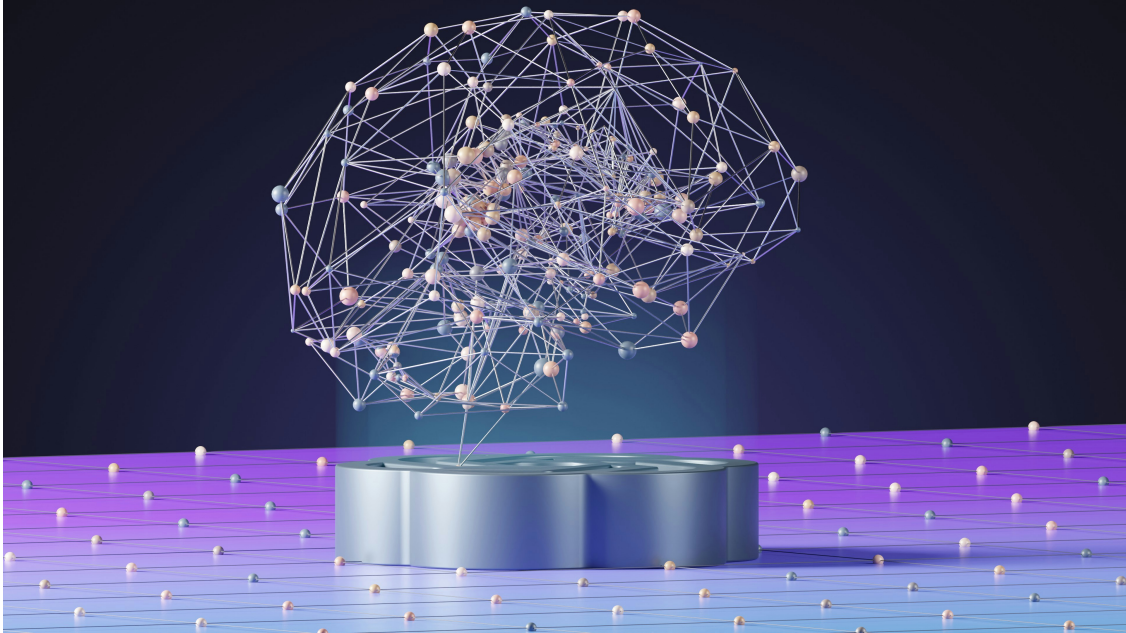




CHALMERS
UNIVERSITY OF TECHNOLOGY



Automated Validation of Test Cases Using Generative AI

Development of program for generating test cases utilizing generative AI for system requirements in automotive industry.

Degree project report in Electrical Engineering

Jarl Dang, Daniel Göransson

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

DEGREE PROJECT REPORT 2025

Automated Validation of Test Cases Using Generative AI

Development of program for generating test cases utilizing generative
AI for system requirements in automotive industry.

Jarl Dang, Daniel Göransson



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Automated Validation of Test Cases Using Generative AI
Development of program for generating test cases utilizing generative AI for system requirements in automotive industry.
Daniel Göransson, Jarl Dang

© Jarl Dang, 2025.

© Daniel Göransson, 2025.

Supervisor: Jonatan Rydberg, Volvo Group
Examiner: Hazem Torfah, Computer Science and Engineering

Degree project report 2025
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Image illustrating the similarities between the neurons in an AI to the neurons of a brain. Photo by Growtika on Unsplash.

Typeset in L^AT_EX
Gothenburg, Sweden 2025

Automated Validation of Test Cases Using Generative AI

Development of a program for generating test cases utilizing generative AI for system requirements in automotive industry.

Jarl Dang, Daniel Göransson
Department of Electrical Engineering
Chalmers University of Technology

Abstract

This project was provided by Volvo and was intended to help automate their current system of creating validation test cases using generative artificial intelligence (AI). Automation of this system reduces development time and costs while enhancing efficiency. This project covered theories regarding AI, Large Language Model (LLM), prompt engineering, test cases and system requirements. Following an evaluation of available open-source LLMs, the model QWQ-32B was selected. By applying prompt engineering techniques, the model was able to generate not only validation test cases but also executable test code. The program was able to generate adequate results according to Volvo staff. However, the results needs some adjustments in order to be viable for a variety of test cases. Potential improvements, such as the integration of Retrieval-Augmented Generation (RAG), are discussed in this report as future directions to address current limitations. As a result of this project, Volvo has been provided with a solid foundation for automating generation of validation test cases and executable test code.

Keywords: automotive, system requirement, test cases, Large Language Models, LLM, AI, Gen AI, RAG, prompt engineering, HIL

Acknowledgements

This project was conducted in collaboration with Volvo Group and would not have been possible without the contributions of several skilled colleagues. We would like to thank Giorgos Georgiadis for his support with the AI component of the project. We are also deeply grateful to Martin Hill and Fabricio Lélis de Melo for their invaluable assistance with the Jama platform. Special thanks goes to our manager, Ivana Jern, for initiating and coordinating the project, and to our Volvo supervisor, Jonatan Rydberg, whose consistent guidance and weekly support significantly contributed to the end result of the project.

Jarl Dang, Daniel Göransson, Gothenburg, March 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
API	Application Programming Interface
CAN	Controller Area Network
DFS	Depth First Search
HIL	Hardware In the Loop
IDE	Integrated Development environment
LLM	Large Language Model
NLP	Natural Language Processing
RAG	Retrieval Augmented Generation
VAS	Volvo Autonomous Systems

Contents

List of Acronyms	ix
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Background	1
1.1.1 AI	1
1.1.2 HIL	2
1.1.3 Requirements Traceability	2
1.2 Purpose	2
1.3 Goals	3
1.4 Delimitations	3
1.5 Contributions	4
2 Theory	5
2.1 AI	5
2.1.1 Neural Networks	5
2.1.2 LLM	6

2.1.3	Generative AI	7
2.2	API	7
2.3	Prompt Engineering	8
2.4	Retrieval Augmented Generation (RAG)	9
2.5	Depth First Search	9
2.6	Testing	11
2.6.1	Test cases	11
2.6.2	System requirements	11
2.6.3	Jama Connect	11
2.6.4	PyTest	11
3	Methods	13
3.1	Researching potential models	13
3.2	Execution	16
3.2.1	Approach	16
3.2.2	Signal interfaces	16
3.2.3	Validation Test Cases Template	17
3.2.4	Generating Validation Test Cases	18
3.2.5	Generating Code	18
3.2.6	Logging the results	19
4	Results and Discussion	21
4.1	Workflow	21
4.2	Automated Validation of Tipping Bucket	23
4.2.1	Generated Validation Test Case	23

4.2.2	Generated Code	25
4.3	Validate Trailer Tipper Bucket State Transitions	27
4.3.1	Generated validation test case	27
4.3.2	Generated Code	31
4.4	Overall	33
5	Conclusion	35
A	Appendix 1	I

List of Figures

2.1	Figure which illustrates the different AI-concepts and subcategories. . .	5
2.2	Simple illustration of a general neural network with a limited number of hidden layers.	6
2.3	The neural network of a typical Large Language Model with only 2 hidden layers illustrated. The input context data and output data is also explicit. The predictive probabilities are taken from the GPT2 model.	7
2.4	Diagram showing the process of using an API [8]	8
2.5	An example of how the architecture of a simple RAG implementation may look.	9
2.6	The order which Depth First Search would explore this specific tree structure.	10
3.1	Jama structure visualised as a tree data structure.	17
4.1	Flowchart showing the program structure	22
4.2	Generated Python Code based on the given validation test case 4.1. . .	27
4.3	Generated Python Code to validate Trailer Tipper Bucket State Transitions based on the given validation test case 4.2.	33
4.4	Figure illustrating the potential feedback loop of having two AIs work in combination to autotomize the principle of prompt engineering. Either by improving the prompt to create a better output, or by improving the output directly.	34

List of Tables

3.1	Comparison of Selected Large Language Models	15
-----	--	----

1

Introduction

1.1 Background

The usage of artificial intelligence, AI, models has grown significantly over the last couple of years since the introduction of ChatGPT. These AI models have helped people in their daily lives by automating boring and tedious tasks, but their impact goes far beyond that. AI is now driving innovation, making critical decisions, and even transforming entire industries, reshaping the way we live and work on a fundamental level. Our opportunities with AI provide us with unlimited potential for the development of society, enhancing efficiency and unlocking innovations we have yet to imagine.

Something that is currently a hot topic on everyone's lips is self-driving cars, or more broadly, the integration of AI in vehicles. This can be uncomfortable for some people, as the idea of machines making decisions raises concerns about safety, reliability, and control. However, we should view this more as an opportunity rather than a threat, as AI has the potential to refine our way of driving.

With cars becoming increasingly dependent on software, the complexity of their codebase continues to grow. Every new feature, whether it's safety related or performance enhancing, requires thousands of lines of code. Every feature requires **system requirements** to ensure the users safety and products functionality. For Volvo, this means that managing and maintaining software has become just as important as designing the physical vehicle itself. But with millions of lines of code, outdated and inefficient part can slow development, making software management a growing challenge.

Our hope with this thesis is to solve these challenges by combining AI with code. By developing a tool which helps developers generate solid foundations for test cases, we are potentially increasing their efficiency. Instead of having to start from scratch with every system requirement, they simply need to do the last finishing touches.

The background to this project can be divided into three main sections. The technology that makes up the foundation of the project (1.1.1), its main area of application (1.1.2), and the problem that it is trying to solve (1.1.3). The delimitations that were made by Volvo and why they were made are also presented (1.4).

1.1.1 AI

Artificial Intelligence (AI) is a technology which has received a lot of attention for its limitless applications. It enables computers and machines to simulate human learning in order to solve problems. One certain technology under this umbrella

concept are LLMs, which are AIs that have been specialized in processing natural languages. Large Language Models (LLMs) are the most common AI models in order to create original content from user input, since it takes input in natural language, thus being relatable and easy to instruct for humans [1]. AI which is used to generate new content is categorized as generative AI and will be the main focus of this project. Generative AIs differs from LLMs since they are free to generate new content in any format, not necessarily just text.

1.1.2 HIL

The automotive industry has been investing enormous amounts of effort and resources in order to seize the advantages that AI could potentially bring, such as autonomous vehicles. Volvo is no different in this regard. One area in which Volvo sees potential benefits is in combination with Hardware in the Loop (HIL) testing. HIL testing is the process of testing the software of a vehicles hardware by simulating real world environments and scenarios without needing the actual vehicle. This is useful since it can be costly, time intensive and dangerous to test the full physical system each time for every system requirement [2]. Instead of attaching the hardware to a real vehicle, data is simulated using computer models and sent via Controller Area Network (CAN) or other signals to the hardware system being tested. The CAN signals acts as the bridge between the simulated data and the physical components and allows the simulated world to communicate with the real hardware.

1.1.3 Requirements Traceability

In order to ensure that products fulfil industry standards and regulations, several requirements have to be implemented and complied with. In order to trace all the system requirements, and their related test implementations and results, the platform named Jama Software is used at Volvo Autonomous Systems (VAS). Vehicles are large systems requiring several thousands upon thousands of system requirements in order to ensure functionality and user safety. Each component needs to fulfil its system requirement, thus the amount of tests needed increases exponentially with the amount of components. Hence writing these tests is time consuming, even though each test may be simple the sheer amount of tests is a challenge. By utilizing generative AI to write these tests, it could potentially provide immense benefits regarding safety and efficiency.

1.2 Purpose

The purpose of this project is to enhance and automate the existing test framework to enable seamless execution of tests in a HIL environment. By leveraging open-source AI models, the project aims to transform human-readable system requirements and test specifications into executable Python code, ready to be tested with HIL equipment. This approach will significantly streamline the process from

requirements to test results, enabling faster feedback and shorter development cycles. The hope is also for the AI to cover more areas that a human may not think of, therefore testing scenarios that may not have been tested before, improving the quality of testing.

1.3 Goals

The goal with this project is to automatize the process of creating test cases from system requirements in HIL environments. This is done via several steps:

- Utilize prompt engineering in order to tailor the LLM output until a satisfactory result is produced, meaning a valid template and functional test case code.
- Create a seamless integration between the Jama requirement, the AI model, and the hardware using python as our programming language and API endpoints for retrieving information.
- Provide a valuable tool for Volvo to further develop and apply to other areas of development.

1.4 Delimitations

These delimitations were set in order to focus only on the important aspects in order to accommodate the given time frame.

1. This project will only focus on LLMs that meet the requirements set by the project team at Volvo. The specific requirements are as following:
 - *Local deployment* - The LLM needed the capability of being deployed on local hardware. A non-local model posed a risk of data leakage and was not assessed to be safe enough.
 - *Large input capacity* - In order for prompt engineering to be unrestricted and receive more context, the AI model had to be able to handle large sets of text strings as input data. The limit that was set due to hardware capacity was 32B parameters.
 - *Processing speed* - The time it takes for the LLM to process a request can not be outside a reasonable time frame in order to keep the promise of increased production speed. A reasonable time frame in this regard is assessed to be within 10 minutes, which was decided after a dialogue with employees at Volvo. This is to optimize time used for engineering the input data.

- *Pre-trained* - To ensure accordance with the time plan and thereby a quick setup process, the LLM had to be pre-trained in natural language and code generation for the task to be fulfilled. It also had to be of reasonable size with regards to the hardware that was available at the Volvo office.
 - *License* - To use this program to develop a commercial product, a free use license had to be used for the LLM. The specific license that was deemed most appropriate was the apache license 2.0. It allowed for free use and free distribution of the software including for commercial use with minimal restrictions [3].
2. The tuning of the LLM will only be with different input prompts and adjusting certain parameters such as temperature since Volvo does not have any large dataset that can be used for fine-tuning the model for this task.
 3. The evaluation of the output from the AI will only be based of previously written test cases the functionality of the generated code.
 4. The project will not cover a user interface in order to focus on the internal functionality of the program.

1.5 Contributions

1. Prompts which are adjusted for QWQ-32B to generate validation test cases from system requirements.
2. Prompts to generate code from validation test cases.
3. A workflow map which illustrates the process of the project from start to finish.
4. Suggestions for potential improvements when creating and implementing an LLM with an existing database.

2

Theory

2.1 AI

AI is an umbrella concept which includes several other terms such as LLMs and Generative AIs. This is illustrated in figure 2.1 which indicates all the technical terms which AI encompasses. As seen, AI is a broad and ubiquitous term within tech, capable of a wide range of diverse tasks. Consequently, it lacks a proper agreed upon definition in the industry and scientific world. In this paper, AI will be defined as per NASA's definition, which in turn references the National Defence Authorization Act of 2019 [4]. As such, AI is defined as computer systems which can perform complex tasks normally done by humans, such as reasoning and decision making.

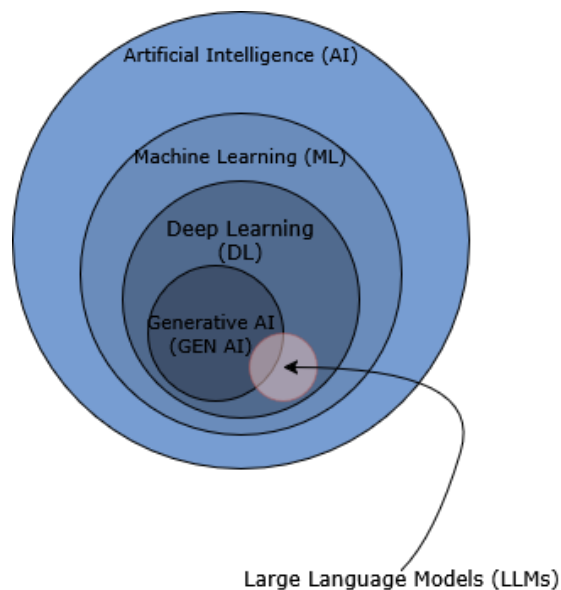


Figure 2.1: Figure which illustrates the different AI-concepts and subcategories.

2.1.1 Neural Networks

Most modern AIs utilizes a concept called neural networks in order to process data. Although there exists several other approaches to train and create AIs. Neural networks are inspired by the human brain and consist of interconnected nodes, so called neurons, in order to process information in layers, similar to humans. These

neural networks often have large numbers of processors working in parallel and arranged in several layers. In figure 2.2 the structure of a typical neural network can be seen. It consists of an input layer, a number of hidden layers and an output layer. A comparison is made in [5] which relates the input layer to the optic nerves in human visual processing. The hidden layers would then represent the neurons in the brain which processes the raw data in order to create an image (output) for us humans to see. The number of hidden layers a neural network contains is also referred to as the depth of the network or AI.

To further delve into the hidden layer and how it processes the raw data, we can imagine it as a control panel consisting of several thousands of dials (nodes). Each dial is associated with an adjustable number (weight). Adjusting these dials modifies the weights, which determine how input signals are transformed and propagated through the network. Each node computes a weighted sum of its inputs and transmits the resulting continuous value to the next layer. Setting these dials to their optimal value for each input is an impossible challenge for humans to solve manually. There are several algorithms in order to adjust the dials which yields different results, some popular alternatives are Deep Learning methods such as Backpropagation and Transformers [5]. All of them utilize large datasets as answer sheets in order to train the AI. A high quality dataset is often key in training an AI model to perform well.

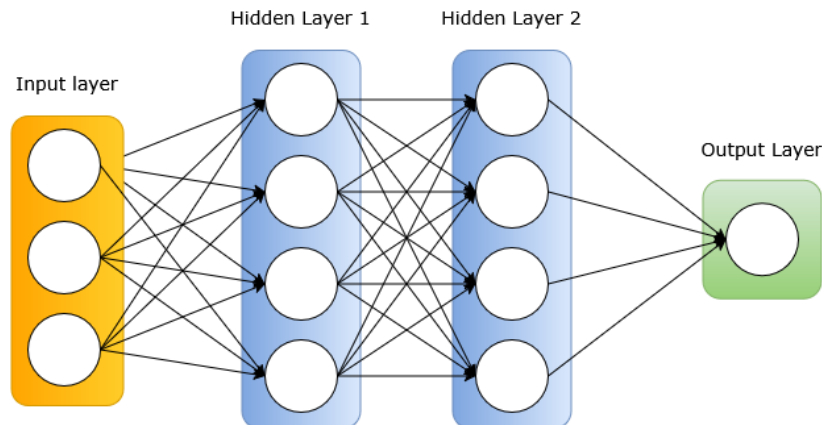


Figure 2.2: Simple illustration of a general neural network with a limited number of hidden layers.

2.1.2 LLM

Today there are several popular LLMs, such as ChatGPT, Claude and DeepSeek. LLMs are AIs which are specialised in processing natural language, such as the languages humans use to communicate. They are statistical models which tries to predict the next **token** given a **context**. Tokens refers to the basic building blocks of natural languages, such as punctuation marks, individual words or Chinese characters. The context includes the user input, but may also include previous chat history or stored memory. The neural network of an LLM can be seen in figure 2.3. It is clear that it follows the general structure of a neural network as previously

illustrated, the difference being that the weights have values which results in it being an expert in processing natural languages.

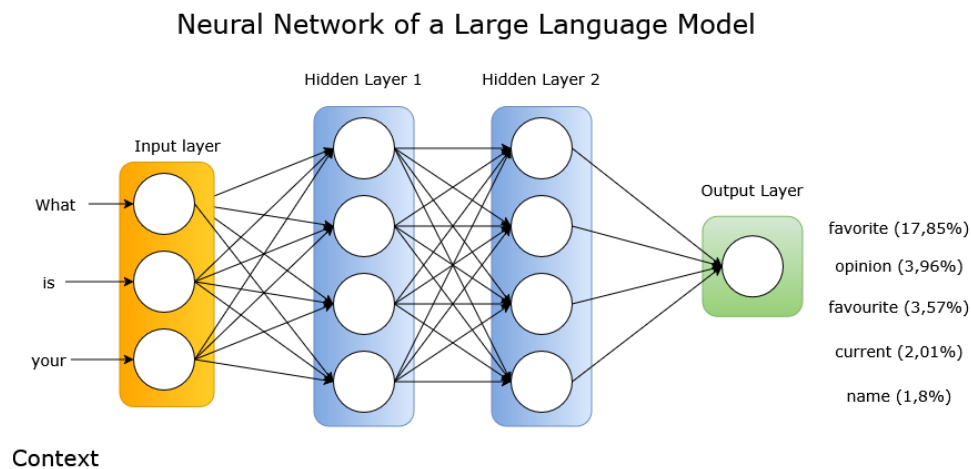


Figure 2.3: The neural network of a typical Large Language Model with only 2 hidden layers illustrated. The input context data and output data is also explicit. The predictive probabilities are taken from the GPT2 model.

2.1.3 Generative AI

Generative AI is a class of artificial intelligence systems designed to generate new, original content based on patterns learned from training data. It does not simply retrieve or remix existing data, instead new content is generated which that resembles its training data. The generative part comes from its ability to create, unlike traditional AI which typically focuses on classification or prediction. This also includes popular LLM models such as ChatGPT which generates text. But it also extends to AIs which focus on other forms of media, such as Midjourney which generates images, or Jukebox which generates raw audio. One major aspect of Generative AI in the industry is its ability to generate code, since code is simply plain text. This has already proven to be an invaluable tool for developers in increasing efficiency and improving the learning process [6].

2.2 API

API stands for Application Programming Interface (API) and is a useful tool that provides a channel for two different software programs to communicate. This is done using a set of rules and protocols that is used to request and provide data between the two programs. This request is done using code and often includes an endpoint URL, a Hypertext transfer protocol (HTTP) method and certain parameters. This process of communication provides a standardized way for users to access other applications data and is used in many businesses and organizations, saving time and money [7].

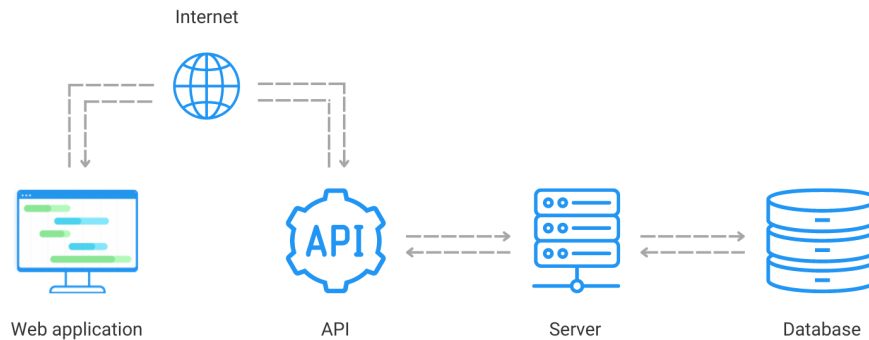


Figure 2.4: Diagram showing the process of using an API [8]

2.3 Prompt Engineering

Prompt engineering is the technique of carefully crafting and refining input prompts to improve the quality and precision of the generated output by the LLM. It is a way of carefully instructing and guiding the LLM closer to the desired outcome. This can be a useful technique in scenarios where training data or time is limited and the complexity of the task is reasonable [9]. It also allows for the user to have more control of the model and ensure that the model's output are aligned with user goals. Prompt engineering is not training the LLM, no weights are being adjusted nor is the neural network of the LLM changing. It is simply a way to better utilise it and unleash more of its potential. Although prompt engineering has many benefits there are also challenges with implementing such a technique. Since LLMs are statistical models, the output will always vary even if the input remains identical. This may lead to inconsistent results and makes it hard to quantify the quality of the input. Another challenge is the limited context window. For complex tasks which may require a large context window, it may not be possible for the LLM to process all of it and thus limiting the LLMs understanding of the problem.

One fundamental and famous technique which a lot of programmers are utilizing is called rubber duck debugging (rubberducking). The principle is that a developer would explain their code in natural language to an inanimate object, such as a rubber duck, in order to better conceptualize their approach and avoid tunnel vision [10]. The potential benefits with combining rubberducking and prompt engineering are exponential, allowing for developers to go through their code and receive instant feedback from the LLM. Thus leading to a key concept within prompt engineering, an iterative feedback cycle where the prompts continually improve to better serve the developers goal.

2.4 Retrieval Augmented Generation (RAG)

RAG is a technique that enables LLMs to access more context before processing a request. A retrieval framework retrieves and incorporates new information by first searching a predetermined database. The retrieved data is used as additional context for the LLM to produce an improved response. The architecture of a simple RAG implementation can be seen in figure 2.5. The clients questions is first sent to a framework which combines the question with contextual data. It is then prompted to the LLM which then generates a response, the response is then processed and sent back to the client.

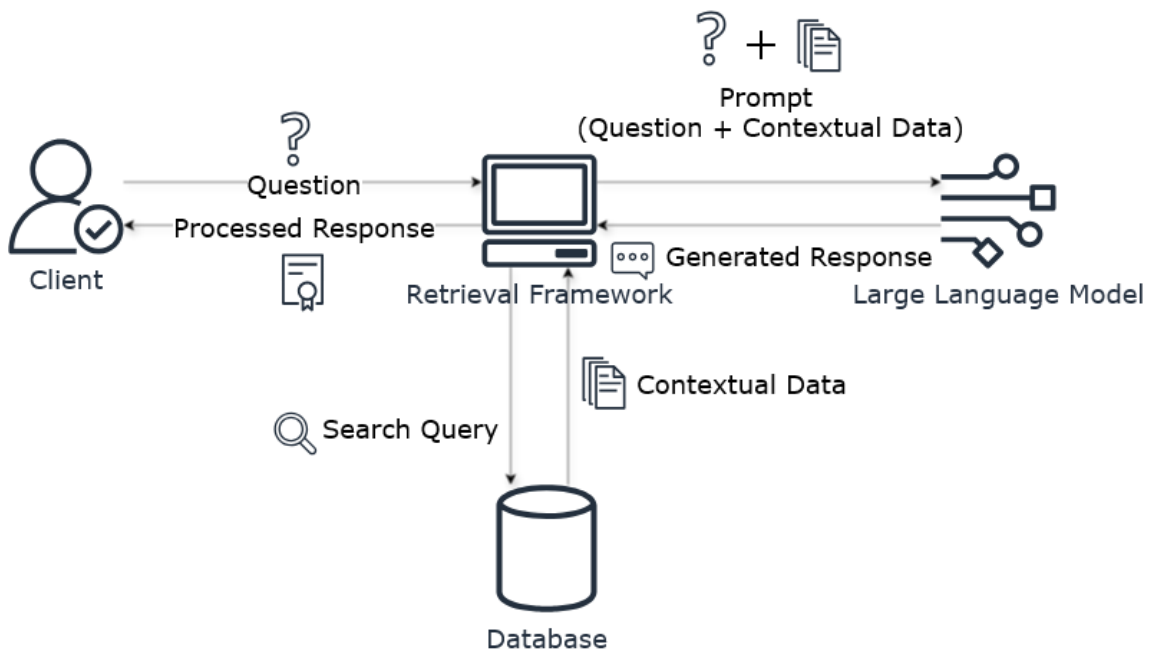


Figure 2.5: An example of how the architecture of a simple RAG implementation may look.

This technique allows for the LLM to access more information in order to process the question with a higher confidence. Consequently, the result should be of higher quality and relevance to the client. A study found that RAG improved the results of their GPT-4 model with 4% and reduced hallucinations [11]. One important factor is that the database provides relevant high quality data, since this will serve as a foundation for the LLM to process the given request.

2.5 Depth First Search

DFS is a search algorithm commonly used to explore data structures such as graphs and trees. Since DFS prioritises the depth first, it is generally the fastest way to search a tree when the depth of the tree is of interest. These data structures consists of vertices and edges. The algorithm initialises in a root node which marks the start of the tree. It then explores adjacent vertices and prioritises depth before breath,

meaning it searches as far deep as possible in a given branch before exploring other paths. After the deepest level has been reached, it backtracks until it finds an unexplored neighbour and starts over again. This is repeated until all vertices have been explored. The search order and depth priority can be seen in figure 2.6.

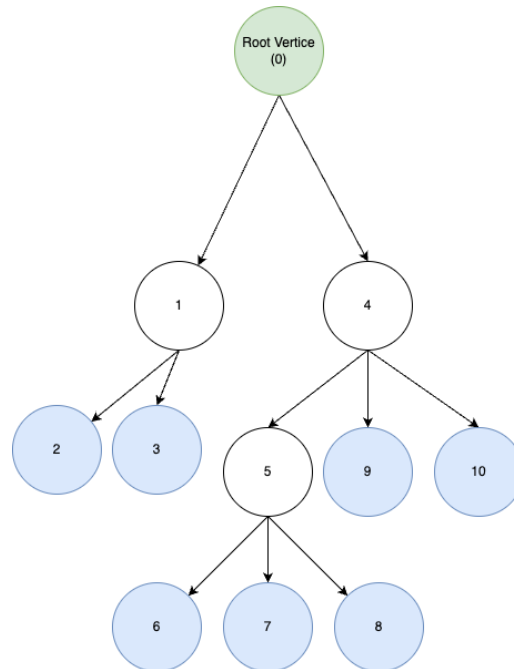


Figure 2.6: The order which Depth First Search would explore this specific tree structure.

2.6 Testing

2.6.1 Test cases

Test cases are an integral part of the production chain in software and hardware development. It is used when testing a product, and can be defined as a pair of inputs and outputs to assess the functionality of software or hardware components. The inputs and outputs are produced according to a test specification and then executed using the tested software, typically named software under test (SUT), and the result is then reported back and documented as a part an evaluation of the software [12]. However, designing a test case and its designated test data requires a lot of resources such as cost and time, and can even account for up to half of the development costs. One attempt to reduce these costs are to automate the process, thereby reducing the labour costs and improving the production efficiency [13].

2.6.2 System requirements

With the rapid development in the automotive industry comes new innovations and increasingly complex systems. To ensure that these technologies are safe, functional and follow the automotive legislations, system requirements are created. A system requirement is commonly created in order to meet costumer expectations, ensure quality and safety, or to meet necessary legislations [14]. It is also used for design purposes by creating internal requirements that the company deem important.

In order to create a standardized set of legislations on an international level, Volvo uses the ISO-standards. The ISO-standards was created by the International Organization for Standardization in order for companies to follow a unionized set of standards when creating a product. This ensures that the product meets a standardized set of regulatory requirements and in turn promotes high quality and safe product [15].

2.6.3 Jama Connect

Jama connect is a requirement management platform for companies within the automotive industry that is designed to support developers with documentation and traceability during the development process. It features compliance support for ISO-standard which is critical in order to create safe and secure products that comply with industry regulations. Another key functionality is providing a framework for managing testing, such as running test cases and logging test runs [16].

2.6.4 PyTest

When developing software in python, the most common framework for testing python code is PyTest. Its main function is easily write and create readable tests that can be scaled to more complex functional tests [17].

3

Methods

In this chapter the methodology of the project will be described. The project began with researching different models and exploring needs and demands of Volvo. There were several meetings with test developers and AI specialists working at Volvo in order to lay out a loose plan. A gantt chart (??) was created in order to plan and structure the progress of the project, although this early on in the project the tasks were not immediately obvious and were prone to change as the project developed. Twice a week, meetings were held with a supervisor at Volvo in order to receive feedback and guidance on the project.

3.1 Researching potential models

To identify the most suitable language model for the process description that was provided by Volvo, a survey of LLMs was conducted. For a particular model to meet the project requirements, it had to satisfy a defined set of criteria. These criteria were created in compliance with the team at Volvo to ensure functional, technical and security constraints.

The requirements, stated and described in delimitations, were as following:

- Local deployment
- Large input capacity
- Processing speed
- Pre-trained
- License

The LLMS that were surveyed were:

- DeepSeek V3
- DeepSeek R3
- QWQ-32B
- Falcon LLM

3. Methods

- Claude Sonnet 3.5

Model	Pros	Cons
DeepSeek V3 [18]	<ul style="list-style-type: none"> • Solid benchmarks • Processes large texts (64k tokens) • Open-source • Cost-effective, fast • Adjustable weights 	<ul style="list-style-type: none"> • No reasoning capability • Low max output (8k) • Possible censorship concerns
DeepSeek-R1-Distill-Qwen-32B [19]	<ul style="list-style-type: none"> • Smaller, faster version of R1. • Less hardware intensive • Cheaper to run • Open-source 	<ul style="list-style-type: none"> • Lower max context tokens • Lower output tokens • Slight performance drop (5-15%) compared to R1
QWQ-32B [20]	<ul style="list-style-type: none"> • Compact model • Good reasoning • Open-source • Quick response time • Solid benchmarks 	<ul style="list-style-type: none"> • Limited adoption • Fewer tools
Falcon 3 LLM [21]	<ul style="list-style-type: none"> • Open-source • Apache 2.0 • Strong benchmark (Falcon 3) 	<ul style="list-style-type: none"> • Large models need compute
Claude Sonnet 3.5 [22]	<ul style="list-style-type: none"> • Beats GPT-4o in benchmarks (math, language, coding) 	<ul style="list-style-type: none"> • Hard to download locally • License needs review • Price plan

Table 3.1: Comparison of Selected Large Language Models

After communication with an AI specialist at Volvo, the selected LLM that was deployed for testing on local hardware was an open-source model named DeepSeek-R1-Distill-Qwen-32B. It showed clear signs of high performance in coding tests and also high performance in english tests, with results that were comparable to more common models like GPT-o1 and Claude-3.5 [23]. The LLM was changed to QWQ-32B after three weeks, a decision that was made by Volvo as they deemed it to be more efficient. This was shown to be true when testing the model, as the LLM followed instructions more thoroughly than before.

3.2 Execution

Visual Studio Code was utilized as the IDE to write the program. The program itself was written in Python Version 3.13 and pip as the package manager. Several dependencies were installed using pip in order to communicate with the AI and Jama API, such as huggingface and py-Jama-rest-client. A code library from Volvo was also supplied to make certain requests possible from Jama utilizing their API and endpoints. Access to their API was given by creating a username and password in accordance to Volvo policy. The program was then built with two main purposes, generating a validation test case or generating executable code.

3.2.1 Approach

In order to instruct the LLM and produce the best results, prompt engineering was leveraged. Several initial approaches were discussed with the engineers at Volvo, such as fine-tuning or RAG. Most were deemed too time and resource intensive, therefore it was settled that prompt engineering would yield sufficient results within the constraints of this project. First a request was sent to the LLM in natural language, in order to generate a validation test case. The context was simply the system requirement alone. This was then improved upon several iterations of analysing the context and having test case experts, as well as ourselves, manually judging the quality of the output. Improvements were then made, such as giving the LLM high quality examples of what the output should resemble or clarifying instructions. After iterating several times, a structured way to input the request was created. The input consisted of an instruction file, a prompt file, and several high quality examples. At this point the user could demand the LLM to generate either a validation test case given a system requirement, or executable code given a validation test case. The different files varied depending on which step was to be fulfilled.

3.2.2 Signal interfaces

In order to retrieve all of the CAN signal interfaces and downstream system requirements the Jama structure was constructed as a tree with nodes, as seen in figure 3.1. The given system requirement or function acted as the root node, the other nodes were then explored via DFS. Since the signal interfaces of the lowest level are of interest, DFS is an efficient way to search and retrieve them. This method also allowed the user to search for nodes other than signal interfaces, depending

on what the Jama API allowed. By using this approach, all the downstream CAN signals were extracted and could be fed into the LLM in order for it to generate code signal names corresponding to the physical hardware. By simply providing a Jama ID for the relevant system requirement, the program could now extract all of the downstream CAN signals.

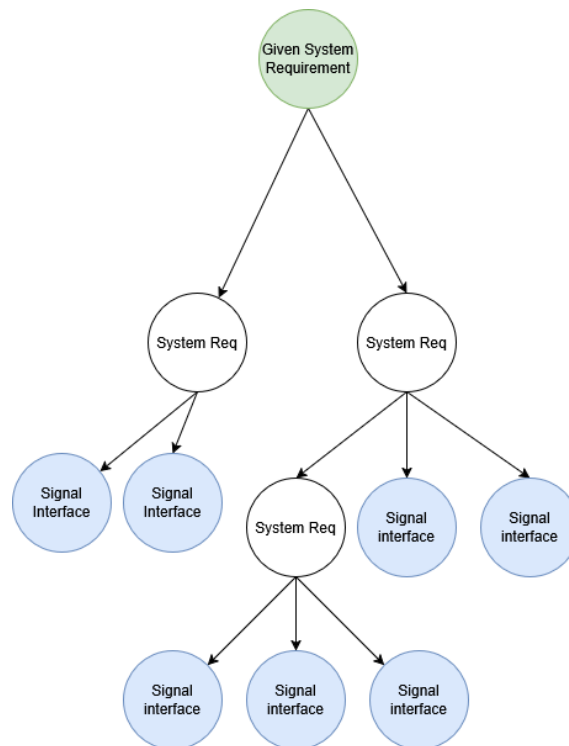


Figure 3.1: Jama structure visualised as a tree data structure.

3.2.3 Validation Test Cases Template

A template was created to provide the LLM with clear instructions and an easy way to fill in the requested information. It was written in markdown language in order to structure the information in sections with headers. The template took inspiration from high quality examples on Jama, given by test engineers at Volvo. The markdown block in figure 3.1 shows a small snippet of the template and clearly shows the benefits of writing in markdown language. Additionally, many LLMs are trained with data written in markdown language, making them suited at understanding that specific format. This template was constructed following high quality validation test cases already uploaded onto Jama, with the goal of producing similar results.

Validation Test Case Template Snippet

```
# TEST CASE TEMPLATE INSTRUCTIONS

You are to complete the template below by replacing fields marked
with '[FILL]' with appropriate content. The '[FIXED]' and '[FILL]'
markers are only guides and **must not appear in your final
output**.

## FORMATTING REQUIREMENTS

-Remove all '[FIXED]' and '[FILL]' markers from your final output
-Preserve all formatting, including bold text, line breaks,
numbering, and spacing
-Use precise technical language appropriate for validation test
cases
-Default values in italics are suggestions to use when no
specific value is indicated
-Leave any field empty with a dash (-) if no relevant information
exists

## TEMPLATE STRUCTURE

[FIXED]# PROJECT ID:
[FILL]
. . .
```

Markdown Code 3.1: A small snippet of the validation test case template written in markdown language.

3.2.4 Generating Validation Test Cases

To generate a validation test case several inputs were given to the LLM. Following the general structure, the LLM was fed instructions, examples and the entire template A.1 to fill in. The LLM then generated a response which included internal thinking. This made it mandatory to extract the validation test case manually from the response. This process of was repeated several times in order to improve the instructions using prompt engineering. When the output was satisfactory for one system requirement, several others were inputted in order to test the stability of the model and instructions.

3.2.5 Generating Code

To generate code a validation test case is inputted into the LLM which acts as the main guideline to shape the code. The purpose of the code is to perform the necessary steps and tests in order to fulfill the requirements of the validation test case. Once again the LLM responded with internal thinking making it mandatory to extract the code manually. This method was then repeated similarly as when generating validation test cases.

3.2.6 Logging the results

A folder and file structure was implemented in order to log the results from each iteration. This was done via utilizing Python's ability to create and edit files. Depending on what is generated the output is different and is logged respectively in their separate folders. Besides the generated output, the logs also include the entirety of the LLM input and context to better track and quantify which inputs yields good results.

4

Results and Discussion

The output can be split into two sections, the generated validation test case and the generated code. In order to test the LLM, several validation test cases were generated. They all share the same instructions and examples which the LLM can learn from. The only difference is the user prompt, which will be the system requirement description. The same principles applies for the generated code results, but the difference being the validation test case from which the code is generated.

4.1 Workflow

The flowchart seen in figure 4.1 is designed in order to visualize and divide each task into smaller sub-tasks in accordance with the project description. This is done with the intention of reflecting each part of the system and how they are connected, but also the order of execution in the program.

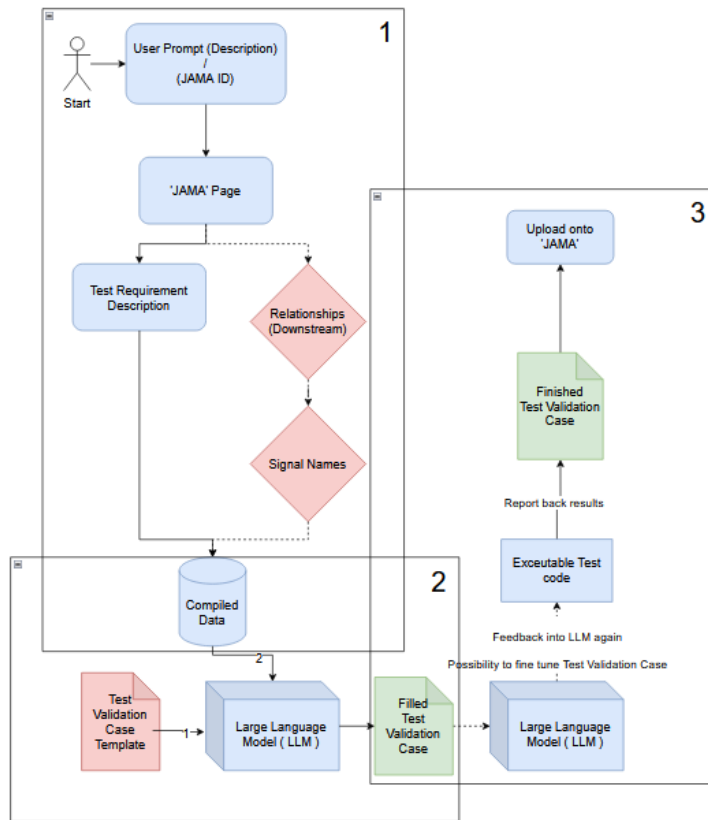


Figure 4.1: Flowchart showing the program structure

To provide additional clarity on the functionality of each major component of the program, the flowchart has been divided into bigger distinct sections. Each section is subsequently described in detail.

1. Data extraction from Jama

The process begins with the user providing a prompt in the form of a requirement description or a specific Jama ID. Based on this input, the system accesses the corresponding Jama data and extracts both the description of the requirement and also the CAN signal names. This is then compiled into a set of data which can be configured for each run to alter the output.

2. Test case generation

In the second part of the program, the compiled data along with a pre-defined template for documentation of test cases is fed into the LLM in order to generate the validation test case document.

3. Code generations

In the last step of the process, the previously generated validation test case is fed back into the LLM for the generation of executable code. The generated code is then executed while connected to the HIL-rig and the result of the test execution is returned.

4.2 Automated Validation of Tipping Bucket

The following results have been generated in order to validate the shaking capability of a truck's bucket at varying angles. The results have been manually extracted from the LLM output and does not show the internal thinking, which was part of the original output.

4.2.1 Generated Validation Test Case

Besides the instructions and examples, the input given to the LLM in order to generate this result is the following system requirement description:

When TrailerBucketIMUStatus indicates that the tipping angle is below BucketShakerThreshold, the Shaker Control Function shall set Shaker Capability to ShakerNotAvailable.

The result is outputted in markdown language, since inserted template was written in markdown language. As seen in the markdown block 4.1, the LLM fills the validation test case template (A.1) with the relevant information in order to satisfy the system requirement. Some steps are left as blank, as the LLM deems that it is lacking sufficient information in order to fill those headers, such as ASSIGNEE, RELEASE and TEST RUN RESULTS. Another thing to note is that the judgment by the LLM is far from flawless, as seen in the markdown block. The threshold values are arbitrarily chosen since this data was missing from the context.

The goal of this system requirement was to validate that the bucket is incapable of shaking when below a certain tipping angle. This is clearly described in the template under **Description**. It can also be seen that the preconditions and configurations for the system are reasonable as well as the steps needed to ensure that the goal is achieved. One engineer at Volvo stated that the angles are arbitrary and that it would be an improvement if the test did a continuous sweep across an interval of angles instead. Furthermore, since a lot of HIL specific context is missing, the validation test case is written more generally and lacking in detail, such as simply **initiating the shaker via a operation command via control interface**. The LLM is missing the necessary context in order to be more specific.

Validation Test Case for Shaker Control Function

```
# PROJECT ID:
CAATS502-HIALS-1888

# GLOBAL ID:
GID-587887

# OBJECT TYPE:
Test

# NAME:
Shaker Control Function -Tipping Angle Below Threshold
```

```
# COMMON DESCRIPTION:

## Description:
When TrailerBucketIMUStatus indicates that the tipping angle is below
BucketShakerThreshold, the Shaker Control Function shall set Shaker
Capability to ShakerNotAvailable.

## Module:
Control System

## Test Type:
System Functional

## Precondition and configurations:
1. The system is powered on and initialized.
2. TrailerBucketIMUStatus signal is operational.
3. BucketShakerThreshold value is pre-configured (e.g., 15 degrees).

## Testing environment(HIL/SIL/Machine/Site):
HIL

## Input data:
TrailerBucketIMUStatus with tipping angle values below
BucketShakerThreshold.

## Expected behavior:
The Shaker Capability output should transition to ShakerNotAvailable when
the tipping angle falls below the threshold.

# SPECIFIC DESCRIPTION:

## Specific equipment:
HiL rig with IMU simulation module

## Specific configuration:
BucketShakerThreshold = 15°

## Specific data collection:
Record Shaker Capability state changes and tipping angle readings.

## Specific environmental conditions:
No external vibrations or disturbances.

# STEPS:
1. Configure HIL rig to simulate TrailerBucketIMUStatus with a tipping
angle of 10° (below threshold).
2. Initiate shaker operation command via control interface.
3. Monitor Shaker Capability signal output.
4. Verify Shaker Capability transitions to ShakerNotAvailable within 2
seconds.
5. Repeat test with tipping angles at 14°, 0°, and -5°.

# ASSIGNEE:
```

```

*(Left blank)*

# RELEASE:
*(Left blank)*

# VALIDATION METHOD:
Test

# STATUS:
Ready for Review

# TEST CASE STATUS:
Blocked

# EXTERNAL ID:
*(Left blank)*

# TEST RUN RESULTS:
*(Left blank)*

# TAGS:
Shaker Control, Tipping Angle, Safety Threshold

```

Markdown Code 4.1: The validation test case which was generated in order to verify shaker capability at various bucket angles.

4.2.2 Generated Code

To generate the code, instructions, past code examples, the relevant CAN signals, and the validation test case of interest was inputted as context for the LLM. As seen in the code 4.2, all of the steps which was required from the validation test case 4.1 have been covered in the test code. The system is initialized and all of the angles and transitions have their own tests. The code properly sets the signals and sends the appropriate commands according to past code examples. When comparing to the existing code at Volvo it has the same structure and way of testing.

The connection to the physical hardware and actually running the code on the HIL-rigs have not yet been executed. It can also be seen that the LLM is missing context since the functions and vehicle class are not imported properly, thus the program would not compile or run at its current state. Once again it would require human intervention in order to be a finished product and ready to execute. The code shows potential but the LLM, once more, is lacking some of the required context in order to produce a fully functional result.

```

1 import pytest
2 from time import sleep
3
4 class TestShakerControlFunction:
5

```

```

6  @pytest.fixture(scope="function")
7  def initialize_system(self, vehicle):
8      # Configure BucketShakerThreshold to 15 degrees as per specific
      ↪ configuration
9      vehicle.set_signal(['ControlSystem', 'BucketShakerThreshold'], 15)
10     yield
11     # Reset any signals post-test if needed
12
13  @pytest.mark.dotest
14  def test_shaker_not_available_below_10_degrees(self, vehicle,
      ↪ initialize_system):
15     """Verify Shaker Capability transitions to NotAvailable when tipping
      ↪ angle is 10°."""
16     vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], 10)
17     # Initiate shaker operation command via control interface
18     vehicle.send_command("InitiateShakerOperation")
19     sleep(2) # Wait for system response within specified timeout
20     assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) ==
      ↪ "ShakerNotAvailable"
21
22  @pytest.mark.dotest
23  def test_shaker_not_available_at_14_degrees(self, vehicle,
      ↪ initialize_system):
24     """Test tipping angle at 14° (just below threshold)."""
25     vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], 14)
26     vehicle.send_command("InitiateShakerOperation")
27     sleep(2)
28     assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) ==
      ↪ "ShakerNotAvailable"
29
30  @pytest.mark.dotest
31  def test_shaker_not_available_zero_degrees(self, vehicle,
      ↪ initialize_system):
32     """Check zero degree input condition."""
33     vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], 0)
34     vehicle.send_command("InitiateShakerOperation")
35     sleep(2)
36     assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) ==
      ↪ "ShakerNotAvailable"
37
38  @pytest.mark.dotest
39  def test_negative_angle_input(self, vehicle, initialize_system):
40     """Validate negative angles (-5°) trigger NotAvailable state."""
41     vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], -5)
42     vehicle.send_command("InitiateShakerOperation")
43     sleep(2)
44     assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) ==
      ↪ "ShakerNotAvailable"
45
46  @pytest.mark.dotest
47  def test_threshold_exact_value(self, vehicle, initialize_system):
48     """Ensure threshold value (15°) does not trigger
      ↪ ShakerNotAvailable."""
49     vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], 15)
50     vehicle.send_command("InitiateShakerOperation")

```

```

51     sleep(2)
52     # Expected to remain available or other state except NotAvailable
53     assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) !=
        ↪ "ShakerNotAvailable"
54
55     @pytest.mark.dotest
56     def test_above_threshold_condition(self, vehicle, initialize_system):
57         """Test angle above threshold (e.g., 20°)."""
58         vehicle.set_signal(['IMUSensor', 'TrailerBucketIMUStatus'], 20)
59         vehicle.send_command("InitiateShakerOperation")
60         sleep(2)
61         assert vehicle.get_signal(['ShakerOutput', 'ShakerCapability']) !=
            ↪ "ShakerNotAvailable"

```

Figure 4.2: Generated Python Code based on the given validation test case 4.1.

4.3 Validate Trailer Tipper Bucket State Transitions

These results have been generated in order to validate a trailer tipper bucket transitions. In order to ensure that the TrailerTipper correctly updates the BucketState based on the BucketDown signals and current BucketAngle.

4.3.1 Generated validation test case

This is a combination of four simple system requirements in order to test the functionality of the bucket. The LLM thus had to combine the four different system requirements into a single validation test case, which is a common practice at Volvo. The results have once again been manually extracted. The user prompt was also slightly modified to clarify that the given system requirements were to be combined. These are the following system requirements:

1. When Bucket angle transitions to Not available, the TrailerTipper State function shall set Bucket_state to BUCKET_STATE_NOT_AVAILABLE
2. When Bucket angle transitions to At or Above Threshold, the TrailerTipper State function shall set Bucket_state to BUCKET_STATE_UP
3. When BucketDown transitions to Inactive, the TrailerTipper State function shall set Bucket_state to BUCKET_STATE_MID
4. When BucketDown transitions to Active, the TrailerTipper State function shall set Bucket_state to BUCKET_STATE_DOWN

As seen in 4.2 the LLM took into account all four of the different system requirements, this is made clear under the expected behaviour section. The description is also accurate and the thresholds are asked to be defined by the user in the pre-conditions and configurations header, since this information was missing from the

system requirements alone. Once again the details are lacking as in the first validation test case 4.1, it would be difficult for someone to follow these steps without any prior knowledge of a HIL setup or testing in general. It shares similar strengths and weaknesses as validation test case 4.1. The general steps are clear and easy to follow, but the missing small details may make it impossible to follow through.

Validation Test Case for Trailer Tipper Bucket State Transitions colback

```
# PROJECT ID:
CAATS502-TIPPER-001

# GLOBAL ID:
GID-678901

# OBJECT TYPE:
Test

# NAME:
Validate Trailer Tipper Bucket State Transitions

# COMMON DESCRIPTION:

## Description:
This test verifies that the TrailerTipper State function correctly updates
the Bucket_state based on BucketDown signal transitions and BucketAngle
threshold conditions.

## Module:
Control System

## Test Type:
System Functional

## Precondition and configurations:
1. The system is in a stable operational state with all subsystems
   initialized.
2. HIL environment configured with simulated bucket sensors and actuator
   models.
3. Provide/Define value for parameter/threshold X (BucketDown transition
   delay)
4. Provide/Define value for parameter/threshold Y (Bucket angle activation
   threshold)
5. Ensure CANoe/Simulation tool is set up to monitor/record signals.

## Testing environment(HIL/SIL/Machine/Site):
HIL

## Input data:
-BucketDown signal (Active/Inactive)
-BucketAngle signal (numeric value in degrees)
-TippingActuationAvailableStatus (TRUE/FALSE)
-TrailerChassiIMUStatus (roll angle, if applicable)
```

```

## Expected behavior:
The TrailerTipper State function shall set Bucket_state to:
-BUCKET_STATE_DOWN when BucketDown transitions to Active
-BUCKET_STATE_MID when BucketDown transitions to Inactive
-BUCKET_STATE_UP when BucketAngle  $\geq$ Threshold Y
-BUCKET_STATE_NOT_AVAILABLE when BucketAngle is marked as Not Available

# SPECIFIC DESCRIPTION:

## Specific equipment:
HIL rig with simulated bucket position sensors, actuator controls, and CAN
communication interface

## Specific configuration:
Control system version P3/P4, Trailer model TA15B, IMU simulation enabled

## Specific data collection:
Record all input signals (BucketDown, BucketAngle) and output states
(Bucket_state) during test execution. Capture timestamped logs for state
transitions.

## Specific environmental conditions:
Ambient temperature within operational range (-20°C to +60°C), stable
power supply, no external vibrations.

# STEPS:
| # | Action | Expected Result | Notes |
|---|-----|-----|-----|
| 1 | Set BucketDown signal to Active | Bucket_state transitions to BUCKET_STATE_DOWN | Verify via HIL simulation and signal monitoring |
| 2 | Set BucketDown signal to Inactive | Bucket_state transitions to BUCKET_STATE_MID | Confirm via control system diagnostics |
| 3 | Set BucketAngle to a value  $\geq$ Threshold Y | Bucket_state transitions to BUCKET_STATE_UP | Ensure threshold Y is defined in preconditions |
| 4 | Set BucketAngle to Not Available | Bucket_state transitions to BUCKET_STATE_NOT_AVAILABLE | Validate error handling logic |
| 5 | Simulate simultaneous transitions: BucketDown=Active AND BucketAngle  $\geq$ Threshold Y | Bucket_state prioritizes BUCKET_STATE_DOWN over UP condition | Confirm state machine priority rules |
| 6 | Revert all inputs to default/neutral states | System returns to initial state (if applicable) | Verify system stability after test |

# ASSIGNEE:
*(Left blank)*

# RELEASE:
*(Left blank)*

# VALIDATION METHOD:
Test

# STATUS:
Ready for Review

```

```
# TEST CASE STATUS:
Not Scheduled

# EXTERNAL ID:
*(Left blank)*

# TEST RUN RESULTS:
*(Left blank)*

# TAGS:
TrailerTipper, StateTransition, SensorValidation, ControlSystem,
BucketState, HILTesting

--
# PROJECT ID:
CAATS502-TIPPER-001

# GLOBAL ID:
GID-678901

# OBJECT TYPE:
Test

# NAME:
Validate Trailer Tipper Bucket State Transitions

# COMMON DESCRIPTION:

## Description:
This test verifies that the TrailerTipper State function correctly updates
the Bucket_state based on BucketDown signal transitions and BucketAngle
threshold conditions.

## Module:
Control System

## Test Type:
System Functional

## Precondition and configurations:
1. The system is in a stable operational state with all subsystems
initialized.
2. HIL environment configured with simulated bucket sensors and actuator
models.
3. Provide/Define value for parameter/threshold X (BucketDown transition
delay)
4. Provide/Define value for parameter/threshold Y (Bucket angle activation
threshold)
5. Ensure CANoe/Simulation tool is set up to monitor/record signals.

## Testing environment(HIL/SIL/Machine/Site):
HIL

## Input data:
```

```

-BucketDown signal (Active/Inactive)
-BucketAngle signal (numeric value in degrees)
-TippingActuationAvailableStatus (TRUE/FALSE)
-TrailerChassiIMUStatus (roll angle, if applicable)

## Expected behavior:
The TrailerTipper State function shall set Bucket_state to:
-BUCKET_STATE_DOWN when BucketDown transitions to Active
-BUCKET_STATE_MID when BucketDown transitions to Inactive
-BUCKET_STATE_UP when BucketAngle  $\geq$ Threshold Y
-BUCKET_STATE_NOT_AVAILABLE when BucketAngle is marked as Not Available

# SPECIFIC DESCRIPTION:

## Specific equipment:
HIL rig with simulated bucket position sensors, actuator controls, and CAN

```

Markdown Code 4.2: The validation test case which was generated in order to verify that the transitions for the trailers tipper bucket are working correctly.

4.3.2 Generated Code

The generated code follows the same principle as before, the only difference being that validation test case 4.2 is fed into the LLM instead. This is then used as the base to generate the code. The results share many of the same strengths and weaknesses as before. Although the LLM did not opt to include fixtures for this code, in contrast to code block 4.2. Otherwise it asserts the signals in the same way and shares a similar architecture. As such it shares the same flaws as previously and would require an improved prompt, more context, or manual improvement in order to make the code executable and fulfil its function. One thing to note is that the code doesn't seem to test the transitions correctly, instead it seems to test the current state. It would also be necessary for the code to retrieve the signal from BucketDown, in a similar fashion to `vehicle.get_signal` from code block 4.2.

```
1 import pytest
2
3 class TestTrailerTipperBucketStateTransitions:
4     def test_bucket_down_active(self):
5         bucket_down = "Active"
6         bucket_angle = 0
7         tipping_available = True
8         imu_status = 0
9         assert self.trailer_tipper_state(bucket_down, bucket_angle,
10            ↪ tipping_available, imu_status) == "BUCKET_STATE_DOWN"
11
12     def test_bucket_down_inactive(self):
13         bucket_down = "Inactive"
14         bucket_angle = 0
15         tipping_available = True
16         imu_status = 0
17         assert self.trailer_tipper_state(bucket_down, bucket_angle,
18            ↪ tipping_available, imu_status) == "BUCKET_STATE_MID"
19
20     def test_bucket_angle_above_threshold(self):
21         bucket_down = "Inactive"
22         bucket_angle = 50 # Assuming threshold Y is 40
23         tipping_available = True
24         imu_status = 0
25         assert self.trailer_tipper_state(bucket_down, bucket_angle,
26            ↪ tipping_available, imu_status) == "BUCKET_STATE_UP"
27
28     def test_bucket_angle_not_available(self):
29         bucket_down = "Inactive"
30         bucket_angle = None # Not Available
31         tipping_available = True
32         imu_status = 0
33         assert self.trailer_tipper_state(bucket_down, bucket_angle,
34            ↪ tipping_available, imu_status) == "BUCKET_STATE_NOT_AVAILABLE"
35
36     def test_simultaneous_transitions(self):
37         bucket_down = "Active"
38         bucket_angle = 50 # Above threshold
39         tipping_available = True
40         imu_status = 0
41         assert self.trailer_tipper_state(bucket_down, bucket_angle,
42            ↪ tipping_available, imu_status) == "BUCKET_STATE_DOWN"
43
44     def test_revert_to_default(self):
45         bucket_down = "Inactive"
46         bucket_angle = 30 # Below threshold
47         tipping_available = False
48         imu_status = 0
49         result = self.trailer_tipper_state(bucket_down, bucket_angle,
50            ↪ tipping_available, imu_status)
51         assert result in ["BUCKET_STATE_MID", "BUCKET_STATE_NOT_AVAILABLE"]
52
53
```

```
47 def trailer_tipper_state(self, bucket_down, bucket_angle,
48 ↪ tipping_available, imu_status):
49     # Placeholder for the actual implementation
50     if bucket_down == "Active":
51         return "BUCKET_STATE_DOWN"
52     elif bucket_down == "Inactive":
53         return "BUCKET_STATE_MID"
54     elif bucket_angle is not None and bucket_angle >= 40 and
55 ↪ tipping_available:
56         return "BUCKET_STATE_UP"
57     elif bucket_angle is None:
58         return "BUCKET_STATE_NOT_AVAILABLE"
59     else:
60         return "BUCKET_STATE_MID"
```

Figure 4.3: Generated Python Code to validate Trailer Tipper Bucket State Transitions based on the given validation test case 4.2.

4.4 Overall

Overall the templates were filled and of decent quality. They would still need to be quality controlled by a human before being finished products. Therefore, the level of quality produced by the LLM is not yet sufficient enough to be autonomous and should instead be seen as a helpful tool to speed up progress. Since every demand may differ from the previous, it is hard to produce the perfect prompt for each instance using prompt engineering. The difference between getting a nearly complete result and an industry ready result is therefore enormous. The LLM is proficient at creating results which are general and cover most scenarios, but in return it lacks the specific context to fill in specific details and create complete and functional results. The AI-generated validation test cases and code are fairly similar to human created ones, especially the validation test cases are written in the same way as already existing ones. The code is also similar but lacks the functionality of the human created one, since it contains small bugs and issues, the structure however is the same.

An improved version could be having the user input more context specific to the system requirement. Such inputs could be how to initialize, preconditions and what environment it will be tested in. The test environment could be fed as context for future runs. RAG is also a technique that is worth exploring in order to add more context for the LLM. The only concern being how to filter the good data from the bad, since not all of Jama can be trusted to be of high quality.

Another issue that made itself apparent was the difficulty of quantifying what a good output is. Especially as two student which lack the work experience to judge the quality of the validation test cases and code which is produced. An improvement could be to create an AI whose sole purpose is to quantify the data and decide its' quality. Using this in combination with a generating AI, such as the one we worked with, could lead to a powerful feedback loop continuously improving the prompts,

and thus the results, based on the principle of prompt engineering 4.4. It could also be used directly to improve the results instead of the prompt. This may lead to superior results but could also decrease flexibility for future issues, since the results can't be reused in the same way as a well constructed prompt.

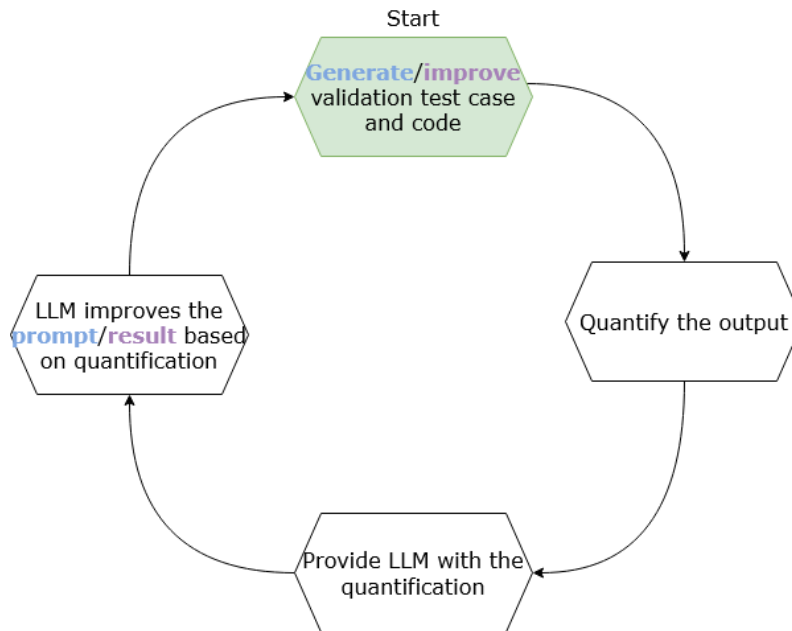


Figure 4.4: Figure illustrating the potential feedback loop of having two AIs work in combination to autotomize the principle of prompt engineering. Either by improving the prompt to create a better output, or by improving the output directly.

Another potential improvement which would be worth exploring is to enable a dialogue, similar to Chat-GPT and many other popular chat bots. This would allow the user to adjust and tweak the output of the LLM directly in order to improve the results. This possibility was briefly explored during development, but was not fully realized due to lack of time. An AI specialist at Volvo had recommended a path to get started, but also hinted at the possibility that it may turn out to be far more complicated than initially thought.

In summary, the results were satisfactory but not enough to make the process completely autonomous. It should still be able to serve as a valuable tool in increasing efficiency and cutting down time regarding testing. It increases existing system requirement testing by providing an easy way to create a strong foundation which can be built upon. The generated validation test case and code covered most of the system requirement. Although it was quite general, thus some details and context were missing from the output and would require additional improvement from a human in order to be fully complete. This issue could potentially be solved in several ways. Such as improving the prompts and providing more context specific to the system requirement at hand, or creating a feedback loop with an AI which quantifies the results. A non-autonomous solution would be to enable a dialogue with the LLM after a request has been sent, potentially improving flexibility as well as results.

5

Conclusion

The project, provided by Volvo, aimed to create an autonomous solution for generating validation test cases using generative AI for HIL-rigs. The result of the project is a tool that can be of assistance for creating validation test cases. The goal of a fully automated process has not been fully achieved but is in its current state a program that can assist and speed up the process of creating the test cases. The final model that was selected for the task was QWQ-32B and the programming language used for creating the program was Python. The validation test cases created so far are adequate but needs some refinement before being uploaded and used in production. Additionally, the generated code needs to be tested in a HIL-environment in order to determine its quality and functionality. After discussing the results with Volvo employees, the result was deemed reasonable but needs further adjustments with the inputs in order for the program to be deemed ready for production. Further improvements stated under results and discussion state ideas such as specific tailored context, allowing for dialogue, RAG, or creating a feedback loop for the LLM. In general, the program will provide Volvo with a base for further improvements in order to automate the process of generating validation test cases and related test code. As a concept, it could have potential to automate testing within several industries, extending beyond car industries and Volvo.

Bibliography

- [1] IBM. “Artificial intelligence.” Accessed: 2025-04-15. (2024), [Online]. Available: <https://www.ibm.com/think/topics/artificial-intelligence>.
- [2] S. Neemeh. “What is hil testing?” Accessed: 2025-04-15. (2023), [Online]. Available: <https://www.lhpes.com/blog/what-is-hil-testing>.
- [3] The Apache Software Foundation, *Apache License, Version 2.0*, <https://www.apache.org/licenses/LICENSE-2.0>, [Online; accessed 6-May-2025], 2004.
- [4] NASA Science, *What is artificial intelligence?* <https://www.nasa.gov/what-is-artificial-intelligence/>, [Online]. Updated May 13, 2024. Available: <https://www.nasa.gov/what-is-artificial-intelligence/>, [Accessed: Apr. 22, 2025], 2024.
- [5] R. T. Kreuzer and M. Sirrenberg, “What is artificial intelligence and how to exploit it?” In *Understanding Artificial Intelligence: Fundamentals, Use Cases and Methods for a Corporate AI Journey*. Cham: Springer International Publishing, 2020, pp. 1–57, ISBN: 978-3-030-25271-7. DOI: 10.1007/978-3-030-25271-7_1. [Online]. Available: https://doi.org/10.1007/978-3-030-25271-7_1.
- [6] R. Ulfsnes, N. B. Moe, V. Stray, and M. Skarpen, *Transforming Software Development with Generative AI: Empirical Insights on Collaboration and Workflow*. 2024, pp. 219–234, Cited by: 9. DOI: 10.1007/978-3-031-55642-5_10. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85205652290&doi=10.1007%2f978-3-031-55642-5_10&partnerID=40&md5=967e70b195ca2c53846b637c15b1218a.
- [7] Oracle, *What is an api?* Accessed: 2025-05-05, 2025. [Online]. Available: <https://www.oracle.com/cloud/cloud-native/api-management/what-is-api/>.
- [8] DHTMLX, *Web apis used in dhtmlx widgets*, Accessed: 2025-05-05, 2025. [Online]. Available: <https://dhtmlx.com/blog/web-apis-used-dhtmlx-widgets/>.
- [9] B. Meskó, “Prompt engineering as an important emerging skill for medical professionals: Tutorial,” *Journal of Medical Internet Research*, vol. 25, no. 1, 2023, Cited by: 228; All Open Access, Gold Open Access, Green Open Access. DOI: 10.2196/50638. [Online]. Available: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85173015974&doi=10.2196%2f50638&partnerID=40&md5=8b7d886ab3df29d5ff01aadb60436ebd>.

- [10] T. Elvira, T. T. Procko, J. O. Couder, and O. Ochoa, “Digital rubber duck: Leveraging large language models for extreme programming,” Cited by: 1, 2023, pp. 295–304. DOI: 10.1109/CSCE60160.2023.00051. [Online]. Available: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85191165473&doi=10.1109%2fCSCE60160.2023.00051&partnerID=40&md5=ff6207b68b2ea1aabdd35970fa2dddb3>.
- [11] F. Miladi, V. Psyché, and D. Lemire, “Comparative performance of gpt-4, rag-augmented gpt-4, and students in moocs,” *Communications in Computer and Information Science*, vol. 2162 CCIS, pp. 81–92, 2024, Cited by: 0. DOI: 10.1007/978-3-031-65996-6_7. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85200739796&doi=10.1007%2f978-3-031-65996-6_7&partnerID=40&md5=b70c533d14dde233565a0b7d3e59686b.
- [12] N. Setiani, R. Ferdiana, P. I. Santosa, and R. Hartanto, “Literature review on test case generation approach,” in *Proceedings of the 2nd International Conference on Software Engineering and Information Management*, ser. ICSIM ’19, Bali, Indonesia: Association for Computing Machinery, 2019, pp. 91–95, ISBN: 9781450366427. DOI: 10.1145/3305160.3305186. [Online]. Available: <https://doi.org/10.1145/3305160.3305186>.
- [13] S. Anand, E. K. Burke, T. Y. Chen, *et al.*, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013, ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2013.02.061>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121213000563>.
- [14] Visure Solutions, *Automotive requirements management*, Accessed: 2025-04-23, 2025. [Online]. Available: <https://visuresolutions.com/blog/automotive/requirements-management/>.
- [15] International Organization for Standardization, *Iso-standards*, Accessed: 2025-04-23, 2025. [Online]. Available: <https://www.iso.org/standards.html>.
- [16] Jama Software, *Jama connect® for automotive solution overview*, Accessed: 2025-04-24, 2025. [Online]. Available: <https://www.jamasoftware.com/solution-overview/jama-connect-for-automotive-solution-overview-thank-you/>.
- [17] pytest-dev, *Pytest documentation*, <https://docs.pytest.org/en/latest/>, Accessed on: 2025-04-22, 2025.
- [18] A. Liu, B. Feng, B. Xue, *et al.*, “Deepseek-v3 technical report,” DeepSeek-AI, Tech. Rep., Dec. 2024, Accessed: 2025-05-05. [Online]. Available: <https://deepseekv3.org/paper>.
- [19] DeepSeek-AI, *Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning*, 2025. arXiv: 2501.12948 [cs.CL]. [Online]. Available: <https://arxiv.org/abs/2501.12948>.
- [20] Qwen Team, *Qwq-32b: Embracing the power of reinforcement learning*, Accessed: 2025-05-17, Mar. 2025. [Online]. Available: <https://qwenlm.github.io/blog/qwq-32b/>.

- [21] Technology Innovation Institute, *Introducing the technology innovation institute’s falcon 3: Making advanced ai accessible and available to everyone, everywhere*, Accessed: 2025-05-17, 2024. [Online]. Available: <https://falconllm.tii.ae/>.
- [22] Anthropic, *Claude 3.5 sonnet model card addendum*, Accessed: 2025-05-17, 2024. [Online]. Available: <https://www.anthropic.com/news/claude-3-5-sonnet>.
- [23] DeepSeek, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025. [Online]. Available: <https://arxiv.org/abs/2501.12948>.

A

Appendix 1

TEST CASE TEMPLATE INSTRUCTIONS

You are to complete the template below by replacing fields marked with '[FILL]' with appropriate content. The '[FIXED]' and '[FILL]' markers are only guides and **must not** appear in your final output.

FORMATTING REQUIREMENTS

- Remove all '[FIXED]' and '[FILL]' markers from your final output
- Preserve all formatting, including bold text, line breaks, numbering, and spacing
- Use precise technical language appropriate for validation test cases
- Default values in italics are suggestions to use when no specific value is indicated
- Leave any field empty with a dash (-) if no relevant information exists

TEMPLATE STRUCTURE

[FIXED]# PROJECT ID:
[FILL]

[FIXED]# GLOBAL ID:
[FILL]

[FIXED]# OBJECT TYPE:
[FILL]

*(Default value: **Test**)*

[FIXED]# NAME:
[FILL]

[FIXED]# COMMON DESCRIPTION:

[FIXED]## Description:
[FILL]

[FIXED]## Module:
[FILL]

*(Default value: **Environment**)*

[FIXED]## Test Type:
[FILL]

```
*(Default value: **System Diagnostic**)*

[FIXED]## Precondition and configurations:
[FILL]

1.
2.
3.

[FIXED]## Testing environment(HIL/SIL/Machine/Site):
[FILL]

*(Default value: **HIL**)*

[FIXED]## Input data:
[FILL]

[FIXED]## Expected behavior:
[FILL]

[FIXED]# SPECIFIC DESCRIPTION:

[FIXED]## Specific equipment:
[FILL]

[FIXED]## Specific configuration:
[FILL]

[FIXED]## Specific data collection:
[FILL]

[FIXED]## Specific environmental conditions:
[FILL]

[FIXED]# STEPS:
[FILL]

[FIXED]# ASSIGNEE:
[FILL]

[FIXED]# RELEASE:
[FILL]

[FIXED]# VALIDATION METHOD:
[FILL]

[FIXED]# STATUS:
[FILL]

[FIXED]# TEST CASE STATUS:
[FILL]

[FIXED]# EXTERNAL ID:
[FILL]
```

```
[FIXED]# TEST RUN RESULTS:
[FILL]

[FIXED]# TAGS:
[FILL]

---

## VERIFICATION STEPS

After filling the template:

1. Ensure all '[FIXED]' and '[FILL]' markers have been completely removed
2. Verify all fields have appropriate content based on the provided context
3. Use default values when appropriate, but customize based on context
4. Ensure all formatting is preserved exactly as shown in the template
5. Check that all required fields have content (or an explicit dash if empty)

## EXPECTED OUTPUT FORMAT

Your output should look like this (example snippet):
```

Markdown Code A.1: Validation Test Case template

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se



CHALMERS
UNIVERSITY OF TECHNOLOGY