Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

A.Y. 2025/2026

Development of an LLM Model for automated Generation of ADAS Test Cases in a Keyword-driven Specification Language

Supervisors:

Prof. Paolo Garza

Eng. Andrea Vergine

Candidate:

Angelo Iannielli

Graduation Session

March 2026

Abstract

The rapid evolution of the automotive sector, especially in Advanced Driver Assistance Systems (ADAS), demands increasingly agile, scalable and reliable validation and testing pipelines. In this context, Large Language Models (LLMs) offer a real opportunity to improve industrial workflows by automating complex and technical tasks.

This thesis, developed in collaboration with the Nardò Technical Center (Porsche Engineering Group GmbH), creates the foundation for a system that automatically generates structured Test Catalogues in JSON format. The goal is to transform functional requirements into technical test catalogue files. This process requires not only semantic reasoning, but also the ability to manage and interpret large amounts of technical data. A key challenge is ensuring that the generated tests remain transparent and consistent with the underlying automotive architecture, a process that requires analysing and correctly matching thousands of heterogeneous signals.

The proposed approach uses generative models combined with a hybrid Information Retrieval system. This system uses both keyword search and semantic embedding models to find and map technical information accurately. The framework is designed to be scalable and secure, using local on-premise solutions to protect industrial data privacy.

By connecting natural-language requirements with structured technical data, this work offers a new perspective on making automotive test development more efficient. It serves as a methodological starting point to explore how modern AI can reduce manual effort.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Large Language Models: Architectures, Capabilities and Industrial Impact

In a period of rapid technological progress, significant efforts are being made to improve communication and interaction between humans and machines. [1] Enabling computers to understand and interpret human language at a deep level unlocks a wide range of possibilities, both in everyday life and in professional or industrial environments. Artificial intelligence, and in particular **Natural Language Processing** (NLP), is increasingly allowing machines to perform tasks such as text analysis, machine translation, speech recognition and text generation. Recent advances in NLP have led to the development and widespread adoption of **Large Language Models** (LLMs), systems based on deep neural architectures capable of learning the structure and patterns of human language. Their capabilities emerge from training on massive amounts of textual data, made possible by *self-supervised learning* techniques, which do not require human-annotated labels. This approach enables the models to learn complex linguistic patterns and to generalize across a wide variety of tasks.

Dual Capabilities: NLU and NLG LLMs are characterized by a dual capability [1]. The first is *Natural Language Understanding* (NLU), which enables the model to interpret an input text sequence by extracting its context, meaning and semantic relations. This ability is essential for tasks that require deep text analysis,

such as sentiment analysis, entity recognition and syntactic parsing. The second component is *Natural Language Generation* (NLG), which allows the model to produce new text sequences, generating content that is coherent, informative and similar to human language. The combination of NLU and NLG makes LLMs highly versatile tools, capable of supporting human–machine interactions in which users can ask questions and receive meaningful and contextually appropriate responses. Common applications include chatbots, machine translation, summarization and content creation. However, the flexibility of this technology enables a much broader range of use cases, spanning domains such as medicine, healthcare, e-commerce, education, finance, banking and automotive.

LLMs in Industrial Environments and Decision Support The adoption of these technologies is gaining increasing importance, not only from a technical perspective but also for their social impact and their role in providing support to individuals [2]. In industrial environments, LLMs have the potential to automate time-consuming and repetitive tasks, while also improving employees' quality of life. By reducing the burden of operational work, they allow workers to focus more on creative and innovation-driven activities. These AI systems can support decision-making processes, including real-time decisions, and contribute to improving overall productivity within organizations. Beyond assisting with the understanding of technical documents, LLMs enable faster access to relevant information, provide contextual explanations and can actively participate in content creation. An especially promising application concerns *test design*, where LLMs can accelerate the development of new features and products. Through the development of virtual assistants, LLMs can also accompany employees in their daily tasks, acting as a colleague who is always available to answer technical questions, offer personalized guidance and even perform simple tasks autonomously. This contributes to making work more efficient, accessible and supportive.

Current Limitations and Technical Challenges Despite their capabilities, these systems are far from free of limitations [3]. Their operation is fundamentally based on estimating probability distributions, which, although highly refined, still guide the generation of outputs in a probabilistic manner. The accuracy of an LLM depends both on the size of its architecture, which is measured in the number

of learned parameters, and on the quantity and quality of the data used during training. It is also important to note that training data may contain *biases*, which inevitably influence the model's behavior. Moreover, the ability of LLMs to produce realistic and convincing text increases the risk of accepting information that appears coherent but is actually the result of *hallucinations*, meaning content that is fabricated or not supported by the underlying data. Another critical aspect concerns energy consumption and hardware requirements. Training large-scale models demands substantial computational resources and even their inference can be costly, especially in environments with limited capabilities. This represents a significant challenge for many industrial and embedded applications.

1.2 Porsche Engineering

Company background Porsche was founded in Stuttgart in 1931 by Ferdinand Porsche and has always been strongly oriented toward innovation and technological advancement in the automotive sector. Porsche Engineering Group GmbH [4] is a fully owned subsidiary of the German manufacturer and is responsible for a wide range of activities, from the development of automotive architectures to AI-based solutions for autonomous driving. Porsche Engineering is a mid-sized company with a global presence, with offices in Germany, China, Italy (NTC), the Czech Republic, Romania and the United States. Its teams work on highly innovative projects, combining advanced engineering expertise, interdisciplinary collaboration and a strong commitment to quality.

Nardò Technical Center (NTC) The *Nardò Technical Center* (NTC) [5] has been conducting automotive testing for more than fifty years, relying on innovative and safe technologies for future mobility. The center supports the integrated development and validation of next-generation vehicles, covering a wide range of ADAS functions and autonomous driving systems. Its activities include both virtual simulations and on-track testing. With a portfolio of virtual proving grounds, state-of-the-art simulation software, physical simulators and high-performance real-time computing hardware, NTC helps engineers bridge the gap between simulation and real-world testing, accelerating the vehicle development process. The facility includes several test tracks, among which the famous high-speed ring stands

out: a circular track with a diameter of about 4 km and a total length of 12.68 km. Its parabolic profile allows vehicles to drive in any lane at a compensation speed of up to 245 km/h, balancing the centrifugal force. This design gives drivers the sensation of traveling on an infinite straight line, enabling high-speed testing with increased safety and reliability.

Chapter 2

State of art

2.1 The advent of Large Language Models: first developments

The advent of Large Language Models (LLMs) is rapidly reshaping everyday life and is being integrated into numerous production processes across a wide range of industries. An LLM can generally be defined as an artificial system capable not only of *understanding* human language but also of *generating* it. Thanks to their remarkable flexibility, these models can be used to tackle a wide variety of tasks. Among the most common are question answering, text summarization, and creative text generation. Their applications become even more compelling when LLMs are used to address tasks that require genuine reasoning abilities.

The functioning of language models is inherently probabilistic: given a sequence of words as input, the system generates the output that is considered most likely. The result may be a representation of the original text or an entirely new sequence.

2.1.1 N-gram model

The *n-gram model* is one of the simplest techniques for building a probabilistic language model. Based on a strong mathematical assumption known as the Markov assumption, it states that the probability of a word w_t depends only on a subset of the n preceding words [6]. Formally:

$$P(w_t|w_{t-1}, ..., w_1) \approx P(w_t|w_{t-1}, ..., w_{t-n+1}) \tag{2.1}$$

These probabilities are estimated by examining how frequently each word in the corpus appears after a specific window of n preceding words, known as the context.

2.1.2 RNNs

Recurrent Neural Networks (RNNs) [7] were among the first neural-network based approaches applied to language processing. Unlike traditional *Fully Connected Neural Networks* (FNNs), which require fixed-size inputs and outputs, RNNs are designed to process sequences of arbitrary length. This architecture also enables the model to retain information about previously processed elements through an internal hidden state that acts as a form of memory.

Despite these advantages, RNNs exhibit several limitations that make them difficult to employ effectively in linguistic tasks. During training through *Back-propagation Through Time* (BPTT), they suffer from the well-known vanishing and exploding gradient problems, which prevent optimization stability and efficiency. Moreover, the hidden state tends to lose information about distant elements in the sequence, making RNNs poorly suited to capture long-range dependencies.

2.1.3 Gated RNNs

Subsequent research led to the development of more advanced variants of RNNs, including *Long Short-Term Memory* (LSTM) [7] networks and Gate*d Recurrent Units* (GRUs) [8]. These architectures were designed to overcome the limitations of standard RNNs, by introducing gating mechanisms that regulate the flow of information. LSTMs employ three main gates:

- **input gate**, which controls how much of the new input should be included into the memory;
- **forget gate**, which determines which past information should be discarded;
- **output gate**, which selects the information to be propagated to the next processing step.

GRUs, instead, use a compact structure based on two gates:

- **reset gate**, which controls how much of the previous state should be forgotten;
- **update gate**, which determines how much of the past information should be retained.

2.2 Transformers

Despite the improvements, these models are unable to match the performance of modern large language models. With the introduction of *Transformers*, first proposed by *Google* in 2017 [9], and thanks to the **attention mechanism**, it became possible to design an architecture capable of modeling long-term dependencies much more effectively. Like RNNs, Transformers can process sequences of variable length, but they offer a crucial advantage: their computation is not strictly sequential. This makes them significantly easier to parallelize and far more suitable for large-scale training.

The original Transformer architecture is organized into two main components: the **encoder** and the **decoder**. The encoder processes the entire input sequence and produces a contextualized representation for each token. These representations are then passed to the decoder, which generates the output sequence *autoregressively*, leveraging both the encoder's output and its own masked self-attention mechanism. Both components consist of a stack of layers, each responsible for a specific processing step. This modular structure enables the model to learn complex relationships within the sequence in a highly efficient and parallelizable manner.

2.2.1 Input embedding layer

The initial text sequence is fed into the first component of the architecture, the input embedding layer, after being divided into elementary units that the Transformer can process. These units, known as **tokens**, are produced through a procedure called *tokenization*, which splits the text into word-level, subword-level, or character-level elements.

Tokenization Several tokenization strategies exist, differing in the desired level of granularity. One option is to represent each character as a token, an approach known as *character-level tokenization*. This method avoids the out-of-vocabulary

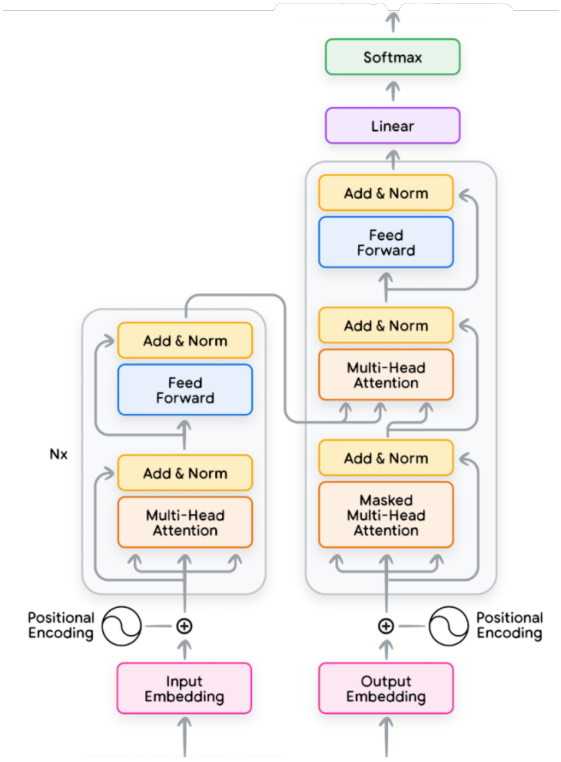

Figure 2.1: Encoder-decoder architecture

problem and is straightforward to implement, but it is inefficient: it produces very long sequences with limited semantic content, significantly increasing the computational cost of the attention mechanism, whose complexity grows quadratically with sequence length. To obtain more informative units, tokenization can instead be performed at the *word* or, more commonly, the *subword level*. Subword techniques offer an effective compromise: they generate tokens that carry meaningful information while still allowing the model to generalize frequent components such as prefixes, roots, and suffixes, enabling the handling of previously unseen words.

Byte-Pair Encoding

An example of subword-level tokenization is *Byte-Pair Encoding* (BPE) [10]. This is a data-driven technique used to construct a token vocabulary from a text corpus. The goal is to obtain a finite set of variable-length tokens, selected according to their frequency in the corpus. The process begins by splitting the text into individual characters. Then, the most frequent adjacent pair of symbols is identified and replaced with a new token. This operation is applied iteratively: at each step, the corpus is updated by merging the selected pair and the resulting token is added to the vocabulary. The algorithm stops once a predefined maximum vocabulary size has been reached. The final vocabulary contains units that may correspond to entire words or to frequent subword fragments.

Embeddings and Positional Encoding For each token, the model assigns a *high-dimensional vector* obtained through a lookup table. These vectors, known as **embeddings**, represent vocabulary items in a multidimensional space, providing each token with an information-rich representation. All embedding values are automatically learned during the training process. However, embeddings defined in this way do not encode any information about the position of tokens within the sequence. To incorporate positional information, additional vectors are added to the token embeddings. The original formulation, introduced in "**Attention Is All You Need**"[9], is defined as follows:

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{model}}}) \tag{2.2}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{model}}}) \tag{2.3}$$

2.2.2 Attention layers

Self-Attention [9] [11, p. 12] After converting tokens into embeddings, these vectors are passed to the **multi-head attention module**. Each module consists of several attention heads operating in parallel, with each head applying an independent self-attention mechanism. Attention is the component that enables the model to capture contextual relationships, allowing it to focus on the most relevant

parts of the sequence and to identify dependencies between distant tokens.

Each attention head is defined by three trainable weight matrices, W_q, W_k, and W_v. By multiplying these matrices with the input embeddings, the model obtains the *Query* (Q), *Key* (K) and *Value* (V) vectors. A Query vector represents a specific token and is used to determine which other tokens in the sequence are relevant to it. The Key and Value vectors play complementary roles: Keys determine how strongly each token should contribute, while Values contain the information to be aggregated.

The attention score is computed as the dot product between each Query and all Keys, indicating the relevance of each token with respect to the one being examined. After normalization and application of the softmax function, the resulting attention weights are used to weight the corresponding Value vectors. The weighted sum of the Values yields a contextualized representation of the original token. Formally:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.4}$$

The use of multiple attention heads proves particularly powerful in tasks that require a deep understanding of the *syntactic* and *semantic* structure of complex texts. Each head processes a different projection of the same input sequence, capturing complementary relationships between tokens. The Transformer can therefore compute multiple independent representations of the sequence, which are then concatenated and transformed into a single output. The resulting representation is rich in *contextual information* and can be viewed as an analysis performed from multiple perspectives at once. This enables the model to capture various patterns, such as long-range dependencies, grammatical relations, or semantic associations.

Masked Self-Attention The *masked self-attention module* is used within the decoder and is responsible for computing attention-based representations over the generated sequence. Its mechanism preserves the autoregressive nature of the output: each token is allowed to attend only to the tokens that precede it, without accessing any future information. This behavior is essential for the decoder, which during inference updates its input sequence step by step by appending the most recently generated token. Since all previously generated tokens are reprocessed at each iteration, the masking operation ensures that the model does not rely on

or alter information from future positions. Computationally, masked attention is implemented by adding a mask to the QK^T matrix, assigning $-\infty$ to all positions corresponding to future tokens and 0 to all valid positions. As a result, the subsequent softmax operation completely suppresses contributions from future tokens.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}} + Mask)V \tag{2.5}$$

Cross-Attention The *Cross-Attention mechanism* enables the decoder to incorporate relevant information from the input sequence. It uses the contextualized representations produced by the encoder as Key and Value vectors, while the Query vectors are derived from the tokens generated by the decoder. This allows the decoder to dynamically attend to the entire encoded sequence, identifying which parts of the input are most useful for generating the next token. Therefore, Cross-Attention serves as the connection point between the model's input and its output, ensuring that the decoder can continuously access the information encoded from the original sequence and produce coherent, context-aware responses.

2.2.3 Feedforward Layer

The representations produced by the attention layers are then processed by the *feed-forward layers* [11, p. 15], whose purpose is to generate a more complex and nonlinear transformation of the input. This module typically applies two linear transformations separated by a nonlinear activation function, such as $ReLU$ or $GELU$. The feed-forward network is applied independently to each token, using the same parameters across all positions. It follows that neither the feed-forward operations nor the attention mechanism inherently encode the original ordering of the tokens. This is due to the mathematical nature of the operations involved: matrix multiplications and point-wise transformations do not incorporate positional information. For this reason, positional embeddings play a crucial role, injecting positional information into each token representation before it is processed by the model.

2.2.4 Add and Norm

Within the Transformer architecture, additional components known as *Add and Norm layers* [11, p. 15] combine a residual connection with a layer normalization step. Their purpose is to improve gradient flow during backpropagation and stabilize the distribution of activations. Placed after the main attention and feed-forward modules, these layers normalize each sample by computing its mean and variance, and then apply a transformation using the trainable parameters γ and β . This process contributes to more stable optimization and faster convergence during training.

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{2.6}$$

Residual connections, on the other hand, add the original input of a layer to its output, enabling efficient propagation of both information and gradients throughout the network. This allows the model to learn transformations that represent slight modifications of the input, significantly mitigating issues related to vanishing and exploding gradients.

2.2.5 Sampling approaches

At the output of the Transformer, a final linear layer followed by a *softmax* produces a probability distribution over the vocabulary tokens. From this distribution, the model selects one token at a time, appending it to the generated sequence. The selection method can vary depending on the desired behavior. In particular, there are two families of strategies that can be distinguished: the first one is about **deterministic** methods, which always produce the same output given the same initial sequence, while the second regards **stochastic** methods, used when a more creative or variable result is desired. Some examples [11, pp. 53–54]:

- **Greedy search**: this is the most intuitive deterministic method. At each step, the token with the highest probability in the distribution is selected, typically producing repetitive and predictable outputs.
- **Random sampling**: the token is chosen according to the probability distribution. In theory, no token is completely excluded, with selection proportional

to its probability. This can lead to more creative text, but also increases the risk of incoherent sequences, since low-probability tokens may occasionally be sampled even when they do not fit the context.

- **Beam search**: at each generation step, the top n alternatives are selected according to the probability distribution. Each alternative is expanded, generating parallel paths that correspond to different candidate sequences. At every subsequent step, all active paths are evaluated and only the most probable ones n are kept, discarding the least promising sequences. This is a deterministic technique.

- **Top-K sampling**: for each new token, the K most probable candidates are selected and all others are excluded. Their probabilities are then renormalized to form a new distribution from which the next token is sampled. In high-entropy situations, this approach may exclude potentially valid tokens; in low-entropy situations, it may include candidates that are not contextually appropriate.

- **Nucleus sampling**: also known as *Top-P sampling*, this method selects a subset of tokens whose cumulative probability reaches a fixed threshold P. The number of candidates adapts dynamically to the model's confidence, allowing for greater diversity when uncertainty is higher.

- **Temperature sampling**: this method reshapes the probability distribution using a parameter T, which controls the level of diversity. For each token, the probability is recalculated as:

$$y_i = \frac{\exp\left(\frac{Z_i}{T}\right)}{\sum_j \exp\left(\frac{Z_j}{T}\right)} \tag{2.7}$$

  where Z_i is the logit associated with the token i. Low temperature values make the distribution sharper and closer to greedy behavior, while higher values increase randomness. In the limit $T \to 0$, the prediction becomes deterministic and equivalent to the Greedy search technique.

2.3 Fine-tuning

The first training phase of large language models is known as **pre-training** [11, p. 45]. It relies on massive unlabeled datasets and a self-supervised objective, in which the model learns to generate text token by token from a given context. This process is extremely resource-intensive, often requiring weeks or months of computation on advanced hardware such as GPUs or TPUs. Pre-training produces models capable of understanding and generating natural language across a wide range of topics. To perform specific tasks, models undergo an additional training phase called **fine-tuning**. This can be carried out in different ways depending on the dataset and the desired objective.

2.3.1 Supervised fine-tuning

Supervised fine-tuning [11, p. 46] uses datasets that are much smaller than those used for pre-training. These datasets consist of carefully curated examples, each containing an input prompt and a target response. Instruction-tuning trains the model to follow instructions, such as summarizing a text or writing code. Dialogue-tuning uses conversational data to teach the model how to handle question-answer interactions. Safety-tuning aims to reduce harmful, biased, or toxic outputs rather than teaching a specific task.

2.3.2 RLHF

Reinforcement learning from human feedback (RLHF) [11, p. 47] aligns model outputs with human preferences. It typically leads to responses that are more helpful, truthful and safe, while reducing the likelihood of undesirable content. The standard procedure consists of three stages:

- *supervised fine-tuning* on human-written examples;
- training a *reward model* based on human preference judgments;
- *reinforcement learning* (PPO) using the reward model as feedback.

This process guides the main model toward outputs that users find more appropriate and aligned with human expectations.

The reward mechanism is counterbalanced by a *KL-divergence term*, which prevents the fine-tuned model from drifting too far from the original one. This regularization is obtained by subtracting from the reward model's score a term proportional to the log-ratio between two probabilities: the likelihood of producing a given output under the new policy and the corresponding likelihood under the original model. The overall reward function is:

$$R(x, y) = r_\theta(x, y) - \beta log[\frac{\pi_{new}(y|x)}{\pi_{old}(y|x)}] \tag{2.8}$$

where x denotes the input, y the generated output, r_θ the score assigned by the reward model, and π_{new}, π_{old} the probabilities of generating y from x under the new and original policies, respectively.

2.3.3 Parameter Efficient Fine-tuning

Traditional fine-tuning techniques require a significant amount of time and computational resources, since they essentially continue to train all of the model's parameters. *Parameter-Efficient Fine-Tuning* (PEFT) [11, p. 49] methods aim to make this process more affordable, enabling the improvement of a pre-trained model even on modest hardware. By drastically reducing the number of parameters that are updated, it is possible to achieve results comparable to those of full fine-tuning. In practice, only a limited subset of parameters is allowed to change, while the rest remain frozen at their pre-training values. Depending on how these parameters are selected or introduced, different PEFT strategies can be defined.

BitFit *Bias-Terms Fine-Tuning*, also known as *BitFit*, is one of the simplest PEFT techniques and consists of keeping all model parameters frozen except for the bias terms in each layer. This allows training to continue on an extremely small subset of values, significantly reducing the computational cost. Despite its simplicity, this approach can achieve performance comparable to full fine-tuning while updating only a tiny fraction of the model's parameters.

Adapter-based Fine-tuning This technique introduces small modules, called adapters, into the Transformer architecture. Each adapter consists of two linear layers: a down-projection that reduces the dimensionality and an up-projection that

restores it, connected through a non-linear activation. During fine-tuning, only the parameters of these additional modules (and also the Norm Layer parameters) are updated, while all other weights remain frozen at their pre-training values. Thanks to the residual connections and the careful initialization of the adapter layers, the model can be adapted without disrupting its original behavior, preserving the knowledge acquired during pre-training.

LoRA *Low-Rank Adaptation* is based on the empirical observation that weight matrices in Transformers can be approximated using low-rank structures, meaning matrices whose rank is much smaller than their full dimensionality. With the original weights kept frozen at their pre-training values, new trainable matrices are introduced and added as a learned update. Instead of learning full-size matrices, LoRA represents this update as the product of two smaller matrices, reducing the number of parameters to optimize while still influencing the targeted layers. LoRA modules are considered plug-and-play: multiple sets of LoRA weights can be trained independently for different tasks. During inference, the appropriate LoRA matrices can be loaded according to the task, without altering the original parameters of the model.

Soft prompting *Soft prompt tuning* freezes the entire model and learns a set of trainable embedding vectors associated with each target task. Despite the name, this technique does not aim to create better textual prompts: the model learns a sequence of continuous vectors that represent the task and are pre-pended to the user's prompt during inference. These embeddings are optimized until the model produces satisfactory outputs for the task, allowing specialization without modifying any of the model's internal parameters. This makes the approach extremely lightweight and computationally efficient.

2.4 Optimization techniques

Language models have improved their performance over time by increasing the number of trainable parameters. However, this has also led to a substantial rise in resource requirements, both in terms of memory consumption and computational

cost. These challenges affect both the training and inference phases, often necessitating expensive hardware even for model deployment. Optimization methods typically introduce a trade-off between response quality, latency and computational cost. For example, inference can be accelerated and model cost reduced by accepting a slight decrease in accuracy, using fewer parameters, or applying numerical approximations. In many situations, especially when the tasks are relatively simple, reducing a model's capacity does not significantly impact its effectiveness, enabling resource savings without noticeably degrading output quality.

2.4.1 Quantization

The *quantization* [11, p. 60] process reduces the numerical precision of a model's parameters and activation functions. Typically, weights are stored in *32-bit* or *16-bit floating-point format*, but they can be compressed into *8-bit* or even *4-bit* representations. Thanks to the reduced memory footprint, quantization enables the deployment of larger models on hardware with limited capacity. Operations on lower-precision numbers also require fewer computational resources and shorter execution times. Quantization can be applied only during inference, or incorporated into training. In the latter case, the technique is known as *Quantization Aware Training* (QAT), which is more robust because it allows the model to recover part of the performance lost due to quantization.

Absmax *Absmax quantization* scales values proportionally and symmetrically within a new numerical range. Starting from the maximum absolute value of a parameter, a scaling factor is computed and applied to all other values, mapping them into the range supported by the chosen format (e.g. int8). This method preserves the exact representation of zero.

Zero-point *Zero-point quantization* instead uses an asymmetric range, assigning the smallest representable value to the lower bound and the largest to the upper bound. All values are shifted and scaled accordingly. This approach makes more efficient use of the available range but may introduce a misalignment for values that were originally exactly zero.

2.4.2 Distillation

Using smaller models improves inference time and reduces resource consumption, but response quality can degrade significantly. *Distillation* [11, p. 61] introduces two models: a smaller student and a larger teacher. Since larger models typically outperform smaller ones, under the same training conditions, they can serve as a reference from which the student learns.

Data distillation In *Data distillation*, the teacher generates additional synthetic data that is incorporated into the student's training set. This approach can be effective, provided that the synthetic data is of sufficient quality.

Knowledge distillation *Knowledge distillation*, on the other hand, aims to align the probability distribution produced by the student with that of the teacher, enabling the smaller model to imitate the behavior of the larger one even without explicit labels.

2.4.3 Output-preserving optimizations

Beyond techniques that modify the model's behavior or reduce its internal complexity, there is a class of optimizations that can speed up inference without affecting output quality. These so-called output-preserving methods do not alter the model's parameters or its generative capabilities; instead, they optimize specific operations from an algorithmic point of view.

Prefix caching During inference, the first processing step is known as the *prefill phase*, in which the attention key and value vectors are computed for every input token across all Transformer layers. Depending on the length of the input sequence, this step can require a substantial amount of computation. The subsequent decoding phase, responsible for generating new text, uses these attention vectors at every iteration. To avoid recompute everything repeatedly, a *KV Cache* is employed to store the precomputed key and value matrices. *Prefix caching* [11, pp. 63–65] applies to a fixed subsequence of text and ensures that the key and value vectors for that prefix are computed only once and reused in subsequent requests. This optimization is effective whenever the initial part of the prompt remains unchanged

between inference calls. Since, in causal self-attention, each token can attend only to preceding tokens, the cached values remain valid. Additional input can then be appended after the cached prefix. This technique is particularly useful in multi-turn scenarios where the model must repeatedly operate on a document provided by the user, which can be processed a single time. The cache may be stored in memory or on disk, and it is crucial that the structure of the input remains identical across requests to avoid invalidating the precomputed results.

Speculative Decoding The decoding phase is autoregressive and therefore difficult to parallelize, unlike the prefill phase where the matrix multiplications required for attention can be executed in parallel. Generating a new token requires that all previous tokens have already been produced. *Speculative decoding* [11, pp. 65–67] aims to accelerate each decoding step by introducing a smaller and faster auxiliary model, known as the drafter, which repeatedly proposes short sequences of n tokens. The main model, which is slower but more accurate, then verifies the proposed block, deciding whether to accept the entire sequence or only part of it. The first token considered invalid is regenerated by the main model and the drafter resumes its work from that point. The effectiveness of this technique depends on how well the two models are aligned: if the drafter's proposals are frequently rejected, the computational advantage disappears. Speculative decoding is considered quality neutral, as it does not degrade the quality of the final output.

Batching During the decoding phase, processing units may not be fully utilized. Executing multiple requests simultaneously on the same hardware can improve both throughput and latency. *Batching* [11, p. 67] is particularly effective during decoding, as it allows GPUs and TPUs to operate at full capacity by processing several sequences in parallel. The main constraint is memory: large batches can easily saturate it, so the number of requests per batch must be chosen carefully.

Parallelization *Parallelization* [11, p. 68] distributes the most computationally intensive operations across multiple devices. Transformers, composed of many layers, scale well horizontally and lend themselves to splitting the workload across hardware units. However, this technique requires precise and efficient synchronization between devices, since slow communication can compromise the advantages gained from

parallel execution.

2.5 Applications

Language models have revolutionized the way information is processed, thanks to their ability to understand structure, meaning, and context, as well as to generate coherent new text. LLMs have numerous real-world applications [11, pp. 68–79], since a wide variety of tasks can be expressed directly in natural language.

Coding Generative models are capable of interpreting requirements and objectives expressed in natural language and translating them into algorithms or code sequences. They serve as effective assistants for developers in many use cases, such as generating code from simple descriptions or supporting debugging activities. Their applications span the entire software development life-cycle, from defining functional requirements to producing test cases and documentation. LLMs are also valuable for refactoring existing code to improve correctness and efficiency, as well as for code translation, when a program needs to be converted from one programming language to another.

Translation The strong linguistic understanding and the ability to generate fluent, high-quality text make these models particularly well suited for translation across different languages. During training, the model must be exposed to a substantial amount of multilingual data in order to learn the relationships that link words and syntactic structures across languages. Compared to traditional translation algorithms, LLMs can produce outputs that are more coherent, natural, and contextually appropriate, even incorporating idiomatic expressions and colloquialisms. The most advanced applications include real-time translation, both for text in global platforms and for spoken conversations.

Summarization The comprehension and reformulation capabilities of LLMs can be leveraged to produce compact content, ranging from document and article summaries to conversational summarization in chat-based applications. Thanks to their ability to identify the most relevant information and express it clearly and

coherently, these models are particularly effective at condensing long or complex texts.

Question answering Unlike traditional QA systems, which mainly relied on keyword matching or simple information-retrieval techniques, LLMs can understand the broader context surrounding a question and produce more accurate and rich answers. Depending on the application, they can be used to build virtual assistants capable of providing users with information, explanations, and guidance. The prompt can be enriched with a set of relevant details retrieved from external sources through advanced search mechanisms. Using prompt engineering techniques, it is possible to craft clear and explicit instructions that the model must follow, improving the overall quality of the generated responses.

Natural Language Inference *Natural Language Inference* (NLI) is a family of tasks that aim to determine whether a given hypothesis is according to, contradicted by, or unrelated to a textual premise. Sentiment analysis, for example, involves classifying content based on the emotions or polarity it conveys. Other applications arise in the medical domain, where the model can support the analysis of clinical histories and associated diagnoses, helping to identify relevant patterns or correlations. Classification tasks may also involve ambiguous or overlapping categories, which traditional algorithms often struggle to handle. LLMs, thanks to their contextual understanding, are better equipped to manage such complexity.

2.6 Popular models

Depending on the components included in the architecture, different types of Transformer models can be defined[11, pp. 21–22]. The original **encoder–decoder** architecture contains all the layers previously described: an encoder that produces a contextualized representation of the input sequence and a decoder responsible for generating the output text. This configuration is typically trained on supervised sequence-to-sequence tasks such as translation, summarization, and question answering. **Encoder-only** models include only the encoder block. They are used for tasks that do not require text generation but instead rely on converting the input into a dense and informative representation, or aim to do classification. A

common training objective is *Masked Language Modeling* (MLM), which requires the model to predict masked tokens within a sentence. **Decoder-only models**, on the other hand, are the most used in modern Large Language Models. In this architecture, the input sequence is processed directly by the decoder, which treats it as part of the autoregressive token stream. As a result, the model does not require a cross-attention module. These models are trained using causal language modeling, a self-supervised objective in which the model predicts the next token given all previous ones.

2.6.1 BERT

BERT [11, p. 25] is an example of an encoder-only model, trained using two self-supervised objectives. The first, Masked Language Modeling (MLM), aims to predict missing words within a partially masked text. Selected tokens are replaced with a special [MASK] token and processed normally through the attention and feed-forward layers. A fully connected layer followed by a softmax is added on top of the model to produce a probability distribution for each masked position and reconstruct the original words. BERT is also trained on the *Next Sentence Prediction* (NSP) task. Here, the model receives two sentences separated by the [SEP] token, and the goal is to determine whether the second sentence is a coherent continuation of the first. The final prediction is obtained by applying a fully connected layer to the embedding associated with the [CLS] token, placed at the beginning of the sequence. Thanks to this training setup, models like BERT exhibit strong natural language understanding capabilities, making them well suited for tasks such as sentiment analysis, text classification, and natural language inference.

2.6.2 GPT-1

GPT-1 [11, pp. 23-24] (Generative Pre-trained Transformer 1) is a decoder-only model developed by *OpenAI* in 2018. Its architecture consists of 12 Transformer layers, each equipped with 12 attention heads, for a total of approximately 117 million parameters. The model supports a maximum context size of 512 tokens, which defines the longest sequence it can process in a single forward pass. GPT-1 was trained on the *BooksCorpus* dataset, containing around 7,000 books from various genres, using a self-supervised pre-training objective based on causal language

modeling, where the model learns to predict the next token given the previous context. It was then fine-tuned on some supervised tasks, such as translation and sentiment classification. The development of GPT-1 demonstrated the effectiveness of large-scale pre-training: this approach enables the model to acquire general linguistic representations, improving its ability to generalize to unseen tasks and allowing the use of much larger datasets than traditional supervised methods. Nevertheless, GPT-1 remains limited: it often produces repetitive text, struggles to maintain coherence over long contexts, and has difficulty capturing long-range dependencies.

2.6.3 GPT-2

In 2019, OpenAI released *GPT-2* [11, p. 25-26], a larger and more capable successor to GPT-1. The model was trained on WebText, a dataset consisting of approximately 8 million documents collected from Reddit, for a total of over 40 GB of text. The largest version of GPT-2 contains 1.5 billion parameters and a context size of 1024 tokens. This increase in scale, in terms of model size and training data, enabled GPT-2 to learn more coherent and contextually rich linguistic representations. Its most significant contribution was demonstrating that larger models achieve better performance, even on tasks they were never explicitly trained for. This behavior, known as zero-shot learning, refers to the model's ability to follow novel instructions and solve unseen tasks, purely based on its pre-training. OpenAI's analysis showed that zero-shot performance improves consistently as model size increases, highlighting large-scale pre-training as a highly effective strategy for building versatile and general-purpose language models.

2.6.4 GPT-3

GPT-3 [11, p. 27] represents a further step forward in terms of scale and performance. With 175 billion parameters, the model can understand and perform a wide range of tasks using only a few examples, or even starting from a simple instruction. Built on GPT-3, **InstructGPT** was developed as an improved version designed to produce responses that are more helpful and better aligned with human preferences. Its training relies on a pipeline that includes a supervised fine-tuning on human-written examples and a phase of reinforcement learning, based on a reward model trained on

human preference ranking. The resulting model was preferred by human evaluators over the base GPT-3 model, despite being significantly smaller (only 1.3 billion parameters).

2.6.5 GPT-4

GPT-4, and already *GPT-3.5* before it [11, p. 24], introduced models capable of handling much larger context windows, with recent variants supporting up to 128K tokens. They can also generate dialogue-optimized outputs, with sequences far longer than the earlier models, reaching up to 4096 token. These versions have demonstrated strong versatility across a wide range of tasks and domains, from mathematics to coding, law, and psychology. GPT-4 also introduced the ability to process visual inputs, enabling the model to interpret images in addition to text.

2.7 Prompt Engineering

A generative LLM produces an output based on the textual sequence provided as input, known as the prompt. The attention mechanism and the autoregressive generation of new tokens rely on the relationships encoded in the initial sequence. For this reason, output quality is closely tied to the prompt, which must not only specify the task but also guide the style, tone, structure and contextual framing of the response. *Prompt engineering* is an iterative process that may require testing multiple formulations before achieving the desired behavior. Several standard techniques exist to make prompts clearer and more effective, but it is also important to recognize that individual models often follow specific patterns shaped by the data and instructions encountered during training.

Zero-shot prompting A prompt is considered *zero-shot* [12, p. 13] when it provides only the description of the task to be performed, along with any necessary input data, without including examples or demonstrations. The structure of the sequence may vary, taking the form of a question, an instruction, or the beginning of a narrative. The term reflects the absence of examples: in this setting, the model has no explicit pattern to imitate, and the generation is less constrained, relying entirely on its pre-training knowledge and on the instructions specified in the prompt.

One-shot and Few-shot prompting A prompt is considered *one-shot* [12, pp. 15–17] when it includes a single example that guides the model toward the correct output. This approach is especially useful when the task requires a fixed structure or a specific output format. When multiple examples are provided, the technique becomes *few-shot* [11, pp. 15–17] prompting. The number of examples depends on several factors, such as task complexity and model capability. Each example should be accurate, well-structured and distinct from the others, ideally covering different scenarios and including edge cases when appropriate.

System prompting Defines [12, pp. 18–20] the overall context and high-level objective, describing how the model should behave. It may include additional requirements to follow as well as specifications for the desired output format. It also plays a crucial role in safety, as it can establish rules that limit the generation of toxic, sensitive or controversial content.

Role prompting *Role prompting* [12, pp. 21–23] adds contextual guidance by assigning the model a specific identity or character. The generated responses tend to align with the assigned role, adopting an appropriate tone of voice and prioritizing information in a manner consistent with that persona.

Step-back prompting *Step-back prompting* [12, pp. 25–28] artificially expands the prompt by first asking the model to answer a more general question and then incorporating that answer into the final reasoning process. This preliminary step helps the model retrieve relevant background knowledge and encourages a more accurate and structured reasoning chain.

Chain of Thought *Chain of Thought* (CoT) [12, pp. 29–31] is a prompting technique designed to enhance the reasoning capabilities of a model. It is applied by explicitly asking the model to reason "step by step", thereby producing intermediate reasoning steps that lead to the final solution. This approach yields more interpretable answers and makes it easier to identify potential errors or failure modes. The model's behavior can be further guided by providing reasoning examples, encouraging it to follow similar patterns. The main drawback is computational: generating longer sequences increases both inference time and memory usage.

Self-consistency and Tree of Thought CoT can be extended by combining sampling with majority voting. *Self-consistency* [12, pp. 32–35] explores multiple reasoning paths and then selects the most consistent final answer. Using high-temperature probabilistic sampling, the model generates several distinct chains of thought and the most frequent outcome is chosen as the final prediction. Similarly, *Tree of Thought* [12, pp. 36–37] enables the exploration of multiple reasoning paths organized in a tree structure. A scoring and selection mechanism applied at intermediate steps prunes less promising branches, allowing the model to continue only along the paths that appear most likely to yield correct solutions.

2.7.1 Best practises

In [12, pp. 54–65], several guidelines are proposed for crafting effective prompts, based on the prompt engineering techniques discussed earlier. One of the most important best practices is to provide a set of examples from which the model can infer patterns, style and tone. More generally, prompts should be written clearly and concisely, avoiding unnecessary information. It is preferable to use positive instructions, relying on verbs that explicitly describe the actions to be performed. Conversely, listing numerous constraints or specifying what the model should avoid may be counterproductive, as it can limit its expressive capacity and lead it to guess the intended behavior. Restrictions should therefore be used sparingly, mainly in cases where they are necessary to mitigate bias or filter sensitive content. The output format can be optimized by requesting a response of a specific length or by appropriately setting the max tokens parameter of the model. To reduce post-processing and more tightly control the model's behavior, it is also possible to enforce a structured format, such as JSON or XML.

2.7.2 Prompt chaining

Prompt Chaining [13] addresses the need for more controllable and transparent model behavior. Inspired conceptually by Chain of Thought, this technique decomposes a complex task into multiple steps, each handled through a separate inference. The output of one sub-task can be injected as additional context into the prompt of the next, forming a chain of transformations that enables the model to tackle problems it would not be able to solve in a single pass. Prompt chaining

also contributes to a richer human-AI experience: defining a sequence of steps and crafting dedicated prompts for each of them help users to better understand the problem, inspect intermediate results and maintain control over the reasoning process. Executing multiple generations in sequence often leads to more concise and accurate outputs. A single long generation is more prone to accumulating errors or repeating information. Several primitive operations can be applied within a chain:

- **Rewriting**: reformulates the input to make it clearer and more model-friendly;
- **Extraction**: concisely selects the most relevant information;
- **Split points**: breaks text into smaller units, such as lists;
- **Compose points**: merges multiple intermediate results into a single coherent answer;

Each sub-task can also be customized by adjusting the temperature: higher values encourage creativity, while lower or zero values are preferable for deterministic tasks such as classification or mathematical computation.

2.8 Agents

Just as humans rely on tools to extend their capabilities, generative models can be combined with external tools and documents to enhance their functionality. By querying databases or performing online searches, the models can access additional information, including real-time data. When properly configured, LLMs can even interact with external systems through dedicated APIs, allowing them to retrieve data, perform computations, or trigger external services. When a model is equipped with external tools and is capable of autonomously planning and executing a sequence of tasks, it is referred to as an *agent* [14]. At the core of any agent system lies a language model, which may be multimodal or fine-tuned according to the specific application requirements.

2.8.1 ReAct

Each agent maintains a session history that enables multi-turn inference. Through a cognitive architecture, information is processed iteratively, influencing subsequent

decisions and progressively refining the action plan. An orchestration layer guides the planning process, ensures coherence across steps, and keeps track of the most relevant information. The process can be supported by various prompt-engineering techniques, including *ReAct* [14, pp. 9–11], which combines reasoning and acting to decompose a problem into a structured sequence of steps. A typical ReAct-like workflow includes:

- **Question**: the request derived from the user input;
- **Thought**: an internal reasoning step used to better understand the request;
- **Action**: selection of a tool to be used, if needed;
- **Action input**: preparation of the input to be passed to the tool;
- **Observation**: acquisition of the tool's output;
- **Final answer**: composition of the final response for the user.

The Thought, Action, Action input, and Observation phases may be repeated multiple times until the task is completed.

2.8.2 Tools

Tools extend the capabilities of a model by providing access to external and up-to-date information, while also serving as a key mechanism for reducing hallucinations. They enable interaction with external systems: this not only improves the accuracy of the agent's output, but also allows it to affect the external environment through concrete actions.

Extensions *Extensions* [14, pp. 13–15] act as a bridge between the agent and third-party APIs. They provide a structured and resilient mechanism for defining the arguments and parameters required by each API and teach the model how to correctly invoke the available endpoints.

Functions Similarly to functions in traditional software, agents can rely on a set of callable *functions* [14, pp. 18–24] designed to perform specific tasks. Like extensions, they define arguments and parameters, but they are executed on the client side. This offers greater control, as their behavior and format can be fully customized, and no additional application layer is required to filter data or manage the security of external calls.

Data stores *Data stores* [14, pp. 27–31] supply additional knowledge to the model by injecting it directly into the prompt, avoiding the need for retraining or fine-tuning. Information is stored in a vector database, where documents are converted into embedding sequences. A common implementation is Retrieval Augmented Generation (RAG): given a user query, the model computes its embeddings and retrieves the most similar vectors from the database. The matched content is returned in textual form and incorporated into the final prompt as additional context.

2.9 Retrieval Augmented Generation

[15] Despite the strong generative capabilities of LLMs, including the ability to produce highly specific content, their use in real-world applications still presents several limitations. Beyond the impossibility of knowing all domain-specific details, the knowledge encoded in a model cannot be easily updated without retraining or fine-tuning, both of which are costly and slow processes. Moreover, LLMs are optimized to produce plausible and well-structured responses, but not necessarily correct ones; when they generate unsupported or fabricated content, this phenomenon is referred to as hallucination. *Retrieval Augmented Generation* (RAG) aims to improve the accuracy, reliability and verifiability of model outputs by integrating external data into the generation process. This allows the model to rely on up-to-date and contextually relevant information, reducing the likelihood of errors and enhancing the overall quality of the response.

2.9.1 Original approach

In its original formulation [16], the RAG model combined two types of memory: a parametric memory, representing the knowledge encoded in the model's weights

during training, and a non-parametric memory, consisting of information retrieved from external sources. This second memory relies on a collection of easily inspectable documents, ensuring a high degree of transparency throughout the generation process. The initial architecture used a pre-trained seq2seq model for the parametric component, while the non-parametric component relied on a neural retriever combined with a dense vector index. Conditioned on the input sequence, the retriever selects a set of relevant documents from the index. These retrieved passages are then used, together with the original input, to condition the subsequent text generation. Document retrieval follows a **top-K approximation strategy**, in which a subset of K documents with the highest embedding similarity is selected.

Retriever The retriever used in the original RAG model is based on a *bi-encoder architecture*, where queries and documents are encoded into embedding vectors using two separate BERT encoders. If a document is relevant to the query, the two vectors should exhibit a high inner product, computed as the sum of element-wise products. This mechanism enables the selection of the most relevant documents for the input.

Generator The generator relies on a pre-trained encoder-decoder model (BART-large in the original paper). The user input and the retrieved documents are combined by simple concatenation, providing the generator with an enriched context. The retrieved documents are treated as latent variables, which are used to marginalize the probability of the output sequence. The paper introduces two different methods for computing the token probability distribution, depending on how each retrieved document contributes during inference.

RAG-Sequence Model In the RAG-Sequence Model, the generator produces a probability distribution over the entire output sequence using one retrieved document at a time. Therefore, each candidate answer is conditioned on a single latent document, and the final output distribution is obtained by combining the probabilities computed for all retrieved documents.

$$p_{RAG-Sequence}(y|x) \sum_{ztop-k(p(\mathring{u}|x))} p_\eta(z|x) \prod_i^N p_\theta(y_i|x, z, y_{1:i-1}) \tag{2.9}$$

Here, y is the target sequence, x the input sequence, z the retrieved document, p_η the retriever distribution, and p_θ the generator distribution.

RAG-Token Model In the RAG-Token Model, the generator may rely on a different latent document for each token. This allows the model to draw from multiple sources during generation, improving coverage and factual accuracy. For every token and every document, a probability distribution is computed and then marginalized; the process is repeated through the entire sequence.

$$p_{RAG-Token}(y|x) \prod_{i}^{N} \sum_{ztop-k(p(\mathring{u}|x))} p_\eta(z|x) p_\theta(y_i|x, z, y_{1:i-1}) \tag{2.10}$$

2.9.2 Modern approaches

Improvements in language models, particularly their ability to handle increasingly large context windows, combined with the strengths of in-context learning, have accelerated the development of RAG techniques. Advances in semantic similarity evaluation between documents and queries, together with modern prompt-engineering strategies, now make it possible to construct highly informative and context-rich prompts. Depending on the complexity of the task, RAG systems can be implemented following different paradigms [17, pp. 3–5], including Naive RAG, Advanced RAG and Modular RAG, each offering a distinct level of control, orchestration, and integration with external systems.

Naive RAG *Naive RAG* is the most straightforward approach and consists of three main stages: indexing, retrieval, and generation.

- **Indexing**: after cleaning and extracting content from various formats (PDF, Word, HTML, ...), the data is converted into plain text. Each document may be split into controlled-size chunks, which are then encoded into embeddings and stored in a vector database.
- **Retrieval**: user input is transformed into the same embedding space. Using a similarity score, the system identifies the most relevant chunks in the database. Retrieved information is ranked by relevance, and the top-K fragments are selected for prompt construction.

- **Generation**: the original query and the retrieved documents are synthesized into a prompt, which is then provided to the model responsible for generating the final answer.

Despite its simplicity, the Naive approach may still suffer from hallucinations, bias or irrelevant outputs. This may depend on suboptimal retrieval, where irrelevant chunks are selected, or from redundancy in the retrieved information, which may lead to repetitive or unfocused responses.

Advanced RAG *Advanced RAG* strategies aim to improve output quality by applying both pre-retrieval and post-retrieval operations. Indexing techniques can be enhanced through mechanisms such as sliding windows, fine-grained segmentation and the inclusion of metadata to guide retrieval more effectively. The query itself can be refined using **query rewriting** and **query expansion**, making it clearer and more informative, and increasing the likelihood of retrieving highly relevant content. Once retrieved, the documents can undergo the following:

- **Compression**: the content may be summarized while preserving only the essential information, reducing prompt length and improving context quality;
- **Re-ranking:** the retrieved documents can be reorganized by importance, placing the most relevant information in the most influential positions of the prompt.

These techniques lead to more accurate responses, reduce redundancy and mitigate hallucinations compared to the basic Naive approach.

Modular RAG *Modular RAG* introduces additional components that extend and specialize the system, enabling more flexible and controlled retrieval-and-generation pipelines. Among the most relevant modules:

- **Search Module**: enhances the retrieval process by adapting the search strategy to the specific scenario. It may rely on dynamically generated code or structured query languages to obtain more relevant results than simple semantic matching.

- **Fusion Module**: expands the user query through a multi-query strategy, generating several variants of the same request to explore the corpus from different perspectives. This increases information coverage and reduces the risk of retrieving overly similar or redundant documents.
- **Predict Module**: aims to reduce noise and redundancy by generating context in a controlled manner. It can summarize, filter or reformulate retrieved documents, producing a more compact and focused version of the material to be injected into the prompt.

2.9.3 Indexing Module

The *indexing module* [18] is one of the key components shared by all RAG architectures. It is responsible for organizing the corpus that contains external resources and information. Typically, documents are divided into chunks and then encoded into more compact representations, for example using sparse retrieval methods such as **BM25**, or dense retrievers based on encoder models like **SBERT**.

Data sources Text sources can vary widely and may exhibit different levels of structure [17, pp. 5–7].

- **Unstructured data**: the most common and consist of plain, unorganized text.
- **Semi-structured data**: text combined with structured elements, such as tables or metadata.
- **Structured data**: more complex and highly organized formats, for example databases or graphs.
- **LLM-generated data**: enriched corpus with synthetic data, produced by language models, which are useful for increasing coverage or improving dataset quality.

Granularity The size of the chunks plays a crucial role in determining the amount and quality of information retrieved from the corpus. A **coarse-grained retrieval strategy** [17, p. 7] uses larger units, which increases the likelihood of

obtaining relevant content, but also introduces redundancy and noise that may reduce model effectiveness. In contrast, a **fine-grained granularity** [17, p. 7] reduces noise by extracting only the core information, but it may compromise semantic integrity when the content is divided into segments that are too small. Chunk units can vary in length, ranging from sentences or short paragraphs to entire documents. A common approach is to define a fixed size in terms of tokens, such as 100, 256 or 512.

Metadata Each chunk can be enriched with additional information [17, p. 8], such as the page number from which it comes, the file name, the author or a timestamp. These metadata enable more efficient filtering of the retrieved content and improve system transparency by making it easier to trace and cite the origin of the data. Metadata can also be generated *artificially*, for example by adding summaries to paragraphs or creating hypothetical questions that the content can answer, with the goal of improving retrieval quality.

2.9.4 Retrieval Module

The second fundamental component of a RAG system is the retrieval module [18]. It acts as a bridge between the user query and the external database, with the goal of retrieving the most relevant documents for the task.

Query analysis The first step is to optimize the form of the query [18] [17, pp. 8–9]. The initial input may be ambiguous or incomplete, so it is often useful to derive a clearer and more informative query, especially for complex tasks.

- **Query rewriting**: rewrites the original input to produce a more complete and disambiguated query.
- **Query expansion**: enriches the query by generating multiple variants that cover different nuances of meaning.
- **Query decomposition**: breaks down a complex question into several simpler and more focused sub-questions.
- **Answer inferring (HyDE)**: generates a hypothetical answer that serves as a pseudo-document to guide retrieval toward semantically similar content.

Retrieval and re-rank The retrieval phase is closely connected to the indexing mechanism. The query is processed in a way that is consistent with the document encoding, allowing their representations to be compared through similarity measures. The *re-ranking* step then orders the retrieved documents according to their relevance to the task, ultimately selecting a subset of the top-k results. Additional summarization and filtering operations can be applied to reduce irrelevant or redundant content.

The retrieval process can also be performed in multiple steps [17, pp. 10–11]:

- **Iterative Retrieval**: the knowledge base is queried repeatedly using both the initial query and the text generated by the model. This enriches the context with additional information, although it may also introduce content that drifts away from the original scope.
- **Recursive Retrieval**: this approach deepens the search results by using documents retrieved in previous steps as new queries.
- **Adaptive Retrieval**: this method enables the LLM to decide dynamically whether to further explore a given piece of content and to assess the relevance and reliability of retrieved documents.

2.10 RAG applications in technical contexts

RAG-based technologies can be particularly effective in highly technical domains, where information is difficult to access and spread across large volumes of documentation. A notable example is *Ask-EDA* [19], an agent designed to support electronic design engineers. This system aims to assist practitioners, especially newly hired engineers with limited experience, by helping them search for and understand the documents required for their tasks.

2.10.1 Hybrid RAG

Ask-EDA adopts a **hybrid** indexing and retrieval approach that combines both *sparse* and *dense* representations. The data sources are heterogeneous and include manuals, tool documentation, Slack conversations and other technical material, for

a total of about 400 MB. The documents are split into chunks of 2048 tokens with an overlap of 256.

For the sparse component, BM25 is used to perform lexical retrieval based on direct term matching between the query and the documents, which is particularly useful for identifying specific technical terminology. For the dense component, a sentence transformer (*all-MiniLM-L6-v2*) encodes the text into dense vectors that capture its semantic content. These vectors are stored in *ChromaDB*, a vector database.

During retrieval, **cosine similarity** is computed between the dense vector of the query and the vectors of the documents, selecting the top-n results. In parallel, the best matches from the BM25 index are also retrieved. The two result sets are then combined using **reciprocal rank fusion** (RRF), which takes into account the ranking position of each document in both lists.

$$RRF_{score}(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)} \tag{2.11}$$

Here, D is the set of the top n_{dense} and n_{sparse} results, d denotes a candidate document, $r(d)$ its rank in each list, R the set of ranking methods, and k a balancing constant empirically set to 60.

2.10.2 Abbreviation De-hallucination

Once the most relevant documents have been selected, a prompt is constructed by combining the query and the retrieved context before generating the output. The *ADH* module identifies the most important technical abbreviations by searching for them in a dictionary through exact matching and enriches the context by adding the full name and a short description. To improve clarity, this information is inserted into the prompt using the format: "abbr is usually short for name, which is desc".

2.10.3 Possible improvements

Document retrieval within the knowledge base can also be performed using another type of representation: sparse encoder retriever models [20]. This approach lies between traditional sparse methods and fully dense models, as it leverages both

exact token matches and the activation of related terms and synonyms.

Indexing can also be organized separately across the different fields that compose the documents:

- **Cross-Fields**: the search spans multiple fields simultaneously, useful when information is distributed.
- **Most Fields**: multiple fields are used and their scores are combined.
- **Best Fields**: selects the field with the strongest match.
- **Phrase Prefix**: prioritizes phrases over individual keywords.

The choice of representation affects both retrieval accuracy and resource usage. Dense vectors, although lower-dimensional than sparse representations, occupy more disk space because every dimension contains an active value. Sparse representations, instead, can be optimized by storing only the non-zero values.

Chapter 3

Case study

3.1 Advanced Driver Assistance System (ADAS)

In the automotive domain, the human driver represents both the key element and the weakest link in driving activities. Most accidents are caused by human error, often due to inattention or failure to follow traffic rules.

Modern vehicles are composed of multiple electronic subsystems that allow access to a wide range of information and enable various functionalities to improve driving safety. *Advanced Driver Assistance Systems* (ADAS) refer to a set of technologies designed to monitor parameters related to the vehicle and its surrounding environment, with the goal of detecting potentially dangerous situations before it becomes too late to react.

Safety systems can be classified as *passive* or *active*. Passive systems aim to reduce the consequences for passengers when an accident occurs, such as seat belts and airbags. Active systems, instead, help maintain vehicle control and intervene in advance to prevent accidents.

3.1.1 Sensors

ADAS functionalities are based on several types of sensors [21], each with specific capabilities and characteristics. Examples include:

- **RaDAR (Radio Detection And Ranging)**: uses radio waves to measure the distance and speed of obstacles. It can operate in short-, mid- or long-range

configurations with different field-of-view angles.

- **LiDAR (Light Detection And Ranging)**: similar to radar, it uses a laser beam reflected by surrounding objects.

- **US (Ultrasonic Sensor)**: relies on the reflection of ultrasonic waves to detect nearby objects, mainly during low-speed scenarios.

- **Optical Camera**: includes mono and stereo cameras used to map the environment. They cannot directly estimate object speed and are sensitive to weather and lighting conditions. They are commonly used together with RaDAR sensors.

3.1.2 Functionalities

The automated support a vehicle can provide to the driver includes a wide range of possible features [21]. Depending on the degree of automation, different levels of assistance can be defined, from partial control to near-complete system management. Through data collection and the use of advanced software, various ADAS functionalities can be implemented. Among the most common:

- **Forward Collision Warning (FCW)**: alerts the driver of an imminent collision through visual or audible signals, helping to avoid or mitigate the impact.

- **Automatic Emergency Brake (AEB)**: automatically applies emergency braking without requiring driver intervention. Sensors continuously monitor the area in front of the vehicle to detect dangerous situations.

- **Adaptive Cruise Control (ACC)**: automatically adjusts speed and maintains a safe distance from the vehicle ahead, without the driver needing to use the accelerator or brake.

- **Lane Keeping System (LKS) and Lane Change Assistant (LCA)**: use optical recognition to detect lane markings and prevent the vehicle from unintentionally leaving its lane.

3.1.3 Testing

As in any development process, the *testing phase* is essential to ensure the correct functioning of ADAS features. Modern systems integrate complex digital components, making it necessary to test not only individual modules but also the entire infrastructure. Experiments can be carried out both in the field and in virtual environments, reducing time and costs. *Computer-based simulation* and modelling methods significantly optimize the validation process.

Hardware-in-the-loop *Hardware-in-the-loop* (HiL) [22] is a simulation technique that allows real hardware components to be tested within a controlled environment. This approach makes it possible to verify the combined behaviour of hardware and software without exposing them to real-world scenarios. By moving the process into the laboratory, any condition, even extreme ones, can be reproduced safely, avoiding risks for operators and equipment and ensuring experiment reproducibility. Moreover, simulations can accelerate processes that would normally require long execution times by adjusting time-related variables.

3.2 Vehicle Architecture

In the automotive domain, the term architecture, or platform, refers to the fundamental structure of a vehicle, which includes its main components. In addition to mechanical elements, the architecture comprises an electronic infrastructure made up of **multiple control units** (**ECUs**) that communicate with each other. The number of ECUs can vary significantly, and signals are managed through different communication protocols.

3.2.1 Protocols

Depending on the requirements, performance needs, and criticality of the signal, different protocols may be used. Among the most common:

- **CAN** [23]: an ISO standard protocol (ISO 11898) for serial communication. It allows each node to process only the relevant messages and includes mechanisms for error handling and retransmission.

- **LIN** [24]: a more economical and simpler alternative, used for non–time-critical and moderately fault-tolerant communications.
- **FlexRay** [25]: a deterministic and fault-tolerant protocol that supports very high communication speeds. It is used in advanced applications, including ADAS.
- **Automotive Ethernet** [26]: enables high-speed communication and the reuse of IP-based software technologies.

3.2.2 Porsche Architectures

The vehicle architectures used in Porsche and relevant to this thesis are MLBevo and E³ 1.2.

- **MLBevo** [27]: introduced in 2015 as an evolution of the MLB platform, it is a highly adaptable structure that can support hybrid or fully electric configurations. The platform relies on the FlexRay communication.
- **E³ 1.2** [28]: based on a core of five high-performance computers that centrally manage the main functionalities of the vehicle. The architecture adopts Automotive Ethernet as its primary communication backbone.

3.3 Company objective

The primary research objective of this thesis is to address the practical operational requirements of NTC. During the testing phase of the ADAS functionalities developed by the company, it is necessary to define requirements and expected behaviors in a structured technical format. The current workflow is predominantly manual, requiring the transformation of natural language specifications into **JSON** files. This process represents a major operational bottleneck as it is poorly scalable and demands significant time investments from highly specialized personnel. The task involves complex logical operations and extensive pre-processing to ensure alignment with the data conventions mandated by Porsche. Furthermore, it requires the ability to comprehend and translate a vast body of technical documentation provided in German, adding a layer of linguistic and domain-specific complexity.

3.3.1 Key values

The goal is to develop a scalable system capable of *optimizing* the production of **test files**, starting from a *high-level natural language description* and a set of *technical documents*. Several quality standards and constraints must be ensured, which can be summarized in four key points:

- **Confidentiality**: documents, parameters, technical information about ECUs and signals, as well as the test files themselves, are strictly confidential and represent a valuable company asset. They cannot be disclosed and all processing steps must be executed locally in a secure environment.
- **Standardization**: the generated outputs must follow the formal rules used to standardize the representation of elements such as aliases, keywords and measurement units.
- **Extensibility**: the code must be sufficiently scalable and as independent as possible from the specific output format. The workflow should therefore be divided into modular and independent sub-tasks.
- **Architecture transparency**: the final test files must be usable regardless of the underlying architecture (E^3 or MLBevo). The process must therefore be generalized and unified to accommodate architectural differences.

3.3.2 Tasks descriptions

The entire project can be divided into three main macro-tasks:

- **Signal mapping**: the state of the vehicle and its surrounding environment is represented by the values of the signals emitted by the ECUs. Although the underlying architectures may differ, it is possible to identify sets of signals that convey the same information. A 1:1 mapping between MLBevo and E^3 signals enables the definition of tests using the same operators (and therefore it simplifies the test definition and reading). Starting from the respective documentation, all possible correspondences should be identified. This also helps users identify the correct signals more easily, thanks to the use of a clearer and more intuitive alias.

- **Alias generation**: after mapping the signal pairs, a representative alias must be defined for each of them, following the standardized naming rules and abbreviations used within the company.
- **JSON generation**: based on the textual description of the test which includes, among other things, preconditions, actions and expected results, a JSON file must be produced. It is required to contain signals, parameters and logical operations used to monitor the vehicle state during experiments and verify its behaviour.

Chapter 4

Task 1: Signal Mapping

4.1 Background

Writing a test file that remains transparent with respect to the underlying vehicle architecture requires the use of signals that are compatible across both MLBevo and E3. The first objective is therefore to identify pairs of signals that represents the same information, enabling the construction of a unified abstraction layer. This task is particularly challenging due to the limited and *heterogeneous* documentation available: two Excel files, each containing a long list of signals accompanied by a set of technical attributes. The material is written almost entirely in German and includes between *10,000* and *20,000* signals per architecture.

Further complicating the process, the number of signals differs significantly between the two platforms, with a discrepancy of several thousand entries. This makes a complete 1:1 mapping almost impossible and requires a more nuanced approach based on semantic similarity, exact correspondence and contextual interpretation. In practice, the goal is to reconstruct a shared "vocabulary" between two systems, with different design and naming conventions that are not always intuitive. Establishing such a correspondence is essential for enabling architecture-independent test generation.

4.1.1 Objectives

The signal-mapping task can be decomposed into a sequence of subtasks, each addressing a specific aspect of the overall workflow. These subtasks, which will be

detailed in the following sections, form the foundation of the entire pipeline and ensure that the final mapping is both reliable and computationally manageable.

- **Pre-processing**: the first step consists of restructuring the available data and converting it into a format suitable for subsequent processing stages. During this phase, the relevant fields are selected, noisy or redundant information is removed and a series of transformations is applied to normalize the data across the two architectures. This ensures that signals become comparable despite differences in naming conventions, formatting, or documentation quality.
- **Multi-Perspective Representation: Lexical, Sparse and Dense Models**: each signal must be encoded in a way that captures its semantic and functional meaning, enabling an efficient and scalable similarity comparison. To achieve this, multiple representation strategies are employed, including sparse and dense encodings such as **BM25**, **SPLADE** and **SBERT**. These methods convert each signal into a *multidimensional vector*, allowing mathematical comparison through similarity metrics.
- **Testing**: a preliminary evaluation is performed on a set of *mockup data* to verify the functional correctness of the pipeline in a controlled scenario. Once the workflow is validated, the same models are applied to a real ground-truth to assess the practical effectiveness of the proposed solutions. This two-step testing strategy ensures both conceptual and empirical reliability, providing a clear indication of how well the system generalizes to real-world documentation.

4.2 Proposed solution

The mapping procedure is structured into several core subtasks, each responsible for a specific phase of the pipeline. This modular organization ensures clarity, reproducibility and the ability to refine or replace individual components without affecting the entire system.

4.2.1 Pre-processing

The difficulty of gathering sufficient and high-quality data is a common challenge in industrial environments, where documentation is often fragmented, inconsistent

or not designed for automated processing. For this reason, a preliminary analysis is required to identify the relevant information and to determine which fields can be discarded due to redundancy or lack of significance.

Each signal is described by approximately 40–50 fields, although the exact structure varies between the two architectures. A substantial portion of these fields is empty, incomplete or irrelevant for the purpose of signal mapping. This makes a preprocessing phase indispensable, involving data cleaning, normalization of formats, translation of key attributes and selection of the features that carry meaningful semantic or functional information. Only after this preparation step is it possible to perform a systematic comparative analysis, which forms the foundation for the subsequent stages of the test generation pipeline.

Reading and Normalization The signal data were first loaded from the source files using the functions provided by the *pandas library*. Then, a preliminary cleaning and validation step was performed, which included removing unnecessary tabulation and whitespace characters (such as newline, carriage return, and tab) and replacing all *NaN values* (unsupported by the JSON format) with empty strings. This normalization ensured a more consistent dataset and facilitated subsequent serialization.

Fields selection The most informative fields, shared across both architectures, were then selected. Specifically, the following attributes were retained: "PDU", "SIGNAL", "BITS", "TRANSMISSION MODE", "INIT", "ERROR", "MIN", "MAX", "UNIT", "OFFSET", "SCALE", "RAW VALUE", "DESCRIPTION" and "COMMENT". These fields exhibit heterogeneous data types, including strings, integers and floating-point numbers with one or two decimal digits. The description and comment fields contain the most detailed and narrative information, written mainly in German with some exceptions in English. The "PDU" and "SIGNAL" fields consist of combinations of technical abbreviations. The "UNIT" field generally follows a standard naming convention, although some inconsistencies appear in signals represented by pure numerical values, which may be associated either with an empty string or with the label “Unit_None”. Finally, the "PHY VALUE" and "RAW VALUE" fields may contain, respectively, a range of values (expressed through minimum and maximum bounds) and a sequence of discrete values.

Serialization After selecting and normalizing the relevant information, two dictionaries were created, one for each architecture. Each dictionary contains a unique key for every signal and a corresponding value represented by a nested dictionary that includes all selected parameters. To ensure key uniqueness, a composite key was defined: for MLBevo, the fields "PDU" and "SIGNAL" were used, while for E3 an additional field "FRAME" was included, since some signals share the same combination of the first two attributes. Once these structures were built, the data were serialized into two separate JSON files, one for each architecture, enabling fast access and consistent handling in the subsequent stages of the pipeline.

Translation The next preprocessing step involved data augmentation through machine translation. Specifically, the "DESCRIPTION" and "COMMENT" fields, which contain the most narrative and semantically rich information, were translated from German into English. To ensure data confidentiality, the entire procedure was carried out locally, without relying on external services. To achieve good translation quality and maintain scalability over the large dataset, a dedicated module was implemented using the Transformer-based model *Helsinki-NLP/opus-mt-deu-en*. The execution was optimized by processing batches of eight text sequences at a time, leveraging the computational acceleration provided by an NVIDIA GTX 1050Ti GPU (4GB GDDR5). Due to the presence of some very long strings and the limited context window of the model (512 tokens), an automatic chunking mechanism was introduced. This ensured that each sequence could be split into smaller segments when necessary, preventing truncation and preserving translation completeness.

MarianMT

The *Helsinki-NLP/opus-mt-deu-en* model [29] is a neural machine translation system based on a Transformer encoder–decoder architecture, with a disk footprint of approximately 298 MB. It belongs to the OPUS-MT family, a large collection of NMT models optimized for specific source–target language pairs. These models rely on **MarianMT** [30], an implementation built using the Marian framework, a machine translation toolkit written entirely in C++ and widely adopted in several European research projects. The efficiency of

the framework is supported by its minimal dependency ecosystem and by native CUDA integration, which enables effective GPU acceleration. Although Marian also supports hybrid configurations, such as RNN-based encoders, the version used in this work fully adopts the Transformer architecture, providing improved stability and translation quality for technical documentation.

Rescaling and final conversion An additional normalization step involved standardizing the "INIT", "MAX", "MIN" and "RAW VALUE" parameters by applying rescaling and offset transformations based on their corresponding "SCALE" and "OFFSET" values. After these operations were applied, the "SCALE" and "OFFSET" fields were removed, as they were no longer useful. This procedure produced values expressed within a consistent numerical space, making comparisons between different signals more straightforward. Since these fields originally exhibited different levels of numerical precision, a uniform representation with one decimal digit was adopted to reduce variability caused by heterogeneous formats. Finally, because the models and algorithms used in the subsequent stages operate on textual inputs, all numerical fields were converted into strings.

4.2.2 Multi-Perspective Representation: Lexical, Sparse, and Dense Models

The textual representation of the signals can be obtained through different retrieval paradigms, each capturing linguistic information in a distinct way. The approaches considered in this work, BM25, SBERT and SPLADE, can be regarded complementary, as they allow text sequences to be compared from multiple perspectives.

BM25 The first approach relies on a lexical retrieval method based on the direct matching between the words in the query (in this case, a signal to be paired) and those contained in a pre-computed corpus, consisting of all signals from the other architecture. For this purpose, the *BM25Okapi* module from the *rank_bm25* library was used.

```json
{
  "SPD_01_ETH": {
    "PDU": "VEH_STATUS_ETH",
    "SIGNAL": "VehicleSpeed_ETH",
    "BITS": 32,
    "TRASMISSION_MODE": "CYCLIC",
    "INIT": 0,
    "ERROR": "",
    "MIN": 0,
    "MAX": 10000,
    "UNIT": "km/h",
    "OFFSET": 0,
    "SCALE": 0.01,
    "RAW_VALUE": "",
    "DESCRIPTION": "Fahrzeuggeschwindigkeit über Ethernet-Gateway
    ↪ erfasst.",
    "COMMENT": "Signalversion für Ethernet-Architektur."
  }
```

```json
{
  "SPD_01_ETH": {
    "PDU": "VEH_STATUS_ETH",
    "SIGNAL": "VehicleSpeed_ETH",
    "BITS": "32",
    "TRASMISSION_MODE": "CYCLIC",
    "INIT": "0.0",
    "ERROR": "",
    "MIN": "0.0",
    "MAX": "100.0",
    "UNIT": "km/h",
    "RAW_VALUE": "",
    "DESCRIPTION": "Fahrzeuggeschwindigkeit über Ethernet-Gateway
    ↪ erfasst.",
    "COMMENT": "Signalversion für Ethernet-Architektur.",
    "ENG_DESCRIPTION": "Vehicle speed acquired via Ethernet gateway.",
    "ENG_COMMENT": "Signal version for Ethernet architecture."
  }
```

Figure 4.1: Mockup signal (synthetically generated) before and after full preprocessing.

BM25 implementation details

Best Match 25 [31], commonly known as BM25, is an algorithm widely used in information retrieval. It integrates **indexing**, **ranking** and **relevance scoring** mechanisms to efficiently store information from a corpus and compute similarity scores with respect to a query. BM25 extends the traditional TF-IDF approach by introducing additional factors, such as *document-length normalization* and a *saturation function* for term frequency. The *rank_bm25* library provides several variants of the algorithm, including ATIRE, BM25L, and BM25+. All of them compute a score for each term in the query by comparing it with the content of each document in the corpus. The final score is obtained by summing the contributions of individual terms, which are proportional to their frequency in the document and inversely proportional to their overall frequency in the corpus. Formally [32]:

$$BM25score_q = \sum_{t \in q} IDF_t * \frac{(k_1 + 1) * tf_{td}}{k_1 * (1 - b + b * \frac{L_d}{L_{avg}}) + tf_{td}} \tag{4.1}$$

where the score for a query q is computed as the sum of the contributions of its terms t. N denotes the total number of documents in the corpus, df_t the number of documents containing term t, and tf_{td} the frequency of term t in document d. The IDF used in the base BM25 implementation [33] is defined as:

$$IDF_t = log \frac{N - df_t + 0.5}{df_t + 0.5} \tag{4.2}$$

During the pre-computation phase, the signals from the two architectures were divided into two groups. The first group, corresponding to MLBevo, was treated as a fixed corpus on which term frequencies were computed and the retrieval index was built. The second group, containing the E^3 signals, was processed one element at a time with the goal of identifying, for each signal, the most similar document within the corpus. Both the corpus documents and the query strings were tokenized using an explicit procedure, which splits sequences composed by alphanumeric characters and dots. To obtain a more accurate analysis, separate corpora were created for each signal field. This allows each value to be compared with a coherent

set of values, ensuring that term rarity, and therefore its importance, is evaluated consistently within the specific context of each field.

SBERT The second approach relies on SBERT models, which provide compact and dense representations of the semantic information contained in text sequences.

SBERT: model details

Sentence-BERT (SBERT) introduces a family of models widely used in information retrieval and semantic textual similarity tasks. SBERT was developed as an extension of the encoder-only BERT architecture, with the goal of making semantic similarity computation efficient and scalable. Earlier models, although capable of performing such tasks, required a very high computational cost: computing similarity across all pairs in a set of 10k sentences could take dozens of hours, since each pair required a separate inference. The key improvement introduced by SBERT is the separation of the process into two stages: an initial **encoding phase**, performed once for each sentence, followed by a much faster **vector-based comparison phase**. This design drastically reduces computation time, allowing the same task to be completed in a matter of seconds. SBERT models generate dense semantic embeddings for each sentence, enabling similarity computation through standard operations such as cosine similarity. All vectors share the same dimensionality and are obtained through pooling operations applied to the token-level embeddings produced by a pre-trained BERT-style model and subsequently fine-tuned. The most common pooling strategy is **mean pooling**, which averages the embeddings of all tokens in the sentence. The original fine-tuning process relies on a triplet objective function, which trains the model to produce similar representations for semantically related pairs (*query–positive*) and distant representations for unrelated pairs (*query–negative*). This results in a well-structured embedding space suitable for semantic retrieval tasks.

As in the previous method, the signals were divided into two sets: one used as the query set and the other as the document corpus. Two Sentence Transformer

models were evaluated to generate dense embeddings of the textual content: *multi-qa-mpnet-base-dot-v1* and *all-MiniLM-L6-v2*. Both models produced comparable results in this setting, despite differences in size and complexity. Using SBERT enables the extraction of deeper **semantic relationships** than purely lexical methods, thereby improving the overall quality of the retrieval process. Similarly to the BM25 setup, a separate corpus was created for each signal field. The splitting of the documents and their analysis for each value individually enabled a more fine-grained comparison, preventing the high heterogeneity of the information from introducing noise or misleading the model. This approach allows SBERT to operate on more homogeneous text sets, improving the quality of the semantic representations and the consistency of the retrieval results.

SPLADE The third strategy occupies an intermediate position between BM25 and SBERT. It produces a sparse representation for both queries and documents, enabling the computation of similarity scores that account not only for exact **lexical matches** but also for the presence of **semantically related terms**. This approach therefore combines the interpretability and precision of lexical matching with the semantic generalization capabilities typical of dense models.

SPLADE model details

The **SPLADE** model [34] was introduced as an extension and improvement of the mechanism proposed in *SparTerm*, which estimates the "importance" of each vocabulary term with respect to the tokens appearing in a query or document. The process begins with tokenization and the corresponding transformation of the input sequence into BERT embeddings. Building on the original SparTerm formulation, SPLADE proposes a revised computation of importance weights designed to prevent the dominance of a small number of terms and to enforce a desirable level of sparsity. The resulting representation is a high-dimensional vector, matching the size of the model's vocabulary, where each component reflects the activation of the corresponding term based on its semantic relevance to the input text. In essence, SPLADE produces an expanded representation of the sequence, capturing not only the words explicitly present but also semantically related terms, such as **synonyms** or conceptually **similar expressions**.

The importance weight of an input embedding h_i with respect to a vocabulary embedding E_j is computed as:

$$w_{ij} = \text{transform}(h_i)^T E_j + b_j$$

where b_j is a bias term and the transformation consists of a linear layer followed by a GeLU activation and normalization.
The final representation of the text is obtained by aggregating the importance weights across all tokens:

$$w_j = \sum_{i \in t} \log(1 + \text{ReLU}(w_{ij}))$$

where w_j denotes the final weight associated with term j, t is the sequence of input tokens, and w_{ij} represents the contribution of token i to vocabulary term j.
Over time, several improvements have been introduced, leading to SPLADE-v2 [35] and SPLADE-v3 [36]. Among these enhancements:

- the replacement of the summation with a max-pooling operation in the aggregation step;
- the removal of query expansion during inference, thus reducing computational cost by computing expanded representations only for corpus documents;
- a re-training of the base model with an increased number of negative examples.

The model used in this work is *naver/splade-v3*, provided through the *SparseEncoder* library. The sparse vectors for the MLBevo signal data were pre-computed in advance, following the same strategy adopted for the other models: the information associated with each field was separated into distinct corpora, allowing the encoder to generate field-specific sparse representations.

Data storage The indices computed with BM25 and the representations generated with SBERT and SPLADE were stored on disk in order to drastically reduce the computational overhead during inference. This design choice makes it possible to fully exploit the strengths of all three techniques, each of which benefits from delegating the most time-intensive operations to an initial pre-computation phase. By separating the expensive encoding step from the online retrieval stage, the system can respond quickly while still relying on rich and expressive representations.

The SPLADE and SBERT vectors were serialized using the functionalities provided by the *torch* library, whereas the BM25 indices were stored using *pickle*. The resulting file sizes differ substantially across the three strategies: the BM25 indices occupy less than 5 MB in total, while the SBERT and SPLADE representations require approximately 400 MB and 100 MB respectively. This discrepancy is fully expected. BM25 stores only statistical information about the terms appearing in the corpora, leading to compact data structures, whereas SBERT and SPLADE produce high-dimensional vectors for each document, resulting in significantly larger storage requirements.

Overall, this pre-computation and serialization pipeline enables efficient retrieval while preserving the expressive power of sparse and dense models, ensuring that the system remains scalable even when applied to large collections of domain-specific documents.

4.3 Experimental results

The experimental phase dedicated to the signal–mapping task aimed not only to evaluate the effectiveness of the three individual strategies (BM25, SBERT and SPLADE), but also to explore the potential benefits of combining two or more of them. Since separate corpora were defined for each descriptive field of the signals, it becomes possible to assign **different relevance weights** to each field independently. This flexibility extends naturally to the ensemble setting, where the contributions of different retrieval approaches can be weighted according to their relative importance or expected reliability. Some weighting choices were defined a priori, based on intuitive considerations about the nature of the fields and the characteristics of the models. However, the experimental campaign also included the exploration of more arbitrary or randomized configurations, with the goal of

assessing the robustness of the ensemble and identifying potential synergies between lexical, sparse and dense representations. This dual approach, combining principled design with empirical exploration, allowed for a more comprehensive understanding of how different retrieval paradigms interact within the signal–mapping task.

4.3.1 Mockup validation

The initial step of the experimental phase was carried out on two small artificially generated datasets, each consisting of five pairs of signals. The goal was to verify the formal correctness of the implementation and to ensure that the task was solvable in a simplified setting. Both datasets were created with the assistance of *ChatGPT*, which was provided only with the names of the fields to be filled and a brief description of the general context (automotive domain). On the first dataset, all three retrieval strategies (BM25, SPLADE, SBERT) achieved 100% accuracy. For the second dataset, GPT was instructed to generate more ambiguous examples in order to increase task difficulty. In this case, BM25 and SBERT maintained 100% accuracy, while SPLADE reached 80%. In all experiments, a *min-max normalization* was applied to the document scores with respect to a fixed parameter. When a *softmax-based normalization* was used, SPLADE also achieved 100% accuracy on the second dataset, due to the stronger separation between scores. In this preliminary phase, all fields were assigned equal weight, allowing the evaluation of the models' behaviour without introducing additional balancing factors.

4.3.2 Setup

Test set The matching of signals between the MLBevo and E3 architectures represents a current challenge within NTC. Despite the large number of signals (over 10,000 for each architecture) the available ground truth is extremely limited, consisting of only **20 known pairs**. While this amount is sufficient to assess whether the proposed solution moves in the right direction, it is not enough to perform a comprehensive evaluation that accounts for a wide range of scenarios. Given the strong numerical imbalance between the two architectures, on the order of several thousand signals, the pre-computation of the corpora was performed on the smaller set. The goal of the experimental phase is therefore to evaluate how

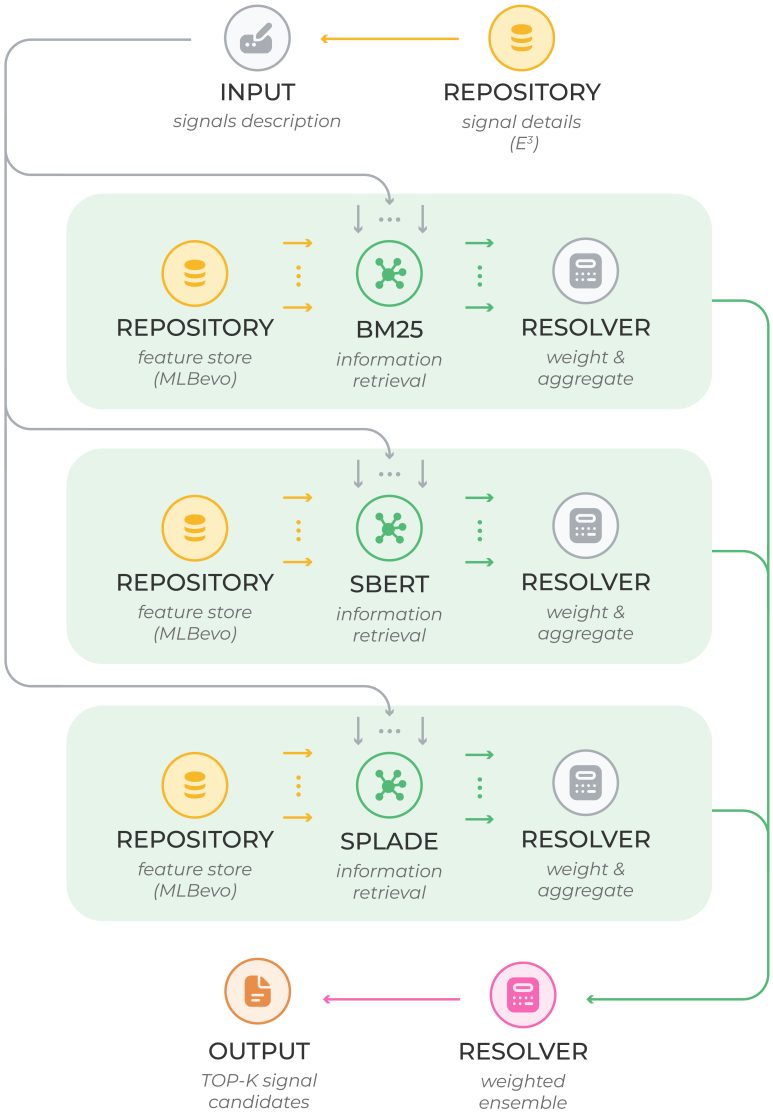

Figure 4.2: The figure illustrates the architecture designed for the signal mapping task. The inputs, consisting of the technical specifications of the signals, are retrieved from the E3 repository. Each descriptive field of the signal is compared against the preprocessed MLBevo corpus and a similarity score is computed for each field. These scores are then combined through a weighted sum, using customizable weights to reflect the relative importance of the different fields. The outputs of the individual models are also weighted and aggregated, ultimately producing the top-k most likely candidates for mapping signals between the two architectures.

accurately the E3 signals in the test set can be matched to the correct counterparts among the more than 10,000 MLBevo signals.

Field weight strategies Since each descriptive field contributes differently to the retrieval process, several weighting approaches were explored to assess how field relevance influences the overall matching performance. In addition, a comparison was carried out between pre- and post-rescaling configurations. The strategies considered can be summarized as follows:

- **Uniform Weighting**: all available fields are activated with unit weight, ensuring that each contributes equally to the similarity computation;
- **Selective Field Activation**: only a subset of fields is enabled. The selection was made empirically, based on the suitability of each field for the specific retrieval model and on preliminary observations about their discriminative power;
- **Weighted Key Fields**: certain fields are assigned higher weights, either because the model is known to perform particularly well on them or because they represent key attributes that two signals must necessarily share to be considered a correct match.

Ensembles In addition to evaluating each model individually, their combination through ensemble techniques was also tested. Before aggregation, the scores produced by each model are normalized to ensure that they lie on compatible scales and that no model dominates the others due to numerical differences. Since each model relies on a different number of fields and custom weights, two fusion strategies were defined:

- **Manually-Tuned Ensemble**: the similarity scores are scaled using manually defined multiplicative coefficients, allowing certain models to be emphasized or down-weighted based on empirical considerations.
- **Inverse Field-Weight Ensemble**: this method automatically balances the contribution of each model according to the total weight of the fields it uses. If a model relies on fewer fields or smaller weights, its score is amplified to

prevent it from being overshadowed by the others. Formally:

$$IFW_{vector} = \left[\frac{|FIELDS|}{\sum weights_{BM25}}, \frac{|FIELDS|}{\sum weights_{SPLADE}}, \frac{|FIELDS|}{\sum weights_{SBERT}} \right]$$

4.3.3 Results

Several tests were conducted to assess the system's ability to correctly identify, on the first attempt, the corresponding signal for a given input. Given the large number of signals and the inherent complexity of a real-world scenario, it was also evaluated the system's capability to locate the correct match within the **top-K** retrieved results, providing a more robust performance measure. The set of experiments allows for a direct comparison of the individual models, highlighting their strengths and limitations. Additional tests were performed to evaluate the impact of preprocessing operations, such as applying rescaling and offset transformations and standardizing numerical precision.

Baseline The results shown in Table 4.1 were obtained using the three models with a *Uniform Weighting* approach and without applying the full pre-processing pipeline. In this setting, SBERT and SPLADE outperform BM25, indicating that the task cannot be reduced to a simple exact-matching across the terms. The additional contextual information captured by sparse and dense models plays a substantial role in the retrieval process. Two milestone values clearly emerge from the results: a **30% top-1** accuracy achieved by SBERT and a **75% top-20** accuracy obtained by SPLADE. These scores constitute the baseline to be improved upon through the ensemble strategies and preprocessing techniques introduced in the previous section.

Accuracy Comparison Across Models

Model	Top-1	Top-10	Top-20
BM25	15%	30%	40%
SBERT	**30%**	50%	65%
SPLADE	25%	50%	**75%**

Table 4.1: Accuracy at multiple top-k levels for each retrieval model, computed without additional pre-processing and using the Uniform Weighting configuration.

Apply full pre-processing The first attempt to improve performance involved applying the full preprocessing pipeline to the data, thereby reducing the number of fields considered and standardizing the representation of numerical values. This was combined with a *Selective Field Activation* strategy: the description and comment fields written in German were disabled for SPLADE and SBERT, since both models are pre-trained in English and do not handle foreign-language tokens effectively. The error, min and max fields were also deactivated, as preliminary tests indicated that they could introduce noise. The results show an improvement for SBERT, shifting the milestone values to **35%** for **top-1** predictions and **65%** for **top-10** predictions. These findings suggest that reducing noise and selectively enabling fields plays a significant role in enhancing matching accuracy

Accuracy Comparison Across Models

Model	Top-1	Top-10
BM25	15%	30%
SBERT	**35%**	**65%**
SPLADE	30%	45%

Table 4.2: Accuracy at multiple top-k levels for each retrieval model, computed after an additional pre-processing and using the Selective Field Activation.

Model-specific improvements Each of the three models was adjusted through targeted modifications designed to enhance their respective strengths. Using the Weighted Key Fields technique, the contribution of the discursive fields (description and comment) was doubled for all models: the English version was used for SPLADE and SBERT, while the German version was used for BM25 to better capture potential matches involving technical terminology. Since it is reasonable to assume that the unit of measurement should match between corresponding signals, the weight assigned to this field was increased by a factor of five. The results indicate an improvement in **top-1** predictions for BM25 and SBERT, which reach **30% and 45%** accuracy respectively. These gains confirm that a more informed weighting of key fields can enhance matching performance.

Ensembles Finally, a series of tests was conducted to evaluate the collaboration between the different models. Three ensemble configurations were examined: an

Inverse Field-Weight Ensemble without full pre-processing and using Uniform Weighting, an Inverse Field-Weight Ensemble combined with pre-processing and Selective Field Activation and a Manually-Tuned Ensemble enhanced with the Weighted Key Fields technique. These three ensembles reflect the improvements observed in the individual models. The best performance in top-10 prediction slightly outperform the previous results.

Accuracy Comparison Across Ensembles

Model	Top-1	Top-10
IFW w/o pre-processing	25%	55%
IFW	35%	65%
MT	**45%**	**70%**

Table 4.3: Accuracy at multiple top-k levels for each retrieval ensembles.

Final Configuration After extensive testing across multiple configurations, the best performance was achieved using a *Manually Tuned Ensemble* with a *Weighted Key Fields* strategy. In this setup, the BM25 component was disabled and SPLADE was assigned a weight of 1.5× relative to SBERT. This configuration resulted in a **75% TOP-5** accuracy, significantly reducing the uncertainty window in the signal-mapping process and providing a compact and highly relevant set of candidate signals.

LLM approach The results indicate that preprocessing strategies, field-weighting techniques and ensemble methods progressively improve the system's performance, especially in its ability to retrieve the correct signal among the top candidates. A natural next step would be to further enhance the top-1 accuracy by integrating a generative model into the pipeline. However, an experiment conducted with the small *LLM gemma3 4B* did not yield additional improvements. This outcome can be explained by the limited capacity of the model and by the high semantic similarity among the top-5 retrieved signals, which makes it difficult for a compact LLM to reliably discriminate between them.

Chapter 5

Task 2: Alias generation

5.1 Background

After identifying the pairs of equivalent signals (one from the MLBevo architecture and the other from the E3 architecture) it becomes necessary to assign a representative **alias** to each pair. The JSON-based test catalogue requires every pair of signals to be associated with a unique and meaningful name, which allows the tests to refer to the signals in an abstract way, independent of their original architecture. The introduction of standardized aliases also provides an important scalability advantage: future architectures can be integrated into the testing framework simply by mapping their signals to the existing aliases, without the need to modify the test definitions or replace architecture-specific signal names.

5.1.1 Objectives

Alias generation requires the design of an operational process capable of acquiring the technical information associated with each signal and, based on this information, producing a semantically meaningful name. Currently, aliases follow a fixed structure composed of three elements:

- The **prefix "AL"**, which is constant and indicates that the identifier is an alias.
- A sequence of **acronyms**, each derived from a keyword describing the characteristics or function of the signals.

- An acronym representing the **unit of measurement**, which completes the alias.

The focus is on the last two components. In particular, the objectives are:

- **Keyword generation**: this is a creative and interpretative task that requires analyzing the technical specifications of the signals to extract relevant concepts. Since this process involves semantic understanding and text generation, the use of a Large Language Model (LLM) is required.
- **Acronym mapping**: neither the keywords nor the units of measurement can be used directly in the alias. They must be transformed into compact acronyms, following rules defined by company requirements. This transformation can occur in two ways:
  - mapping to a predefined dictionary of acronyms,
  - applying formal rules for manually constructing the acronyms.

5.2 Proposed solution

The proposed solution is a system based on a prompt chain, supported by information retrieval capabilities. The overall architecture, shown in Figure 5.1, integrates components for data extraction, semantic generation and transformation of the outputs into structured aliases.

5.2.1 Keyword generation

The first step consists of *retrieving* the technical details associated with each signal in the pair through a repository function. Only the fields containing the name, unit of measurement, description and comment were selected, as they represent the most descriptive and interpretable portion of the specifications. These elements provide enough contextual information to understand the purpose of the signal and to identify meaningful concepts from which keywords can be derived. For keyword generation, an LLM was used, running locally on an NVIDIA GTX 1050Ti 4GB GPU through the *Ollama* framework. Running the model locally ensures that

sensitive data remains within the company infrastructure, avoiding the need to send information to external cloud services.

Ollama

Ollama [37] is a platform that enables local execution of a wide range of language models, offering a simple interface and automatic management of hardware resources. Compared to using traditional Python libraries such as transformers, Ollama simplifies model execution by:

- automatically handling GPU usage when available;
- downloading and running quantized model versions;
- providing minimal configuration and fast startup.

When combined with **Docker**, it is possible to create a dedicated container that hosts both Ollama and the LLM, keeping the service always active and accessible as if it were an external server. Integration with Python code is supported by the ollama library, which allows customization of generation parameters, output formats and communication with the Docker container.

5.2.2 Acronym mapping

The Keyword generation phase produces a list of terms for each pair of signals. These terms must then undergo an **abbreviation** step, where each keyword is converted into a compact acronym. Two different approaches are used to obtain these abbreviations: by looking in a technical vocabulary ad using an LLM.

Lookup in a technical vocabulary The first method consists of searching for abbreviations in a technical dictionary. The **AUTOSAR** glossary, which is publicly available online and contains around 1300 terms with their corresponding acronyms, was selected for this purpose. After downloading the glossary, embeddings were generated for each term using an SBERT model, specifically *all-MiniLM-L6-v2*. This made it possible to identify not only the exact abbreviations for keywords present in the dictionary, but also semantically related terms for keywords that were not included explicitly. Semantic similarity was computed using cosine similarity

[38], allowing the system to associate each keyword with the closest matching terms.

$$cosine_{similarity}(d_1, d_2) = \frac{V(d_1) \cdot V(d_2)}{|V(d_1)||V(d_2)|} \tag{5.1}$$

Where d_1 and d_2 are the source and target words, while $V(d_1)$, $V(d_2)$ their vector representations (SBERT embeddings).

This approach proved effective for:

- handling *morphological variations* (singular/plural forms, different verb tenses);
- recognizing *synonyms* or alternative formulations;
- increasing *robustness* against linguistic variability introduced by the generative model.

Automatic derivation of abbreviations The second method is used when a keyword has no match in the AUTOSAR dictionary. In these cases, the goal is to preserve the widest possible vocabulary while still producing consistent and readable acronyms. A *company guideline* was used, containing a set of general rules for deriving abbreviations from any word. These rules include both mechanical criteria (such as removing vowels and repeated letters) and more subjective decisions based on readability, which cannot be easily encoded in a traditional algorithm (for example, after removing all vowels, it may be necessary to reintroduce some of them if the resulting acronym is difficult to interpret). To handle this non-deterministic component, an **LLM** was used again. The guideline was included in the prompt and the model was instructed to produce a structured output in the form of a *keyword → acronym dictionary*, simplifying the post-processing phase.

5.2.3 Alias assembly and conflict handling

Once the dictionary mapping each keyword to its corresponding abbreviation has been generated, an LLM is queried again. In this step, the model receives the list of keywords, the fixed prefix “AL” and the abbreviation of the unit of measurement, which is also obtained through the previously described procedures. Although the assembly of the alias could be implemented through a purely mechanical process, a

language model was intentionally involved to introduce variability and flexibility. The model is instructed to select only a *subset* of the available abbreviations, choosing those it considers most relevant, and to arrange them in the order it finds most appropriate. This prevents the creation of overly rigid or repetitive aliases. To ensure alias uniqueness and make the system robust at runtime, this node of the prompt chain was designed to be **recursive**. If the model produces an alias that already exists, it is queried again, this time including examples of conflicting aliases to avoid. The cycle can be repeated multiple times until a unique result is obtained. To mitigate the impact of potential model hallucinations, a final *validation step* is performed. The system checks that the alias follows the required structure (the "AL" prefix, the sequence of abbreviations and the unit-of-measurement acronym) with each component separated by an underscore. Any special or invalid characters are also removed, ensuring that the final output is well-formed and compliant.

5.3 Experimental results

The experiments were conducted on a subset of 20 pairs of already-mapped signals, the same used in the evaluation of task 1. The generative model employed was *Gemma3 4B*, running locally inside a container through Ollama. The AUTOSAR dictionary was enriched beforehand with a set of official abbreviations for units of measurement, ensuring broader semantic coverage.

General observations The system's behavior was evaluated by repeatedly running the alias-generation process on the entire subset. Thanks to the fast execution time (approximately *2–3 minutes* for all 20 pairs) multiple iterations could be performed efficiently. The model was able to extract a meaningful number of keywords from the signal information. The *comment field* was preferred, as the other fields contain highly technical details from which it is more difficult to abstract general concepts suitable for keyword generation. The *conflict-handling mechanism* successfully resolved cases of duplicate aliases, especially when similar signals produced overlapping keyword subsets. The recursive generation step ensured that a unique alias was eventually produced for each pair.

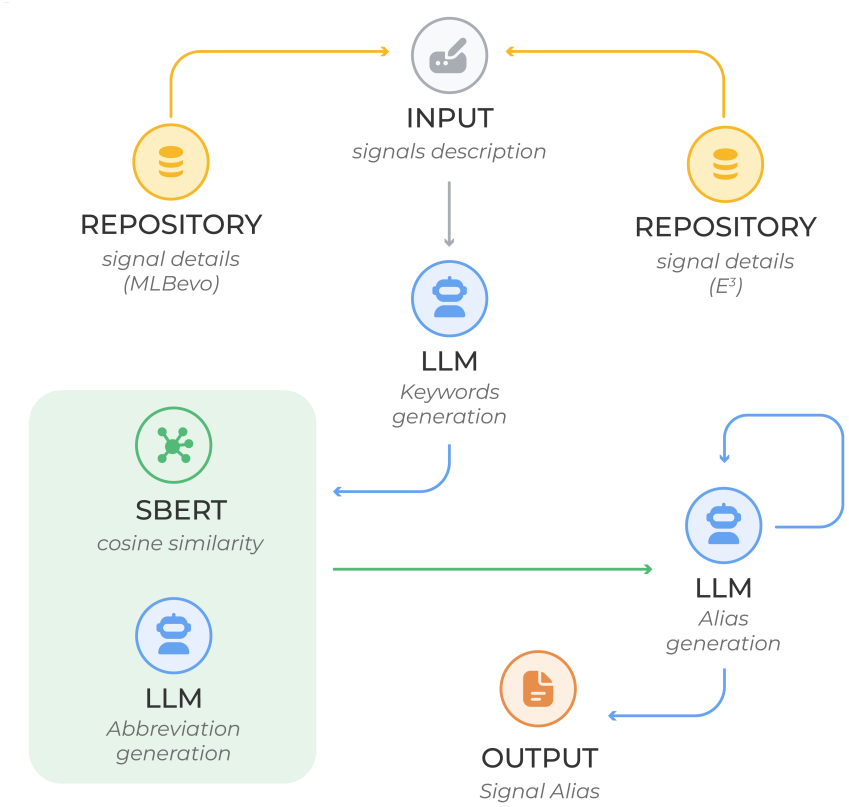

Figure 5.1: The figure illustrates the architecture of the alias generator. The input data consists of signal details retrieved from the MLBevo and E3 repositories. Based on this information, a set of keywords is produced and forwarded to the module responsible for generating abbreviations. The process unfolds in several stages. First, SBERT is used to search for potential matches within the AUTOSAR dictionary. If no suitable abbreviation is found, an LLM is invoked to generate a new abbreviation at runtime, which is then added to the existing dictionary, allowing it to grow over time. The resulting set of abbreviations is then passed to another LLM, whose task is to assemble the final alias. In the event of conflicts, such as duplicate aliases, the model reiterates the generation process until a unique and valid alias is produced.

Weaknesses Since the system relies on a relatively small and quantized model, the quality of the outputs is not always guaranteed. The presence of multiple rules for constructing abbreviations, some of which are non-deterministic, may lead the model to generate unexpected aliases, for example by including special characters or ignoring certain constraints. For this reason, an additional post-processing stage was introduced to:

- remove *invalid* characters;
- algorithmically *assemble* the three components (prefix, keyword abbreviations, unit of measurement), leaving only the central part to the generative model.

Example Here, an example from the generated ones:

- **Generated keywords**: “object”, “speed”, “relative”, “radial”
- **Unit of measurement**: "meterpersecond"
- **Generated alias**: AL_ObjSpdRelRd_mps

The content of the signal description is not reported, for confidentiality reasons.

Chapter 6

Task 3: JSON Test catalogue Generation

6.1 Background

The third task focuses on generating structured files that contain the descriptive specifications of the tests used to validate system functionalities. Each test encodes a specific behavior to be observed and breaks it down into a sequence of operations involving the reading and manipulation of numerical values transmitted by the vehicle's control unit signals. Each file includes all the information required to describe the entire testing pipeline and is organized into the following blocks:

- **Signal list**: This block contains a sequence of objects representing the signals involved in the test, each identified by an alias. Every alias internally refers to the actual signal identifiers for each supported architecture. In this case study, the architectures are MLBevo and E^3 and each pair of equivalent signals is associated with an alias consistent with the mapping defined in task 2.

- **Parameters**: A list of parameters used to customize the test. These may include, for example, timeout values or numerical thresholds associated with specific operations.

- **Preprocessors**: Preprocessors can be considered *virtual signals*. They allow

certain computations on real signal values to be performed once at the beginning of the test, simplifying the subsequent steps. Their purpose is to reduce redundancy by precomputing recurring expressions that may be reused across multiple parts of the test.

- **Detectors**: This is the core and most complex block of the test. It contains sequences of arithmetic and logical operations applied to signals and preprocessors to verify the conditions required by the test and to evaluate the vehicle's state and behavior. The block includes:
  - **Global condition**: conditions describing the overall system state, which must remain valid throughout the entire test.
  - **Precondition**: initial conditions that must be satisfied before the test begins.
  - **Action**: a set of actions that should be performed or happen once the preconditions become valid, within a certain time.
  - **Expected results**: the final state of the system, right after the previous actions.

6.1.1 Objectives

The goal of this study is to improve the efficiency of generating the JSON file that represents a test, starting from a **natural-language description**. Since the parameters section is highly customizable and depends on the specific experimentation, it is not considered relevant for this analysis. To further simplify the problem, the preprocessors section was also excluded. The operations normally contained in this block were integrated directly into the steps where they are used, avoiding the need to generate intermediate structures.

Case studies Three case studies were examined, each defined by the type of input information available and the desired level of automation:

- **Signal detection**: The aim is to automate the identification of the signals required for the test, starting from a short and generic description of the test objectives. The detected signals can then be used to construct the signal list.

- **Structured conversion of detectors**: In this scenario, the user provides a textual input describing the operations to be performed in each step, along with the signals involved. The goal is to convert this description into a structured format that matches the detector specification.

- **Detector deduction**: The most advanced level of automation evaluates whether the system can automatically infer all the required operations, using only the set of signals and a high-level description of the expected behavior.

6.2 Proposed solution

The proposed solutions involve the implementation of architectures that integrate multiple functionalities, combining information-retrieval strategies with generative language models and advanced prompt-engineering techniques. It is important to note that the amount of data available within the company is extremely limited and often not of high quality. In addition, the previously mentioned constraints related to *data confidentiality* and the scarcity of available *hardware resources* further restrict the design space. These factors made the development of effective solutions particularly challenging. The **lack of sufficient data** prevents the training or fine-tuning of customized models, while **hardware limitations** restrict the use of large-scale architectures. As a result, the system had to be designed to maximize the capabilities of lightweight models, compensating for their limitations through targeted *prompt-engineering strategies* and information-retrieval mechanisms.

6.2.1 Signal detection

Identifying the signals required for test generation is a particularly challenging task, as it involves combining two highly heterogeneous input sources: a high-level natural-language description and a structured set of technical signal information. The proposed architecture addresses this complexity by integrating the **information-retrieval techniques** used in task 1 with an extensive preprocessing phase aimed at improving the representation of the available data. This approach helps reduce the semantic gap between free-form text and structured data, enabling the system to more effectively identify the signals relevant to the test definition.

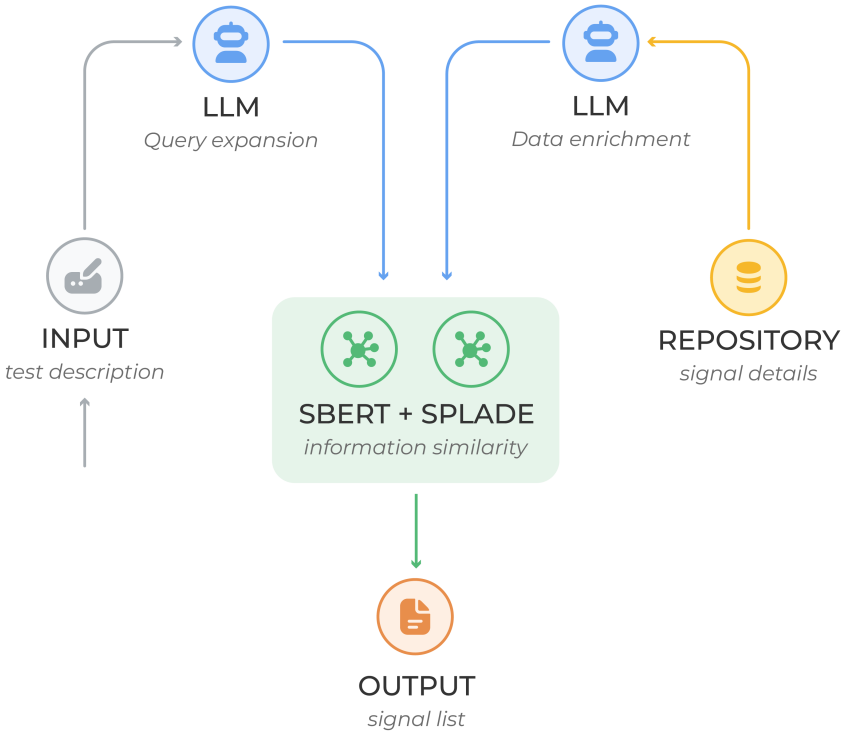

Figure 6.1: The figure provides a graphical representation of the main components of the architecture. The diagram illustrates the high-level pipeline and the overall flow of data across the different modules. Although there is a logical connection between the LLM-based data enrichment block and the SBERT + SPLADE ensemble, the size of the dataset required this process to be executed separately and only once, storing the results in the repository.

Data preprocessing The technical information associated with the signals, structured as in the previous example 4.1, is often difficult to interpret. It mainly consists of numerical values, abbreviations and very short technical descriptions, which do not easily support the extraction of meaningful semantic concepts. In contrast, the input test descriptions use a more narrative and high-level language, expressing the general objectives and conditions to be verified. To reduce this semantic gap and make the two data sources more comparable, a **data-enrichment** process is applied to the signal information. Specifically, two new descriptive fields are introduced:

- **enhanced description**: provides a more detailed and narrative explanation of the technical specifications of the signal;
- **possible uses**: suggests a set of potential applications or usage contexts for the signal.

Both fields are generated using an LLM and are designed to produce content that is more similar in style and structure to the natural-language test descriptions. With this transformation, the enriched fields (considered as *documents* in the information retrieval phase) become more comparable to the test descriptions (*queries*). To further improve query quality, a query-expansion step is also introduced, where an LLM reformulates and enriches the user's input description, making it more informative and better suited for retrieval.

An information-retrieval system Building on the results obtained in task 1, an **ensemble** composed of an SBERT module and a SPLADE module was adopted. Using the test description as the query and treating the signal information as documents, an information-retrieval system was implemented with the goal of selecting the documents most semantically aligned with the initial query. To improve retrieval quality and representation granularity, a separate corpus was created for each of the five available fields:

- *signal name*,
- *description*,
- *comment*,

- **enhanced description**,
- **possible uses**.

The description and comment versions were the same previously used in task 1, translated using *MarianMT*. This separation allows the semantic contribution of each field to be evaluated independently and enables more flexible combination strategies, ultimately improving the system's ability to identify the most relevant signals for test construction.

6.2.2 Structured conversion of detectors

Automatically converting natural-language descriptions into a structured format significantly accelerates the creation of a test file. The structure of the file is strict, hierarchical and must comply with specific formatting rules to ensure that the resulting JSON is syntactically valid. In contrast, providing a literal, narrative description of the test content is far more user-friendly and considerably faster for the user.

Prompt-chain structure The proposed architecture is based on a *prompt chain*. The input consists of a textual description that specifies the signals involved and the operations to be performed in each step of the test (*global condition, precondition, action and expected results*). The pipeline is divided into smaller, specialized tasks:

- a *node* responsible for assembling the **signal list**, based on the signals mentioned in the description;
- a dedicated *node* for each of the *four test steps*, responsible for generating the substructures that compose the **detectors section**.

Each node relies on an LLM, which receives a detailed explanation of the task to be performed. Several prompt-engineering techniques are used to guide the model toward a correct and coherent output:

- **Role engineering**: used to provide technical context and shape the model's response style;

- **System prompting**: applied to specify the formal rules of the task and the exact structure of the expected JSON output;
- **Few-shot prompting**: useful to supply examples that the model can imitate to reproduce the desired format.

The interaction with the model is managed through a **chat template** [39], which cleanly separates system instructions, few-shot examples and the user query, in line with the model's training paradigm.

Chat template

All language models are causal: they are trained to generate a sequence of tokens conditioned on an input text, which strongly influences the final output. After the pre-training phase, carried out on large amounts of data, many models undergo fine-tuning to improve their behavior in user-oriented scenarios, such as chat-based interactions involving questions and answers. To better guide the model's behavior, fine-tuning introduces specific markers (**special tokens**) that delimit structured information, such as the message role, enabling the model to recognize and interpret them effectively. Using the same tokens during inference is therefore an excellent way to improve output quality, as the model receives an input sequence formatted in a way that is "familiar" from training. This configuration can be implemented manually by inserting the special tokens directly into the prompt, or automated through library functions that construct the message sequence. This allows the creation of a structured interaction composed of:

- **System message**: which contains task details and formatting rules;
- **A sequence of user and assistant messages**: used to simulate a chat history and provide few-shot examples;
- **Final user message**: that contains the actual request.

This organization mirrors the format used during model fine-tuning and leads to responses that are more coherent, technically accurate, and aligned with the desired output structure.

6.2.3 Detector deduction

The final investigation aimed to assess whether the system can autonomously infer the operations required for each step of the test, using only a **high-level description**. Unlike the structured-conversion scenario, no explicit instructions are provided regarding how the available signals should be used. The proposed architecture mirrors the prompt-chain structure described earlier but introduces two key components: a signal repository and a reasoning node.

Access to the signal repository The system has access to the full signal repository, the same used in task 1 (signal mapping) and task 2 (alias generation). For each signal, both the *technical specifications* and the *enriched descriptive fields* are available. In this setting, the model must independently determine which signals are relevant and how they should be used, relying solely on their descriptive information.

Preliminary reasoning phase Before generating the JSON sections for the test steps, an additional interaction with the LLM is introduced. In this phase, the model is tasked with:

- *analyzing* the high-level test description;
- examining the *specifications* of the available signals;
- inferring the **arithmetic and logical operations** required to perform the test, for each test step (global condition, precondition, action and expected results).

This reasoning stage acts as an intermediate layer between the abstract description and the structured JSON output, enabling the model to construct a logical representation of the test flow before converting it into the final format.

6.3 Experimental results

The experiments were carried out independently for each of the three scenarios. Although, in a real-world application, it would be natural to combine signal detection with JSON generation in a cascaded pipeline, the purpose of this analysis was

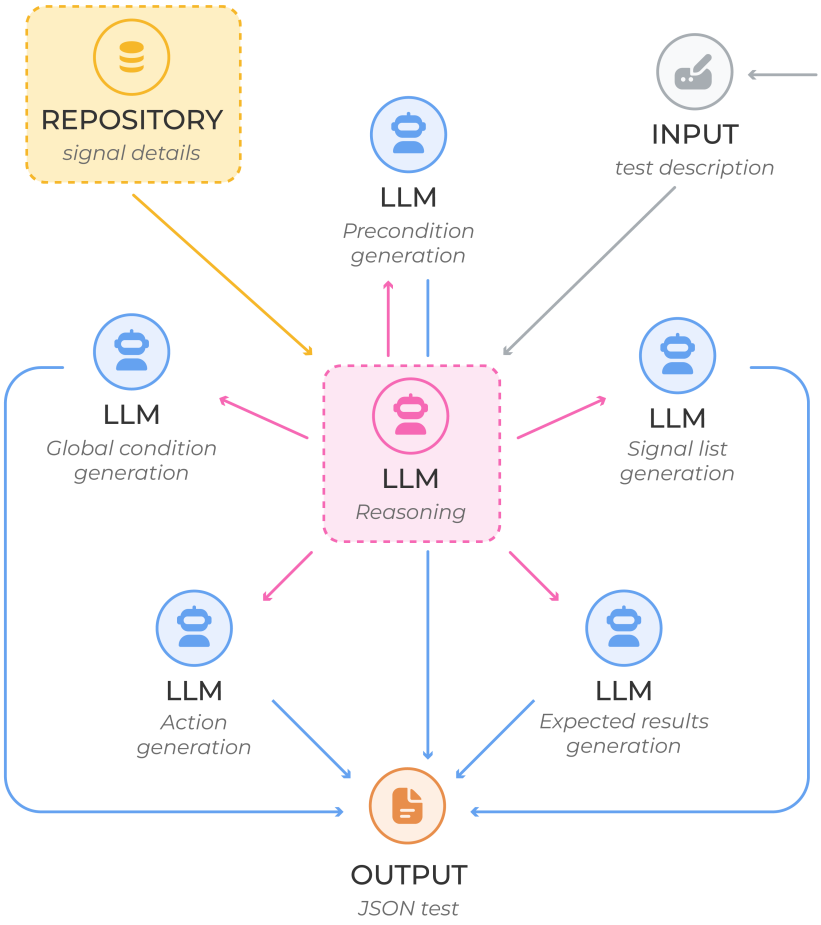

Figure 6.2: The figure illustrates the architecture proposed for the detector deduction task. At the core of the system lies the reasoning component, which generates a description of the operations to be performed within each step. The final JSON file is obtained by composing the intermediate outputs, each produced through a dedicated interaction with the LLM. The architecture follows a modular design: each step is processed independently, reducing error propagation and ensuring better control over the generation pipeline. A similar structure can also be applied in the guided scenario. In that case, the reasoning block is replaced by a human operator, who provides the operational logic to be converted into the structured format. The LLM is then responsible for translating this logic into the required JSON representation, while the rest of the pipeline remains unchanged.

to evaluate each component separately, assessing its effectiveness and identifying potential challenges. This experimental setup makes it possible to isolate the contribution of each module, preventing upstream errors from affecting the evaluation of subsequent tasks. As a result, it becomes easier to determine which parts of the pipeline are already robust and which require further refinement before being integrated into a full end-to-end workflow.

6.3.1 Signal detection: experimentation

The ground truth used for the experimentation consists of a set of **26 JSON tests**, each accompanied by a generic description and a guided description of the individual steps. For the signal detection task, only the generic test description was used, since the guided descriptions contain explicit references to the signals and would have biased the evaluation. The first step involved performing data enrichment on the signal specifications in the dataset, generating the fields "enhanced description" and "possible uses". These fields were produced using a lightweight LLM, *Qwen2.5 3B*, executed locally through Ollama and accelerated by an NVIDIA GTX 1050Ti 4 GB GPU. To speed up processing, the signals were handled in batches of five. The entire procedure required more than *40 consecutive hours* of computation.

Detection procedure Candidate signals were identified through an information retrieval approach, where the test description serves as the query and the signal specification fields serve as the documents. Each field was evaluated independently, as well as in selected combinations, to assess which representations contributed most effectively to retrieval quality.

Since each test may involve a variable number of signals, the evaluation metric was an accuracy computed over a **window** whose *size matches the correct number* of signals for that test. This allows measuring the proportion of correct signals that the system is able to place within the top positions of the ranking, providing a realistic assessment of retrieval performance.

Baseline The initial results reveal a partially unexpected outcome: among the original fields of the signal specifications, the **signal name** yields the best preliminary performance when used independently. This behavior can be explained by two main factors.

Accuracy on True Window - (Baseline)

Model	Eng Description	Eng Comment	Signal Name
SBERT	0%	0.55%	17.02%
SPLADE	0.96%	1.1%	20.14%
Ensemble	2.47%	1.1%	**22.79%**

Table 6.1: Accuracy on true window using independently one of the original field.

First, the high-level test descriptions are often *too abstract* compared to the technical descriptions of the signals. The semantic gap between the two domains is therefore quite large, making it difficult for the system to establish meaningful correspondences based on descriptive content. Second, some of the generic test descriptions contain *acronyms or keywords* (e.g., ACC) that appear explicitly within the signal names. This gives the signal name field a direct advantage, as the presence of identical or highly similar terms facilitates alignment with the query. As a result, despite its brevity, the signal name turns out to be the most effective field in the initial retrieval phase, especially when the test description implicitly or explicitly references concepts encoded directly in the signal nomenclature.

Data Enrichment Due to the limited size of the model used to generate the additional descriptions, the initial semantic-enrichment step did not lead to any improvement in retrieval performance. The generated descriptions were often generic, imprecise and in several cases affected by *hallucinations*, which reduced their usefulness as textual representations of the signals. For this reason, a **targeted refinement** was applied: more accurate descriptions were provided only for the signals involved in the detection task. Despite the noise introduced by the remaining ~20,000 signals, whose enriched fields remained low-quality, this strategy produced a measurable improvement. Specifically, as shown in 6.2, the overall result improved further when combining these fields with the signal name, which remains highly informative due to the presence of acronyms and technical terms shared with the test descriptions.

Query expansion Another experiment involved applying an automatic query expansion step to the test description used as input. Since these descriptions

Accuracy on True Window - (Data Enrichment)

Model	Enhanced Description (ED)	Possible Uses (PU)	ED + PU	Signal Name + ED + PU
SBERT	10.39%	2.47%	14.37%	22.38%
SPLADE	9.43%	12.55%	13.31%	22.38%
Ensemble	13.19%	14.15%	19.4%	**24.85%**

Table 6.2: Accuracy on true window using different combinations of descriptive fields.

typically consist of a single sentence, the goal was to enrich them semantically, introducing additional nuances and potential keywords. The effect was, in some respects, the opposite of what was previously observed. The expanded descriptions become more *narrative* and tend to use fewer acronyms. As a result, the effectiveness of retrieval based on the signal name decreases, since it no longer benefits from the presence of identical or highly similar terms between the query and the signal name. Conversely, an improvement is observed in the matching performance of the more descriptive fields. The best results are achieved by combining "enhanced description", "possible uses" and the original English comment, indicating that a richer query better aligns with fields that contain more elaborate textual content. These findings suggest that query expansion can be *beneficial*, but its impact strongly depends on the type of field used for retrieval and on the linguistic characteristics of the original test description.

Accuracy on True Window - (Query Expansion)

Model	Signal Name	Signal Name + ED + PU	Eng Comment + ED + PU
SBERT	1.51%	25.4%	23.17%
SPLADE	5.49%	23.52%	24.31%
Ensemble	6.46%	24.48%	**32.04%**

Table 6.3: Accuracy on true window using Query Expansion.

A larger window Since the reference tests involve between 1 and 7 different signals, it is reasonable to explore the effect of *enlarging* the evaluation window. By extending the window to **three times** the number of correct signals, the number of retrieved candidates remains manageable for a human user, who would still benefit from a reduction of three orders of magnitude compared to the full dataset. In this scenario, the best configuration (obtained by combining "enhanced description", "possible uses" and the original English comment within the query-expansion setting) reaches an accuracy of **44%**. Although still insufficient for full automation, this result represents an improvement over the previous configurations.

Final considerations The experiments on the signal detection task highlight a strong dependence on the quality of the available data, both in terms of the textual descriptions of the tests and the technical specifications of the signals. Unlike task 1, where the system compared pairs of semantically similar signals, this task requires aligning elements that differ substantially in structure, abstraction level and linguistic form. The results show that the techniques employed (an ensemble of retrieval models, data enrichment and query expansion) do contribute to performance improvements. However, the overall accuracy remains too low to consider the system capable of fully automating this task.

6.3.2 Structured conversion of detectors: experimentation

This experimental phase involved **36 test files**, divided into a set of *34* used as the *test set* and *2* used as *few-shot examples*. Each test includes a *detailed description* of the operations to be performed in every step (global condition, precondition, action and expected results). The system operates in a guided setting: its task is to convert these textual instructions into a JSON file that complies with the strict structure and formatting rules defined by the company. All experiments were conducted using the *Gemma3 4B* model.

Example of detailed description *"1) Global Condition: ACC status (SIGNAL_NAME_1) == 3 and PACC system status (SIGNAL_NAME_2) == 3, 2) Precondition: predicted speed (SIGNAL_NAME_3) < target set speed (SIGNAL_NAME_4), 3) Action: predicted speed (SIGNAL_NAME_3) > target set speed (SIGNAL_NAME_4), 4) Expected Result: SIGNAL_NAME_5 == 1."*

Signal list The correct set of signals to be used is assumed to be known. The first task assigned to the system is the generation of the signal list section, which contains the information of the signals across different architectures and their aliases. This task is essentially a *fill-the-blank* operation: the model must populate a predefined structure with the appropriate values. As expected, the model handled this step without particular difficulty.

Detectors The most challenging part concerns the generation of the detectors, the section that encodes the *mathematical and logical operations* between signals in a structured format. The model was provided with all the necessary information:

- **textual descriptions** specifying which signals to use and how,
- a prompt describing the required JSON structure and **formatting rules**,
- a dictionary of **allowed operations**, which must be expressed in the JSON using the exact syntax defined by the specification.

A preliminary preprocessing step was applied to split the textual description into fragments corresponding to each test step. This ensures that each node of the prompt chain receives only the information relevant to its specific step, reducing noise and improving the consistency of the generated output.

Metrics To evaluate the performance of the proposed architecture, the metrics **precision** and **recall** were employed. These indicators are essential for assessing both the correctness and completeness of the structured output generated by the model. Together, precision and recall provide a balanced assessment of the model's performance, capturing both the accuracy and the coverage of the generated detectors: two aspects that are crucial when producing structured JSON content.

Precision and Recall

Precision measures the proportion of correctly generated elements with respect to the total number of elements produced by the model. It reflects how accurate the output is and how often the model introduces incorrect or unnecessary information (i.e., hallucinations). Formally:

$$precision = \frac{TP}{TP + FP} \tag{6.1}$$

where TP (True Positives) denotes the number of correctly generated elements, while FP (False Positives) represents the elements generated that do not appear in the ground truth. *Recall*, on the other hand, evaluates the model's ability to retrieve all expected elements. It measures how complete the generated output is. Formally:

$$recall = \frac{TP}{TP + FN} \tag{6.2}$$

where FN (False Negatives) indicates the elements that should have been generated but are missing from the output.

Results (Lightweight model) Precision and recall were computed on the operations contained in each step, requiring an exact match between the three components of every operation (**operator** and **two operands**). To account for minor errors, *relaxed versions* were also defined, where a match is considered valid even if one of the fields contains an error. The results show very strong performance: in almost all steps precision and recall exceed **0.82**. The only exception is the action step, where the values drop by roughly one tenth. However, the customized metrics reveal an important insight: in the action step, the model fully recovers the gap. This indicates that most of the additional errors affect only one field of the triplet, while the other two are correct. As a result, these mistakes are easy to fix through human supervision.

Operations Precision and Recall

Step	Precision	Recall	Relaxed Precision	Relaxed Recall
Global Condition	0.858	0.853	0.902	0.897
Precondition	0.828	0.941	0.880	0.971
Action	0.733	0.733	0.912	0.885
Expected Results	0.858	0.853	0.902	0.897

Table 6.4: Stats on the operations contained in detector's steps, using Gemma3 4B, in the context of structured conversion of detectors.

Operations Precision and Recall

Step	Precision	Recall	Relaxed Precision	Relaxed Recall
Global Condition	0.847	**0.885**	**0.933**	**0.968**
Precondition	**0.848**	0.941	0.877	0.971
Action	**0.750**	**0.821**	0.829	0.885
Expected Results	0.847	**0.885**	**0.933**	**0.968**

Table 6.5: Stats on the operations contained in detector's steps, using Gemma3 27B, in the context of structured conversion of detectors.

Results (Mid-range model) An additional experiment was carried out in this guided scenario using a larger model, *Gemma3 27B*, accelerated by a workstation equipped with an NVIDIA RTX 4070Ti (12 GB). The results are largely comparable to those obtained with the smaller model, as the baseline performance was already strong. However, slight improvements in recall can be observed, particularly in the action step, where the gap with the other models was reduced. This indicates that increased representational and reasoning capacity can enhance coverage, even when the task is already well-defined.

6.3.3 Detector deduction: experiments

The experimental phase of the final and most challenging task aimed to assess whether a general-purpose language model can autonomously infer the **full logic** and pipeline of a test, starting only from a generic description and the specifications of the signals involved. The dataset used for this task is smaller than in the previous experiment. Tests lacking additional information about their signals were excluded, since those signals would have been effectively unknown to the repositories. Moreover, a manual preprocessing step was required:

- *removing all preprocessors* from the original files,
- *adding the missing operations* within each step to obtain a complete and reliable ground truth.

It is also important to note that the textual input does not explicitly describe all the conditions and behaviors that the test is meant to verify. In most cases,

it only conveys the general purpose of the test, without detailing the underlying logic. This makes the task particularly challenging, as the model must bridge a substantial information gap and reconstruct the intended reasoning process on its own.

Example of high-level description *"Check if the predicted set speed crosses the target set speed from below to above while ACC is active."*

Operation metrics (Lightweight model) As in the previous scenario, precision and recall were computed separately for each step, considering the operations contained within them. Each operation is represented as a triplet consisting of an operator and two operands and both exact and partial matches were evaluated (the latter allowing one of the three fields to be incorrect). As expected, the results are significantly *lower* than in the guided scenario. The average values are around **0.2**, with a slight increase when partial matches are included, where the highest values observed are **0.378** for precision and **0.5** for recall. This task introduces a substantial level of difficulty: it requires deep technical knowledge and a detailed understanding of the test logic, which a general-purpose model does not possess. Moreover, the textual input is highly generic and does not explicitly describe the operational conditions to be verified. As a result, it is far from guaranteed that the model can reconstruct such a specific technical pipeline from such limited information.

Operations Precision and Recall

Step	Precision	Recall	Relaxed Precision	Relaxed Recall
Global Condition	0.111	0.083	0.178	0.150
Precondition	0.263	0.333	0.338	0.500
Action	0.278	0.283	0.378	0.367
Expected Results	0.111	0.083	0.178	0.150

Table 6.6: Stats on the operations contained in detector's steps, using Gemma3 4B, in the context of Detectors deduction.

Operation metrics (Mid-range model) An experiment was also carried out in this more advanced scenario using the *Gemma3 27B* model. In this case, the improvement is more noticeable, consistent with the enhanced reasoning capabilities provided by a larger model. The reported metrics show gains across all steps, with a particularly interesting result in the action step, which becomes the most accurately predicted component. Precision increases to **0.4**, while recall exceeds **0.58**, and relaxed recall goes beyond **0.8**. These results indicate that a larger model can better capture the logical structure of the operations and generalize more effectively, even when starting from high-level descriptions.

Operations Precision and Recall

Step	Precision	Recall	Relaxed Precision	Relaxed Recall
Global Condition	0.178	**0.325**	0.255	**0.385**
Precondition	0.342	0.433	0.394	0.550
Action	**0.406**	**0.583**	**0.642**	**0.821**
Expected Results	0.178	**0.325**	0.255	**0.385**

Table 6.7: Stats on the operations contained in detector's steps, using Gemma3 27B, in the context of Detectors deduction.

Input metrics In addition to evaluating the model's ability to infer the operation pipeline, an analysis was conducted on its capability to assign the correct signals to the steps in which they should be used. The results show high recall values, above **0.7** with a maximum of **1**, while precision averages around **0.6**. These results can be explained by the model's tendency to include more signals than necessary in each step. This "*over-inclusive*" behavior makes it easier for the model to capture all the relevant signals (thus increasing recall), but it also leads to the inclusion of unnecessary ones, which lowers precision. In other words, the model is generally able to identify the correct signals, but struggles to avoid adding irrelevant ones. This behavior is consistent with the nature of the task: when the input is vague and underspecified, a general-purpose model tends to err on the side of inclusion rather than risk omitting important elements.

Input Precision and Recall

Step	Precision (4B)	Recall (4B)	Precision (27B)	Recall (27B)
Global Condition	**0.583**	**0.733**	0.497	0.592
Precondition	0.574	1.0	**0.589**	1.0
Action	0.689	0.767	**0.767**	**0.800**
Expected Results	**0.583**	**0.733**	0.497	0.592

Table 6.8: Stats to evaluate the capability to assign the correct signal to the correct step, using both Gemma3 4B and Gemma3 27B.

Final considerations Given the intrinsic difficulties highlighted in the previous analyses, it is clear that the JSON-generation process cannot be fully automated when starting from such generic inputs and relying on a general-purpose model. The lack of explicit information in the test description and the technical complexity of the required operations make it unrealistic for the model to reconstruct the entire pipeline reliably on its own. However, the results obtained in the guided scenario indicate that this approach can be effectively used as a support tool in intermediate applications. In these contexts, a human operator can provide more precise instructions so that the model can generate a structured draft of the test. The resulting output serves as a useful starting point, enabling the user to refine and correct the content without having to build the entire JSON from scratch.

Chapter 7

Conclusion

7.1 Final considerations

The use of artificial intelligence tools, in particular language models, both *generative* and *non-generative*, has the potential to become a turning point in industrial and corporate environments. Automating complex and repetitive tasks can significantly improve *efficiency*, reducing the time and resources required compared to traditional workflows. However, the adoption of these technologies is neither immediate nor straightforward. It first requires a clear understanding of the needs and requirements of the tasks to be optimized, followed by a careful process of selecting, studying and experimenting with the most suitable tools. The experiments conducted in this work demonstrate that research and methodological design are essential: combining multiple techniques leads to better results than applying one or more models in a naive manner. Even under challenging conditions, where data and resources are not ideal, it is still possible to achieve meaningful outcomes. This confirms that, when integrated thoughtfully and systematically, artificial intelligence can serve as a valuable and strategic support within real-world industrial processes.

The results obtained gain additional significance when considered within the real application context. In a highly specialized industrial environment, tools of this kind can provide a solid starting point on which human experts can build more efficiently. The tools developed in this work cannot offer a fully automated solution, nor can they eliminate the need for human supervision. This limitation is largely due to the extreme specificity of the problems involved: these tasks are

complex even for domain specialists and therefore cannot be reliably solved by small general-purpose language models. Despite these constraints, the proposed architectures are able to generate structured drafts in a matter of seconds, greatly reducing the amount of manual work required. They also help narrow down the search space in several related sub-tasks, such as cross-architecture signal mapping. In this sense, the system acts as a practical support tool and a process accelerator, while still relying on human expertise for final validation and refinement.

7.1.1 Considerations on Task 1: Signal Mapping

Ensuring that the initial information is coherent and well-structured is essential for any problem. Task 1 was designed precisely to address a gap in the company's documentation, where the different architectures were not explicitly linked to one another. The signal mapping task achieved a highly significant result: the process of identifying the corresponding signal across two distinct architectures can be greatly simplified, reducing the uncertainty window from thousands of candidates to only a few. On the test set, in **75%** of the cases, the correct counterpart of an E^3 signal appears within the system's **TOP-5** retrieved candidates. Reaching this level of performance required several preprocessing and processing steps. Selecting the most informative descriptive fields and assigning weighted importance based on their relevance proved crucial, as did the combination of multiple techniques. In particular, **SBERT** and **SPLADE** enabled an efficient retrieval process that would otherwise be infeasible due to the size of the dataset. Representing each signal (composed of multiple technical fields) in both dense and sparse vector spaces allowed for retrieval that was not only fast but also semantically meaningful. The comparison between two signals could therefore incorporate their semantic information and rely on similarity computations that remain computationally manageable. It is also noteworthy that the integration of an LLM did not lead to further improvements. This suggests that, despite relying on relatively simpler tools, the resulting system is already highly optimized and difficult to outperform.

Future works and possible improvements To further improve and stabilize the results, a first enhancement should focus on the quality and richness of the available data. Enriching the initial dataset with more detailed signal descriptions, although challenging due to the large number of signals involved, would enable more

reliable mapping predictions, especially for subsets of signals that are highly similar to one another. Another promising direction is the collection of pairing heuristics, by investigating the conditions that make a mapping more reliable even when signals appear different at first glance. This, however, requires additional analysis and the involvement of a technical team with deep knowledge of the documentation and design logic of all architectures. Finally, further improvements could be achieved by strengthening the tools already used. Employing more advanced embedding models may enhance retrieval quality, particularly in the most ambiguous cases.

7.1.2 Considerations on Task 2: Alias Generation

The second task, focused on alias generation, highlighted how even an apparently simple problem may require a fairly complex architecture. The task combines a *creative* component, generating semantically meaningful keywords from the input descriptions, with a set of *strict rules* needed to produce a correctly formatted alias. What is essentially a "nickname" becomes challenging once constraints and formatting requirements are introduced. The presence of an official abbreviation dictionary required the use of an embedding model to efficiently search for matches starting from the generated keywords. Due to the size of the dictionary, including its entire content directly in a prompt would have been ineffective: it would exceed the context window and reduce the model's ability to maintain attention, increasing the likelihood of errors. Building a prompt chain was essential to break the problem into a sequence of simpler subtasks. This design made it possible to orchestrate the different phases (keyword generation, abbreviation retrieval, alias assembly) while also handling conflicts, for example when the LLM produced non-unique aliases. The iterative mechanism ensured that the final alias was both unique and compliant with the required conventions.

Future works and possible improvements As in the previous task, the most impactful improvements would concern the quality of the input data. Ideally, providing clearer and more detailed descriptions for each signal would enable the system to generate more accurate keywords and, consequently, more coherent aliases. Another promising direction involves the use of more advanced models, which could better interpret the technical content of each signal, handle a richer vocabulary and follow formatting rules more reliably. A further approach, inspired

by the strategy adopted in Task 3, is the use of data enrichment. Associating more narrative and descriptive text to each signal could help extract additional concepts, leading to a richer set of keywords. However, this method requires high-quality enrichment fields: as observed earlier, LLM-generated descriptions tend to be vague or occasionally hallucinated, which may introduce noise instead of improving the process.

7.1.3 Considerations on Task 3: JSON Test Catalogue Generation

Task 3 represents the core of this thesis. The creation of structured test files, used to validate automotive functionalities during simulation, is a time-consuming activity that requires both precision and deep technical expertise. Optimizing this process can significantly increase productivity and reduce the workload associated with such a complex task. The analysis focused on three different but complementary scenarios. The first investigated whether it is possible to automatically identify the set of signals containing the information needed to construct the test. The results showed that retrieval techniques based on *dense and sparse representations* (such as SBERT and SPLADE) can be effectively used for this purpose. Although full automation was not achieved, support techniques like data enrichment and query expansion proved valuable in improving performance. The other two scenarios focused on the actual generation of the JSON file, considering both a *guided* and a more *deductive* use case. The use of a **prompt chain** made it possible to break the process into smaller, more manageable steps. Through *prompt-engineering* strategies, excellent results were obtained in the guided scenario and encouraging results in the deductive one. The experiments showed that, when properly described, test operations can be *consistently translated* into the structured format used within the company. The results also indicate that, in guided scenarios, models of different sizes perform similarly. The most challenging case, however, required deducing all operations from a high-level description and the set of available signals. Here, the best results were achieved with the largest LLM, demonstrating that stronger reasoning capabilities, combined with a few examples, can already produce a useful *initial draft*. Although full automation is not yet feasible, the system can generate starting points in just a few seconds, enabling human operators to refine the output

without having to build the entire test from scratch.

Future works and possible improvements Beyond testing the system with more powerful models, a key improvement concerns the quality of the textual specifications provided as input. The excellent results obtained in the guided scenario, together with the promising outcomes in the deductive one, suggest that medium-detail descriptions may be sufficient to generate reliable initial drafts of the test. A more advanced architecture could also involve technical personnel directly in the process, allowing them to refine or adjust the draft through natural-language interactions. This human-in-the-loop approach would preserve expert oversight while benefiting from the speed of automation. Another promising direction is the development of a specialized model. In addition to providing more few-shot examples (the experiments used only two per subtask), it would be possible to fine-tune a generative model. This would enable the creation of a domain-expert assistant with more vertical capabilities and a deeper understanding of test structures and logical processes. However, this would require collecting a sufficiently large number of samples (likely in the hundreds) to enable an effective fine-tuning process. At present, the number of labeled test cases available is extremely limited, which represents a major constraint for training a specialized model. A larger and more diverse dataset would allow the model to better learn the structure of the tests, the underlying logical processes and the relationships between signals, ultimately improving the quality of the generated outputs. The model could also be trained on the technical documentation of the signals, giving it a clearer understanding of the available resources and constraints. In any case, external data support (for example through Retrieval-Augmented Generation) would be essential to reduce hallucinations and improve reliability.

Appendix A

Prompts

```
1  You are an assistant for an automotive company.
2  You will receive:
3    Keywords: {keywords}
4
5  Your task is:
6  Generate an abbreviation for each by following these rules:
7      1. Remove all vowels except those at the beginning or end of
       the word
8      2. Eliminate any repeated letters (no double letters)
9      3. For verbs: Replace the suffix "-ed" with "-d" and "-ing"
       with "-g", unless the root ends in "g", in which case use "-n".
10     4. The first letter is capital
11
12 Focus on abbreviations that are compact yet recognizable.
13 If the result is too difficult to read or sounds unclear,
       reintroduce minimal vowels.
14
15 Do NOT add any comments (do NOT use markdown). Return DIRECTLY the
       result formatted like that:
16 { "keyword1": "abbreviation1", ... }
```

Listing A.1: Prompt for abbreviation generation.

```
1  You are an expert assistant in automotive signal analysis.
2  Your role is to support the generation of aliases for pairs of
       equivalent signals coming from different network architectures
       or data sources.
```

```
3
4  Your task is:
5  - Identify the core concepts that best describe the shared meaning
       of the two signals, based on their names, descriptions, and
     comments.
6  - Focus on semantic meaning, not acronyms or protocol names.
7  - Extract as many as possible keywords you can.
8  - Convert each extracted concept into a single English keyword (
     one word per concept).
9
10 Return ONLY an rich array of s i n g l e word elements.
11 Do NOT use acronyms, special characters, or m u l t i word
     expressions.
12 Do NOT include any introductory text or explanations.
13
14 EXAMPLE
15 First Signal name: PEDAL_CAN::AccelPedalPosition
16 Description: Position of the accelerator pedal measured by the
     pedal position sensor.
17 Comment: CAN signal representing the d r i v e r s acceleration
     request.
18 Second Signal name: PEDAL_ETH::ThrottlePedalPos
19 Description: Throttle pedal position acquired from the position
     sensor and distributed via Ethernet.
20 Comment: Ethernet variant of the pedal position information.
21 ["pedal", "position", "acceleration"]
22
23 Now generate the keyword array using the following data:
24 First Signal name: {name1}
25 Description: {description1}
26 Comment: {comment1}
27 Second Signal name: {name2}
28 Description: {description2}
29 Comment: {comment2}
```

Listing A.2: Prompt for keywords generation

```
1  You are an assistant for an automotive company.
2
3  Your task is:
4    1. Combine the keywords and the unit into a single compact alias
       string (do NOT use special characters to combine them).
```

```
  2. Every alias starts with "AL_". Order the abbreviation in a
   logical way. The unit of measurement is always at the end.
  3. Do not necessarily use all keywords: only include the most
   relevant.
  4. Take care the alias is not in the conflicts examples (if any)
   .
  5. If no alternatives are possible except the conflicts, modify
   the given abbreviations.
  6. Do NOT return an alias in the conflicts list. Instead, use
   similar abbreviations.

Return ONLY the alias string in a JSON format.
Examples:
Keywords: Vehicle: Veh, Speed: Spd
Unit: Kph
Conflicts: []
{"alias": "AL_VehSpd_Kph"}

Keywords: Engine: Eng, Temperature: Temp, Oil: Oil
Unit: Celsius
Conflicts: ["AL_EngTemp_Celsius"]
{"alias": "AL_EngOilTemp_Celsius"}

Keywords: Battery: Batt, Charging: Chrg, State: Sts, Voltage: Volt
Unit: V
Conflicts: []
{"alias": "AL_BattChrgSts_V"}

### Now it's your turn:

Keywords: {keywords}
Unit: {unit}
Conflicts: {conflicts}
```

Listing A.3: Prompt for alias generation.

```
You are a Test Logic Architect.
Your objective is to write a "Reasoning" section for an automotive
    test case.

# TASK:
Synthesize the entire test logic into a cohesive narrative.
```

```
6
7  # INSTRUCTIONS:
8  1. Explain the "Why" and "How" of the test using the provided
       General Description and Signals.
9  2. You must use Signal Aliases (e.g., AL_Signal_Name) for all
       references.
10 3. Structure the text into: Test Intent, Global Conditions, and
       the 3 Pillars (Precondition, Action, Expected Results).
11 4. Describe the mathematical and logical flow clearly. Refer to
       the Operations Dictionary to ensure the verbal description
       matches the JSON logic (e.g., if using TrueDivisionOperator,
       describe it as a "True Division").
12 5. Keep the tone professional, technical, and concise.
```

Listing A.4: System prompt used in Task 3 (Detector Deduction) to produce a reasoning, starting from a general test description and a signal list

```
1  You are a Test Logic Architect.
2  Your objective is to write a "Reasoning" section for an automotive
       test case.
3
4  # TASK:
5  Synthesize the entire test logic into a cohesive narrative.
6
7  # INSTRUCTIONS:
8  1. Explain the "Why" and "How" of the test using the provided
       General Description and Signals.
9  2. You must use Signal Aliases (e.g., AL_Signal_Name) for all
       references.
10 3. Structure the text into: Test Intent, Global Conditions, and
       the 3 Pillars (Precondition, Action, Expected Results).
11 4. Describe the mathematical and logical flow clearly. Refer to
       the Operations Dictionary to ensure the verbal description
       matches the JSON logic (e.g., if using TrueDivisionOperator,
       describe it as a "True Division").
12 5. Keep the tone professional, technical, and concise.
```

Listing A.5: System prompt used for reasoning, in the JSON generator.

```
1  You are an automotive assistant specialized in generating
       structured test catalogues.
2
```

```
# TASK:
You will receive a list of aliases and each of them is characterized by couples of architecture name / real signal name.

# INSTRUCTIONS:
1. Extract the three types of information.
2. Produce a JSON object with the following structure:
{
  "signal_list": [
    {
      "alias": "<ALIAS>",
      "architecture": [
        {
          "name": "<ARCHITECTURE_NAME>",
          "signal_item": "<REAL_SIGNAL_NAME>"
        }
      ]
    }
  ]
}
3. Do not invent aliases or names. Extract only what is explicitly present in the input.
4. Output must contain ONLY the JSON object. No explanations, no comments, no markdown, no surrounding text.
```

Listing A.6: System prompt used for the signal list generation

```
You are a JSON Sequence Developer.
Your task is to write the first "test_steps" object where "step_type" is "precondition".

# INSTRUCTIONS:
1. Return a json object containing the precondition, with the operations explained in the section "PRECONDITION" of the input REASONING.
2. Mapping: Ensure all "input" signals used in the operations are declared in the "input" list of that specific step.
3. Operations Dictionary: Strictly follow the provided dictionary for operator naming and argument structure.

# STEP DATA STRUCTURE:
```

```
Each step object must contain:
- "step_number": A string index ("10", "20", "30" etc.).
- "step_type": "precondition".
- "type": Always "StandardTestStep".
- "operations": A list of operation objects.

# OPERATION OBJECT STRUCTURE:
Each operation must have:
- "name": A CamelCase string.
- "type": The Category from the Allowed Operations dictionary.
- "operator": The specific Operator from the Allowed Operations
    dictionary.
- "arguments": A dictionary where values can be signal names or
    numeric constants.

# OPERATIONS DICTIONARY:
{OPERATIONS}
```

Listing A.7: System prompt used for precondition generation, in the JSON generator.

Bibliography

[1] Mubashar Raza, Zarmina Jahangir, Muhammad Bilal Riaz, Muhammad Jasim Saeed, and Muhammad Awais Sattar. «Industrial applications of large language models». In: *Scientific Reports* 15.1 (2025), p. 13755. ISSN: 2045-2322. DOI: 10.1038/s41598-025-98483-1. URL: https://doi.org/10.1038/s41598-025-98483-1 (cit. on p. 1).

[2] Pradosh Kumar Gantayat, Tejinder Kaur, Madhusmita Majhi, Uttam Kumar Jena, Swatismita Das, and Soni. «From Efficiency to Innovation: LLM 5.0 and the Future of Industry». In: *2025 International Conference on Multi-Agent Systems for Collaborative Intelligence (ICMSCI)*. 2025, pp. 551–558. DOI: 10.1109/ICMSCI62561.2025.10894335 (cit. on p. 2).

[3] Seokjae Heo and Seunguk Na. «Ready for departure: Factors to adopt large language model (LLM)-based artificial intelligence (AI) technology in the architecture, engineering and construction (AEC) industry». In: *Results in Engineering* 25 (2025), p. 104325. ISSN: 2590-1230. DOI: https://doi.org/10.1016/j.rineng.2025.104325. URL: https://www.sciencedirect.com/science/article/pii/S2590123025004062 (cit. on p. 2).

[4] Porsche Engineering. *About Porsche Engineering - Your Global Technology Partner*. URL: https://www.porscheengineering.com/it/peg/about-us/ (cit. on p. 3).

[5] Porsche Engineering. *Nardò Technical Center*. URL: https://www.porscheengineering.com/it/nardo/ (cit. on p. 3).

[6] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. «A Neural Probabilistic Language Model». In: *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press,

2000. URL: https://proceedings.neurips.cc/paper_files/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf (cit. on p. 5).

[7] Aditi Mittal. *Understanding RNN and LSTM*. URL: https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e (cit. on p. 6).

[8] Ravjot Singh. *GRU Explained: The Simplified RNN Solution for Sequential Data*. URL: https://ravjot03.medium.com/gru-explained-the-simplified-rnn-solution-for-sequential-data-c706d0d149c5 (cit. on p. 6).

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (cit. on pp. 7, 9).

[10] Philip Gage. «A new algorithm for data compression». In: *The C Users Journal archive* 12 (1994), pp. 23–38. URL: https://api.semanticscholar.org/CorpusID:59804030 (cit. on p. 9).

[11] Mohammadamin Barekatain et al. *Foundational Large Language Models and Text Generation*. White paper. Google, 2025 (cit. on pp. 9, 11, 12, 14, 15, 17–25).

[12] Lee Boonstra. *Prompt Engineering*. White paper. Google, 2025 (cit. on pp. 24–26).

[13] Tongshuang Wu, Michael Terry, and Carrie J. Cai. «AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts». In: *CHI Conference on Human Factors in Computing Systems (CHI '22)*. New York, NY, USA: ACM, Apr. 2022, pp. 1–22. DOI: 10.1145/3491102.3517582. URL: https://doi.org/10.1145/3491102.3517582 (cit. on p. 26).

[14] Julia Wiesinger, Patrick Marlow, and Vladimir Vuskovic. *Agents*. White paper. Google, 2025 (cit. on pp. 27–29).

[15] Muhammad Arslan, Hussam Ghanem, Saba Munawar, and Christophe Cruz. «A Survey on RAG with LLMs». In: *Procedia Computer Science* 246 (2024). 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024), pp. 3781–3790. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2024.09.178. URL: https://www.

sciencedirect.com/science/article/pii/S1877050924021860 (cit. on p. 29).

[16] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401 (cit. on p. 29).

[17] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997 (cit. on pp. 31, 33–35).

[18] Yongjie Wang, Yue Yu, Kaisong Song, Jun Lin, and Zhiqi Shen. *When Retrieval Succeeds and Fails: Rethinking Retrieval-Augmented Generation for LLMs*. 2025. arXiv: 2510.09106 [cs.CL]. URL: https://arxiv.org/abs/2510.09106 (cit. on pp. 33, 34).

[19] Luyao Shi, Michael Kazda, Bradley Sears, Nick Shropshire, and Ruchir Puri. *Ask-EDA: A Design Assistant Empowered by LLM, Hybrid RAG and Abbreviation De-hallucination*. 2024. arXiv: 2406.06575 [cs.CL]. URL: https://arxiv.org/abs/2406.06575 (cit. on p. 35).

[20] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. «Blended RAG: Improving RAG (Retriever-Augmented Generation) Accuracy with Semantic Search and Hybrid Query-Based Retrievers». In: *2024 IEEE 7th International Conference on Multimedia Information Processing and Retrieval (MIPR)*. 2024, pp. 155–161. DOI: 10.1109/MIPR62202.2024.00031 (cit. on p. 36).

[21] Bassim Abdulbaqi Jumaa, Anwaar Mousa Abdulhassan, and Ammar Mousa Abdulhassan. «Advanced Driver Assistance System (ADAS): A Review of Systems and Technologies». In: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* (2019) (cit. on pp. 38, 39).

[22] R. Isermann, J. Schaffnit, and S. Sinsel. «Hardware-in-the-loop simulation for the design and testing of engine-control systems». In: *Control Engineering Practice* 7.5 (1999), pp. 643–653. ISSN: 0967-0661. DOI: https://doi.org/10.1016/S0967-0661(98)00205-6. URL: https://www.sciencedirect.com/science/article/pii/S0967066198002056 (cit. on p. 40).

[23] Kvaser. *About CAN: A Flexible and Powerful Protocol*. https://kvaser.com/about-can/ (cit. on p. 40).

[24] Kvaser. *Introduction to the LIN bus*. https://kvaser.com/about-can/can-standards/linbus/ (cit. on p. 41).

[25] National Instruments. *FlexRay Automotive Communication Bus Overview*. https://www.ni.com/en/shop/seamlessly-connect-to-third-party-devices-and-supervisory-system/flexray-automotive-communication-bus-overview.html (cit. on p. 41).

[26] Siemens. *Automotive Ethernet*. https://www.sw.siemens.com/it-IT/technology/automotive-ethernet/ (cit. on p. 41).

[27] Roger Biermann. *Audi's All-New Premium Platform Combustion Isn't As 'All New' As We Thought*. https://carbuzz.com/audi-premium-platform-combustion-not-all-new/ (cit. on p. 41).

[28] Audi MediaCenter. *The New Audi E3: 1.2 Electronic Architecture Brings "Vorsprung durch Technik" to Life*. https://www.audi-mediacenter.com/en/press-releases/the-new-audi-e3-12-electronic-architecture-brings-vorsprung-durch-technik-to-life-15925 (cit. on p. 41).

[29] Hugging Face. *MarianMT*. https://huggingface.co/docs/transformers/model_doc/marian (cit. on p. 47).

[30] Marcin Junczys-Dowmunt et al. *Marian: Fast Neural Machine Translation in C++*. 2018. arXiv: 1804.00344 [cs.CL]. URL: https://arxiv.org/abs/1804.00344 (cit. on p. 47).

[31] Sunny Dhokane, Chinmay Deshmukh, Akshit Bollabattin, Siddhesh Karande, Bhakti Karangale, and Pradip S. Varade. «BM25 Implementation For Information Retrieval: Candidate Shortlister For Recruitment Process». In: *2024 Intelligent Systems and Machine Learning Conference (ISML)*. 2024, pp. 722–727. DOI: 10.1109/ISML60050.2024.11007378 (cit. on p. 50).

[32] Andrew Trotman, Antti Puurula, and Blake Burgess. «Improvements to BM25 and Language Models Examined». In: *Proceedings of the 19th Australasian Document Computing Symposium*. ADCS '14. Melbourne, VIC, Australia: Association for Computing Machinery, 2014, 58–65. ISBN: 9781450330008. DOI:

10.1145/2682862.2682863. URL: https://doi.org/10.1145/2682862.2682863 (cit. on p. 50).

[33] dorianbrown. *rank_bm25.py*. URL: https://github.com/dorianbrown/rank_bm25/blob/master/rank_bm25.py#L96 (cit. on p. 50).

[34] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. «SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking». In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '21. Virtual Event, Canada: Association for Computing Machinery, 2021, 2288–2292. ISBN: 9781450380379. DOI: 10.1145/3404835.3463098. URL: https://doi.org/10.1145/3404835.3463098 (cit. on p. 52).

[35] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. *SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval*. 2021. arXiv: 2109.10086 [cs.IR]. URL: https://arxiv.org/abs/2109.10086 (cit. on p. 53).

[36] Carlos Lassance, Hervé Déjean, Thibault Formal, and Stéphane Clinchant. *SPLADE-v3: New baselines for SPLADE*. 2024. arXiv: 2403.06789 [cs.IR]. URL: https://arxiv.org/abs/2403.06789 (cit. on p. 53).

[37] Ollama. *Ollama*. URL: https://ollama.com/ (cit. on p. 63).

[38] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. «The Vector Space Classification». In: *Introduction to Information Retrieval*. Cosine similarity formula at p. 14 of the PDF (p. 121 of the book). Cambridge University Press, 2008, pp. 121–123. URL: https://nlp.stanford.edu/IR-book/pdf/06vect.pdf (cit. on p. 64).

[39] Hugging Face. *Chat Template*. URL: https://huggingface.co/learn/llm-course/chapter11/2 (cit. on p. 74).

[40] AUTOSAR. *AUTOSAR Keyword Abbreviation*. URL: https://www.abbretech.com/autosar/.