AI-driven Software Validation

How can AI be used in software testing?

IVAN BOGOSAVLJEVIC & SEBASTIAN MALMQVIST
BACHELOR´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

AI-driven Software Validation

How can AI be used in software testing?

LUND
UNIVERSITY

Authors: Ivan Bogosavljevic, Sebastian Malmqvist

Department of Computer Science
Faculty of Engineering, Lund University

Supervisor: Yamen Albdeiwi, Christin Swahn

Examiner: Christian Nyberg

2025-06-07

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The process of writing test specifications from software requirements is time consuming and relies on manual effort. This thesis explores how a large language model (LLM) can automate parts of this validation process by generating test specifications from software requirements. Using a dataset extracted from Polarion consisting of 186 requirements and 371 test specifications, the Mistral-7B model was used along with various data augmentation techniques to generate additional training samples. The data augmentation produced 1203 new samples which were used to train the AI model. Evaluation was conducted using both automatic similarity metrics such as BLEU, ROGUE and cosine similarity and through human evaluation which involves engineers at the automotive company scoring the generated test specifications. The results show that AI generated test specifications can match or be slightly worse than the human written specifications. This suggests that LLMs can be a valuable tool in the software validation workflow. Future work should focus on refining the data augmentation process as a larger dataset is needed and prompt engineering strategies to get a better result.

Keywords: NLP, LLM, software testing, test specification, requirements analysis, prompt engineering, Mistral-7B, data augmentation, evaluation

Sammanfattning

Processen att skriva testspecifikationer utifrån mjukvarukrav är tidskrävande och bygger på manuellt arbete. Detta examensarbete undersöker hur en stor språkmodell (LLM) kan automatisera delar av denna valideringsprocess genom att generera testspecifikationer från mjukvarukrav. Med ett dataset hämtat från Polarion som bestod av 186 krav och 371 testspecifikationer, användes modellen Mistral-7B tillsammans med olika dataaugmenteringstekniker för att generera ytterligare träningsdata. Dataaugmenteringen resulterade i 1203 styck av ny data som användes för att träna AI-modellen. Utvärderingen genomfördes både med automatiska likhetsmått som BLEU, ROUGE och cosinuslikhet samt genom manuell granskning där ingenjörer på fordonsföretaget bedömde de genererade testspecifikationerna. Resultaten visar att AI-genererade testspecifikationer kan matcha eller vara något sämre än de människoskrivna specifikationerna. Detta tyder på att LLMs kan vara ett värdefullt verktyg i mjukvaruvalideringsprocessen. Framtida arbete bör fokusera på att förfina dataaugmenteringsprocessen då ett större dataset behövs samt vidareutveckla prompttekniker för att uppnå bättre resultat.

Nyckelord: NLP, LLM, mjukvarutestning, testspecifikation, kravanalys, prompt engineering, Mistral-7B, dataaugmentering, evaluering

Foreword

First and foremost we would like to thank our supervisor Yamen Albdeiwi for his help, guidance and expertise throughout the thesis. His feedback and support was invaluable during every stage of the process. We would also like to express our gratitude toward the automotive company for giving us the opportunity to write the thesis with them. We would also like to give a special thanks to Robin for believing in us to give us the opportunity even though we are new to the field of AI. Furthermore, we would like to express appreciation for our supervisor Christin Swahn at Lund for her consistent guidance, feedback and encouragement throughout the process. Her expertise helped shape the direction and quality of the report. Completing this thesis has been a significant learning experience for both of us and has given us real world experience in AI. We hope that the work presented here contributes to the field and can be the value to others.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter provides a concise overview of the work, outlining the challenges of validating software requirements in the automotive sector and highlighting the thesis's core objectives. It discusses the repetitive nature of creating test cases for similar or recurring requirements and proposes an AI-driven approach to solve this. By automating the generation of test specifications, the thesis aims to reduce redundancy and free up valuable resources for more critical tasks.

1.1 Background

This thesis aims to improve the process of validating software requirements at an automotive company. Compliance with safety regulations through validation is critical in this sector, where software directly impacts vehicle performance and safety.

The current workflow begins with the customer providing requirements, which are then negotiated and finalized. Test cases are manually created based on these finalized requirements.

The primary challenge lies in the fact that many of these requirements are generic, applicable across multiple projects, or nearly identical with slight modifications. Every vehicle manufacturer needs to adhere to regulations and ISO standards, which impose specific constraints on requirements. As a result, requirements often share similar patterns, structures, and semantics but may differ in phrasing or contain minor variations. Despite these similarities, resources are frequently spent rewriting these requirements and creating new test cases for each project, even though similar or identical requirements have already been implemented and tested in previous projects.

To address this, this thesis proposes developing an AI-based prototype capable of analyzing requirements, identifying similarities with previously implemented requirements, and retrieving their related test specifications. The AI model will generate new test specifications based on these retrieved references, incorporating necessary adjustments where required. This approach focuses specifically on automating the generation of test specifications, not the complete test cases, to improve efficiency while maintaining high accuracy and compliance.

It is important to note that this thesis is part of a larger project aimed at improving the overall software validation process. While the thesis contributes to this broader effort, the results and findings will be presented separately. The effectiveness of the proposed solution will be evaluated by analyzing the number of test specifications generated by

the AI prototype, its accuracy, and the amount of time saved compared to the current manual process. By automating a significant portion of the workflow, this solution aims to streamline the validation process, reduce redundancy, and free up valuable resources for more critical tasks

1.2 Purpose

The purpose of this thesis is to explore whether the implementation of test cases can be improved through automation using an AI -prototype. The expected outcome is that the prototype will be able to generate test specifications for frequently recurring requirements and provide a reliable template for less common requirements.

1.3 Goals

This thesis explores whether automatic test implementation of recurring requirements is possible through the help of an prototype. If possible the prototype will be able to take a requirement, find similar implemented requirements in old or ongoing projects and use these as basis for generating test specifications. To achieve this goal several sub-tasks much be completed:

1. **Data Engineering:** Establish a data engineering pipeline that:
    - Collects raw data using web scraping and APIs.
    - Cleans the collected data by removing duplicates and filtering out irrelevant content.
    - Normalizes the data to a standardized format and tokenizes it using the same tokenizer as the pre-trained models.
    - Maps data elements by establishing clear connections between source and target schemas, ensuring accurate data transformation and integration.
2. **Model Development:** Select and implement an optimal modeling approach by:
    - Conducting research into available models.
    - Exploring different application strategies such as fine-tuning a selected model on a labeled dataset or applying prompt engineering techniques.
3. **Evaluation:** Determine the efficiency and accuracy of the implemented approach. This involves:
    - Addressing the challenges in evaluating language-based outputs.
    - Using multiple evaluation techniques (e.g., NNI, BLEU, ROUGE) to measure the quality of the generated test specifications. This includes human evaluation.

1.4 Problem Formulation

The following questions will be addressed through this thesis to identify and implement potential improvements in the requirement validation process:

1. How are software requirements currently validated?
2. Which parts of the current test process are repetitive and resource intensive?

3. How can an AI model identify similarities between new requirements and previously implemented requirements?
4. How can automation of test case generation improve the efficiency and accuracy of the validation process?
5. What criteria should be used to evaluate the effectiveness of the proposed prototype?

1.5 Motivation of thesis

The motivation for this thesis is to ease the validation process for all automotive engineers. Unexpected vehicle malfunctions can sometimes emerge during operation. By implementing automated testing, engineers can dedicate more time to developing and improving systems, while automatic tests verify new implementations continuously. This could potentially minimize manual validation and strengthen the testing process by catching potential issues early.

This approach benefits companies by reducing costs and automating validation, and it also has significant global implications. Vehicles are a critical part of everyday life, ensuring their reliability and safety is important. Our motivation is for this proof of concept to encourage automotive companies worldwide to adopt more efficient validation processes, contributing to safe and dependable vehicles.

1.6 Work distribution

This section outlines how the work was divided throughout the thesis. Table 1.1 presents the distribution of responsibilites across the main thesis activities.

Table 1.1: Table illustrating the work distribution of general tasks in percentages (%).

	Ivan Bogosavljevic	Sebastian Malmqvist
Analysis	50	50
Design	50	50
Implementation/Construction	50	50
Report and presentation	50	50

Chapter 2

Technical Background

This chapter details the technical concepts underlying the thesis, from foundational artificial intelligence principles to the specific frameworks and tools used for data engineering. It starts by exploring artificial intelligence and machine learning, highlighting how models learn patterns from data. It then dives deeper into deep learning, covering essential ideas like neural networks, transfer learning, and natural language processing techniques such as transformers. The chapter also introduces the libraries and platforms most important to the implementation, including PyTorch, TensorFlow, NumPy, and Pandas. Lastly it covers the different aspects of data handling like scraping and API calls.

2.1 Artificial Intelligence

Artificial intelligence (AI) is a term used for a multidisciplinary field of computer science focused on building systems capable of performing tasks that typically require human cognitive abilities. These tasks include problem solving, reasoning, and decision making

AI is traditionally divided into two categories:

1. General AI
    - General AI is a term used for a theoretical construct of a model where the system possesses human like adaptability across multiple domains. Humans have not managed to build this type of model yet.
2. Narrow AI
    - Narrow AI is a term used for a model where the system excels at specific tasks. These systems are widely used today in various fields, such as managing spam filters for email providers or powering typical chatbot assistants.

As of 2025 most of AI researchers and practitioners are focusing on developing generative AI (gen AI), a model capable of creating original content that has never been made by a human before. There are many subdomains inside AI which are built upon different types of technologies that need their own introduction[1]. A overview of the different subdomains of AI can be found in figure 2.1

2.2 Machine Learning

Machine learning is a subset of artificial intelligence that enables computers to improve performance after making observations about the environment[2]. This allows the com-

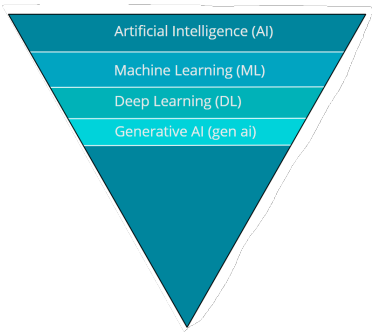


Figure 2.1: Subdomains of AI showing that all the subdomains are built on top of Artificial Intelligence as the base.

puter to learn patterns from data and make decisions on its own rather than being explicitly programmed for every task. The idea is to provide the computer enough data so that it can learn patterns and adjust its internal parameters, commonly referred to as weights. This process is known as training. After being fed numerous examples, the adjusted weights will, in theory, generate accurate predictions or decisions for new, unseen data. Machine learning is especially useful for problems that humans solve intuitively. For example, identifying a friend in a crowd involves subtle cues like facial features and body language that are difficult to hard-code. Instead, a machine learning model can be trained on data to recognize these patterns and accurately identify your friend.

Machine learning techniques can be categorized into three types:

1. **Supervised Learning:** In supervised learning, each training example consists of an input together with its correct output, or label. The model learns to predict the label given the input. If the labels are real numbers, the task is called regression. If the labels come from a fixed set of categories, the task is called classification.
2. **Unsupervised Learning:** The model works with unlabeled data and attempts to identify underlying patterns or structures. Common applications include clustering as an efficient way of grouping unlabeled data[2].
3. **Reinforcement Learning:** An agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal is to learn a policy that maximizes the cumulative reward.

2.2.1 Neural Computation

In many ML models, the basic building block is the cell (or neuron). A layer is a group of neurons. Each neuron performs calculations by taking its inputs (the outputs of all neurons in the previous layer), multiplying each by its corresponding weight, adding a bias term, and finally applying an activation function. The calculation can be formalized as:

$$\hat{y} = \phi\Big(\sum_{i=1}^{n} w_i\, x_i + b\Big)$$

where

- x_i is the scalar output of the ith neuron in the preceding layer (so the full vector of inputs is $\mathbf{x} = [x_1, \ldots, x_n]^\top$),

- w_i is the weight applied to x_i (forming the weight vector $\mathbf{w} = [w_1, \ldots, w_n]^\top$),
- b is the bias term,
- ϕ is the activation function,
- $\hat{y}$ is the scalar output of the current neuron, also known as the prediction.

2.2.2 Training Process

Every model must iteratively adjust the weights of the model's neurons to improve prediction accuracy. This procedure is known as the training process; see figure 2.2 was improved by Rumelhart et al. (1986) who replaced the simple perceptron-convergence procedure with back-propagation, which is widely used today. Modern implementations typically use normalization techniques to standardize the input to each layer by ensuring consistent mean and variance. The steps include:

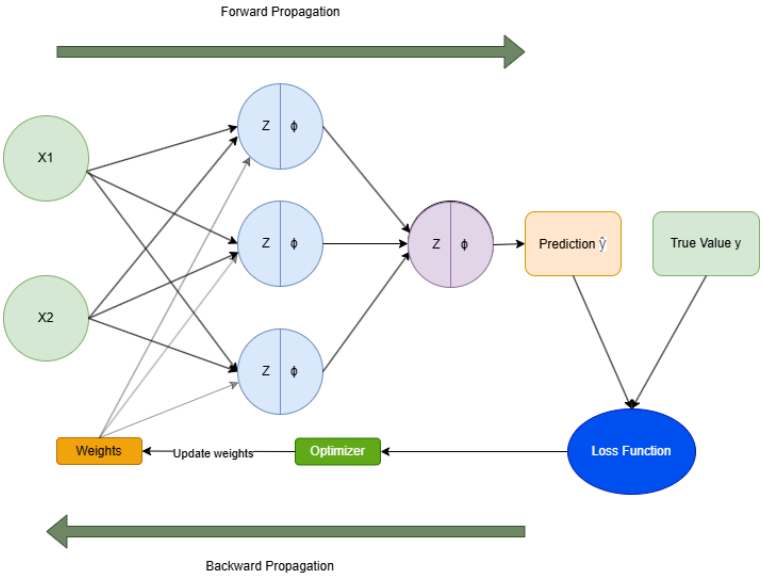


Figure 2.2: Training process cycle

1. **Forward Pass:** The network computes predictions for a given input by propagating the data through its layers. For a single neuron, the computation is:

$$z = \sum_{i=1}^{n} w_i x_i + b,$$

which represents the weighted sum before applying the activation function. The neuron's output is then given by:

$$\hat{y} = \phi(z).$$

2. **Loss Function:** A loss function evaluates the model's predicted outputs, $\hat{y}$, and the true target values, y. One widely used loss function is the Mean Squared Error, defined as:
$$L = \text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\big(y_i - \hat{y}_i\big)^2.$$
which computes the average of the squared differences between the predicted and actual values.

3. **Backward Pass (Backpropagation):** The error calculated by the loss function is propagated backwards using the chain rule to compute the gradients of each weight and bias. Given the loss function L, the gradient to a weight w_i is computed using:
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \phi'(z) \cdot x_i,$$
and the gradient to the bias: b is:
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y} \cdot \phi'(z).$$

4. **Optimization:** Using the computed gradients, the weights can be updated iteratively to minimize the loss. This optimization can be done using gradient descent or any optimization technique. For gradient descent the weight and bias update per iteration can be computed using:
$$w \leftarrow w - \eta \frac{\partial L}{\partial w}, \qquad b \leftarrow b - \eta \frac{\partial L}{\partial b}.$$

5. **Stopping Criteria:** Training typically stops when one of the following occurs:
   - The gradients $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$ become (near) zero, indicating convergence to a local minimum.
   - A fixed number of epochs is reached.
   - No improvement in validation loss for a specified number of epochs (early stopping).
   - The norm of the gradient $\|\nabla L\|$ falls below a small threshold.

2.2.3 Inference

After training is complete, the next step is inference which is when the trained model is introduced to new, unseen data to obtain predictions without any further adjustment of its parameters. Given an input x, the model performs a forward pass $\hat{y} = f(x; \mathbf{w}, b)$ using the fixed weights $\mathbf{w}$ and biases b. No gradients are computed and no weight updates occur during inference. In regression or classification tasks, inference produces a single prediction $\hat{y}$. Inference is typically much faster than training, since it requires just one forward pass through the network.

2.2.4 Activation Functions

An activation function (such as sigmoid, tanh, ReLU, or softmax) is used to determine the neuron's output based on the computed sum. Activation functions are crucial for introducing non-linearity into the model, which enables the network to learn complex patterns beyond simple linear relationships[2].

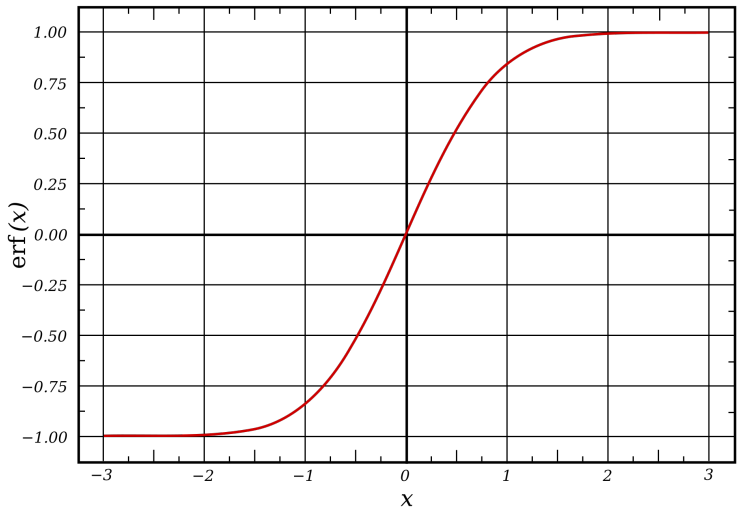


Figure 2.3: Sigmoid function (Wikipedia sigmoid 2024)

In such cases, the output of the network might represent probabilities for each class rather than a single continuous value. For example as shown in figure 2.3, the sigmoid activation function (a) varies with an output from 0 to 1 which is very useful for classification problems.

2.2.5 Features

Features are characteristics or properties of the input data. For example, when predicting housing prices, features might include square footage or the number of bedrooms. During training, the model learns to assign importance to these features. Features can also be manually engineered or automatically learned by neural networks[3]. In traditional machine learning feature engineering which is manual creation of features is crucial.

2.3 Deep Learning

Deep learning (DL) is a subfield of machine learning that constructs neural networks with many stacked layers capable of learning hierarchical features from both labelled and unlabelled data[4]. As data propagates through each layer, the network transforms raw inputs into progressively more abstract representations. Early layers might detect simple patterns like edges or textures, middle layers capture shapes or design, and deeper layers recognize complex concepts. Whereas classical ML models rarely exceed two or three layers, modern DL architectures can extend to hundreds or even thousands, enabling them to model highly non-linear relationships. These networks are trained with the same forward/backward propagation loop outlined in section 2.2 with additional refinements:

- **Weight initialization**: Very essential and directly impacts the stability of convergence. Convergence is the moment when optimization process settles meaning the training loss no longer decreases and the model's parameters change very little

indicating it's no longer learning. If weights are too large or too small, gradients can explode or vanish, making training unstable and preventing convergence.

- **Residual/skip connections**: Shortcut paths allow gradients to bypass multiple layers, helping with gradient problems that may occur.
- **Adaptive optimizers**: Algorithms such as Adam or RMSProp adjust learning rates per parameter and speed convergence.
- **Regularization**: Dropout, data augmentation, weight decay, and early stopping reduce over fitting in over parametrized models.

2.4 Natural Language Processing

Natural Language Processing (NLP) is a branch of AI that enables machines to understand, manipulate, and generate human language [5]. There are important preprocessing steps of any input text that a model must perform before any linguistic task. Text preprocessing involves three successive steps:

1. Tokenization: Split raw text into tokens, tokens can be words, subwords, or characters.
2. Token ID mapping: Assign each token a unique integer ID based on the model's vocabulary. Figure 2.4 represents these convertions.

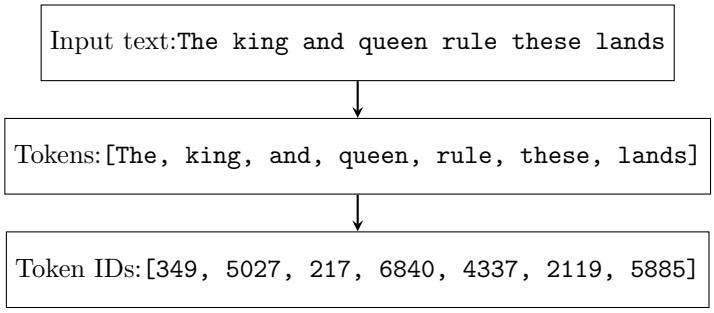


Figure 2.4: From text to token IDs.

3. Embeddings: Transform each token ID into a dense vector that captures semantic similarity (e.g. "king" and "queen" have nearby embeddings in vector space), Figure 2.5 shows an example of the embeddings king and queen.

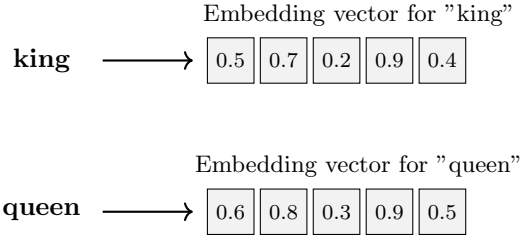


Figure 2.5: Mapping of tokens to embedding vectors.

After converting the text into numerical and embeddings, the model begins processing these vectors to understand context and meaning. In most NLP architectures, particularly transformers, this is achieved through the processes of contextual encoding and decoding.

- **Encoding:** The encoder converts the sequence of embeddings into high-dimensional contextual representations, capturing the meaning of each token in relation to the entire sequence.
- **Decoding:** The decoder generates output sequences e.g., translated text, summaries based on the encoded information, often autoregressively producing one token at a time while attending to both the encoder output and previously generated tokens.

Some models use only encoders, focusing on understanding input text, while others use decoders for text generation. Encoder-decoder models combine both for tasks like translation or summarization.

Beyond general encoding and decoding[5], four core areas enhance a model's ability to process and comprehend text:

- **Language Modeling:** Estimating probabilities of word sequences, fundamental for tasks like text generation and predictive typing.
- **Morphological Analysis:** Examining word internal structure like roots, prefixes or suffixes.
- **Parsing:** Recovering sentence syntax to understand grammatical relations between words.
- **Semantic Analysis:** Capturing meaning at the word, phrase, and sentence levels.

While both traditional ML and modern DL techniques have driven progress in NLP, most recent breakthroughs use DL models built on the transformer architecture[5]. By implementing self-attention mechanisms and vast pretraining on large texts, transformer-based systems excel at a diverse range of NLP tasks with minimal task-specific engineering. Common NLP tasks involve text summarization, translation and text generation.

2.4.1 Transformer models

The transformer model[6], is a neural network based on attention mechanisms that captures dependencies between sequential inputs, allowing parallelization. Transformers are currently the standard architecture for building large language models and have changed the field of NLP completely. The proposed architecture, shown in figure 2.6, consists of 6 encoder layers and 6 decoder layers[6].
Before processing, the model computes token embeddings with positional encoding, providing information about the initial word sequence. The positional encodings are computed as:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \quad \text{and} \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where pos is the word's position, i is the dimension, and d_{model} is the model's dimensionality. Figure 2.7 illustrates positional encoding applied to an input sequence.

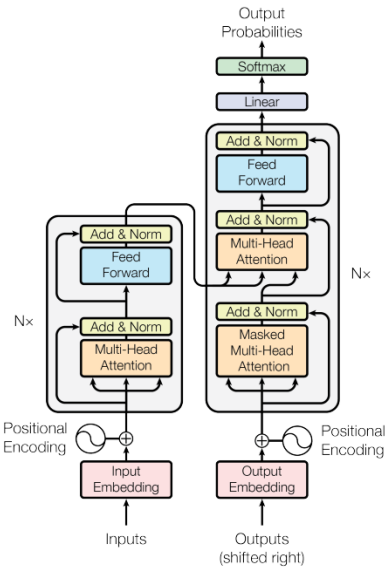
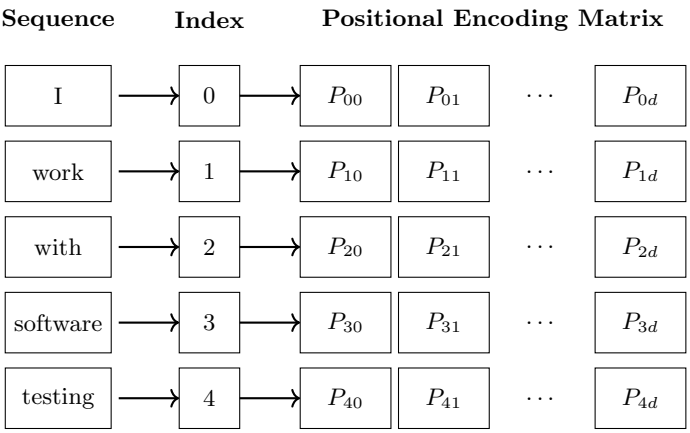


Figure 2.6: The Transformer architecture (Vaswani et al., 2017)[6].



Positional Encoding Matrix for the sequence "I work with software testing"

Figure 2.7: Visualization of positional encoding aligned to index positions.

The encoder maps tokens to learned embedding vectors (plus positional encodings) and processes them through each of the 6 layers. Each encoder layer contains:

- Multi-head self-attention: Every token attends to all other tokens in the sequence.
- Feed-forward network: Processes the outputs of the attention mechanism.

Each decoder layer contains:

- Masked multi-head self-attention: Allows the decoder to attend only to previous positions.
- Encoder-decoder attention: Lets the decoder attend to the encoder's output representations.
- Feed-forward network.

Both encoder and decoder layers rely on self-attention, which allows each token to weigh its relationships with other tokens. This is computed using an attention form created by Vaswani et al.(2017) called Scaled Dot-Product Attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where:

- Q = Query vectors
- K = Key vectors
- V = Value vectors
- d_k = Dimension of the keys

This calculation determines how much each token should pay attention to every other token in the sequence. Figure 2.8 illustrates this attention mechanism.

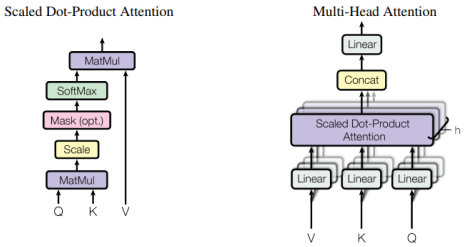


Figure 2.8: Scaled Dot-Product Attention mechanism used in self-attention.Vaswani et al(2017)

Now instead of performing attention once, multi-head attention runs the self-attention mechanism multiple times in parallel. This allows the model to focus on different parts of the sequence simultaneously.

2.4.2 Large Language Models

Large language models(LLM) are a type of deep neural networks. Modern LLMs are typically based on the transformer architecture, which has been pretrained on massive texts to learn about different languages and world knowledge. During pretraining, an LLM is taught to predict each next token given its preceding context by minimizing the loss function. By iteratively repeating this task, the model learns to encode distributional patterns, syntax, and semantics

All knowledge acquired during pretraining resides in the model parameters(weights and biases). Depending on how many parameters a model has, retraining can be very challenging. To be able to provide LLMs domain specific information without retraining, one

can combine them with an external retrieval system. In Retrieval-Augmented Generation(RAG), at inference time a retriever fetches relevant documents from a corpus and combines it with the user input to generate responses grounded on both user input and external knowledge. User input to an LLM is more commonly known as a prompt.

Given a prompt, an LLM generates text one token at a time. To decide each next token, it can use several decoding strategies:

1. Greedy decoding: At each step, select the single token with highest probability. Often results in very generic texts.
2. Sampling:Draws the next token at random from the full probability distribution. This introduces variation and can produce more creative outputs. The variation can be adjust by an input parameter that is known as the temperature.
3. Beam search: Maintain the top k most likely partial sequences the "beam" at each step, expanding each and keeping only the best k overall.

After pretraining, LLMs can be fine-tuned on downstream tasks by training on task specific labeled data, which adapts the model's parameters for that domain. Alternatively, without changing any weights, they can be prompt-engineered by reconstructing the input prompt. Guiding the pretrained model to perform new tasks it wasn't trained on. Both techniques make LLMs powerful across a wide range of applications.

2.4.3 Mistral 7B

Mistral-7B is a 7.3 billion-parameter, decoder-only transformer that specializes in natural language generation (NLG). The Mistral-7B was first introduced as a solution to improve efficiency without increasing modelsize[7]. Benchmark results show that Mistral-7B outperforms all models with the same number of parameters on overall NLP tasks, and even surpasses models with up to 34 billion parameters on many benchmarks see figure 2.9.

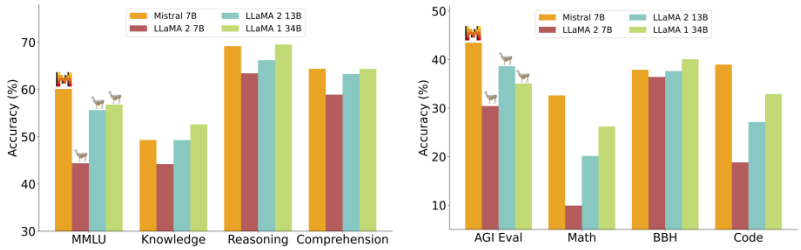


Figure 2.9: Mistral-7B benchmark compared to other models (Jiang et al.,2023).

Mistral uses a decoder-only architecture, meaning it only applies unidirectional context to predict the next token in a sequence iteratively. In other words, Mistral attends only to previous tokens when generating the next token, a process also known as autoregressive attention.

Its architecture is based on the original Transformer but without an encoder block and includes several structural differences such as Sliding Window Attention, Grouped-Query

Attention and Rotary Position Embeddings. These improvements together with some key strategies contribute to efficiency and can be summarized as follows:

- Root Mean Square Layer Normalization (RMSNorm): a simplified alternative to LayerNorm[8]. RMSNorm avoids mean subtraction and normalizes only by the root mean square of the input values:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\text{mean}(x^2) + \epsilon}} \cdot \gamma \tag{2.1}$$

  In Mistral 7B, RMSNorm improves training stability and efficiency, especially in deeper transformer layers, while reducing computational overhead.

- Grouped-Query Attention (GQA): GQA is a hybrid attention mechanism[9]. A compromise between Multi-Head Attention (MHA)[6] and Multi-Query Attention (MQA)[10]. In MHA, each attention head has unique key (K) and value (V) matrices. In contrast, MQA shares a single K and V across all heads. GQA introduces an hybryd approach where query heads (Q) are grouped, and heads within each group share K and V matrices. This reduces memory usage while preserving most of the performance of MHA.

- Sliding-Window Attention (SWA): an efficient attention variant where each token attends only to a fixed-size local window around it, rather than the entire sequence[11]. This reduces the quadratic cost of standard attention $O(n^2)$ to linear $O(n)$ by limiting attention to nearby tokens. SWA allows Mistral 7B to process longer sequences without excessive memory usage. Figure 2.10 shows the sliding-window attention.

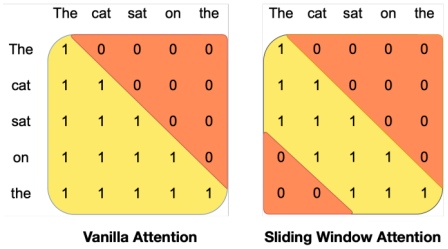


Figure 2.10: Sliding-window attention. Vaswani et al.(2017).

- Switchable Gated Linear Units (SWiGLU): the SwiGLU activation[12] combines the Swish activation[13] and Gated Linear Units (GLU)[14]

$$\text{SwiGLU}(x) = \text{Swish}(xW_1 + b_1) \odot (xW_2 + b_2) \tag{2.2}$$

  where $\odot$ denotes element-wise multiplication. In Mistral 7B, SWiGLU enhances model capacity and improves convergence compared to traditional activations like ReLU and GELU.

- Rotary Position Embedding (RoPE): RoPE[15] encodes positional information by rotating the query (Q) and key (K) vectors based on token positions:

$$\text{Attention}(Q, K) = \text{softmax}\left(\frac{(QR_m)(KR_n)^\top}{\sqrt{d}}\right) \tag{2.3}$$

  Here, R_m and R_n are rotation matrices encoding the positions m and n. RoPE allows Mistral 7B to generalize to longer sequences and preserve relative positional information more effectively than fixed or learned positional embeddings.

- Rolling Buffer Cache: a memory optimization first explored in Transformer-XL[16] and further developed by Rae et al. (2019)[17]. The rolling buffer stores only recent key and value vectors in a limited cache. This helps to reduce memory usage since when the cache reaches its limit it overwrites previous values. It also enables long-context inference without excessive memory growth.In Mistral 7B, rolling buffers work together with SWA to support efficient decoding over long sequences.

These are all key features which enable the Mistral-7B to keep high performance and efficiency while maintaining a lower parameter count effectively decreasing computing power. The Mistral-7B model specifications can be found in Table 2.1.

Table 2.1: Mistral 7B architecture specifications.

Parameter	**Value**
Architecture	Decoder-Only
Hidden size	4,096
Number of attention heads	32
Attention heads (query)	32
Attention heads (key, value)	8
Feedforward dimension	14,336
Total parameters	7 billion
Vocabulary size	32,000
Context window (sequence length)	8,192 tokens

Mistral-7B has two versions: base and instruct. The base model is designed for general-purpose language modeling and can be fine-tuned for a wide range of downstream tasks. The instruct model is trained with instruction-following datasets and optimized to respond to human instructions. This method is known as instruction fine-tuning and has shown improvements in performance when prompting a model in unseen tasks Wei et al. (2022).Jiang et al.(2023) results show strong performance for the instruct model, often outperforming models of similar or larger size on instruction-following benchmarks.

Among the Mistral vocabulary size, 1000 tokens are reserved for what is known as control tokens. Control tokens are used in the encoding process to represent specific instructions or indicators. This can be utilized during prompt engineering to create constructive guidance for the model to perform a task(Mistral AI 2025). Table 2.2 lists the control token currently used from the 1000 total reserved tokens.

2.5 Prompt Engineering

A prompt is the input provided to a language model that guides it to generate an output. Prompt engineering is the practice of constructing prompts to generate accurate responses from LLMs. Prompt structure has a significant impact on model performance(Brown et al., 2020).

An effective prompt clearly conveys the intended task. The intention can be expressed using clear instructions, example-driven learning, role-setting, or formatting. Finding the optimal prompt is often challenging. Providing too much or too little information can lead to inaccurate or fabricated outputs, commonly referred to as hallucinations. Achieving the right balance requires a trial and error approach. Since each language model differs in its capabilities and behavior, prompt effectiveness can vary significantly across models and tasks.

Table 2.2: Mistral 7B control tokens and their purpose (Mistral AI, 2025).

Control Token	Purpose
<unk>	Represents unknown tokens not found in the model's vocabulary.
<s>	Marks the beginning of a sequence (BOS - Beginning Of Sequence).
</s>	Marks the end of a sequence (EOS - End Of Sequence).
[INST]	Indicates the start of an instruction in instruction-tuning format.
[/INST]	Indicates the end of an instruction.
[AVAILABLE_TOOLS]	Marks the beginning of a list of tools available to the model.
[/AVAILABLE_TOOLS]	Marks the end of the available tools list.
[TOOL_RESULTS]	Marks the beginning of tool execution results.
[/TOOL_RESULTS]	Marks the end of tool execution results.
[TOOL_CALLS]	Marks the beginning of tool call information.
<pad>	Used for padding sequences to a uniform length.
[PREFIX]	Indicates the prefix segment in a structured prompt.
[MIDDLE]	Indicates the middle segment in a structured prompt.
[SUFFIX]	Indicates the suffix segment in a structured prompt.

2.5.1 Prompt techniques

There are several common techniques in prompt engineering, many of these leverage in-context learning (ICL) which refers to the ability of AI to learn skills by providing examples or instructions within the prompt:

Zero-shot prompting

The task is given with no examples. This works well if the model has seen similar patterns during pretraining, but might fail on specific or domain-heavy tasks. A example of this prompt can be found in Figure 2.11. Models trained using instruction fine-tuning has shown improvement in this area for unseen tasks (Wei et al., 2022).

```
Generate a test specification using following title:
UDS Service - 0x11 ECUReset
```

Figure 2.11: Zero-shot example

Half-shot prompting

Half-shot prompting is when the model is given an input x without showing the corresponding expected output y.This provides partial context about the input format but leaves the model to infer the output pattern independently, acting like a 'black box' since the user cannot directly guide the output logic with examples, see figure 2.12.

```
Generate a test specification for the following requirement:
"The system shall revert to default mode after a 5 second
timeout"
```

Figure 2.12: Half-shot instructional prompt example

Few-shot prompting

One or more examples of how a correct input–output pair should look is provided. This improves reliability, especially for structured tasks like test generation since it provides the model with ICL and a template to follow (Brown et al., 2020). A example of few-shot prompting can be found in Figure 2.13

```
Using the following example.
Requirement: Fail safe shall trigger when the system receives
code 0x03.
Test Specification: The system shall trigger fail safe when
receiving 0x03.
Test ID: T0
Title: Failsafe 0x03
Steps: Send code 0x03 to the system.
Expected Result: Fail safe triggered.

Generate a test specification for the following requirement.
Requirement: The system shall log out after 15 minutes of
inactivity.
Test Specification:
```

Figure 2.13: Few-shot prompting example

Well structured few-shot prompts have proven to have competitive performance even against fine-tuned approaches (Brown et al., 2020).

Common techniques within shot-prompts

Common techniques within shot prompting are role prompting and style prompting. Figure 2.14 and 2.15 illustrates these techniques.

```
Given a requirement, generate a test specification including
Test ID, Title, Steps, and Expected Result.
```

Figure 2.14: Style prompt example

```
You are a QA engineer. Based on the requirement below, generate
a test specification.
```

Figure 2.15: Role prompt example

Thought Generation

Prompts a language model to communicate its reasoning process for solving a problem step by step before reaching the final conclusion. There are several techniques within thought generation, among them is Chain-of-Thought (CoT) prompting which leverages few-shot and has proven to increase performance in reasoning tasks (Wei et al., 2022). A example of this is illustrated in Figure 2.16.

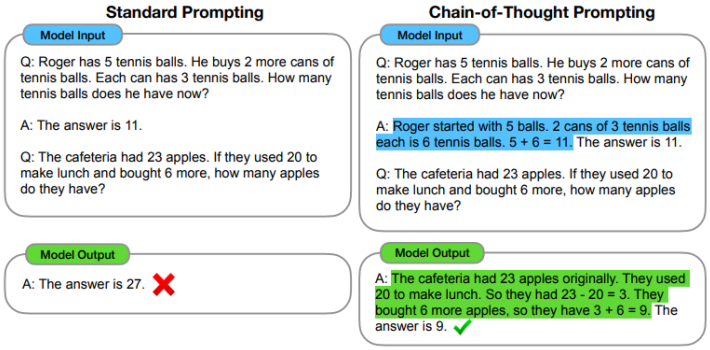


Figure 2.16: CoT example prompt (Wei et al., 2022)

Thought generation is a powerful tool for preventing hallucinations, optimizing prompts or when working with structured outputs like test specifications.

Self-Criticism

A technique where the model is prompted to evaluate its own outputs. This can improve reliability by encouraging internal consistency and error checking. In practice, this can be implemented using self-verification, where multiple candidate solutions are created using CoT reasoning and the model is then prompted to rank the best candidate. Based on this evaluation, the model can revise its answer by avoiding the flaws observed in lower ranked responses, leading to iterative refinement (Weng et al., 2023).

Chat Completion

Chat completion is a prompt strategy that uses a role-based message structure to interact with the language model (Llama 2024). Prompts are formatted as a list of messages, each labeled with a role such as `system`, `user`, or `assistant`. This structure enables the model to keep conversational context and role-specific behavior.

There are various parameters to set when loading a model used for chat completion, among them is the chat format, see figure 2.17:

```
from llama_cpp import Llama
llm = Llama(
  model_path="path/to/llama-model.gguf",
  chat_format="llama-2"
)
```

Figure 2.17: Loading a Llama model with the `chat_format="llama-2"` parameter.

The `chat_format` parameter controls how a list of role-based messages (`system`, `assistant`, `user`) is serialized into the raw text prompt, which special tokens and separators are inserted so that the underlying model correctly interprets the boundaries and order of system instructions, example context, and user queries. By specifying `chat_format="llama-2"`, `llama-cpp-python` uses Meta's built-in Llama 2 chat template to wrap messages with the exact tokens and separators that the Llama 2 models expect, enforcing conversational behavior(see Figure 2.18).

```
from llama_cpp import Llama
llm = Llama(model_path="path/to/llama-model.gguf",
chat_format="llama")

output = llm.create_chat_completion(
  messages=[
    {"role": "system", "content": "You are a knowledgeable
assistant."},
    {"role": "user", "content": "What's the distance to
Mars?"},
    {"role": "assistant", "content": "At closest approach,
about 54.6 million kilometers."}
  ]
)
```

Figure 2.18: Example of chat completion showing all three roles.

Role descriptions:

- system: Provides instructions or configuration for the model's behavior.
- assistant: Supplies example context or previous assistant outputs to guide the model.
- user: Conveys the user's actual query or task prompt.

Llama-cpp also supports response formatting in JSON, token-level generation control using temperature, repetition penalty, and penalty-based enforcement of structure. If formatting constraints are violated, the model can be penalized during inference to reinforce compliance with the prompt structure.

Directional Stimulus Prompting

Directional Stimulus Prompting (DSP), introduced by Li et al. (2023), is a technique in which a small, domain-tuned policy model is used to guide the inference of a large language model (LLM). During generation, the policy model suggests token-level preferences based on its domain knowledge. These suggestions act as directional stimuli, biasing the LLM's sampling process toward more contextually appropriate outputs.

Because the policy model is significantly smaller than the LLM, it can be trained efficiently on in-domain data with lower computational cost. This makes it practical to adapt the system to new domains or evolving requirements without retraining the full LLM. The combination allows the larger model to benefit from domain-specific knowledge while maintaining general-purpose capabilities. Figure 2.19 illustrates this mechanism.

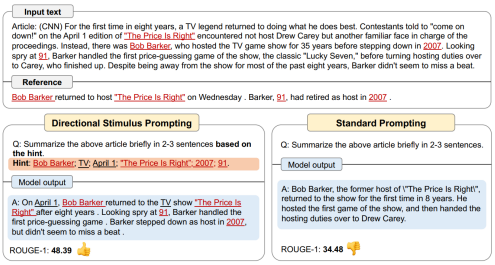


Figure 2.19: Directional Stimulus Prompting mechanism (Li et al., 2023).

2.5.2 Hyperparameters

When prompting a language model, two of the most important parameters to choose are the maximum token length (context window) and the sampling temperature.

When prompting the model the input is broken down into tokens the same goes for the response. Each model has a fixed context window. For example, Mistral-7B supports up to 4096 tokens. The prompt and the model's response must both fit within this window. If exceeded, generation may be truncated or fail.

The temperature $T \in [0, 2]$ controls the randomness of token sampling, this determines how much freedom the model has in it's response.

- $T = 0$: deterministic (always picks the highest-probability token).
- $0 < T < 0.3$: very low randomness for highly consistent, formulaic text
- $T \approx 0.7$: balanced creativity and coherence
- $T > 1.0$: high randomness for diverse or novel outputs.

2.5.3 Prompt Optimization

Speculative Decoding, introduced by Leviathan et al. (2023), is a method for accelerating text generation in large language models (LLMs) by using a smaller, faster draft model to propose a sequence of tokens in advance. The LLM then verifies the proposed tokens. If the tokens are accepted, meaning they match what the LLM would have generated, the decoding process skips ahead, saving time. If the tokens are rejected, the LLM reverts

to generating from that point onward using its standard decoding process.

This approach allows for speed-ups during inference without compromising output quality. Speculative decoding operates without needing to modify the LLM itself or retrain it, making it a lightweight and efficient optimization strategy.

2.5.4 Control Tokens and Formatting

When using instruction-tuned models like Mistral 7B, control tokens help define the structure of prompts. Table 2.2 outlines the control tokens supported by Mistral.

Using these tokens correctly helps the model interpret instruction boundaries and response expectations(Mistral AI, 2025). Figure 2.20 shows an example prompt constructed using tokens from Table 2.2. Here, `<s>` denotes the start of the sequence, and the `[INST]...[/INST]` block encapsulates the user instruction. This formatting ensures the model recognizes the task as an instruction and separates it from the generated output.

```
<s> [INST] Generate a test case for the following requirement
using the template below as guidance:

Requirement: The system shuts down when exceeding voltage usage
of x.

Template:
Test ID: T001
Title: <Short description>
Steps: <Step-by-step actions>
Expected Result: <What should happen>
[/INST]
```

Figure 2.20: Structured prompt example using control tokens and a formatting template.

2.6 Evaluation

Evaluating natural language generation (NLG) is crucial to ensure that the model's outputs are both syntactically and semantically aligned with expected results. Since multiple correct answers may exist for a given prompt, automatic metrics help assess how close a generated text is to a reference text. There are different metrics to use depending on which similarity to look for, all the metrics in this chapter ranges from a value of 0 to 1, where 1 indicates an identical match in all cases.

2.6.1 BLEU (Bilingual Evaluation Understudy)

Introduced by Papineni et al.(ACL 2002) is one of the earliest and most widely used metrics in machine translation and text generation evaluation. It measures how many

n-grams (consecutive sequences of n words) in the generated output overlap with the reference. The higher the n-gram overlap, the better the BLEU score.

An example:

- Reference: *The system shall have two safety measures for handling low voltage.*
- Generated output: *The system must include two safety measures to handle low voltage.*

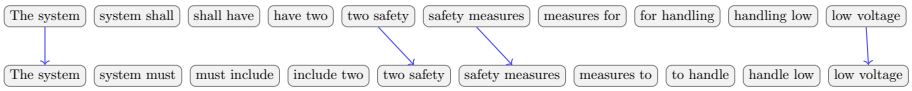


Figure 2.21: 2-gram overlaps between reference and candidate sentences in BLEU scoring.

As shown in Figure 2.21, the BLEU score would reward the candidate for lexical overlaps but includes a brevity penalty to discourage overly short outputs that match only fragments. The brevity penalty is defined as:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{if } c \leq r \end{cases}$$

where:

- c is the length of the candidate sentence
- r is the length of the reference sentence

2.6.2 METEOR (Metric for Evaluation of Translation with Explicit ORdering)

Introduced by Banerjee and Lavie (2005) METEOR was designed to overcome some of the limitations of BLEU by offering a more semantically aware evaluation. While BLEU relies on n-gram precision it lacks flexibility for word variation and order. METEOR aligns words between the candidate and reference sentences using:

- Stemming: Allows matching words that share a common root e.g. run and running.
- Synonym matching: Accounts for semantically similar words using lexical resources.
- Word order penalties: Penalizes significant deviations in the order of matching segments.

METEOR is particularly useful when evaluating generation tasks like summarization or paraphrasing, where exact lexical overlap may be low but semantic similarity remains high.

2.6.3 ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

Introduced by Lin(2004) ROUGE is commonly used in summarization tasks. It focuses on using recall to evaluate how much of the reference text appears in the generated text by using n-grams, word sequences and word pairs. ROUGE can be calculated using different variants:

- ROUGE-N: Measures n-gram recall by counting overlapping n-grams between the candidate and reference summaries.

- ROUGE-L: Computes the longest common subsequence (LCS) between the candidate and reference texts. It counts similarity as long as word order is preserved, even if the matched words are not consecutive. In Figure 2.22, X is the reference, and Y_1 and Y_2 be candidate summaries. Both candidates achieve the same ROUGE-L score since they contain the same subsequences as X, despite Y_2 being less orderly.
- ROUGE-W: A weighted variant of ROUGE-L that gives higher scores to longer consecutive matches. In the same example, Y_1 would receive a higher ROUGE-W score than Y_2 due to its consecutive matching span.

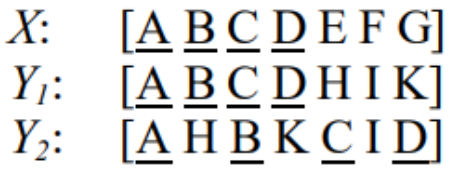


Figure 2.22: Longest subsequence comparison between candidates Y_1 and Y_2 with reference X (adapted from Lin, 2004).

- ROUGE-S: Also called ROUGE-Skip-Bigram, this metric measures the overlap of bigrams allowing for inconsistent gaps between them, making it more flexible in assessing partially reordered content.

2.6.4 Cosine Similarity

Cosine similarity compares two text embeddings [18]. It measures the angle between two high-dimensional vectors using the following formula:

$$\text{cosine similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

where:

- A and B are the vector representations of the two texts,
- $A \cdot B$ is the dot product of the vectors,
- $\|A\|$ and $\|B\|$ are the Euclidean norms of A and B.

The score indicates semantic relation between the vectors A and B depending on the width of the angle. Wider angle implies less semantic relation.

2.6.5 Jaccard Similarity and Distance

The Jaccard similarity first introduced by Jaccard [19] measures how alike two samples are by comparing the size of their intersection to the size of their union. Given two sets A and B,

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

The corresponding Jaccard distance, which is a proper metric, is simply

$$d_J(A, B) \;=\; 1 - J(A, B).$$

A common application is to convert each sample into text embeddings and comparing each text document as a set of terms and compute J to gauge document similarity for tasks such as clustering or near-duplicate detection.

2.6.6 Natural Language Inference

Natural Language Inference (NLI), also known as textual entailment [20], is the task of determining whether a premise X entails a hypothesis Y. Because entailment is directional, evaluation is done by

$$X \to Y \;=\; \Pr\big(\text{Entailment} \mid \text{Premise} = X,\ \text{Hypothesis} = Y\big).$$

then compare $X \to Y$ against a threshold τ to decide if X entails Y.
Implementation strategies include:

- Embedding-based: Compute cosine similarity

$$\text{sim}(X, Y) \;=\; \frac{\mathbf{v}_X \cdot \mathbf{v}_Y}{\|\mathbf{v}_X\|\,\|\mathbf{v}_Y\|},$$

  and interpret $\text{sim}(X, Y) \geq \tau$ as entailment.

- Prompt-based with pretrained LLMs: Send X and Y in a structured prompt, for example:

```
Premise:  Requirement(X)
Hypothesis:  Test specification(Y)
Question:  Does the premise entail the hypothesis?
```

2.6.7 Human evaluation

Human evaluation can mean many things, in this thesis it means simply asking humans to evaluate models [21]. The humans the thesis refers to are the engineers that work at the automotive company. Even though different metrics might say that the output of the model is good, it is only if it passes the human evaluation when it matters.

2.7 Data Engineering

Data engineering is the process of creating systems that collect, store and prepare data for use [22]. Companies gather data from many different sources and this data often come in different formats and quality levels. Data engineers build tools and processes called data pipelines to bring all this data together then clean it up and organize it in a way that makes it easier to understand. This data can then be used by people in the company like analysts and data scientists to find useful information and make decision gathered from the data.

2.7.1 Pandas

Pandas is an open source Python library commonly used for data analysis and data manipulation. Pandas is fast, powerful and easy to use [23]. It provides data structures that are designed to specifically handle datasets withing the Python API. These data

structures are called *Dataframe* and *Series* that can handle data efficiently. Pandas implements operations such as filtering, merging, handling missing values and much more. It has support for structured data like tables, matrices and time series and supports importing and exporting datasets from various file formats like CSV, SQL and spreadsheets. Pandas achieves high performance by implementing these using C or Cython. These capabilities make it well suited for preparing and analyzing datasets in data driven projects like AI.

2.7.2 BeautifulSoup

Beautifulsoup (bs4) [24] is one of many commonly used libraries in python for extracting data from web pages. Beautifulsoup functions as a parser, converting HTML or XML documents into a structured parse tree. This allows for easy navigation, searching and modification of the document's content using standard Python syntax.

2.8 Data Augmentation

Data augmentation (DA) is a technique used to expand a dataset by generating new samples based on existing ones. In natural language processing(NLP), this is beneficial when working with small or similar data samples. The goal is to create diverse variants of original data samples. Data augmentation helps improve model generalization while reducing overfitting and overall accuracy[25].

Unlike in vision tasks, where simple transformations e.g. rotating or flipping the images may suffice, the challenge in NLP data augmentation is preserving meaning while modifying the lexical or structural diversity. The goal is generating new samples that are neither too similar nor diverse. Similar new samples gained through DA will contribute little to a LLM when learning and instead introduce bias in the dataset, risking overfitting[25]. Different data samples introduce noise and label inconsistencies and can degrade the performance of a model.

DA can be categorized into different levels: word, sentence, and document. Each offers unique trade-offs between diversity and semantic preservation. Below are some widely used methods suitable for NLP datasets.

2.8.1 Back-Translation

Back-translation involves translating a sentence into another language e.g. English to French and then translating it back to the source language. This approach is an effective method for increasing syntactic and lexical diversity using monolingual data[26]. However, it's acknowledged that semantic meaning is not always preserved during round-trip translation, and that the effectiveness of the technique can vary depending on the quality of the translation models and the quality of the data[26]. Despite this limitation, back-translation has become a widely adopted data augmentation strategy for expanding data samples.

Figure 2.23 shows an example of how a sentence can be translated back and forth to create new data samples which express the same context in a different way. To perform back-translation, the model can be prompted using the following example prompts:

1. **Prompt 1:**
   `Translate the following English sentence into French. Respond with French only|no commentary or quotes.`
2. **Prompt 2:**
   `Translate the following French sentence back into English. Respond with English only|no commentary or quotes.`

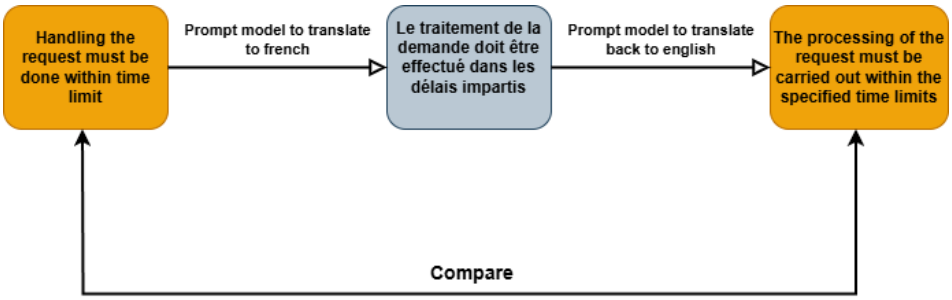


Figure 2.23: Example back-translation

The resulting back-translated English text is then compared both to the original sentence, making sure context and semantics are intact, and against the full corpus to detect and eliminate duplicates.

2.8.2 Contextual Augmentation

Contextual augmentation is a data augmentation strategy that replaces words in a sentence with alternatives predicted by a LM using the words surrounding to provide context in it's calculations. The original contextual augmentation[27] used a bidirectional masked language model to compute, for each target word w_i in a sentence $S = (w_1, \ldots, w_n)$, the probability distribution:

$$P(w \mid \text{context}_i) \quad \text{where} \quad \text{context}_i = (w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_n).$$

Building on this idea this thesis will use an LLM to perform contextual data augmentation using the LLM's learned embeddings and next-token predictions. Given a sentence S and a selected index i, we mask w_i, feed the masked text into the LLM, and obtain a list of candidate tokens from the model's output distribution at that position. By sampling the top-k predictions, new sentences can be generated that maintain context and semantic similar to the source sentence. Figure 2.24 demonstrates how an example sentence has been used to compute negative and positive contextual substitutes for a given word in a sentence. Models using this approach has a significant improvement in accuracy compared to using the old synonym-based approach[27].
There are several approaches when prompting the model to control the augmentation process:

- **Single-word replacement:** mask one token and sample a replacement.
- **Multi-word contextual swap:** mask up to k tokens per sentence.
- **Controlled paraphrasing:** prompt the model to rephrase the entire sentence.
- **Few-shot masked prompting:** provide exemplar masked sentences with outputs.

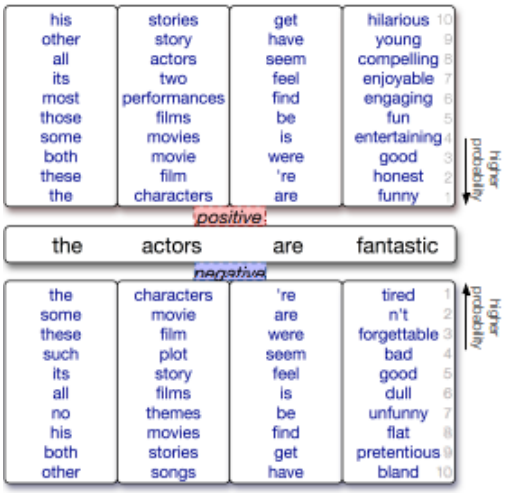


Figure 2.24: Contextual augmentation(Kobayashi 2018)

- **Batch generation of variants:** request N different augmentations in a single call.

2.8.3 Easy Data Augmentation (EDA)

EDA[28] is a set of a data augmentation NLP transformation strategies at the word level:

- **Synonym Replacement (SR)**: Randomly replaces words with their synonyms.
- **Random Insertion (RI)**: Inserts random synonyms into the sentence.
- **Random Swap (RS)**: Swaps the position of two words randomly.
- **Random Deletion (RD)**: Deletes words from the sentence with a given probability.

These operations introduce controlled noise that help model robustness on small datasets or short text samples. For bigger datasets the improvement plateaus[28].

2.9 Polarion

Polarion is an Application Lifecycle Management (ALM) tool developed by Siemens [29]. It's designed to support the end to end development cycle of software systems. It provides a browser based platform that enables organizations to define, build, test and manage their software. Polarion integrates different features of software developement into a single enviroment, including requirements management, coding, testing, and more. It ensures full traceability across all stages of the software lifecycle and enables collaboration trough workflows and access control. Polarion is widely used across industries such as automotive, aerospace, and healthcare.

2.10 Polarion library

To collect data from Polarion, a third-party Python library called `polarion` was used. This library provides a convenient interface for interacting with Polarion's web services [30]. It uses the SOAP network protocol, with the help of the Python libraries `requests` and `zeep`, to access various items on the Polarion server. It enables automated access to work items, documents, and metadata by connecting directly to the Polarion system.

2.11 SQLite

SQLite is a lightweight, free and serverless database engine that is widely used for embedded database applications [31]. SQLite does not require a separate server making it ideal for browsers, operating systems, mobile phones and other embedded systems. It stores the entire database in a single file locally. SQLite generally follows PostgreSQL syntax but does not follow the same set of rules by default.

Chapter 3
Methodology

This chapter outlines the methods used to explore AI driven automation of test specification generation. It begins with interviews conducted at the automotive company to understand current validation workflows and identify where an AI model can help with automation. Next the data engineering process is described, covering how data was extracted from Polarion, cleaned, structured, mapped and prepared for model training. The chapter also explains the selection of the AI model focusing on performance and resource efficiency. Then the data gathered went trough a exploratory data analysis to assess data quality. To improve model performance, some data augmentation techniques such as back-translation and contextual augmentation were applied. Lastly the evaluation process is detailed, including both automated metrics such as BLEU and ROGUE and then human evaluation to validate the quality of the generated test specifications.

3.1 Use of AI in thesis

AI tools have been used throughout the thesis. ChatGPT was used to ensure that sentences are clear and concise, to bounce ideas for clarity and to maintain an easy to follow structure through the whole thesis. Github Copilot was used to assist with some parts of coding such as writing documentation for functions. The authors of the thesis assert that all code and text were written by themselves and AI was only used as a support tool.

3.2 Interview with the workers

Before any work begins, it's good to get a clear picture of the data and the current validation workflow. The first step to this process is to conduct focused interviews with the engineers who draft and maintain requirements and test specifications on a daily basis. The interview guide is built around four core prompts-listed in Figure 3.1 this will answer two of the thesis problem formulation questions: "How are software requirements currently validated?" and "Which parts of the current test process are repetitive and resource intensive?"

Table 3.1: Main questions asked in interviews

Question	Purpose
How are the requirements and test specifications written?	Give the authors a deeper insight into the problem to be solved
Is there a standard template or format that must be followed?	Give the authors knowledge that the AI needs to follow
How is the link between requirements and test specifications established	Give the authors more insight in the process
How many requirements and test specifications are there	Give the authors a scope of the available data

3.3 Data Engineering

One of the key pillars of any AI project is the quality and quantity of the data available. Both factors play a crucial role in shaping the outcome of the final model. The data engineering phase usually consists of more than 50% [32] of the total work done in making the AI model, so it's crucial to get the stage done right. Since the automotive company used Polarion to document all their requirements and test specifications, the first step in the process is to become familiar with this system to be able to automate the collection of data. The data engineering is done in steps. These steps can be found in figure 3.1.

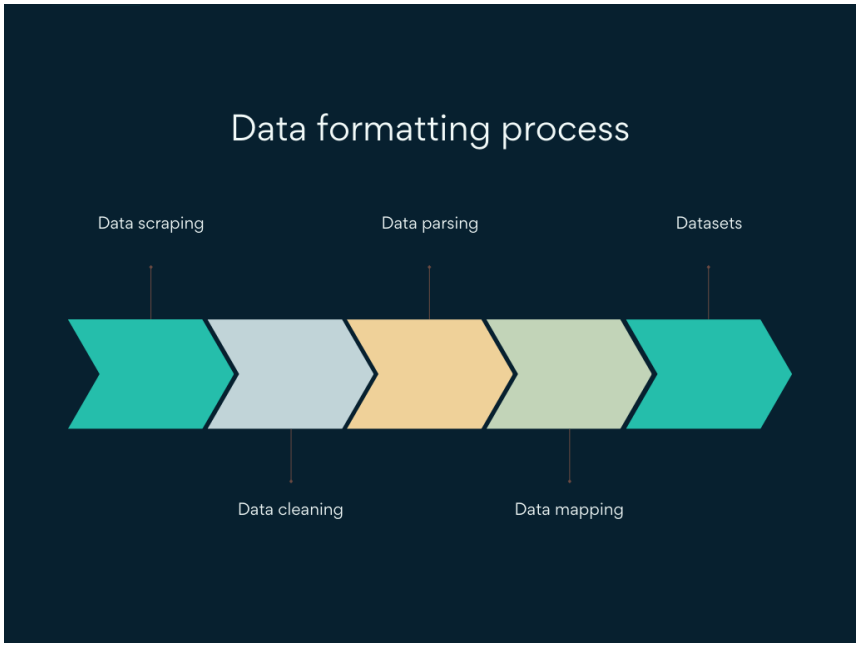


Figure 3.1: Data Engineering stages

3.3.1 The data engineering stages

Following figure 3.1 data was collected through the Polarion SOAP interface: all requirement work items and their associated test-specification documents were exported, converted from XML to JSON, and stored in a SQLite database.

A summary of the data engineering pipeline consist of:

1. **Scraping** - Remove some HTML text with `BeautifulSoup`.
2. **Cleaning** - Cleaning of the text, special characters, contractions simplified, remove template words and attribute reduction.
3. **Parsing** - Literal evaluation, HTML tag stripping and filter attributes.
4. **Mapping** - Each requirement is paired with its linked test specifications.
5. **Datasets** - The processed data is organized into JSON files and split into training, validation and test sets for model development and evaluation.

3.4 Model selection

A suitable Large Language Model (LLM) will be selected based on its ability to handle text generation task while meeting constraints related to hardware resources and deployment requirements. The chosen model should offer a balance between performance and efficiency making it work within the company's existing infrastructure for the proof of concept.

3.5 Exploratory data analysis

Before model training, an Exploratory Data Analysis (EDA) will be carried out to verify data integrity and guide the choice of features.

1. **Structure and completeness**. Load the requirement–specification pairs into a `pandas DataFrame`, inspect `head()`, `info()` and `describe()` to identify missing values, duplicate rows and non-unique attribute columns.
2. **Low-variance attributes**. Compute the number of unique values in every textual attribute.
3. **Null handling**. Count `NaN`/empty tokens per attribute.
4. **Length statistics**. Record token counts for each requirement and its linked specification. A scatter plot of the two counts will later be generated.
5. **Lexical profiling**. Produce unigram–to–4-gram frequency tables and TF–IDF scores to distinguish boiler-plate wording from unique text.
6. **Visual diagnostics**. Generate histograms of word counts, heat-maps of attribute sparsity and frequency plots of the top n n-grams; final figures are reported in the Results chapter.

3.6 Data augmentation

Data augmentation will be applied to increase dataset diversity and improve model generalization. Techniques such as back translation and contextual augmentation will be used to generate variations of the original requirements and test specifications. This

helps the model learn to handle different phrasings of the same sentences and reduces overfitting on the limited available data. All augmented data will be reviewed to ensure they maintain semantic consistency with the originals.

3.6.1 Data Augmentation via Prompt-Based Generation

When facing input-only scenarios—such as requirements that lack corresponding test specifications, the task of generating labeled data becomes a weakly supervised learning problem. This thesis will address such a scenario by generating synthetic test specifications from unlabeled requirements, applying recent advances in prompt-based data augmentation (Feng et al., 2021; Schulhoff et al., 2025).

To achieve this, a prompt-based generation pipeline will be implemented, combining instructional, role-based, and few-shot prompting strategies. Specifically, a one-shot prompting approach will be adopted, where a single example of an input–output pair is presented alongside a structured test specification template. This approach is supported by prior research demonstrating that few-shot prompting can guide large language models to generate task-consistent outputs without fine-tuning (Brown et al., 2020; Wei et al., 2022).

Multiple prompt formulations will be explored and compared. The most effective ones will be identified using automatic evaluation metrics such as Cosine Similarity, Jaccard, BLEU, and Natural Language Inference (NLI). NLI-based evaluation will be performed by prompting the Mistral-7B to assess whether the generated test specification logically follows from the input requirement. The different prompt formulations will also be tested using different prompt settings such as temperature and max tokens. Since the input requirements and the generated test specifications follow different structural templates, a preprocessing step will be included to strip away template-specific tokens from both before applying evaluation metrics, this ensures the results aren't skewed by the template. As a reference point a sample from the training set will be picked to undergo the same evaluation.

The generation pipeline will be executed using the Mistral-7B language model (Jiang et al., 2023). To improve inference throughput, prompt batching will be employed. This batching will allow multiple prompt instances to be processed simultaneously, utilizing available GPU memory more efficiently and preventing memory overload.

Two primary prompt structures will be tested, as shown in Figure 3.3 and Figure 3.4. The first is a one-shot instructional prompt with a provided sample and template. The second enforces stricter output formatting, supporting generation of multiple specifications per prompt and requiring conformity to a structured format. Both prompts will share a role-based instruction component shown in Figure 3.2, which sets the context for the model's behavior. The template input will remain the same since it's company standard but the input example will be shifting between different trials.

```
You are an expert software tester specializing in automotive
software testing.
Your task is to generate a test case strictly based on the
provided example and test specification template.
Do NOT add or assume any new requirements.
Only use what is given.
```

Figure 3.2: Role-based instruction used for context. Used in both Figure 3.3 and Figure 3.4.

```
{Role-based instruction}
Below is the example requirement and test specification,
followed by the test specification template.
Using the following example:
{requirement-test example}
And the following template:
{template_example}
Generate a detailed test specification for the requirement
below.
{requirement}
```

Figure 3.3: Prompt 1 example: Instructional and one-shot prompting with template guidance.

3.6.2 Back-Translation for Data Augmentation

Back-translation will be employed to augment both the original labeled dataset and the synthetic samples generated via prompt-based methods (see Section 3.6.1). The objective is to introduce diversity while preserving the semantic intent of requirement–test specification pairs. By translating samples through chains of multiple languages and then back to English.

To implement this process, the Mistral-7B language model will be used as a translator. While Mistral-7B is not a dedicated translation model, it demonstrates proficiency in several languages including English, French, German, Spanish, and Chinese (Jiang et al., 2023). These languages will be used in multi-step translation pipelines e.g. English → Chinese → French → English.

A primary challenge in back-translation is maintaining the integrity of technical and structural elements, particularly given the general-purpose nature of Mistral-7B. To reduce this, masking will be applied to both the requirement and test specification before translation. Template field names, technical terms, and named entities will be wrapped in `[MASK_]` tags to prevent alteration. For example, the sentence:

```
description:  ECU triggers wakeup timer every 1 seconds when in ECUawake
mode.
```

will be masked as:

```
hh{Role-based instruction}
### Task
For the requirement below, produce one or more complete test
specifications.
• Each spec must reproduce every label and colon from the
template exactly.
• Replace only the bracketed text [ ...] with concrete values.
• No commentary or extra fields.
### Example (fully-filled)
{example}
### Template to fill
BEGIN_TEMPLATE
{template}
END_TEMPLATE
### Requirement
{req_text}
### Output format
Return N  1 specs, each:
1. Starts with title: on its first line.
2. Ends immediately before the delimiter line ---.
3. Matches every label in the template identically.
Separate consecutive specs with a single line containing only
three dashes:
---
(title of next spec continues here)
Generate the filled template(s) now.
```

Figure 3.4: Prompt 2 example: Structured multi-spec generation using formatting constraints.

```
[MASK_description]:[MASK_ECU] triggers wakeup timer every 1 seconds
when in [MASK_ECUawake] mode.
```

These tags are intended as soft constraints; models typically avoid modifying bracketed content when instructed to do so (Schulhoff et al., 2025). A validation mechanism will be applied post-translation. Both the requirement and test specification must meet the following conditions to be accepted:

- All masked tokens are preserved.
- BLEU score exceeds 0.4 when compared to the original input.

Quality of generated samples will also be evaluated using automatic metrics such as cosine similarity, Jaccard, BLEU and ROGUE. If a sample fails validation, it will be retranslated using a different modified prompt instruction. The prompt used for literal translation behavior will follow a constrained structure, as illustrated in Figure 3.5. It directs the model to avoid structural changes and preserve exact formatting during translation.

```
[INST] You are a literal translator. Follow these rules
exactly:
- Do NOT add, remove, repeat, merge, or split any words, lines,
or sections.
- Do NOT reorder content or add commentary.
- Preserve punctuation, whitespace, and line breaks exactly as
in the input.
Translate {src_lang}→{tgt_lang}: {masked_text} [/INST]
```

Figure 3.5: Strict translation prompt used in the back-translation pipeline.

3.6.3 Contextual Augmentation

Contextual data augmentation will be used to increase dataset diversity while preserving the accuracy of the content. A large language model (LLM) will predict contextually appropriate replacements for selected words or phrases by masking tokens and sampling from the model's output distribution. This process will generate new sentence variations that have the same meaning as the original but introduce variance. The augmentation will primarily focus on single word replacements and controlled paraphrasing to avoid introducing semantic errors in critical technical content. This approach aims to improve the models robustness to different phrasings and reduce overfitting to specific word choices present in the original dataset.

3.6.4 Evaluation of Data Augmentation

Every generated sample should be validated through the following steps:

1. **Duplicate removal:** Compare each new sample to the entire corpus using embedding based similarity, discard identical or near duplicate examples.
2. **Intrinsic evaluation:** Compute perplexity using the LLM's token probabilities and cosine similarity of hidden states between the generated and source samples.
3. **Extrinsic evaluation:** Prompt the LLM using the augmented data and benchmark the output's accuracy.
4. **Human evaluation:** Human evaluations are important for the samples that are on the borderline from the intrinsic and extrinsic evaluation. Here experienced users can weigh in by reading the augmented sample and comparing to the source sample.

3.7 Datasplit

The dataset will be partitioned using a 70/20/10 split for training, validation, and test sets. After the initial data engineering phase, all valid samples will be converted into a one-to-one format, if a single requirement is associated with multiple test specifications, it will be duplicated so that each instance forms a unique requirement–test one-to-one pair.

For each phase of data augmentation: prompt-based generation, back-translation, contextual augmentation, and paraphrasing, newly generated samples will be accumulated,

adding to the existing training, validation, and test sets. Original distribution will remain from phase to phase. This ensures the integrity of the evaluation by preventing data leakage between sets.

3.8 Prompt Engineering

Three prompt approaches are tested: control-token instruction, chat-completion, and schema-enforced chat. Each is evaluated in zero-, half-, and few-shot variants (Figures 3.8–3.17). Five high-quality training examples serve as references in the half- and few-shot settings. All methods generate outputs for the full test set, which are scored by BLEU, METEOR, ROUGE, F1-score, and Cosine Similarity to identify the best prompt configuration. For each different approach implementing the type-shots, different context is provided:

- Zero-shot prompts: Includes only the requirement title and the instruction.
- Half-shot prompts: Includes instruction with provided test specification template and an entire requirement input.
- Few-shot prompts: Includes instruction with provided test specification template along with 1-5 requirement-test specification pair examples and an requirement input to generate from.

3.8.1 Control-token Instruction Prompts

Each prompt is wrapped in the tokens `<s>[INST]...[/INST]` so that the model treats everything between as a self-contained instruction. Implementation uses three Python functions described in figure 3.8, 3.9, and 3.10.

```
You are an expert software tester specializing in automotive
software testing. Your task is to generate a test case strictly
based on the provided example and
test specification template.
Do NOT add or assume any new requirements. Only use what is
given.
```

Figure 3.6: Instruction header.

```
test_spec_template = """
title: [Test Case Title]
description:
  Test purpose: [Brief description]
  Expected result: [Expected outcome]
  ...
  Input parameters: [Parameters]
"""
```

Figure 3.7: Test-specification template.

```
def zero_shot_inst(req_text):
  title = req_text.split('\n',1)[0]
  prompt = f"""<s>[INST]
    Generate a test specification for:
    {title}
  [/INST]""".strip()
  return model(prompt, max_tokens=2048)["choices"][0]["text"]
```

Figure 3.8: Zero-shot instruction prompt.

```
def half_shot_inst(req_text, template):
  prompt = f"""<s>[INST]
    {instruction_header}
    Here is the test specification template:
    {template}
    Requirement to generate from:
    {req_text}
  [/INST]""".strip()
  return model(prompt, max_tokens=2048)["choices"][0]["text"]
```

Figure 3.9: Half-shot instruction prompt.

```
def n_shot_inst(req_text, examples, template, n=0):
  examples_block = "".join(...)
  prompt = f"""<s>[INST]
    {instruction_header}
    Here is the test specification template:
    {template}
    Here are {n} examples:
    {examples_block}
    Requirement:
    {req_text}
  [/INST]""".strip()
  return model(prompt, max_tokens=2048)["choices"][0]["text"]
```

Figure 3.10: Few-shot instruction prompt.

3.8.2 Chat-Completion Prompts

Prompts are represented as sequences of role-tagged messages and sent to Llama's `create_chat_completion()` function. The model is initialized with `chat_format="llama-2"` (Section 2.5). The Python implementations are shown in Figures 3.11, 3.12, and 3.13. The context from

figure 3.6 along with the template from 3.7 is used for the half and n-shot functions as input parameters.

```
def zero_shot_chat(req_text):
  messages = [
    {"role": "system", "content": f"""You are an expert
software tester specializing in automotive software
testing.\nYour task is to generate a test specification using
the following requirement title:\n""".strip()},
    {"role": "user", "content": f"req_text"},
  ]
  resp = model.create_chat_completion(messages=messages)
  return resp["choices"][0]["message"]["content"]
```

Figure 3.11: Python implementation of the `zero_shot_chat` function.

```
def half_shot_chat(req_text, examples, template, context):
  messages = [
    {"role": "system", "content": f"context\nHere is the test
specification template:\ntemplate"},
    {"role": "user", "content": f"Requirement to generate
from:\nexamples[0]['req_text']"},
    {"role": "assistant", "content":
examples[0]['req_text']},
    {"role": "user", "content": f"Requirement to generate
from:\nreq_text"}
  ]
  resp = model.create_chat_completion(messages=messages)
  return resp["choices"][0]["message"]["content"]
```

Figure 3.12: Python implementation of the `half_shot_chat` function.

```
def n_shot_chat(req_text, examples, template, context, n=0):
  messages = [
    {"role": "system", "content": f"context\nHere is the test
specification template:\ntemplate"}
  ]
  for i in range(min(n, len(examples))):
    messages.append({"role":"user", "content": f"Requirement
to generate from:\nexamples[i]['req_text']"})
    messages.append({"role":"assistant", "content":
examples[i]['test_text']})
  messages.append({"role":"user", "content": f"Requirement to
generate from:\nreq_text"})
  resp = model.create_chat_completion(messages=messages)
  return resp["choices"][0]["message"]["content"]
```

Figure 3.13: Python implementation of the `n_shot_chat` function.

3.8.3 Schema-Enforced Chat Prompts

This variant uses the same sequence of role-tagged messages as the standard chat prompts (Figs. 3.11–3.13), but includes the JSON schema from Figure 3.14 via the `response_format` parameter in each call to `create_chat_completion()`. By supplying the schema at generation time, Llama is instructed to generate a JSON object that exactly matches the required fields and types. The Python implementations, including the response formats, are illustrated in Figures 3.15, 3.16, and 3.17. This approach functions as a form of DSP, guiding the model toward fully structured outputs by hinting at the structure.

```
test_spec_schema = {
  "type": "object",
  "properties": {
    "[redacted]": { "type": "string" },
    "[redacted]": {
      "type": "object",
      "properties": {
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "string"
},
        "[redacted]": { "type": "string" },
        "[redacted]": { "type": "array", "items": { "type":
"string" } },
        "[redacted]": { "type": "array", "items": {
"type": "string" } },
        "[redacted]": { "type": "array", "items": { "type":
"string" } },
        "[redacted]": { "type": "array", "items": { "type":
"string" } },
        "[redacted]": { "type": "string" }
      },
      "required": [ [redacted], [redacted],
[redacted], [redacted],
            [redacted], [redacted],
[redacted], [redacted],
            [redacted],
[redacted], [redacted],
            [redacted], [redacted], [redacted],
[redacted] ]
    }
  "required": [ [redacted], [redacted] ]
};

response_format = {
  "type": "json_object",
  "schema": test_spec_schema
};
```

Figure 3.14: JSON schema and response format definitions with sensitive field names redacted.

```
def zero_shot_dsp(req_text):
  title = req_text.split('\n', 1)[0]
  resp = model.create_chat_completion(
    messages = [
      { "role": "system", "content": "Generate a test
specification for the following requirement title." },
      { "role": "user", "content": f"title" },
    ],
    response_format = response_format
  )
  return resp["choices"][0]["message"]["content"]
```

Figure 3.15: Python implementation of the zero_shot_dsp function.

```
def half_shot_dsp(req_text, template):
  resp = model.create_chat_completion(
    messages = [
      { "role": "system", "content": (
        "You are an expert software tester specializing in
automotive software testing.\n"
        "Your task is to generate a test specification strictly
based on the requirement\n"
        "and the test specification template.\n"
        f"
nHere is the test specification template:\ntemplate"
      ) },
      { "role": "user", "content": f"req_text" },
    ],
    response_format = response_format
  )
  return resp["choices"][0]["message"]["content"]
```

Figure 3.16: Python implementation of the half_shot_dsp function.

```
def n_shot_dsp(req_text, examples_df, template, n=0):
  messages = [
    { "role": "system", "content": (
      "You are an expert software tester specializing in
automotive software testing.\n"
      "Your task is to generate a test specification strictly
based on the requirement\n"
      "and the test specification template.\n"
      f"
nHere is the test specification template:\ntemplate"
    ) }
  ]
  for i in range(min(n, len(examples_df))):
    req_i = examples_df[i]['req_text']
    test_i = examples_df[i]['test_text']
    messages.append({ "role": "user", "content": req_i })
    messages.append({ "role": "assistant", "content": test_i
})
  messages.append({ "role": "user", "content": req_text })
  resp = model.create_chat_completion(
    messages = messages,
    response_format = response_format
  )
  return resp["choices"][0]["message"]["content"]
```

Figure 3.17: Python implementation of the `n_shot_dsp` function.

3.9 Human Evaluation of Model Output Quality

The final step in the evaluation process will involve a blind human assessment of the model outputs. A custom built tool will be used to display the original input alongside the generated outputs, without revealing which model produced each one, aiming to reduce bias in the judgments.

Engineers will be tasked with ranking the outputs based on how relevant, fluent, and appropriate they are to the prompt without influence from automatic metric scores or prior expectations. These human evaluations will provide an essential layer of assessment, capturing qualitative aspects that automatic metrics might overlook. The compiled rankings will then be used to determine which type of model consistently produces the best outputs from a human perspective.

3.10 Source criticism

The references used in this thesis can be found in Chapter 6. The references come from a variety of sources, including websites, articles, and books. It is important to highlight why the thesis has used these sources.

- **Academic books**
  - References [2, 33]
  - Reason: Standard textbooks by reliable authors in AI and NLP, used in academia. Used in university courses and provide theoretical foundations
- **IBM sources**
  - References [1, 4, 22]
  - Reason: IBM is a leader in AI and cloud computing. Their content is reliable, they are active in the research area and they have published a lot of papers. This makes them a reliable source, but they may be biased towards their own solutions
- **Peer reviewed conference and journal papers**
  - References [6, 3, 26, 25, 28, 27, 34, 35, 36, 16, 37]
  - Reason: Published through accepted conferences like ACL and EMNLP and also through journals like Nature which ensures high research standards
- **arXiv**
  - References [38, 7, 5, 9, 10, 8, 11, 13, 14, 12, 15, 17, 39, 40, 41, 42, 43]
  - arXiv is a free open access repository for research papers. These papers are usually preprints which are early versions of their papers before formal review, meaning the papers provided in arXiv don't have to be peer review to be submitted. These papers should be evaluated before used.
- **Official Documentation**
  - References [24, 29, 30, 31, 23, 44, 21]
  - Reason: Official sources for software documentation. These are reliable for understanding the tool.
- **Industry whitepapers**
  - References [32]
  - Reason: Provides structured methods from a commercial perspective but is probably biased toward promoting services.

Chapter 4
Results

This chapter presents the results from the data engineering pipeline and highlights the different transitions the data takes throughout the different stages. It then shows the results from the data augmentation and how many new requirements and test specifications was made. Lastly it shows the results from the different evaluation stages.

4.1 Interview Findings

Interviews were conducted with engineers responsible for writing and maintaining requirements and test specifications to understand the current validation process and determine the scope of the available data.
The following key takeaways were gathered based on the interview questions in Table 3.1:

- **Writing Process:** Requirements and test specifications are written manually by engineers using company internal guidelines. While certain teams follow informal templates, no strict formatting standard is enforced across all projects.
- **Templates and Formats:** Although no universal template exists, several projects use structured headings and predefined fields within Polarion to maintain consistency. This variation implies that the AI model must handle both well structured and free text inputs.
- **Linking between requirements and text specifications:** The connection between requirements and test specifications is established through Polarion's built in linking functionality, typically using parent child relationships. However, this practice is not always consistently applied, leading to some orphaned requirements.
- **Data Availability:** The project contains a total of **186 requirements** and **371 test specifications**. It was also noted that some requirements and test specifications exist in draft form or as incomplete outlines, which were excluded from the initial dataset but considered for potential future analysis.

A graph showing the amount of test cases that linked to each requirement can be found in figure 4.1

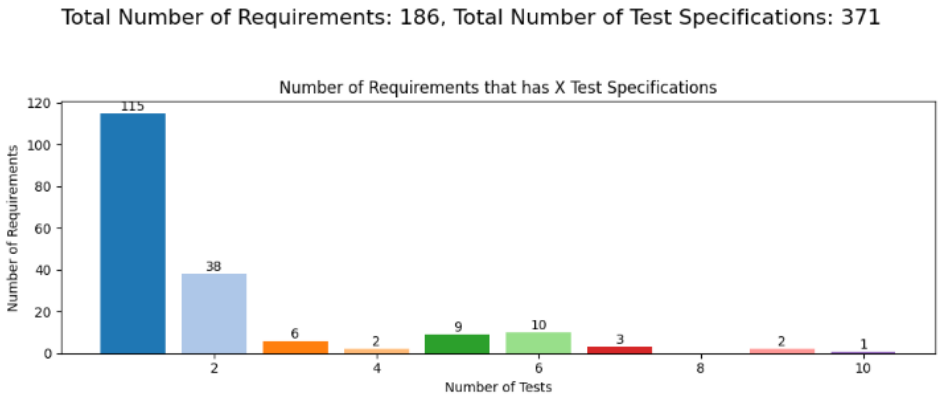


Figure 4.1: Number of requirements that has x test specifications linked to it.

4.2 Data Engineering

Data engineering was the most time consuming part out of the whole thesis. This stage took the longest as the data needed to be refined from its raw form. The data engineering phase successfully transformed the unstructured dataset from Polarion into a clean, structured format suitable for AI processing. The initial automated data collection retrieved 186 requirements and 371 test specifications meeting the target set during interviews. There were struggles with the Polarion API when collecting data so a switch to the SOAP protocol was made to extract the data instead. The extracted raw data initially consisted of 59 JSON attributes per item which was then reduced to 19 after cleaning, filtering, parsing and mapping. The attributes could include HTML tags from the Polarion data which was cleaned using BeautifulSoup, unnecessary special characters were removed. Lastly a lightweight SQLite database was implemented to store the processed data.

4.2.1 Automated data collection

Before developing scripts to automate the data collection process, it was necessary to estimate the volume of available requirements and test specifications. After the interview with the head of documentation employee he established target figures for the number of work items to retrieve. This provided a reference to verify the entirety of the data extraction. The target was 186 requirements and 371 test specifications mentioned in the interview process earlier in the thesis, it was also made known that there were more unfinished requirements and test specifications that had an outline and that some of them could have value to the AI modell. After discussion with the supervisor, it was decided to extract everything available but keep all the useful requirements and test specifications separate from the other that was unknown if they might be useful for testing the AI at a later stage.

Using the Polarion API proved to be complex due to the fact that a special setting was needed to be on the internal server to make the REST API work. It was decided to see if there are any workarounds to this problem to not bother the server admins. After digging deeper into Polarion it was made known that Polarion could also be used by using the

SOAP protocol to retrieve items from the server. Time was invested in understanding how to establish a basic connection, retrieve documents, and access individual work items. The Python library polarion, a third-party library with relatively clear documentation, was key in helping this process. By experimenting with smaller code snippets to retrieve individual items, gradually a complete automation script for data extraction was built.

To store the extracted data, the JSON format was used. This decision was based on JSON's readability and compatibility with Polarion's structure, where each work item contains clearly defined attributes. Each work item was converted into a JSON object, preserving its attributes for future processing. Everything gathered from the Polarion work item was saved, resulting in each JSON object having 56 attributes which was quite a lot. The layout snippet of the raw data from polarion is shown in figure 4.2.

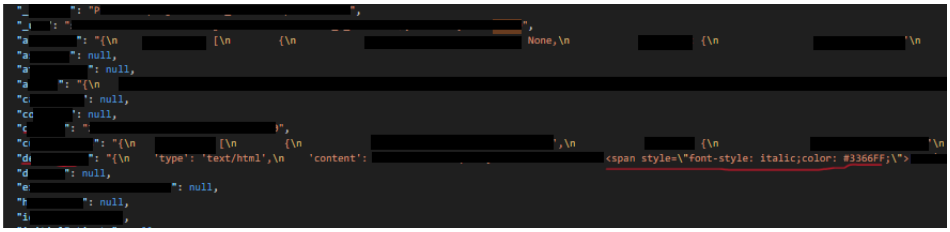


Figure 4.2: Redacted snippet of raw data from polarion

4.2.2 Scraping

As seen in figure 4.2 the raw data could contain strings that have the same structure as a Python dictionary, it could also contain HTML/CSS code snippets. This is something that would not only confuse the AI model but also the ones handling the data. Given the extensive HTML formatting in the raw data, different Python libraries capable of parsing and cleaning HTML content were evaluated. BeautifulSoup was selected, a widely used library for web scraping, which also supports parsing strings containing HTML. Each JSON file was processed using a custom script that iterated through its items, extracted the relevant attributes, and passed them through BeautifulSoup. The output was a more manageable looking text, some of the HTML tags were kept because they were necessary for keeping the same structure as was seen in Polarion with plans to remove them later in the pipeline.

4.2.3 Cleaning

Before parsing, cleaning on the text was applied to ensure that the AI model could correctly understand the information in each string. The cleaning was done in phases:

1. **Attribute Reduction:** We performed feature selection to reduce the number of attributes from 59 down to 13, removing those with low variance or high missing-value rates.
2. **Special-Character Removal:** All special characters were removed, except for those needed for sentence boundary detection. For example, semicolons `;` were uniformly replaced with colons `:`.
3. **Contraction Expansion:** Common English contractions were expanded (e.g. `They're` → `They are`) to improve token consistency.

4. **Template and Placeholder Cleanup:** Removed template markers such as `TBD` and bracketed placeholders (e.g. `[PLACEHOLDER]`), that did not contribute semantic content.

This cleaning phase ensures that the subsequent parsing and embedding steps operate on well-structured and uniform text. [33].

4.2.4 Parsing

After cleaning and getting the attributes into values that are more optimal for the model, the next step in the process is parsing. The main goal for the parsing is to fix three things:

1. **Literal Evaluation:** In each attribute that contains a string that has the format of a Python dictionary/list convert it back into it's original form. Use Python's ast library to safely parse and extract Python literals (eg dicts, lists) embedded in the strings.
2. **HTML Tag Stripping:** Using BeautifulSoup like previous steps in the data engineering pipeline to process each HTML element. In each attribute that has saved HTML code tags (eg, `<li>` `<table>`) convert each list hierarchy into a key-value pair dictionary. Convert each table into a key-value pair where each header is the key and the value is the value under the header in the original table. Do this for every saved HTML tag in every saved JSON attribute.
3. **Filter the less useful attributes:** Lastly some kind of filtering is needed to go over the 13 saved attributes from the cleaning phase but also some of the expanded attributes gained from step one in the parsing phase.

After the parsing process the total amount of attributes each JSON object had was 19. This means that the number of attributes from raw data went from 59 to 13 in the cleaning phase back to 19 in the parsing phase when expanding the strings containing Python literals. This doesn't mean that the model will be able to use all 19 attributes, but rather that 19 of these attributes might have value to the model and that more research into these attributes was needed.

4.2.5 Mapping the data

The mapping phase had two main functions. The first function is to fetch every requirement and map it to each corresponding test specification that validates it. It does this by checking one of the attributes that is contained in the requirement which has links to its "children" which are test specifications that validate the requirement. Each requirement/test specification pair gets put into a new JSON object that contains the pair. If the requirement has more test specification that validate it the requirement will be included on multiple rows but with each requirement/test specification pair being unique.

The second function is to put every requirement that has no test specification that validate it into its own JSON object.

4.2.6 Database

Given that this is a proof of concept, at the end of the pipeline is a database. The purpose of the database is to automatically update whenever a new work item is uploaded in Polarion where it has gone through the data engineering pipeline. Instead of making the time and effort to set up a real database with a server, it was opted to settle for a more

light version run on each computer. SQLite was chosen as it's quick to set up and no need for a server to host the database on meaning more time could be dedicated for the AI model. The schema for the database had three tables, one table for all the requirements available with every attribute, one table for all the test specifications available with every attribute, and one for each link in the mapping process containing only the id for the requirement and the test specification that validate it. The schema for the database can be found in figure 4.3. This meant that it was possible to query the database to get all the usable requirements and test specifications with a SQL join query. But also possible to query if there was a need to get the data that had no links. The database gets filled by checking the JSON from the mapping step making sure that every link actually has a link but also taking in every requirement that had no links.

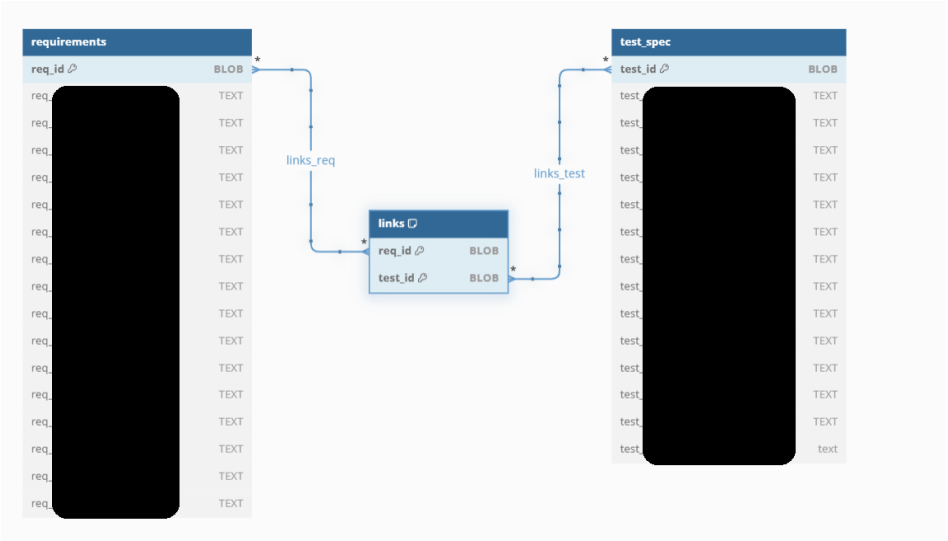


Figure 4.3: Redacted picture of the database schema

4.3 Choice of AI model

Originally the plan was to evaluate different base models on the initial data, and depending on which model produced the best result, that would be chosen. Sadly this turned out to not be the case, mainly from two issues. One was that it would have produced a time constraint which the thesis was already constrained on. Secondly it was an internal issue from the automotive company where each model would need to get approved and that would further complicate issue one. The model chosen was a hugging face model called Mistral-7B Instruct v0.3. The Mistral-7B is a good fit because it is already instruction tuned so prompts like "generate a structured test specification from these requirements" match its training objective and benchmarks (e.g., MT-Bench) show it rivals or exceeds larger models (Fig. 2.9). At 7 billion parameters the model runs entirely on local hardware, keeping company data protected.

4.4 Explorative Data Analysis

Exploratory Data Analysis (EDA) is the process where the data is examined before being passed to the AI model. The goal of the EDA is to understand the structure of the data, check for missing values, duplicates, inconsistencies, and get a general overview of the contents. EDA helps to identify what kind of patterns the model might learn and it's useful to adjust anything that could lead to misleading results. This process ensures that the data is in a reasonable state before moving on to the modeling. Instead of just passing the data to the model, then trying to understand its output, EDA provides an understanding of what is being used in the first place.

Doing EDA on the requirements and test specifications mentioned in figure 4.1 gave a lot of insights into the data. The first step was to check if there were any missing values or incomplete values like null. Using pandas built-in methods for dataframes like head(), info() and describe() to get a great overview of the data, it was discovered that many of the kept attributes weren't unique. One specific attribute had only two unique values: "N/A" and "Not applicable" which shortens to "N/A" meaning that there was in fact nothing unique in this specific attribute. This was one example of many that needed to be dealt with. You can find the redacted Figure 4.4 below which shows that some of the attributes didn't have much variation in them. A decision was made to remove these attributes before passing them to the model as they didn't contain anything useful and to not confuse the model later on.

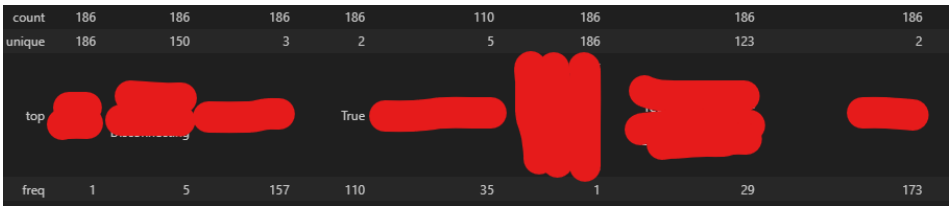


Figure 4.4: Redacted picture of the output from all the requirements with the attribute names hidden. There are more attributes than shown above.

The next step was to handle empty or null data. Using the method info() on the dataframe shows what type of data is in the attribute and counts the number of null values. This showed that a lot of the data that was missing was usually just one word instead of meaningful text, so it was decided to discard the whole attribute. The motivation for just discarding was that if an attribute in the requirement had a single word then that same word in the same attribute was present in the test specification, so it had no impact on the output.

After getting an overview of the data, it's important to go more into detail. As the thesis is about a NLP model, this means a deeper dive into the words and sentences in the data. A good question to ask is if there is any correlation between the word count of a requirement and the word count of a test specification. Figure 4.5 shows that there is a high correlation between the word counts. The analysis shows that certain clusters are more prone to specific outcomes than others. This means that it can be used later as quality assurance, for example the graph shows that for a requirement of word count 25-50 it should generate test specifications of about 60-120 words. This is of course not given as there are outliers as well, but if the model generates an outlier it should warrant

a closer look.

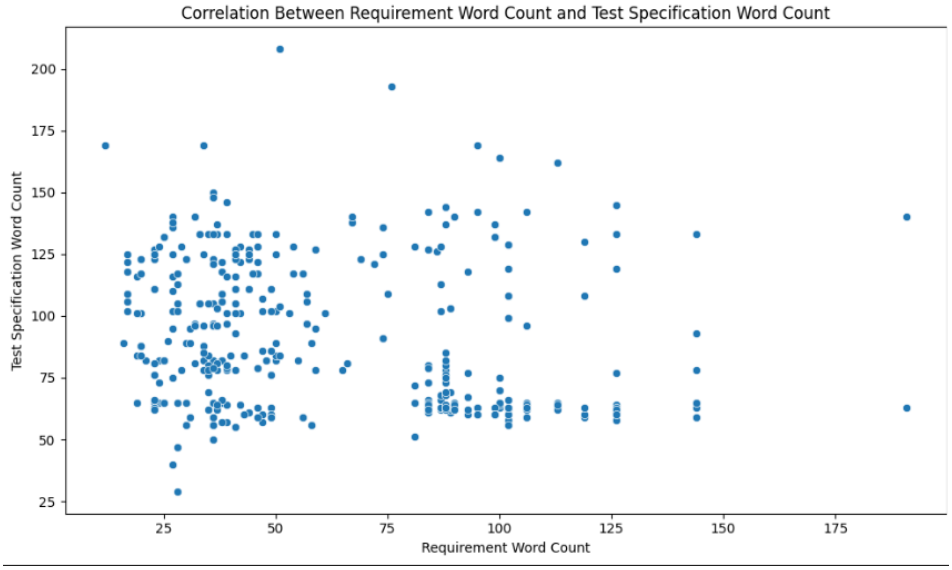


Figure 4.5: correlation between the word count of a requirement and the word count of a test specification.

The last step in the EDA process contains different kinds of graphs e.g histograms and heatmaps. These figures contains redacted information and without them it would be hard to convey what they are trying to show. The findings show what type of words are most common in the requirement and test specifications. It's done by counting the number of n-grams where n was one to four. A 1-gram is just counting the most frequent word like "test" while a 4-gram shows the 4 most common words one after another like "test the Bluetooth feature" in each work item combined. This gave information about the most common words in different n-grams, and what type of template words are the most common which will be needed later. It also gave information about if some specific words are found in the requirement then they have a high chance to also be in the test specification. One graph was also the Term Frequency-Inverse Document Frequency (TF-IDF) that turns the text into numerical vectors. TF-IDF is a method that figures out how important a word is by checking how often it appears in a single document compared to how common it is across all documents. Words that appear often are considered unimportant and assigned a low value. This is an important feature as a lot of the documents contain boiler plate text that are almost always present in the text. By ignoring the boiler plate template words it highlights the non boiler plate text.

4.5 Data Augmentation

To expand the training dataset beyond the original 371 requirement–test specification pairs, a total of 4761 synthetic samples were generated using various augmentation strategies, including prompt-based generation, back-translation, contextual augmentation, and paraphrasing. After applying validation filters, 1612 of these samples passed automatic checks. Out of the validated samples, 1203 were further accepted following manual human evaluation.

The automatic validation was based on the following criteria:

- All masked tokens must be preserved during generation.
- The BLEU score between the generated sample and its original input must exceed 0.4.

Human evaluation was conducted on a random subset of validated samples. Samples were assessed on the logical consistency between requirement and test specification, linguistic quality, and whether the test case appropriately validated the requirement.

4.5.1 Data Augmentation via Prompt-Based Generation

The objective was to generate test specifications for 205 requirements that lacked corresponding test cases. Same prompt-based generation approach as in section 3.6.1 was employed. Before generating the final version, experiments regarding the best hyperparameter tuning, input example selection, and prompt template were conducted.

A fixed prompt (Figure 3.3) served as the basis for hyperparameter tuning. Through trial and error, the following parameters were found to generate good results:

- max_tokens: 2048
- temperature: 0.3

Max token values below 2048 led to premature termination or repetitive output (cosine similarity$<$0.4). Temperature values above 0.4 produced valid content but deviated from the required format and lacked coherence.

Next, the impact of input example selection was evaluated by testing how different example inputs influence the quality of the generated output. The prompt takes an example input–output pair to serve as reference when generating a test specification from a new requirement, as depicted in figure 3.4. All requirements were used as input, while the example provided to the prompt was varied across three different samples from the training set. The examples used were:

- First: the first sample in the training set.
- Random: a randomly selected sample.
- Quality: a manually chosen, high-quality sample.

Using the optimized hyperparameters, one output was generated per reference example input for all 205 requirements and evaluated using automatic metrics and NLI-based assessment after preprocessing. In cases where multiple test specifications were generated for a single requirement, they were split into one-to-one pairs with the source requirement enable individual evaluation. The final score was computed as the average similarity between each generated test specification and its input requirement. Table 4.1 presents the average scores for each example input type. A reference value is also included, obtained by evaluating a randomly selected training sample against itself, serving as a baseline for comparison.
The results show minimal variation across outputs generated using different reference examples, suggesting that either the training data is of consistently high quality or that the model is capable of producing strong outputs regardless of the specific example provided. All variants achieve scores close to the self-evaluated reference sample, indicating that the generated specifications are of comparable quality. While the overall scores are slightly lower than the reference, the Jaccard similarity is notably higher for generated

Table 4.1: Similarity and NLI evaluation metrics for each input example type.

Sample	Cosine (ST)	Cosine (TF–IDF)	Jaccard	NLI
Reference	0.9417	0.7277	0.1569	9
First	0.9305	0.6190	0.2081	8
Random	0.9169	0.5807	0.1881	8
Quality	0.9180	0.6111	0.1958	8

samples, reflecting a broader lexical range. The First sample produced the highest overall scores among the alternatives and was therefore selected for continued use in generation.

The next step involves selecting the most effective prompt format. Several prompt types were evaluated, examples include those shown in Figure 3.3 and Figure 3.4. The prompt format in Figure 3.4, which combines one-shot, instructional, and role-based prompting with clear formatting rules, consistently yielded the best results. Additional variants that modified the role description or instruction phrasing were tested, but none outperformed this configuration.

Using the optimal prompt template, hyperparameters and input example determined, a full scale generation was performed on the 205 requirements lacking test specifications and 481 test specifications were generated. Two filtering steps were applied:

- **Validation Check**: 70 samples were discarded due to missing or malformed template elements.
- **Near-Duplicate Removal**: 143 samples with BLEU > 0.9 and cosine similarity > 0.95 to any other sample were flagged, human evaluation confirmed negligible variation with only one or two word differences.

This yielded 268 valid, diverse specifications with the following metrics score that can be found in Table 4.2

Table 4.2: Average evaluation scores comparing cleaned generated samples to a reference sample.

Sample Set	Cosine (ST)	Cosine (TF–IDF)	Jaccard	NLI
Reference	0.9417	0.7277	0.1569	9.0
Generated Samples	0.9244	0.6325	0.1911	8.0

4.5.2 Back-Translation

Back-translation was explored using two experimental strategies, referred to as Version 1 and Version 2.

- **Version 1** used the following language chains:
  - English → French → English
  - English → German → English

Masking was applied only to template words using the tag `[MASK]`.

- **Version 2** used the following language chains:
  - English → French → English
  - English → German → English
  - English → Spanish → English

  In this version, masking was extended to include template words, technical terms, acronyms, and placeholders. The masking tag `X_X_X` was used, and prompts included stricter instructions explicitly directing the model not to modify or translate these tokens.

In total, 3627 back-translated samples were generated: 783 per chain for Version 1, and 639 per chain for Version 2. Each sample consisted of a requirement and test specification. Validation was conducted on both separately, and a sample was only marked valid if both preserved all masked tokens and achieved a BLEU score above 0.4 compared to their respective originals. If either component failed, the entire pair was invalid.

From Version 1, only 4 samples passed validation. In contrast, 405 samples passed from Version 2, primarily due to the improved masking and stricter prompting strategy. Table 4.3 shows the number of valid pairs and individual requirements that passed from each chain.

However, further analysis of the 409 validated samples exposed significant quality issues. Despite passing validation and achieving high scores on automatic metrics, including BLEU, ROUGE, cosine similarity, and Jaccard. Manual inspection revealed the presence of untranslated foreign words, incorrect or partial masking, and semantic inconsistencies. Attempts to use automated filtering led to false positives and failed to generalize across diverse errors. Due to these challenges and limited resources, manual review of all 409 samples was deemed impractical, and they were ultimately excluded from the final dataset.

Table 4.4 presents the average similarity metrics for all back-translated samples, including those that failed validation. These results initially suggested strong lexical and semantic alignment with the original samples, as indicated by relatively high BLEU, ROUGE, and cosine scores. However, these metrics proved to be poor indicators of actual quality. Despite these scores, many samples contained critical errors such as translated placeholders, foreign words, or altered technical terms that violated the structural and semantic constraints of the domain.

Table 4.5 reports similarity scores exclusively for the subset of samples that passed the automatic validation checks. These scores are even higher than those observed across the full dataset but were ultimately found to contain violations when manually inspected. These findings reinforce that while metrics such as BLEU and ROUGE can quantify surface similarity, they fail to capture deeper violations that matter in controlled, domain-specific generation tasks like test specification synthesis.

Table 4.3: Number of valid requirement–test pairs and requirements per translation chain.

Chain	Valid Requirement–Test	Valid Requirements
Chain 1 (EN→FR→EN)	158	467
Chain 2 (EN→DE→EN)	139	422
Chain 3 (EN→ES→EN)	108	411
Chain 4 (EN→FR→EN)	0	15
Chain 5 (EN→DE→EN)	4	70

Table 4.4: Average similarity metrics for all back-translated samples (including invalid).

Chain	Cosine (TF-IDF)	Jaccard	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Requirement Similarities (All Samples)						
Chain 1	0.772	0.562	0.495	0.797	0.618	0.764
Chain 2	0.749	0.572	0.483	0.777	0.570	0.737
Chain 3	0.766	0.586	0.502	0.781	0.588	0.744
Chain 4	0.666	0.480	0.394	0.730	0.474	0.681
Chain 5	0.681	0.461	0.392	0.751	0.536	0.717
Test Specification Similarities (All Samples)						
Chain 1	0.774	0.610	0.552	0.797	0.663	0.768
Chain 2	0.769	0.658	0.573	0.813	0.643	0.769
Chain 3	0.804	0.706	0.634	0.851	0.715	0.817
Chain 4	0.721	0.506	0.404	0.744	0.508	0.696
Chain 5	0.733	0.516	0.444	0.776	0.586	0.747

Table 4.5: Similarity metrics for validated back-translated samples.

Chain	Cosine (TF-IDF)	Jaccard	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Requirement Similarities (Valid Samples Only)						
Chain 1	0.833	0.661	0.617	0.862	0.719	0.840
Chain 2	0.815	0.683	0.627	0.844	0.683	0.817
Chain 3	0.829	0.702	0.644	0.856	0.706	0.835
Chain 5	0.864	0.593	0.574	0.814	0.676	0.792
Test Specification Similarities (Valid Samples Only)						
Chain 1	0.838	0.691	0.658	0.857	0.750	0.834
Chain 2	0.841	0.715	0.653	0.851	0.707	0.815
Chain 3	0.865	0.758	0.698	0.880	0.765	0.855
Chain 5	0.856	0.634	0.631	0.864	0.738	0.846

4.5.3 Contextual Augmentation

Contextual augmentation was used on the 639 available requirement and test specifications pairs, 371 of them were from the original dataset and 268 were from the prompt generation. This was done to further diversify the dataset. Different prompting techniques were tried such as role-based prompting and Zero-shot prompting where the prompt

looked like Figure 4.6. This turned out to be the prompt that gave the best results.

```
{Zero-shot prompting} I need you to improve the following

technical text for clarity and readability while keeping the
exact meaning only do it on everything that is after:

[Template words that are redacted]

I want you to keep the same format as given to you but only change
the things I ask for, so you will need to give me back the same
format as the text given.

text below:
{requirement_text} or {test_text}
```

Figure 4.6: Zero-shot instruction for contextual augmentation

More advanced prompts didn't work as well as the zero-shot prompt, and there was no more time to find a better prompt so it was settled to use the zero-shot. The final result from the contextual augmentation was 438 valid samples and 201 invalid ones. The invalid ones were discarded for not following certain criteria like not all template words found or simply bad metric scores. The metric score of the valid samples can be found in Table 4.6.

Table 4.6: Similarity Metrics for Requirements and Test Specifications for contextual augmentation

Type	Cosine (TF-IDF)	Jaccard	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
req_text	0.802316	0.526422	0.441661	0.751025	0.557572	0.712805
test_text	0.899300	0.635678	0.554017	0.843877	0.694724	0.811302

The results shows promising results and after a closer human evaluation done on the outliers that had the worst score, they were deemed to be worthy and passed the human evaluation.

4.5.4 Paraphrasing Augmentation

Just like contextual augmentation the same dataset of 639 available requirements and test specifications was used. Given the promising results from contextual augmentation, expectations were high that paraphrasing would yield similar or even better results. Paraphrasing augmentation was performed by instructing the model to rephrase the text while preserving its original meaning and maintaining the required template structure. Different prompting techniques were evaluated including zero-shot and role-based prompting. After testing, just like contextual the zero-shot prompt provided the most reliable results. The final prompt that was used can be seen in Figure 4.7 which turned out to be almost indentical with the contextual augmentation.

```
{Zero-shot prompting} I need you to paraphrase everything that

is after:

[Template words that are redacted]

I want you to keep the same format as given to you but only change
the things i ask for, so you will need to give me back the same
format as the text given.

text below:
{requirement_text} or {test_text}
```

Figure 4.7: Zero-shot instruction for Paraphrasing augmentation

The final output of the paraphrasing augmentation process resulted in 491 valid samples and 148 invalid ones. The invalid ones were subject to the same reason the contextual augmentation ones where they did not follow the criteria. The similarity metrics for the valid paraphrased samles are presented in Table 4.7

Table 4.7: Similarity metrics for requirements and test specifications in paraphrasing augmentation.

Type	Cosine (TF-IDF)	Jaccard	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
req_text	0.612695	0.517228	0.480339	0.718081	0.492883	0.686850
test_text	0.760292	0.623536	0.593548	0.812485	0.635871	0.784254

While the results are slightly lower compared to contextual augmentation, a final human evaluation confirmed that out of the 10 randomly selected samples all paraphrased samples were suitable to be used for the AI model. There is no time check all 491 valid samples so a random selecting was best. A quick check was also done on outliers. When augmenting the data it is expected that scores drop lower for each augmentation step as the data gets more noisy after every iteration, making each augmentation step use this added noisy data the next time.

4.5.5 Prompt Engineering

Prompt engineering plays a crucial role in guiding an AI model to generate accurate and structured test specifications from software requirements. Because of the domain specific language used in the automotive industry, generic prompts often fail to produce relevant results. To address this, several prompt strategies were explored focusing on how different prompt structures affect the quality of the generated outputs.

There were three prompt strategies that were tried:

1. **Control-token instruction prompts:**
    - Utilizing mistrals control tokens such as [INST] and [/INST] to define instruction boundaries.
2. **Chat-completion prompts:**

- Structured prompts using formats with system, user and assistant roles

3. **Schema-enforced (dsp) chat prompts:**
    - Same as chat-completion prompts but a strict response template that follows the test specification template that guides the model to output the same structure as the template.

Each prompt strategy was run seven times in total for every type-shot variant with 56 samples each from the test set. Each strategy used:

1. **Zero-shot:** The model is given a task with no examples.
2. **Half-shot:** The model is given only the requirement to write the test specification from
3. **n-shot:** The model is given one to five requirements and test specifications in its context window as guidance to write the new test specification. n denotes what type of "shot" it is. If n is equal to one it's called a "1-shot" and the model is given one requirement and test specification and so forth.

The results from each strategy:

Instructional prompt:

Table 4.8: Evaluation for the instructional prompt, values are rounded to the 3rd decimal.

Instance	Cosine (ST)	Cosine (TF-IDF)	Jaccard	BLEU	F1	ROUGE-1	ROUGE-2	ROUGE-L
zero_shot_inst	0.898	0.307	0.091	0.012	0.145	0.240	0.045	0.128
half_shot_inst	0.920	0.374	0.230	0.094	0.302	0.395	0.140	0.258
1_shot_inst	0.926	0.390	0.294	0.183	0.391	0.481	0.249	0.351
2_shot_inst	0.932	0.438	0.359	0.265	0.461	0.555	0.348	0.445
3_shot_inst	0.930	0.433	0.366	0.284	0.467	0.558	0.354	0.447
4_shot_inst	0.932	0.439	0.391	0.305	0.491	0.571	0.372	0.469
5_shot_inst	0.931	0.443	0.388	0.299	0.485	0.566	0.371	0.463

Chat-completion prompt:

Table 4.9: Evaluation Metrics for Chat-completion, values rounded to the 3rd decimal

Instance	Cosine (ST)	Cosine (TF-IDF)	Jaccard	BLEU	F1	ROUGE-1	ROUGE-2	ROUGE-L
zero_shot_chat	0.910	0.353	0.115	0.025	0.199	0.317	0.079	0.160
half_shot_chat	0.908	0.361	0.159	0.048	0.233	0.344	0.097	0.200
1_shot_chat	0.933	0.413	0.335	0.226	0.442	0.516	0.285	0.395
2_shot_chat	0.937	0.460	0.426	0.322	0.519	0.591	0.401	0.496
3_shot_chat	0.935	0.424	0.398	0.297	0.487	0.572	0.375	0.470
4_shot_chat	0.937	0.434	0.405	0.307	0.502	0.577	0.386	0.488
5_shot_chat	0.934	0.409	0.403	0.299	0.492	0.569	0.377	0.482

Schema-enforced or dsp:

Table 4.10: Evaluation Metrics for DSP Instances, values rounded to the 3rd decimal

Instance	Cosine (ST)	Cosine (TF-IDF)	Jaccard	BLEU	F1	ROUGE-1	ROUGE-2	ROUGE-L
zero_shot_dsp	0.900	0.301	0.130	0.031	0.166	0.283	0.091	0.181
half_shot_dsp	0.919	0.351	0.261	0.124	0.356	0.419	0.160	0.288
1_shot_dsp	0.933	0.401	0.331	0.218	0.439	0.512	0.275	0.384
2_shot_dsp	0.935	0.444	0.405	0.299	0.507	0.578	0.372	0.476
3_shot_dsp	0.935	0.431	0.413	0.318	0.508	0.580	0.378	0.477
4_shot_dsp	0.938	0.450	0.399	0.307	0.507	0.574	0.372	0.473
5_shot_dsp	0.935	0.435	0.405	0.308	0.507	0.572	0.369	0.470

As seen from the metrics given in the tables 4.8 4.9 4.10 there are big improvements to be had up until the 2-shot prompt in every strategy. After that the jump in improvement is marginal. The automatic metrics can only show so much so a human evaluation of the outputs are needed. Upon closer inspection none of the 10 samples from zero-shot follow the test specification template, even in the dsp strategy where a template is given. The half-shot prompts had the same issues as zero-shot but in some cases it followed the template but not as well as the others. 1-5 shot in every strategy followed the template but the quality of the output varied, as seen in the automatic metrics. The best metrics was around 2 and 3-shot prompting and the human evaluation had the same conclusion. The 4 and 5-shot didn't give the model a better output and made the model more biased with the given context. The best result was gotten trough the 2-shot chat-completion prompt and this is what the human evaluation from the thesis workers also concluded.

4.6 Human Evaluation of Model Output Quality

Before the engineers could begin evaluating the model outputs, a dedicated tool had to be developed to support the process. A fast, lightweight frontend tool was set up to run on the internal servers, designed specifically for this evaluation task. The tool presented each requirement and displayed both the human-written test specification and the AI-generated test specification side by side. Crucially, the tool was built to hide the origin of each output, so the engineers wouldn't know which was created by a human and which was produced by the model. This was done to reduce bias as much as possible and ensure that the scoring focused purely on the quality of the outputs themselves.

Once the frontend was completed and tested internally, a survey was sent out to the engineers working in the department. They were asked to participate in the evaluation when they had free time, scoring each pair of outputs based on how well they aligned with the requirements, how fluent and clear they were, and how useful they would be in practice. The engineers were not given access to any automatic metric results and were instructed to rely only on their own judgments.

After the survey period ended, the collected human evaluation data was compiled and analyzed. The results showed that, overall, the automatic metrics and the human judgments pointed in the same direction. Both types of evaluation agreed that the AI-generated outputs were usually weaker than the human-written ones. However, interestingly, there were some specific cases where the generated outputs managed to score evenly with the human ones, suggesting that the models are capable of producing competitive results under certain conditions. This alignment between human and automatic evaluation added confidence to the overall assessment and gave valuable insights into where the models currently stand in terms of practical usefulness.

Chapter 5
Discussion and conclusions

This chapter presents reflection on the outcomes of the thesis, analyzing both the strengths and limitations of the proposed AI driven approach for test specification generation. The results from automatic and human evaluation are discussed to determine how well the AI prototype met the initial goals and problem formulation. The chapter also explores how these findings align with the expectations set at the beginning of the thesis and what implications they have for the automotive industry. Finally the chapters identifies limitations encountered during the thesis and suggest future development and improvements.

5.1 Reflection on results

The thesis set out to investigate whether a large language model (LLM) could automate the generation of test specifications from software requirements. Throughout the thesis the results have highlighted both the capabilities and limitations of this approach.

5.1.1 Data Engineering

The data engineering phase consumed the most time out of all the steps. This is typical in AI projects when starting with unstructured or raw data. Despite several challenges in the data engineering phase such as dealing with incomplete data and inconsistent formatting, a strong pipeline was developed. The pipeline effectively transformed unstructured raw data into a structured JSON format and reduced the original 59 attributes per item down to 19 useful attributes.

5.1.2 Prompt Engineering

On the modeling side, the Mistral-7B model was used exclusively due to resource limitations and internal issues from the automotive company. While this constrained the exploration of alternative models, the Mistral-7B still demonstrated promising results. Various data augmentation techniques were applied, including prompt-based generation, back-translation, contextual augmentation and paraphrasing augmentation. Although Mistral-7B is not equally well-suited for all types of augmentation techniques, the overall efforts still expanded the dataset, contributing over 1200 new samples. However, the model showed varying levels of effectiveness across different augmentation strategies. Back-translation yielded the weakest results, despite generating 3627 samples across five

language chains, only 409 passed the validation checks, and ultimately none were included in the final dataset due to recurring quality issues. These included translated placeholders, untranslated foreign words, and inconsistencies. While automatic metrics such as BLEU (up to 0.64), ROUGE-1 (up to 0.88), and cosine similarity appeared strong (see Table 4.5), manual inspection revealed that these metrics were insufficient indicators of actual semantic and structural correctness. Longer back-translation chains, such as EN→FR→DE→EN, were also tested but yielded even worse results. This is a known issue in iterative translation settings (Sennrich et al., 2016), where the original meaning becomes increasingly distorted with each translation step. This is a known issue in iterative translation settings (Sennrich et al., 2016), where the original meaning becomes increasingly distorted with each translation step. In contrast, paraphrasing and contextual augmentation, while more controllable and structural, offered fewer variations. These methods were initially considered less suitable for large-scale augmentation, as it was deemed they would not yield enough varied samples. Therefore back-translation was considered promising for large-scale augmentation, even though it ultimately proved less effective in this domain-specific setting.

In contrast, contextual and paraphrasing augmentation techniques were far more successful. The contextual augmentation pipeline produced 438 valid samples, and paraphrasing yielded 491. Manual evaluation on a random subset of these samples confirmed their suitability for model training. These findings suggest that while Mistral-7B may not be robust for augmentation techniques like back-translation, it can reliably handle controlled rewriting tasks such as paraphrasing and contextual improvement when paired with strict prompts.

The evaluation results show that the AI generated test specifications can achieve comparable results to human written specifications based on automatic metrics such as BLEU, ROUGE and cosine similarity. However, automatic metrics alone were not a fully reliable criterion of true quality. A human evaluation was conducted confirming that even samples with lower metric scores could meet quality expectations. This shows the importance of combining automatic metrics with human evaluations in tasks where domain knowledge plays a significant role.

An important observation from the evaluation phase was that increasing the number of in-context examples (shots) did not lead to consistent performance gains beyond the 2-shot level. While results improved significantly from zero-shot to 2-shot prompting, metrics plateaued or slightly declined with 3 to 5 shots across all prompt strategies. Several factors may explain this behavior.

First, the inclusion of too many examples could have caused prompt overload. According to findings from Brown et al. (2020), attention-based models like transformers can become confused when overloaded with too many signals, especially in structured generation tasks. This was evident in some generations where hallucinations occurred.

A second factor is the technical constraint of the model's context window. As Mistral-7B has a fixed limit on the number of tokens it can attend to (Jiang et al., 2023), adding more examples reduces the available space for processing the actual input requirement and generating a coherent response. This increases the risk of truncation or reduced focus.

These insights explain why 2-shot prompting appears to be optimal, offering enough structure to guide generation without overwhelming the model with redundant or distracting input. This highlights the need for prompt strategies that prioritize the quality

and diversity of examples over quantity, especially in domain-specific applications like automotive test specification generation.

5.1.3 Conclusion

In conclusion, this thesis demonstrates that LLMs can assist in automating parts of the software validation workflow by generating test specifications from requirements. While the results are promising, especially with structured prompt engineering, the process is not yet fully reliable without human oversight. Prompting showed diminishing returns beyond two examples, suggesting that more context does not always lead to better output. Among the data augmentation techniques, only contextual and paraphrasing approaches produced consistently usable samples, while back-translation suffered from quality and validation issues. Further improvements in prompt diversity and augmentation precision are needed to enhance generation quality and robustness.

5.2 Goals of the thesis

There were three goals that were set for the thesis but how many were achieved? The first goal was to establish a data engineering pipeline that included some sub goals. This was reached and more was added to it. But it didn't go smoothly and had some hiccups on the way to reach the goal. Most of the time during the thesis was spent in the data data engineering part. This is usually the case for AI projects if the data is not already processed and ready to work on. The data engineering part had to be redone a couple of times when new discovery's were made. It was hard to write modular code that was easily adaptable like the one learned trough school. The data engineering worked perfectly but after coming to a later stage discovering that some of the HTML tags were good to keep until a later stage or that some feature was removed that might be needed. This meant that the data engineering code got refactored a lot of times and it was not only because the original code wasn't adaptable it was also because some of the changes were huge meaning that whole functions had to be rewritten. This sometimes caused a chain effect where for example some regex functions relied on certain words to be able to work. Or that one thing removed in the filtering process could mess something up in the mapping process relying on a certain attribute to work. But in the end the goal was reached and some things got added like the database.

The second goal was model development which didn't go as good. The plan was to conduct research into different models but the thesis only had access to one. This wasn't really that big of a problem, but there might be other models that are better suited for the job than the Mistral-7B was. The second sub goal was exploring different application strategies which was achieved. Not everything was done solely by the thesis workers like fine-tuning but a lot of it can be followed trough the methology.

The third and final goal was to evaluate the prototypes effectiveness. This was achieved through both automatic and human evaluations. The automatic evaluation used estrablished similarity metrics such as BLEU, ROGUE and cosine similarity to measure the similarity between manually written test specifications and AI generated ones. These metrics showed how closely the generated specifications matched human expectations. However automated metrics alone were never intended to be good enough assess the quality of the outputs. Human evaluations were conducted where engineers at the automotive company reviewed a small set of the generated test specifications. They assessed

whether the specifications correctly validated the corresponding requirements. Although time constraints limited the size of the human evaluation step the feedback confirmed that many AI generated specifications were sometimes as good as the ones written by the engineers but on average they were worse.

5.3 Answering problem formulation questions

A summary of the problem formulation questions that are answered through the thesis:

- **How are software requirements currently validated?**
  - The customer provides a requirement for a specific feature, which is then reviewed by the automotive company. If the requirement meets initial standards, it is approved and forwarded to the test team. The test team creates a corresponding test specification and script, followed by an internal review to ensure logic and compliance with company guidelines.
- **Which parts of the current test process are repetitive and resource intensive?**
  - The writing of test specifications for each requirements is repetitive, especially when requirements follow similar patterns. Engineers at the automotive company often copy-paste structure, modify parameter names and rewrite some manually.
- **How can an AI model identify similarities between new requirements and previously implemented requirements?**
  - By embedding both new and historical requirements and test specifications into a vector space using a transformer model like Mistral-7B, the cosine similarity between embeddings can be computed. This allows the model to retrieve similar requirements or test specifications which it has in its knowledge base where it can reference from.
- **How can automation of test case generation improve the efficiency and accuracy of the validation process?**
  - Automation reduces the manual efforts required by the engineers to generate boilerplate test cases. Using tools like LLMs, test cases can be produced faster and then verified or refined by a human. This speeds up the validation process and frees up time.
- **What criteria should be used to evaluate the effectiveness of the proposed prototype?**
  - The evaluation of test specifications underwent multiple steps, each step can be found in section 2.6. If the output from the model ever scored bad results through any of the evaluation methods, this usually ment that the test cases were vague or irrelevant. This in turn ment that a reprompt to the model was needed. Human evaluation was always needed at the last step offering a final step of validation.

5.4 Limitations

There were some key limitations in the thesis which are good to discuss. Firstly, this thesis was a part of a bigger project within the automotive company. This meant that the thesis already had a plan in mind and there was little space for the thesis workers to plan around. There was no need for new ideas as much of the work had already been thought of. While this brings a lot of upsides, there were some negatives as well. One major downside was that everything written by the thesis workers had to be reviewed before continuing, as it's the same workflow for the workers at the company. This meant that some code or findings were not up to company standards and had to be done before continuing. This led to some things planned in the sprints having to be pushed to the next ones, which contributed to some of the time constraints.

Another key limitation the thesis had was a hardware one. As the project only had access to one computer with a strong graphics card, this meant that it had to be shared and only one thing at a time could be run on the computer. This in turn made the tasks that had to be run on the computer into a priority list, and if task had higher priority that got to run first. This type of priority list worked quite good but it was a nuisance sometimes having to wait to test your code on the AI. This was only a problem at the later stages when needing to make code for the AI, because the data engineering was run locally on laptops.

5.5 Ethical aspects

Throughout the thesis there has been secrecy concerning the data used. The thesis uses requirement and test specification data from an automotive company. Although internal access to the data was shared, it was made sure to anonymize all the data and avoid any leakage of proprietary or sensitive business information. All data was stored locally and was never shared externally and used only for the purpose of the project. No access to customer related data was ever accessed or processed.

The developed AI model serves the purpose of improving internal efficiency by reducing repetitive manual work for the engineers at the company. AI models were used in hope of improving the human decision process in making the test specification making not replace it. The prototype generates test specifications but requires human validation before deployment, this ensures accountability. The prototype is designed for internal validation tasks only and is not intended to be used for safety critical decision in vehicle systems. This could free up time for more critical work that the engineers might focus on instead.

Although the data is not shared the methodology for the data processing pipeling and model configuration are documented in great detail to ensure reproducibility. Steps like data collecting, data cleaning and augmentation are described in detail and should work for task of the same caliber. The structure and approach used for building and evaluating the AI system are transparent and could be replicated on similar datasets.

Because the model is smaller, it needs less computational power, which in turn lowers its environmental impact. Large models typically require expensive hardware to be able to run the computation needed for both training and inference which would lead to increased energy consumption. The selected model Mistral 7B still offers competitive performance while needing less resource usage. This choice also aligns with the company's needs to

keep everything local which eliminates the need for costly cloud computers making our research more sustainable.

5.6 Future development

A good base has been built and a great proof of concept that shows that it could work. Right now the data that was used was not customer data but internal generic requirements and test specifications. Given that the generic dataset is more abstract than implemented customer requirements and test specifications, the real data should yield better results than the already good results the thesis got. Given that the thesis got good results these are the steps that could be developed more. The local run pipeline built by the thesis could instead be adapted into a real automatic pipeline which takes each new uploaded work item from Polarion through the data engineering and into the database. Then it will be given to the AI model so it can expand its knowledge base. This in turn should theoretically make the AI better with more requirements and test specifications added. If real customer data could be used, which is not as abstract as the generic dataset, better results might be achieved.

The results the thesis got should be able to be replicated and executed at other automotive companies, making it the new standard in the industry.

Chapter 6

Bibliography

[1] IBM. *What is artificial intelligence (AI)?* https://www.ibm.com/think/topics/artificial-intelligence. Accessed: February 23, 2025. 2025.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. 4th. Pearson, 2020.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[4] Jim Holdsworth and Mark Scapicchio. *What Is Deep Learning?* Accessed: 2025-04-29. 2024. URL: https://www.ibm.com/think/topics/deep-learning.

[5] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. "A Survey of the Usages of Deep Learning in Natural Language Processing". In: *CoRR* abs/1807.10854 (2018). arXiv: 1807.10854. URL: http://arxiv.org/abs/1807.10854.

[6] Ashish Vaswani et al. "Attention is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[7] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[8] Biao Zhang and Rico Sennrich. "Root Mean Square Layer Normalization". In: *CoRR* abs/1910.07467 (2019). arXiv: 1910.07467. URL: http://arxiv.org/abs/1910.07467.

[9] Joshua Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245 [cs.CL]. URL: https://arxiv.org/abs/2305.13245.

[10] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. arXiv: 1911.02150 [cs.NE]. URL: https://arxiv.org/abs/1911.02150.

[11] Iz Beltagy, Matthew E. Peters, and Arman Cohan. "Longformer: The Long-Document Transformer". In: *CoRR* abs/2004.05150 (2020). arXiv: 2004.05150. URL: https://arxiv.org/abs/2004.05150.

[12] Noam Shazeer. *GLU Variants Improve Transformer*. 2020. arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202.

[13] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.

[14] Yann N. Dauphin et al. "Language Modeling with Gated Convolutional Networks". In: *CoRR* abs/1612.08083 (2016). arXiv: 1612.08083. URL: http://arxiv.org/abs/1612.08083.

[15] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[16] Zihang Dai et al. "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 2978–2988. DOI: 10.18653/v1/P19-1285. URL: https://aclanthology.org/P19-1285/.

[17] Jack W. Rae et al. *Compressive Transformers for Long-Range Sequence Modelling*. 2019. arXiv: 1911.05507 [cs.LG]. URL: https://arxiv.org/abs/1911.05507.

[18] Harald Steck, Chaitanya Ekanadham, and Nathan Kallus. "Is Cosine-Similarity of Embeddings Really About Similarity?" In: *arXiv preprint arXiv:2403.05440* (2024). Presented at ACM Web Conference 2024 (WWW 2024 Companion). URL: https://arxiv.org/abs/2403.05440.

[19] Paul Jaccard. "Étude comparative de la distribution florale dans une portion des Alpes et du Jura". In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 37 (1901), pp. 547–579. DOI: 10.5169/seals-65331. URL: https://doi.org/10.5169/seals-65331.

[20] Adam Poliak. "A Survey on Recognizing Textual Entailment as an NLP Evaluation". In: *Proceedings of the First Workshop on Evaluation and Comparison of NLP Systems*. Ed. by Steffen Eger et al. Online: Association for Computational Linguistics, Nov. 2020, pp. 92–109. DOI: 10.18653/v1/2020.eval4nlp-1.10. URL: https://aclanthology.org/2020.eval4nlp-1.10/.

[21] Hugging Face. *Human Evaluation Basics*. https://github.com/huggingface/evaluation-guidebook/blob/main/contents/human-evaluation/basics.md. Accessed: 2025-05-14. 2025.

[22] Ivan Belcic and Cole Stryker. *What Is Data Engineering?* Accessed: 2025-05-12. 2024. URL: https://www.ibm.com/think/topics/data-engineering.

[23] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[24] Leonard Richardson. *Beautiful Soup Documentation*. https://www.crummy.com/software/BeautifulSoup/. Accessed: 2025-05-06. 2024.

[25] Steven Y. Feng et al. “A Survey of Data Augmentation Approaches for NLP”. In: *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Ed. by Chengqing Zong et al. Online: Association for Computational Linguistics, Aug. 2021, pp. 968–988. DOI: 10.18653/v1/2021.findings-acl.84. URL: https://aclanthology.org/2021.findings-acl.84/.

[26] Rico Sennrich, Barry Haddow, and Alexandra Birch. “Improving Neural Machine Translation Models with Monolingual Data”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 86–96. DOI: 10.18653/v1/P16-1009. URL: https://aclanthology.org/P16-1009/.

[27] Sosuke Kobayashi. “Contextual Augmentation: Data Augmentation by Words with Paradigmatic Relations”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 452–457. DOI: 10.18653/v1/N18-2072. URL: https://aclanthology.org/N18-2072/.

[28] Jason Wei and Kai Zou. “EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 6382–6388. DOI: 10.18653/v1/D19-1670. URL: https://aclanthology.org/D19-1670/.

[29] Siemens. *Polarion Documentation*. 2025. URL: https://polarion.plm.automation.siemens.com/.

[30] Python-polarion. *Polarion Package*. 2025. URL: https://pypi.org/project/polarion/.

[31] SQLite. *SQLite Documentation*. 2025. URL: https://www.sqlite.org/.

[32] Cloudera. *AI Data Lifecycle Checklist*. 2025. URL: https://www.cloudera.com/content/dam/www/marketing/resources/whitepapers/ai-data-lifecycle-checklist-cloudera-whitepaper.pdf?daqp=true.

[33] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released January 12, 2025. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/.

[34] Kishore Papineni et al. “Bleu: a Method for Automatic Evaluation of Machine Translation”. In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Ed. by Pierre Isabelle, Eugene Charniak, and Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040/.

[35] Satanjeev Banerjee and Alon Lavie. "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments". In: *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ed. by Jade Goldstein et al. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 65–72. URL: https://aclanthology.org/W05-0909/.

[36] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/.

[37] Zekun Li et al. "Guiding Large Language Models via Directional Stimulus Prompting". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 62630–62656. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/c5601d99ed028448f29d1dae2e4a926d-Paper-Conference.pdf.

[38] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[39] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[40] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652.

[41] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[42] Yixuan Weng et al. *Large Language Models are Better Reasoners with Self-Verification*. 2023. arXiv: 2212.09561 [cs.AI]. URL: https://arxiv.org/abs/2212.09561.

[43] Sander Schulhoff et al. *The Prompt Report: A Systematic Survey of Prompt Engineering Techniques*. 2025. arXiv: 2406.06608 [cs.CL]. URL: https://arxiv.org/abs/2406.06608.

[44] Mistral AI. *Tokenization — Mistral AI Large Language Models*. Accessed: 2025-05-10. n.d. URL: https://docs.mistral.ai/guides/tokenization/.

Printed by Tryckeriet i E-huset, Lund 2025

LUND
UNIVERSITY

Series of Bachelor`s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2025-1069
http://www.eit.lth.se