplified test sequences and requirements when the noise in the form of unrelated test code is introduced. Building on the prompting techniques, we also explore more agent-like LLM-based systems that produce a compliance decision in multiple steps. The results for the agent systems generally do not improve upon the simpler prompting techniques and, although showing some promising research avenues, suffer from accumulating errors in the intermediate steps.

### 6.3 Limitations

The conducted project has some limitations and constrains, which are listed and discussed in this section.

Due to the available data and project context, all analyses were done solely on a text basis. This means the test scripts were provided solely as text files, and no access to an execution environment or the language engine existed. This constrained the exploration of the language space, and no intermediate feedback through the execution of the test scripts or parts of them was available.

The problem statement of deciding the compliance of a test script with a requirement has been simplified from considering a whole corpus of test scripts and requirements. This was mainly done for two reasons. The compliance check of one test script with one requirement can be seen as a subproblem of considering the whole corpus when tests cover requirements in full, which is the case for the provided dataset. We argue that first, showing that the sub-problem is solvable is a necessary step to scale the system in the future, and in this case, the sub-problem is not trivial in itself. The length of the sub-problems in tokens is substantial, making considering the whole or a subset of the corpus at once not feasible with the capabilities of currently available open-source LLMs.

Due to the substantial time requirements of experiments with LLMs, the search space for different hyperparameters and prompt combinations had to be limited. Here, we

focused on the most promising aspects to improve the system’s performance guided by the literature review, requirements of the supervisors, and observations made in limited qualitative trials during the development process. Additionally, a similar balance has to be found between the analysis of achieved results and the exploration of further techniques to improve the system. Section 6.5 further describes future research directions that were not explored or can be extended.

### 6.4 Contribution

This work can be classified as an applied research problem. Our main contributions are twofold. First, we propose an evaluation scheme for LLMs used in classification problems that makes use of the input structure to apply targeted input perturbations and probe their effect on the system output. Using this evaluation scheme, we probe distinct system configurations using different LLMs, in-context learning techniques, and the inclusion of external knowledge in an RAG approach for a specific domain and use case. We show that the explored systems exhibit distinct behaviors and react differently to a variety of input perturbations but are incapable of reliably solving the complex task of deciding if a test script in the automotive domain formulated in an unknown language to the model is compliant with a requirement specified in natural language. However, with selectively good performance in some metrics, the systems can be employed as assistants that help developers by making suggestions. In further analysis with synthetic data we show that the length of the test scripts and, thus, prompts is a major factor in the performance of systems.

### 6.5 Future research

Although this work showed some promise for LLM-based approaches in determining the compliance of domain-specific tests with natural language requirements, several research directions and extensions remain unexplored.

With in-context learning and RAG approaches not leading to performance improvements like those common in the literature, further exploration is required to leverage these techniques better. Possible avenues to improve the usage of these techniques are the formatting and placement of the additional information in the prompts and selectively choosing information that is most relevant to the problem. Additionally, the agent approach to the system design can be further extended and a reasoning step introduced after external information is retrieved. This reasoning step could give the system more direction in what information to use and how it can be useful for the compliance decision. This would provide further insights into how these techniques are generally regarded as valuable and can be adapted for specific domains.

Additional experiments could also explore more and different hyperparameter choices. In addition to newly released LLMs, this work excluded hyperparameters, like temperature or decoding techniques, that can be tested for their impact on the system performance.

Although computationally expensive, with different LLMs showing distinct behavior, combining them to form an ensemble that could leverage their differences to provide

better performance.

With prompt length being conjectured as one of the challenges of this use case, approaches that decompose the full task are a direction to explore. Although some attempts have been made in this work regarding task decomposition, further research is required. This is especially true for the decomposition of the requirement into distinct sub-requirements, which can be evaluated by themselves, reducing the required reasoning capabilities. Next to the decomposition of the requirement, the test script can also be a decomposition target. Although the function evaluation revealed some promise as an approach to discard unnecessary information, further research is needed to improve upon the results. Possible directions for this research could be the exploration of the required context required to make a good evaluation decision, focusing on the trade-off between performance and computational cost. Additionally, more fine-grained aspects could be considered when making the decision. The decision could contain multiple parts covering if the function is necessary, sufficient or even contrary to the fulfillment of the requirement.

This approach could further be extended to the general framing of the problem by dropping the binary classification approach and either introducing more classes, such as "partly fulfilled" or "uncertain", or taking a more interactive approach, where the system supports a developer in an interactive "conversation" to iteratively solve the problem guided by the developers feedback.



# Bibliography

- [1] K. Juhnke, M. Tichy, and F. Houdek, “Challenges concerning test case specifications in automotive software testing: Assessment of frequency and criticality,” *Software Quality Journal*, vol. 29, no. 1, pp. 39–100, Mar. 2021, ISSN: 1573-1367. DOI: 10.1007/s11219-020-09523-0. (visited on 11/23/2023).
- [2] ISO, *Road vehicles – Functional safety*, Norm, 2011.
- [3] V. Q. W. G. 1. / A. SIG, *Automotive spice: Process reference model process assessment model. 3rd edn.* 2017.
- [4] W. X. Zhao, K. Zhou, J. Li, *et al.*, “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, 2023.
- [5] A. Fan, B. Gokkaya, M. Harman, *et al.*, “Large language models for software engineering: Survey and open problems,” *arXiv preprint arXiv:2310.03533*, 2023.
- [6] R. Fernandes and A. Cowie, “Capturing informal requirements as formal models,” Ph.D. dissertation, Deakin University, 2004.
- [7] I. Drave, S. Hillemacher, T. Greifenberg, *et al.*, “SMArDT modeling for automotive software testing,” *Software: Practice and Experience*, vol. 49, no. 2, pp. 301–328, 2019, ISSN: 1097-024X. DOI: 10.1002/spe.2650. (visited on 11/23/2023).
- [8] S. P. Miller, A. C. Tribble, and M. P. Heimdahl, “Proving the shalls,” in *International Symposium of Formal Methods Europe*, Springer, 2003, pp. 75–93.
- [9] A. Marchetto, G. Scanniello, and A. Susi, “Combining Code and Requirements Coverage with Execution Cost for Test Suite Reduction,” *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 363–390, Apr. 2019, ISSN: 1939-3520. DOI: 10.1109/TSE.2017.2777831. (visited on 11/24/2023).
- [10] OpenAI, *Chatgpt*, <https://openai.com/blog/chatgpt>, 2022. (visited on 11/24/2023).
- [11] J. Achiam, S. Adler, S. Agarwal, *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [12] B. Roziere, J. Gehring, F. Gloeckle, *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [13] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: Survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, 2024.
- [14] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, “Towards generating functionally correct code edits from natural language issue descriptions,” *arXiv preprint arXiv:2304.03816*, 2023.

- [15] J. Fischbach, J. Frattini, and A. Vogelsang, "Cira: A tool for the automatic detection of causal relationships in requirements artifacts," in *Joint Workshops of the 27th International Conference on Requirements Engineering, REFSQ 2021-OpenRE, Posters and Tools Track, and Doctoral Symposium, Essen, Germany, 12 April 2021*, CEUR-WS, vol. 2857, 2021.
- [16] M. Tufano, S. Chandel, A. Agarwal, N. Sundaresan, and C. Clement, "Predicting code coverage without execution," *arXiv preprint arXiv:2307.13383*, 2023.
- [17] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale, "Software testing: Survey of the industry practices," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, IEEE, 2018, pp. 1449–1454.
- [18] M. Tuteja, G. Dubey, *et al.*, "A research study on importance of testing and quality assurance in software development life cycle (sdlc) models," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 3, pp. 251–257, 2012.
- [19] B.-Y. Tsai, S. Stobart, N. Parrington, and B. Thompson, "Iterative design and testing within the software development life cycle," *Software Quality Journal*, vol. 6, pp. 295–310, 1997.
- [20] K. E. Wiegers and J. Beatty, *Software requirements*. Pearson Education, 2013.
- [21] J. Y. Zhou, "Functional requirements and non-functional requirements: A survey," Ph.D. dissertation, Concordia University, 2004.
- [22] S. W. Ali, Q. A. Ahmed, and I. Shafi, "Process to enhance the quality of software requirement specification document," in *2018 International Conference on Engineering and Emerging Technologies (ICEET)*, IEEE, 2018, pp. 1–7.
- [23] T. ur Rehman, M. N. A. Khan, and N. Riaz, "Analysis of requirement engineering processes, tools/techniques and methodologies," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 5, no. 3, p. 40, 2013.
- [24] M. Daun, A. M. Grubb, V. Stenkova, and B. Tenbergen, "A systematic literature review of requirements engineering education," *Requirements Engineering*, vol. 28, no. 2, pp. 145–175, 2023.
- [25] S. Balaji and M. S. Murugaiyan, "Waterfall vs. v-model vs. agile: A comparative study on sdlc," *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.
- [26] T. Shah and S. Patel, "A review of requirement engineering issues and challenges in various software development methods," *International Journal of Computer Applications*, vol. 99, no. 15, pp. 36–45, 2014.
- [27] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, 1998. DOI: 10.1109/IEEESTD.1998.88286.
- [28] "Ieee guide for the use of ieee standard dictionary of measures to produce reliable software," *IEEE Std 982.2-1988*, pp. 1–153, 1989. DOI: 10.1109/IEEESTD.1989.122630.
- [29] L. Rosenberg, "Generating high quality requirements," in *AIAA Space 2001 Conference and Exposition*, 2001, p. 4524.

- 
- [30] G. Génova, J. M. Fuentes, J. Llorens, O. Hurtado, and V. Moreno, “A framework to measure and improve the quality of textual requirements,” *Requirements engineering*, vol. 18, pp. 25–41, 2013.
- [31] A. Shaout and S. Pattela, “Model based approach for automotive embedded systems,” in *2021 22nd International Arab Conference on Information Technology (ACIT)*, IEEE, 2021, pp. 1–7.
- [32] N. Nowdehi, A. Lautenbach, and T. Olovsson, “In-vehicle can message authentication: An evaluation based on industrial criteria,” in *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*, IEEE, 2017, pp. 1–7.
- [33] E. Bringmann and A. Krämer, “Model-based testing of automotive systems,” in *2008 1st international conference on software testing, verification, and validation*, IEEE, 2008, pp. 485–493.
- [34] S. S. Barhate, “Effective test strategy for testing automotive software,” in *2015 International Conference on Industrial Instrumentation and Control (ICIC)*, 2015, pp. 645–649. DOI: 10.1109/IIC.2015.7150821.
- [35] K. Juhnke, M. Tichy, and F. Houdek, “Challenges concerning test case specifications in automotive software testing: Assessment of frequency and criticality,” *Software Quality Journal*, vol. 29, pp. 39–100, 2021.
- [36] C.-V. Briciu, I. Filip, and F. Heininger, “A new trend in automotive software: Autosar concept,” in *2013 IEEE 8th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2013, pp. 251–256. DOI: 10.1109/SACI.2013.6608977.
- [37] *Autosar software architecture*. [Online]. Available: <https://www.nxp.com/design/design-center/software/automotive-software-and%20tools/autosar:AUTOSAR-HOME> (visited on 03/25/2024).
- [38] N. Espfors, “Canoe-simulink integration of vehicle model in existing test environment,” Lund University, 2018.
- [39] H. Chen and J. Tian, “Research on the controller area network,” in *2009 International Conference on Networking and Digital Society*, IEEE, vol. 2, 2009, pp. 251–254.
- [40] J. Persson, “Continued integration of simulated vehicle model for software verification in canoe and simulink,” Lund University, 2020.
- [41] R. Polizzano, “Model-based design for automotive control unit,” Università di Pisa, 2015.
- [42] W. X. Zhao, K. Zhou, J. Li, *et al.*, *A survey of large language models*, 2023. arXiv: 2303.18223 [cs.CL].
- [43] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [44] M. Shanahan, “Talking about large language models,” *Commun. ACM*, vol. 67, no. 2, pp. 68–79, Jan. 2024, ISSN: 0001-0782. DOI: 10.1145/3624724. [Online]. Available: <https://doi.org/10.1145/3624724>.
- [45] J. Achiam, S. Adler, S. Agarwal, *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [46] H. Touvron, L. Martin, K. Stone, *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.

- [47] T. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [48] H. W. Chung, L. Hou, S. Longpre, *et al.*, “Scaling instruction-finetuned language models,” *Journal of Machine Learning Research*, vol. 25, no. 70, pp. 1–53, 2024.
- [49] Y. Zhu, R. Kiros, R. Zemel, *et al.*, “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.
- [50] *Common crawl*. [Online]. Available: <https://commoncrawl.org/> (visited on 02/18/2024).
- [51] *Wikipedia*. [Online]. Available: [https://en.wikipedia.org/wiki/Main\\_Page](https://en.wikipedia.org/wiki/Main_Page) (visited on 02/18/2024).
- [52] R. Lou, K. Zhang, and W. Yin, “A comprehensive survey on instruction following,” *arXiv preprint arXiv:2303.10475*, 2024.
- [53] A. Neelakantan, T. Xu, R. Puri, *et al.*, “Text and code embeddings by contrastive pre-training,” *arXiv preprint arXiv:2201.10005*, 2022.
- [54] R. Nakano, J. Hilton, S. Balaji, *et al.*, “Webgpt: Browser-assisted question-answering with human feedback,” *arXiv preprint arXiv:2112.09332*, 2021.
- [55] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep reinforcement learning from human preferences,” *Advances in neural information processing systems*, vol. 30, 2017.
- [56] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, “Direct preference optimization: Your language model is secretly a reward model,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [57] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv preprint arXiv:1503.02531*, 2015.
- [58] C.-M. U. P. P. D. O. C. SCIENCE, *Speech Understanding Systems. Summary of Results of the Five-Year Research Effort at Carnegie-Mellon University*. 1977.
- [59] Y. Wu, M. Schuster, Z. Chen, *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.
- [60] A. Q. Jiang, A. Sablayrolles, A. Roux, *et al.*, “Mixtral of experts,” *arXiv preprint arXiv:2401.04088*, 2024.
- [61] *Phind-codellama-34b-v2*. [Online]. Available: <https://huggingface.co/Phind/Phind-CodeLlama-34B-v2> (visited on 04/02/2024).
- [62] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, “Mistral 7b,” *arXiv preprint arXiv:2310.06825*, 2023.
- [63] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.
- [64] V. Liu and L. B. Chilton, “Design guidelines for prompt engineering text-to-image generative models,” in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–23.

- 
- [65] J. White, Q. Fu, S. Hays, *et al.*, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” *arXiv preprint arXiv:2302.11382*, 2023.
- [66] Q. Dong, L. Li, D. Dai, *et al.*, “A survey for in-context learning,” *arXiv preprint arXiv:2301.00234*, 2022.
- [67] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, “What makes good in-context examples for gpt-3?” *DeeLIO 2022*, p. 100, 2022.
- [68] J. Wei, X. Wang, D. Schuurmans, *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.
- [69] T. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [70] Y. Gao, Y. Xiong, X. Gao, *et al.*, “Retrieval-augmented generation for large language models: A survey,” *arXiv preprint arXiv:2312.10997*, 2023.
- [71] J. Kim and M. Min, “From rag to qa-rag: Integrating generative ai for pharmaceutical regulatory compliance process,” *arXiv preprint arXiv:2402.01717*, 2024.
- [72] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M. Chang, “Retrieval augmented language model pre-training,” in *International conference on machine learning*, PMLR, 2020, pp. 3929–3938.
- [73] M. Aumüller, E. Bernhardsson, and A. Faithfull, “Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms,” in *International conference on similarity search and applications*, Springer, 2017, pp. 34–49.
- [74] F. Cuconasu, G. Trappolini, F. Siciliano, *et al.*, “The power of noise: Redefining retrieval for rag systems,” *arXiv preprint arXiv:2401.14887*, 2024.
- [75] A. Madaan, N. Tandon, P. Gupta, *et al.*, “Self-refine: Iterative refinement with self-feedback,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [76] X. Wang, J. Wei, D. Schuurmans, *et al.*, “Self-consistency improves chain of thought reasoning in language models,” *arXiv preprint arXiv:2203.11171*, 2022.
- [77] C. Shi, H. Yang, D. Cai, *et al.*, “A thorough examination of decoding methods in the era of llms,” *arXiv preprint arXiv:2402.06925*, 2024.
- [78] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.
- [79] T. R. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths, “Cognitive architectures for language agents,” *arXiv preprint arXiv:2309.02427*, 2023.
- [80] A. Newell, H. A. Simon, *et al.*, *Human problem solving*. Prentice-hall Englewood Cliffs, NJ, 1972, vol. 104.
- [81] S. Adams, I. Arel, J. Bach, *et al.*, “Mapping the landscape of human-level artificial general intelligence,” *AI magazine*, vol. 33, no. 1, pp. 25–42, 2012.
- [82] F. Doshi-Velez and B. Kim, “Towards a rigorous science of interpretable machine learning,” *arXiv preprint arXiv:1702.08608*, 2017.
- [83] H. Zhao, H. Chen, F. Yang, *et al.*, “Explainability for large language models: A survey,” *ACM Transactions on Intelligent Systems and Technology*, 2023.

- [84] Z. Wu, Y. Chen, B. Kao, and Q. Liu, “Perturbed masking: Parameter-free probing for analyzing and interpreting bert,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4166–4176.
- [85] H. Mohebbi, A. Modarressi, and M. T. Pilehvar, “Exploring the role of bert token representations to explain sentence probing results,” in *The 2021 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2021, pp. 792–806.
- [86] C. W. Granger, “Investigating causal relations by econometric models and cross-spectral methods,” *Econometrica: journal of the Econometric Society*, pp. 424–438, 1969.
- [87] M. Treviso, A. Ross, N. M. Guerreiro, and A. Martins, “CREST: A joint framework for rationalization and counterfactual text generation,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds., Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 15 109–15 126. DOI: 10.18653/v1/2023.acl-long.842. [Online]. Available: <https://aclanthology.org/2023.acl-long.842>.
- [88] A. Ross, A. Marasović, and M. Peters, “Explaining NLP models via minimal contrastive editing (MiCE),” in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., Online: Association for Computational Linguistics, Aug. 2021, pp. 3840–3852. DOI: 10.18653/v1/2021.findings-acl.336. [Online]. Available: <https://aclanthology.org/2021.findings-acl.336>.
- [89] S. Huang, S. Mamidanna, S. Jangam, Y. Zhou, and L. H. Gilpin, “Can large language models explain themselves? a study of llm-generated self-explanations,” *arXiv preprint arXiv:2310.11207*, 2023.
- [90] J. Zhou, S. Z. Arshad, S. Luo, and F. Chen, “Effects of uncertainty and cognitive load on user trust in predictive decision making,” in *Human-Computer Interaction–INTERACT 2017: 16th IFIP TC 13 International Conference, Mumbai, India, September 25-29, 2017, Proceedings, Part IV 16*, Springer, 2017, pp. 23–39.
- [91] M. Xiong, Z. Hu, X. Lu, *et al.*, “Can llms express their uncertainty? an empirical evaluation of confidence elicitation in llms,” *arXiv preprint arXiv:2306.13063*, 2023.
- [92] Y. Zhang, Y. Li, L. Cui, *et al.*, “Siren’s song in the ai ocean: A survey on hallucination in large language models,” *arXiv preprint arXiv:2309.01219*, 2023.
- [93] J. Frattini, J. Fischbach, and A. Bauer, “Cira: An open-source python package for automated generation of test case descriptions from natural language requirements,” in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, IEEE, 2023, pp. 68–71.
- [94] J. Huang, X. Chen, S. Mishra, *et al.*, “Large language models cannot self-correct reasoning yet,” *arXiv preprint arXiv:2310.01798*, 2023.

- [95] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, “Winogrande: An adversarial winograd schema challenge at scale,” *Communications of the ACM*, vol. 64, no. 9, pp. 99–106, 2021.
- [96] D. Hendrycks, C. Burns, S. Basart, *et al.*, “Measuring massive multitask language understanding,” *arXiv preprint arXiv:2009.03300*, 2020.
- [97] T. Kwiatkowski, J. Palomaki, O. Redfield, *et al.*, “Natural questions: A benchmark for question answering research,” *Transactions of the Association for Computational Linguistics*, vol. 7, L. Lee, M. Johnson, B. Roark, and A. Nenkova, Eds., pp. 452–466, 2019. DOI: 10.1162/tac1\_a\_00276. [Online]. Available: <https://aclanthology.org/Q19-1026>.
- [98] T. Sawada, D. Paleka, A. Havrilla, *et al.*, “Arb: Advanced reasoning benchmark for large language models,” *arXiv preprint arXiv:2307.13692*, 2023.
- [99] Y. Bai, X. Lv, J. Zhang, *et al.*, “Longbench: A bilingual, multitask benchmark for long context understanding,” *arXiv preprint arXiv:2308.14508*, 2023.
- [100] X. Zhang, Y. Chen, S. Hu, *et al.*, “ $\infty$ Bench: Extending long context evaluation beyond 100k tokens,” *arXiv preprint arXiv:2402.13718*, 2024.
- [101] J. Sun, C. Zheng, E. Xie, *et al.*, “A survey of reasoning with foundation models,” *arXiv preprint arXiv:2312.11562*, 2023.



# A

## Appendix

### A.1 Prompts

You are an expert code reviewer, you are given a test script written by a developer and a requirement that it should fulfill. Your task is to evaluate if the test script strictly complies with the provided requirement. A test covers/is in compliance with a requirement when the test script tests the necessary system functionality to ensure that the tested system fulfils the requirement. You are expected to provide an answer in a valid json format.

The decision should be Yes/No on whether the test script meets the requirements. As a reason, you should break down the requirements and point out which parts of the test script fulfills them. Try to interpret the requirement and reason if the tests meets the intention behind the requirement. Some things are not directly mentioned in the requirement but are implied, so you should also look for those in the test script.

The Test Script is a sequence of functions, which are separate lines. Each function has a name followed by a ':'. Inside the function one or more (nested) parameters are called, with their argument in brackets '()'.  
`function_name: parameter1 parameter2 parameter3`

Listing 2: Base Prompt Mixtral

You are an expert code reviewer, you are given a test script written by a developer and a requirement that it should fulfill. Your task is to evaluate if the test script strictly fulfills the provided requirement. A test covers/is in compliance with a requirement when the test script tests the necessary system functionality to ensure that the tested system fulfils the requirement. You are expected to provide an answer in a valid json format.

The decision should be Yes/No on whether the test script strictly meets the requirements. You are not allowed to make any assumptions about the test script and you should also look for exact requirements and variables fulfilling the requirements inside the test script to make your judgement. Indirect fulfillments of the requirements should be counted as a "No". You have to be extra critical as there are more negatives than positives, so be 100 percent sure while answering 'yes'.

The Test Script is a sequence of functions, which are separate lines. Each function has a name followed by a ':'. Inside the function one or more (nested) parameters are called, with their argument in brackets '()'.  
'

Listing 3: Base Prompt CodeLlama

Test Script and Requirement:

This is the test script: *{test\_script}*

This is the requirement: *{requirement}*

Follow this json format:

```
{  
  reason: Your reason for making the decision,  
  decision: Your decision (Yes/No)  
}
```

Listing 4: Input prompt part

Make your decision by thinking step by step.

Strictly stick to the provided json format and don't add any additional information.

### Response

Listing 5: Final instructions prompt part

Inside the test script you'll find some signal names. These signals might have some states. You can find the states (if any) here: *{signal\_states}*

Listing 6: Signal info prompt part

Inside test script you'll find some function names. These functions will have additional documentation about them. You can find the additional documentation (if any) here: *{function\_docs}*

Listing 7: Function info prompt part

Given below are two examples for a pair of test script, requirement and their corresponding decision and reason:

Listing 8: Two-Shot prompt part

You are an expert code reviewer, you are given a single test function out of a test script written by a developer and a requirement that it should fulfill. A test covers/is in compliance with a requirement when the test script tests the necessary system functionality to ensure that the tested system fulfils the requirement. Your task is to reason about if the single function is contributing to the test script fulfilling the requirement or not. You are expected to provide an answer in a valid json format.

The decision should be Yes/No on whether the function contributes to the potential fulfillment of the requirement by the test script. Try to interpret the requirement and see if the function has any relation to the requirement and could contribute to testing it. Also consider that the full test script consist of multiple functions, so consider that the provided function might interact or be necessary in combination with other functions to fulfill the requirement. The Test Script is a sequence of functions, which are separate lines. Each function has a name followed by a ':'. Inside the function one or more (nested) parameters are called, with their argument in brackets '()'.  
'()'

Test Script and Requirement:

These are the test functions: *{test\_script}*

This is the requirement: *{requirement}*

Follow this json format:

```
{  
  reason: Your reason for making the decision,  
  decision: Your decision (Yes/No)  
}
```

Inside the function you'll find some signal names. These signals might have some states. You can find the states (if any) here: *{signal\_states}*

These functions will have additional documentation about them. You can find the additional documentation (if any) here: *{function\_docs}*

Make your decision by thinking step by step. When making the decision consider two main aspects, does the function syntactically match the function information and does the functions content contribute to test the requirement. Strictly stick to the provided json format and don't add any additional information, the only valid decisions are yes and no.

### Response

Listing 9: Function evaluation prompt version 2

You are an expert code reviewer, you are given a slice of test functions out of a test script written by a developer and a requirement that it should fulfill with a single function out of the test script enclosed with `***`. Your task is to reason about if the single function enclosed by `***` is contributing to the test script fulfilling the requirement or not. You are expected to provide an answer in a valid json format.

The decision should be Yes/No on whether the function contributes to the potential fulfillment of the requirement by the test script. Try to interpret the requirement and see if the function has any relation to the requirement and could contribute to testing it.

The Test Script is a sequence of functions, which are separate lines. Each function has a name followed by a `:`. Inside the function one or more (nested) parameters are called, with their argument in brackets `()`.

Test Script and Requirement:

These are the test functions: `{test_script}`

This is the requirement: `{requirement}`

Follow this json format:

```
{  
  reason: Your reason for making the decision,  
  decision: Your decision (Yes/No)  
}
```

Inside the function you'll find some signal names. These signals might have some states. You can find the states (if any) here: `{signal_states}`

These functions will have additional documentation about them. You can find the additional documentation (if any) here: `{function_docs}`

Make your decision by thinking step by step. When making the decision consider two main aspects, does the function syntactically match the function information and does the functions content contribute to test the requirement. Strictly stick to the provided json format and don't add any additional information, the only valid decisions are yes and no.

`### Response`

Listing 10: Function evaluation prompt version 3

You are an expert requirement engineer. A previous requirement engineer has written a requirement for an automotive system. Your task is to improve the requirement according to the given criteria without changing the semantic meaning of the requirement. The criteria you should consider are:

Verifiability: the engineer must be able to check (verify) that the produced system meets the specified system. Understandability: the requirement is correctly understood without difficulty. Unambiguity: there exists only one interpretation for the requirement. Precision: all used terms are concrete and well defined.

Reformulate the requirement to better match the given criteria without changing the purpose and semantic content of the requirement.

Here is the requirement: *{requirement}*

Follow the following json template, while keeping the structure of requirement name and requirement description intact. Only answer with the reformulated requirement without any explanations and formulate the description in one line as a test and not a list. { "requirement name": "requirement description": }

Listing 11: Requirement reformulation prompt