

DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale Ingegneria Informatica

Enhancing Fault Isolation in Hardware Systems Using Large Language Models (LLM)

Miglioramento dell'isolamento dei guasti nei sistemi hardware utilizzando modelli linguistici di grandi dimensioni (LLM)

Relatore:

Prof. Gianfranco Lombardo

Tesi di Laurea di:

Matteo Gianvenuti Matricola: 364979

Table of Contents

[Introduction 2](#_Toc202526641)

[1. State Of The Art 3](#_Toc202526642)

[1.1 Natural Language Processing (NLP) 3](#_Toc202526643)

[1.2 Large Language Models (LLMs) 4](#_Toc202526644)

[1.3 Self-Supervised Learning (SSL) 5](#_Toc202526645)

[1.4 Prompt Engineering 5](#_Toc202526646)

[1.5 Generative Pre-trained Transformer (GPT) 6](#_Toc202526647)

[1.6 Transformer 7](#_Toc202526648)

[1.7 Large Language Model Meta AI (Llama) 7](#_Toc202526649)

[1.8 Phi 7](#_Toc202526650)

[1.9 Hardware Diagnostics Papers with AI 7](#_Toc202526651)

[2. Method 8](#_Toc202526652)

[2.1 Preliminary activities 8](#_Toc202526653)

[2.1.1 Questions dataset 9](#_Toc202526654)

[2.1.2 Prompt structure 10](#_Toc202526655)

[2.1.3 Textual representation 11](#_Toc202526656)

[2.1.4 Json representation 12](#_Toc202526657)

[2.1.5 Tabular representation 13](#_Toc202526658)

[2.1.6 Model dimension limit 14](#_Toc202526659)

[2.1.7 Models results 16](#_Toc202526660)

[2.1.8 PoC conclusion 18](#_Toc202526661)

[2.2 Dataset 18](#_Toc202526662)

[2.2.1 Introduction 18](#_Toc202526663)

[2.2.2 Database structure 18](#_Toc202526664)

[2.2.3 System definition 20](#_Toc202526665)

[2.3 Scenarios 20](#_Toc202526666)

[2.4 ICL 20](#_Toc202526667)

[2.4.1 System prompt without few-shot 21](#_Toc202526668)

[2.4.2 System prompt with few-shot 21](#_Toc202526669)

[2.5 System Prompt Experiments 22](#_Toc202526670)

[2.6 Few-shot Experiments 22](#_Toc202526671)

[Bibliography 23](#_Toc202526672)

Lombardo’s structure:

Introduzione

- Ambito dell’attività di tesi

- Obiettivi dell’attività di tesi

Stato dell’arte

- Documentare tutto ciò che esisteva prima della tesi in termini di tecnologie, metodologie e modelli

2/3 capitoli di metodologia in cui spiega cosa ha effettivamente svolto, progettato e realizzato (architettura software, addestramento di modelli, dati utilizzati e loro composizione etc..)

- 1 capitolo su i risultati

- 1 breve capitolo di conclusioni in cui ricapitola il tutto traendo le conclusioni e proponendo degli sviluppi futuri.

# Introduction

This work thesis was developed during an internship program in MBDA ITALIA SPA. In this thesis work we want to explore the use of Large Language Models (LLMs) in hardware diagnostics. LLMs have shown good memorization and "understanding" capabilities of natural language, even in reasoning tasks. Especially in the case in which the models know the environment in which they operate.

In our case, we initially want to make the LLMs understand how a hardware architecture is structured so that the model is able to solve the diagnostics at its best. Subsequently, we want to exploit the capabilities of the LLMs to identify any problems in the hardware by analysing system logs, telemetry data and hardware functionality test results. To carry out these activities it was necessary to create a specific dataset and a simulation environment. We also implemented the Open WebUI graphical interface.

So, we have two objectives, the first is to verify that the LLMs are able to understand the hardware architecture. While the second is to verify that the LLMs are able to understand and solve hardware problems. This work was based on In-Context Learning (ICL) and Prompt Engineering. In particular, we compared the performance of different models according to various metrics.

# 1. State Of The Art

## 1.1 Natural Language Processing (NLP)

NLP is an interdisciplinary sub-branch of linguistics, computer science, and artificial intelligence (AI) that deals with the interaction between computers and natural language or human language. In particular, it uses algorithms to analyse and extract the content of documents in natural language with the aim of making a computer “understand” the content of documents, including the contextual nuances of the language, as well as classify and categorize the documents themselves.

This process is difficult and complex due to the intrinsic characteristics of ambiguity of natural language. In fact, NLP is considered a difficult problem in computer science because the “laws” that govern natural language are difficult for computers to interpret, in fact sometimes they can make mistakes and attribute incorrect meanings that lead to incorrect and difficult to understand results.

## 1.2 Large Language Models (LLMs)

LLMs are machine learning models with many parameters, designed for natural language processing (NLP) tasks, especially text generation. These models are typically built using deep neural networks, particularly transformer architectures, and are trained on massive corpora of textual data. They are trained with self-supervised learning on a vast amount of text. Their strength lies in the ability to capture complex syntactic, semantic, and contextual patterns, allowing them to perform a wide range of tasks such as translation, summarization, question answering, and reasoning.

LLMs can be fine-tuned for specific tasks or guided by prompt engineering, as done in this work. These models acquire predictive power regarding syntax, semantics, and ontologies inherent in human language corpora, but they also inherit inaccuracies and biases present in the data they are trained in.

The core idea behind LLMs is to learn a statistical model of language by predicting the next word (or token) in a sentence, given a context. This is achieved using self-supervised learning, where the model learns from raw text without the need for labelled data. During training, the model builds internal representations that encode meaning, structure, and dependencies across sequences of words.

The largest and most capable LLMs are generative pre-trained transformers (GPTs).

Modern LLMs like OpenAI’s GPT and DeepSeek’s DeepSeek series or Meta’s LLaMA models are characterized by having billions or even trillions of parameters. The performance of these models tends to improve significantly as the number of parameters and the size of the training data increase, a phenomenon often referred to as “scaling laws.”

In addition to linguistic capabilities, recent LLMs demonstrate emergent properties such as reasoning, commonsense inference, and even basic code generation. However, they also exhibit limitations, including sensitivity to prompt phrasing, hallucination of facts, and the presence of biases derived from the training data. So, in the context of training LLMs, datasets are typically cleaned by removing low-quality, duplicated, or toxic data. Cleaned datasets can increase training efficiency and lead to improved downstream performance.

As all the others machine learning (ML) algorithms, LLMs process numbers rather than text, the text must be converted to numbers through the tokenization process. Tokenization is a fundamental process in LLM pipelines. Before text is processed by a model, it is converted into tokens. In the first step, a vocabulary is decided upon, then integer indices are arbitrarily but uniquely assigned to each vocabulary entry, and finally, an embedding is associated to the integer index.

Most results previously achievable only by (costly) fine-tuning, can be achieved through prompt engineering, although limited to the scope of a single conversation (more precisely, limited to the scope of a context window). Due to the high training costs and the huge amount of data needed, we did not implement a fine-tune, instead we worked in the field of ICL. Also, because it is already sufficient to achieve our goals.

In our thesis work, LLMs serve a dual role: first, as systems capable of understanding the architecture and components of hardware environments through descriptive prompts; second, as tools capable of interpreting system logs and telemetry to support fault diagnosis.

## 1.3 Self-Supervised Learning (SSL)

SSL is a paradigm in machine learning where a model is trained on a task using the data itself to generate supervisory signals, rather than relying on externally provided labels. In the context of neural networks, self-supervised learning aims to leverage inherent structures or relationships within the input data to create meaningful training signals. SSL tasks are designed so that solving them requires capturing essential features or relationships in the data.

Auto-associative self-supervised learning is a specific category of SSL where a neural network is trained to reproduce or reconstruct its own input data. In other words, the model is tasked with learning a representation of the data that captures its essential features or structure, allowing it to regenerate the original input.

## 1.4 Prompt Engineering

Prompt engineering is the process of structuring or crafting an instruction in order to produce the best possible output from a generative AI model.

In case of text-to-text language models a *prompt* is natural language text describing the task that an AI should perform. It can be a query, a command, or a longer statement including context, instructions, and conversation history. Prompt engineering may involve phrasing a query, specifying a style, choice of words and grammar, providing relevant context, or describing a character for the AI to mimic.

There are many different types of prompt engineering:

* Chain-of-Thought (CoT): it is a reasoning technique that goes through several intermediate steps before giving a definitive answer to a question.

The CoT prompting improves reasoning ability by inducing the model to answer a multi-step problem with steps of reasoning that mimic a train of thought. These techniques were developed to aid common sense reasoning and multi-step reasoning.

The prompt specifies some examples of questions and answers (Q&A); this improved the results considerably. Even just writing "let's think step by step" improved the results a lot. Allowing you to generalize and avoid writing many specific examples that consume the context.

* In-Context Learning (ICL): it is the ability of models to learn information temporarily. The information of interest is specified in the prompt, typically in the initial part. This information is then used by the model in the session to answer questions by generating text, this allows the model to be trained without necessarily having to perform fine-tuning.

The weights of the network are not updated, so the information is temporary. Starting a new instance the model does not retain this knowledge.

* Prompting to estimate model sensitivity: research consistently demonstrates that LLMs are highly sensitive to subtle variations in prompt formatting, structure, and linguistic properties. Linguistic features significantly influence prompt effectiveness—such as morphology, syntax, and lexico-semantic changes—which meaningfully enhance task performance across a variety of tasks.

To address sensitivity of models and make them more robust, several methods have been proposed. Format Spread facilitates systematic analysis by evaluating a range of plausible prompt formats, offering a more comprehensive performance interval. Similarly, Prompt Eval estimates performance distributions across different prompts, enabling robust metrics such as performance quantiles and accurate evaluations under constrained budgets.

* Retrieval-augmented generation (RAG): it is a technique that enables generative AI models to retrieve and incorporate new information. It modifies interactions with a LLM so that the model responds to user queries with reference to a specified set of documents, using this information to supplement information from its pre-existing training data. This allows LLMs to use domain-specific and/or updated information.
* Few-shot: the model is already pre-trained and fine-tuned to become instruct. Few-shot means that in inference in the prompt are put from zero to 4/5 task examples before the request for an actual task.

There are many other techniques, but they are not relevant to this work. In this work we mainly focused on ICL and CoT, building a customized few-shot technique.

## 1.5 Generative Pre-trained Transformer (GPT)

A GPT is a type of LLM and a prominent framework for generative AI. It is an artificial neural network that is used in NLP by machines. It is based on the transformer deep learning architecture, pre-trained on large datasets of unlabelled text, and able to generate novel human-like content. As of 2023, most LLMs had these characteristics and are sometimes referred to broadly as GPTs.

The first GPT was introduced in 2018 by OpenAI. OpenAI has released significant GPT foundation models (models trained on board data at scale such that they can be adapted to a wide range of downstream tasks) that have been sequentially numbered, to comprise it “GPT-n” series. Each of this was significantly more capable than the previous, due to increased size (number of trainable parameters) and training. Such models have been the basis for their more task specific GPT systems, including models fine-tuned for instruction following.

GPT is based on the transformer architecture.

## 1.6 Transformer

## 1.7 Large Language Model Meta AI (Llama)

Llama is a family of LLMs released by Meta AI. In the last years Meta AI released many models with various parameters size. Furthermore, they released also fine-tuned versions (instruct). The Llama model demonstrated to be stronger than many other models in NLP tasks.

The Llama model architecture is autoregressive decoder-only transformer, similar to GPT-3 with minor differences in activation function, embeddings, and optimizer.

## 1.8 Phi

## 1.9 Hardware Diagnostics Papers with AI

In this chapter, I analyse the SOTA papers about the diagnostics and log analysis. I mainly analysed four papers and articles.

* “LLM4SecHW: Leveraging Domain-Specific Large Language Model for Hardware Debugging” (paper 2024): This paper realizes, a fine-tune of various models to help in identifying hardware bugs during the design phase. The fine-tune is based on a dataset built on a GitHub repository of machine code.

This system has two workflows, one to identify a bug and another to generate a possible solution. The solution is compared to the one in the dataset, through the Rouge-N F1 score. Since the Rouge metric cannot capture all the LLMs skills, the authors manually evaluated the models replies. The authors experimented fine-tuned and not fine-tuned models. The fine-tuned models have more specific information then they were able to complete the authors tasks in all cases while the not fine-tuned in some cases affirmed, they were unable to complete their tasks, in other cases they just identified the problem without proposing a solution.

In conclusion, with more data is possible to reach better results.

* “[Structuring Unstructured Data with LLMs | by Story\_Teller | Medium](https://medium.com/@vivekvjnk/structuring-unstructured-data-with-llms-13a19810174d)” (article 2024): LLMs thanks to their ability of understanding unstructured data can process raw input and transform it in structured output prompts.
* “FD-LLM: Large Language Model for Fault Diagnosis of Machines” (paper 2024): The method proposed in the paper, pre-process the raw signals data before
* “LogGPT: Exploring ChatGPT for Log-Based Anomaly Detection” (paper 2023):

# 2. Method

## 2.1 Preliminary activities

The purpose of these firsts activities is to evaluate the ability of LLMs in understanding the hardware structure of a sample system and its interconnections, and to determine whether representation of the system architecture is more effective in making the model understand the system architecture. This is therefore an initial proof of concept (PoC).

I have experimented three system architecture representations: textual, JSON and tabular representation of the network architecture. All these representations are always in text format since LLMs only understand text.

We have experimented with various models according to the available hardware capacity for this phase, especially meta-llama/llama3.2-3B-Instruct and microsoft/Phi-4-mini-reasoning.

For this phase I had an NVIDIA GA104GLM RTX A3000 Mobile GPU with 6GB of memory. Consequently, I had to perform the tests with small models and the use of 8-bit and 4-bit quantization.

To test the models on the understanding of the connection topology (the architecture of the System) I created a dataset of 94 questions, in which it is asked what is connected from port/interface A to B, and vice versa. From this dataset I extract a subsample of questions to ask the model to verify if it has "understood" and remembers the architecture. Since the GPU used in this phase has only 6GB of memory, it is not possible to ask all the questions together.

Since the LLMs answer open questions with generally always different and long text, I taught the model (specifying rules in the system prompt) to answer by indicating only the name of the connected component. In this way it is possible to automatically parse the answers to evaluate the accuracy.

The system prompt consists of a short description of the task that the model must perform and its role, a set of rules on the representation of the architecture and the names of the devices, a list of the devices, the representation itself in its format and finally some examples of questions and answers. This PoC is therefore based on a combination of the prompt engineering techniques shown in the previous chapter.



### 2.1.1 Questions dataset

I manually built a dataset with 94 questions and corresponding answers about the system interconnections topology. The dataset contains questions like “What is connected to A?” for all ports and interfaces of all devices. So, if A is connected to B there is the dual question (the answer is B when asking about A and vice versa). In the dataset there are also few questions about components that do not exists. The responses are of the type "The C is connected to D", "Nothing" if there is nothing connected or "The interface/port D does not exist for device G".

To calculate the accuracy, a subset of questions is randomly extracted and asked to the model. Due to GPU memory limitations, I could not ask the models all questions at once.

I calculate the accuracy in reporting the asked questions and in correctly responding to questions. All the models can report the questions (the accuracy was always 100%), so they know the request but in answering/reasoning they had more difficulties.

### 2.1.2 Prompt structure

The prompt is specific for each type of representation. At the beginning of the prompt is defined the model task. In Json and tabular representations, the task description varies slightly as I specify what type of representation is shown.



In the successive part initially, I put the components definitions



After some experiments I removed the definitions because it was not helpful in interconnections understanding and the size of the prompt matter.

In the successive part there are the naming conventions and other rulers depending on the representation and the representation itself.

In the last part of the prompt, I put some Q&A examples. This teaches the model to respond in the specified format. At this point of the work, it is mainly useful to parse the answers and automatically evaluate the accuracy.

Not all models clearly understand that they must respond in the specified way. So, I added a parenthetical note to the prompt to clarify it, but models like mistralai/Mistral-7B-Instruct-v0.3 sometimes still do not follow it, while meta-llama/llama3.2-3B-Instruct already understood it correctly from the start.

Here the last part of the prompt, notice that the last generalized sentence allows you to narrow down the space of possible output values ​​and therefore narrow the field to just your domain of interest. This is powerful.

A screenshot of a computer program

AI-generated content may be incorrect.

### 2.1.3 Textual representation

In this representation the system interconnections are described in natural language. Before the description of the system, I put some naming rules and a list of all devices.



Here the system representation:



### 2.1.4 Json representation

In this representation I described the network and its components as Json objects. I introduced this representation in the prompt as there:

A screenshot of a computer program

AI-generated content may be incorrect.

Here an example of one component in Json format, the switch-1:

A screen shot of a computer

AI-generated content may be incorrect.

### 2.1.5 Tabular representation

As in all the other representation there is a part with rules and the components list:



After the system architecture topology is in a tabular format. The table is manually hardcoded, since it must be text:

A screenshot of a computer program

AI-generated content may be incorrect.

### 2.1.6 Model dimension limit

The tests for this part have been done with meta-llama/llama3.2-3B-Instruct model and the textual representation.

In the first experiments I noticed most errors were caused by a similar nomenclature in multi-device components (e.g., Switch-1, Switch-2, Swtich-3). In questions like “What is connected to the Workstation-1-eth0?” the model answered more than one time like “The Switch-3-port-2” while the correct answer was “The Switch-2-port-2”. Then to better highlight the number that makes the devices different I tried a different representation by separating the device name and number from the port/interface with the colon symbol (‘:’).

With that names representation I did the following experimental test with twenty questions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Device name format | Run 1 | Run 2 | Run 3 | Mean |
| Colon | 85.0% (17/20) | 70.0% (14/20) | 100.0% (20/20) | 85.0% |
| Dash | 90.0% (18/20) | 87.5% (17.5/20) | 100.0% (20/20) | 92.5% |

After this test I did the tests in the section [2.1.7](#_2.1.6_Models_results) (I expanded the context, so more questions, resulting in a worsening of accuracy). With the [2.1.7](#_2.1.6_Models_results) tests it became clear that the problem was due to the size of the model also because the model did not answer correctly to other questions (like “What is connected to ...?”) where the answer was clearly defined in the prompt where the topology is defined. Indeed, a smaller size model has less storage/reasoning capacity especially with a “large” context. To empirically demonstrate this, I did another experiment: I reduced the size of the system (maintaining more questions) resulting in an improvement.

Information that is already available from scientific literature (papers) on diagnostics. However, the company wanted to experiment with it. I also feed the system architecture to larger models like GPT-4 and DeepSeek to verify it again. Obviously, these models correctly understand and answer questions.

Here the smaller system architecture:

A diagram of a computer system

AI-generated content may be incorrect.

To calculate the accuracy for the experiment two I defined a smaller dataset with 34 questions (to test all the interconnections of a smaller system). I feed to the model all the 34 questions for five runs. For simplicity I used only the “dash” representation for device names, also because as I said before it does not really matter. Here the results of the experiment:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean |
| 88.2% (30/34) | 94.1% (32/34) | 91.2% (31/34) | 97.1% (33/34) | 94.1% (32/34) | 92.94% |

### 2.1.7 Models results

#### 2.1.7.1 meta-llama/llama3.2-3B-Instruct

I tested this model with 8-bit double quantization.

This model has been tested on all the representation. In few runs the model did not answer all the questions, so I ran it again in those cases.

In the following table the accuracy in answering the 30 questions with the textual representation:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Device name format | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean |
| Colon | 86.7% (26/30) | 83.3% (25/30) | 73.3% (22/30) | 73.3% (22/30) | 86.7% (26/30) | 80.66% |
| Dash | 80.0% (24/30) | 83.3% (25/30) | 73.3% (22/30) | 83.3% (25/30) | 90.0% (27/30) | 81.98% |

In the following table the accuracy in answering the 30 questions with the JSON representation:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Device name format | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean |
| Colon | 80.0% (24/30) | 93.3% (28/30) | 83.3 (25/30) | 80.0% (24/30) | 76.7% (23/30) | 82.66% |
| Dash | 63.3% (19/30) | 83.3% (25/30) | 83.3% (25/30) | 73.3% (22/30) | 90.0% (27/30) | 78.64% |

In the following table the accuracy in answering the 30 questions with the tabular representation:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Device name format | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean |
| Colon | 63.3% (19/30) | 70.0% (21/30) | 66.7% (20/30) | 80.0% (24/30) | 93.3% (28/30) | 74.6% |
| Dash | 86.7% (26/30) | 80.0% (24/30) | 50.0% (15/30) | 53.3% (16/30) | 63.3% (19/30) | 66.66% |

The tabular representation has the worst performances.

#### 2.1.7.2 microsoft/Phi-4-mini-reasoning

I tested this model with 8-bit double quantization.

This model is focused on reasoning, it has the best performances, but it is a bit bigger (3.84B parameters) than the used llama so after two test with 30 and one with 25 questions the hardware stopped working (always out of memory errors).

Since it is focused on reasoning was better able to understand logical relationships and answer correctly.

In this table I report the results of the few tests (dash textual representation):

|  |  |  |  |
| --- | --- | --- | --- |
| Run 1 | Run 2 | Run 3 | Mean 30 quests |
| 96.7%  (29/30) | 90.0%  (27/30) | 84.0%  (21/25) | 93.35% |

This model is 2x or 3x slower than the llama used due to the reasoning phase. In most other cases it answered but only to 15-20 of 25-30 questions.

#### 2.1.7.3 mistralai/Mistral-7B-Instruct-v0.3

I tested this model with 4-bit double quantization, the hardware did not support the 8-bit quantization. I reduced also the number of questions to 25.

The model struggles to follow the Q&A output format, often ignoring the prompt and answering however it wants. I had to run it multiple times before getting the responses in the desired format. Its poor performance is likely due to quantization or hardware limitations.

In the following table the answers accuracy with the dash textual representation, the hardware didn't hold up with the other representations:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean |
| 76.0% (19/25) | 64.0% (19/25) | 72.0% (18/25) | 72.0% (18/25) | 92.0% (23/25) | 75.2% |

### 2.1.8 PoC conclusion

The hardware available for this phase allowed only these experiments to be performed, however quantizing below 16-bit generally makes neural networks decidedly unstable and extensively imprecise. Despite this, the results obtained have an average accuracy of around 80-90 percent. So, the PoC was successful even if carried out in very precarious conditions. Surely, as expected, having more resources it is possible to make the model “understand” enough the system architecture to then carry out the diagnostics.

## 2.2 Dataset

### 2.2.1 Introduction

In MBDA the diagnostics activity is almost completely based on the HMI software. That application allows users to automatically test a system architecture through a system definition file that contains a list of the system components and tests to perform.

The HMI application save the tests results on a database. It allows also to save the data in pdf or html format.

### 2.2.2 Database structure

The database of the HMI is composed by six tables with the following scheme:

* acceptanceTest (id, begin, end, open)
* generalInformations (id, infoLabel, infoData): this table contains information about the operator
* note (id, session, testId, infoData): this table contains only notes
* notifications (id, type, testID, session, timestamp, testStatus, testResult, error, updateType, updateValue, report):

this is the most important table, because it contains the notifications exchanged between the agents that perform the tests and the tested system components. The most important fields of that table are these:

* id: integer that represents notification arrival order. It is the primary key with autoincrement
* type: integer that indicates the notification type. It can be 0 = Unknown, 1 = Init, 2 = Running, 3 = Update, 4 = Terminated, 5 = Stopped, 6 = Deleted and 7 Error
* testID: string that uniquely identifies the test of the notification. It is the same of testCode filed of the table testIdVocabulary where is located the corresponding label, it is also the test identifier in the system definition
* seission: integer that represents the ID of the test session or process. So, the number of test executions
* timestamp: hour and date of the notification
* testStatus: integer that represents the actual test state. It can be 0 = Default, 1 = Init, 2 = Running, 3 = Terminated, 4 = Stopped, 5 = Deleted and 6 Error
* testResult: Integer that represents the final test result. It can be 0 = Default, 1 = NotExecuted, 2 = Successful and 3 = Fail
* error: description of an eventual error. It can be empty (the default value is the dash ‘-‘)
* report: This is the most important field, it contains the results of the test and all its subtests
* sqlite\_sequence (name, seq): this table contains the number of instances for each other table
* testIdVocabulary (id, testCode, testLabel, subsystemId, subsystemLabel): this table contains the corresponding names/labels for the system codes/tags.

The most important table is “notifications”, because it contains the notifications exchanged between the agents that perform the tests and the tested system components. The most important fields of that table are these:

* id: integer that represents notification arrival order. It is the primary key with autoincrement
* type: integer that indicates the notification type. It can be 0 = Unknown, 1 = Init, 2 = Running, 3 = Update, 4 = Terminated, 5 = Stopped, 6 = Deleted and 7 Error
* testID: string that uniquely identifies the test of the notification. It is the same of testCode filed of the table testIdVocabulary where is located the corresponding label, it is also the test identifier in the system definition
* seission: integer that represents the ID of the test session or process. So, the number of test executions
* timestamp: hour and date of the notification
* testStatus: integer that represents the actual test state. It can be 0 = Default, 1 = Init, 2 = Running, 3 = Terminated, 4 = Stopped, 5 = Deleted and 6 Error
* testResult: Integer that represents the final test result. It can be 0 = Default, 1 = NotExecuted, 2 = Successful and 3 = Fail
* error: description of an eventual error. It can be empty (the default value is the dash ‘-‘)
* report: This is the most important field, it contains the results of the test and all its subtests

In this work thesis I focused on the last field for the notifications with type 4 (a successfully completed test with result pass/fail) and type 7 (error case).

I modified the base database structure by adding two fields reportLabel and errorLabele. These fields contain a reduced version of the same data of the corresponding fields report and error to reduce the context for the ICL activity.

### 2.2.3 System definition

The system definition XML file is composed by a tree structure. Inside the “Diagnostics” root element there are the following elements:

* System: defines the system components
* Agents: define the agent testers which perform the tests
* TestCategories, MonitorCategories, StressCategoires: contains a set of test categories for the following elements
* testList: contains all the tests to be performed interactively (a human operator must manually start/stop them)
* MonitorList: contains a set of tests that are iteratively repeated if started by a human operator. In this case it is empty
* StressList: contains a set of tests that can be iteratively executed without the need of a human operator, they can be monitored by MonitorList. In this case it is empty

## 2.3 Scenarios

For the sample system we defined three scenarios:

* In the first scenario all the tests are successful
* In the second scenarios, the switch 2 is completely broken so all the tests on it and on the linked devices fail
* In the third scenario, the SBC2 is completely broken and so all the tests on it fails

For all the three scenarios we have defined a configuration file, a database with the described structure, a pdf report with a summary of all the executed tests and their results, finally various system prompt with and without few-shot.

The configuration file allows to run the tests in a simulation environment without the need of physical hardware.

## 2.4 ICL

This part of the thesis work is the most important, it consists in teaching to the model how to do a diagnostics task. So, how to analyse the test results and how to infer to identify the causes of any possible issue. For this part of the work, I experimented various system prompt with and without few-shot.

### 2.4.1 System prompt without few-shot

In this prompt I described:

* At the beginning, the model role and briefly its task
* In the successive part I explained the rules to interpretate the test results. In particular, the hash ‘#’ near to a test result mean that the test failed (the result is different from the expected), the star ‘\*’ mean that the test can have any result. I put also some examples to let the model understand the pattern. At the end of that section, I put an explanation about the result for a test which is ‘PASS’ if all the subtests are successful and ‘FAIL’ if even just one fail
* In the next part there is the format I decided to use to show each test and its information. I used a Python script to retrieve the tests data from the database and format them
* At the end of the prompt, there is an explanation about its task that is analyse the test results and find the causes of any possible issue

Before arriving to this prompt version, I did many other system prompts with more details, but they led to a worsening in results. The model was unable to understand its task and correctly answer the questions.

Instead, with this prompt the examples are feed to the model as user input (few-shot). This prompt had the best results in replying to questions.

### 2.4.2 System prompt with few-shot

For each scenario, I realized a system prompt with one-shot. The prompt contains:

* A little description of the model task
* The sample system components with their ports/interfaces status. The status can be ‘up(1)’, ‘down(2)’ or ‘Unknown’
* A textual description of the network topology (the system architecture)
* The tests interpretation rules.
* A description of the tests format
* All the tests with their results in that format
* Finally, the most important part, an example of answer. It is a logical deduction and a results analysis

This prompt is widely longer than the others without examples. All this information overhead the model and lead it to unrelated replies. With two-shot the results are worse, increasing the prompt size results in a performance degradation.

These experiments and colleague discussions lead to define a system prompt without few-shot included.

## 2.5 System Prompt Experiments

I experimented with various system prompts. At the beginning, I tested the system prompt with few-shot included with one example. With this system prompt the model was unable to correctly answer the questions. In particular, the model did not identify the failed tests in many cases, consequently the model did not find the problems and their possible causes.

Subsequently, I experimented with a short system prompt without few-shot. This prompt has the best performances in replying to questions. In many cases, the model correctly identified the issue and their possible causes. Since the model did not identify the correct solution in all the cases, I tried to improve the prompt.

Then, I created another longer system prompt with an explanation of how to logically identify any issue. But this prompt led to worse performance. In fact, the model was again unable to identify the failed tests. So, I decided to maintain the previous prompt.

## 2.6 Few-shot Experiments

I experimented two ways to do few-shot. One where I put the example and its solution directly in the system prompt and another where I put the example with its solution in the user prompt. I composed this prompt, with a first part where I briefly repeated the model and the network components and their ports/interfaces status, and a second part where I put all the tests with their results. For this method I used the short system prompt, that led to better performance, as I said before.

# Bibliography

[] NLP (Online) <https://en.wikipedia.org/wiki/Natural_language_processing>.

[] LLMs. (Online) <https://en.wikipedia.org/wiki/Large_language_model>.

[https://web.stanford.edu/~jurafsky/slp3/](https://eur01.safelinks.protection.outlook.com/?url=https%3A%2F%2Fweb.stanford.edu%2F~jurafsky%2Fslp3%2F&data=05%7C02%7Cmatteo.gianvenuti%40studenti.unipr.it%7C3eb6fbcc055f4e8a0aff08dd8bd29bee%7Cbb064bc5b7a841ecbabed7beb3faeb1c%7C0%7C0%7C638820458860462928%7CUnknown%7CTWFpbGZsb3d8eyJFbXB0eU1hcGkiOnRydWUsIlYiOiIwLjAuMDAwMCIsIlAiOiJXaW4zMiIsIkFOIjoiTWFpbCIsIldUIjoyfQ%3D%3D%7C0%7C%7C%7C&sdata=R76CP%2Fvpd1I40a8MReCrxBrrjUE6ColECYnknFGpswY%3D&reserved=0) book various arguments (it should cover almost all)

[] SSL (Online) <https://en.wikipedia.org/wiki/Self-supervised_learning>.

[] Prompt engineering (Online) <https://en.wikipedia.org/wiki/Prompt_engineering>.

[] Transformer <https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)>.

[]