Exploring the use of LLMs for Web Scraping

João Cardoso, No. 50465, a50465@alunos.isel.pt, tel.: 933840873

Francisco Antunes, No. 50497, a50497@alunos.isel.pt, tel.: 962225712

Rúben Said, No. 47526, a47526@alunos.isel.pt, tel.: 963756360

Supervisor: Eng. Paulo Pereira, e-mail: paulo.pereira@isel.pt

June 9, 2025

Contents

1 Abstract

Companies like Tryp.com face significant challenges when automating interactions with third-party websites due to frequent semantic changes in web structures. Traditionally, these tasks are handled through web scraping techniques; however, these approaches are fragile and highly susceptible to failure, especially given the dynamic nature of modern web pages. This project explores the application of Large Language Models (LLMs) and generative AI as decision-making components in the automation process.

The goal is to develop an adaptive and resilient system capable of improving the robustness of web scraping and automating corrective actions when failures occur. The proposed solution involves a software agent that mimics human behavior to perform actions on changing web platforms more effectively and autonomously.

2 Introduction

Automating actions on third-party websites has become increasingly important for companies like Tryp.com, which rely on reliable data extraction and interaction with dynamically changing platforms. This process is commonly carried out using web scraping — a technique that involves programmatically extracting data from websites. However, due to frequent structural and semantic changes on modern websites, traditional scraping tools often break, requiring constant maintenance and manual intervention.

The fragility of these scraping systems poses a significant challenge, especially in environments where scalability and adaptability are crucial. Minor changes in the HTML structure of a webpage—such as altered class names or reorganized elements—can render a scraper ineffective. As a result, companies are forced to invest time and resources into continuously updating their scraping logic.

In light of these limitations, this project explores a novel approach that leverages Large Language Models (LLMs) and generative AI to build more resilient, adaptive scraping systems. By treating scraping failures as reasoning problems rather than hardcoded logic errors, we aim to develop a system that can analyze the failure context, infer the intended behavior, and autonomously generate corrected code. This approach mimics how a human developer would debug and repair a broken scraper.

Due to the inherent costs associated with large-scale, deployed LLMs—such as ChatGPT—one of the main focuses of this project is to determine whether similar results can be achieved using smaller, locally deployed models.

3 Key Concepts

3.1 Natural Language Processing (NLP)

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human language. It enables machines to understand, interpret, generate, and respond to text or speech in a way that is meaningful and useful. NLP combines linguistics, computer science, and machine learning to handle tasks like language translation, sentiment analysis, speech recognition, and text summarization.

3.2 Large Language Models (LLMs)

LLM (Large Language Model) is a type of artificial intelligence model designed to understand and generate human-like language. It is trained on massive amounts of text data and uses deep learning, especially transformer architectures, to perform tasks such as text completion, translation, summarization, question answering, and conversation. LLMs, like GPT (Generative Pretrained Transformer), can adapt to various language-based tasks without task-specific training.

3.3 Prompt

A prompt is a piece of text or instruction given to a large language model (LLM) to guide its response. It serves as the starting point for the model's output, shaping how it interprets and generates language. Prompts can be simple questions, detailed instructions, or even examples of desired output. The effectiveness of a prompt often determines the relevance and accuracy of the model's response, making prompt design a key skill in using LLMs effectively.

3.4 Transformer Architecture

Transformer architecture is a type of deep learning model introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017.[6] It revolutionized natural language processing by enabling models to process entire sequences of data simultaneously rather than step-by-step like previous models (e.g., RNNs). The core innovation of the transformer is the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence, regardless of their position. This makes transformers highly effective for understanding context and relationships in language. They form the foundation of modern LLMs like GPT, BERT, and others.

3.5 Context Window

A context window in the realm of large language models refers to the maximum amount of text (measured in tokens) that the model can "see" or consider at once when generating a response. This includes both the input (prompt) and the output generated so far. Tokens are chunks of words—like words, subwords, or characters—depending on the model's tokenizer.

3.6 Parameter Sets

In the context of language models, a parameter set refers to the number of learnable weights the model contains—essentially, the model's internal "knowledge capacity." Larger parameter sets generally allow models to capture more complex patterns, reason more effectively, and perform better on a wider range of tasks. However, they also require more computational resources to run and may introduce latency or cost constraints, especially in real-time or local deployments. Smaller parameter sets, while less powerful, are more efficient and can often deliver adequate performance for targeted or simpler tasks. Comparing models across different parameter sizes helps evaluate trade-offs between performance, cost, and deployment feasibility.

4 Known Challenges in the use of LLMs

4.1 Task Specialization

While large language models (LLMs) exhibit strong generalization across a broad range of tasks, they often face challenges in achieving high performance within specialized domains. [1]

This limitation stems from the fact that LLMs are typically trained on diverse, general-purpose datasets that may lack the depth and specificity required for niche areas such as legal analysis, biomedical research, or advanced engineering. As a result, LLMs may produce outputs that are factually incorrect, overly generic, or fail to meet the rigorous standards expected in expert-level tasks. This highlights the need for targeted adaptation techniques.

4.2 Hallucinations

LLM hallucination refers to the generation of content by a large language model that is factually incorrect, logically inconsistent, or unsupported by available data or context. These outputs may appear plausible or well-formed linguistically, but diverge from objective truth or known information. Hallucination can occur in various forms, such as fabricated facts, incorrect citations, or misleading reasoning, and poses a significant challenge for the reliability and trustworthiness of LLM-generated responses, particularly in high-stakes or domain-specific applications.

There have been many works that attempt to reduce the extent of hallucination. These efforts have mostly been empirical so far, which cannot answer the fundamental question whether it can be completely eliminated. [3]

This limitation arises in part from the inherent variability in the generation process and the probabilistic nature of LLMs. While LLMs are capable of learning complex patterns from data, theoretical results from computational learning theory demonstrate that no model can learn all computable functions. Consequently, errors and hallucinations are unavoidable, particularly when LLMs are used as general-purpose problem solvers.

4.3 Multi-step reasoning

Despite their advanced capabilities, LLMs often struggle with tasks that require complex, multi-step reasoning.

A study titled *"Faith and Fate: Limits of Transformers on Compositionality"* [2] found that LLMs tend to reduce multi-step reasoning to linear patterns without truly developing systematic problem-solving skills. Additionally, research has shown that as the number of reasoning steps increases, the performance of LLMs significantly drops, indicating difficulties in maintaining consistency and accuracy over extended reasoning chains.

5 An overview of Model Adaptation Techniques

5.1 Definition

Model Adaptation Techniques refer to strategies employed to tailor LLMs to specific tasks or domains, enhancing their performance and relevance without necessitating training a new model from scratch.

These techniques leverage the foundational knowledge embedded in pre-trained models and adjust them to meet particular requirements.

5.2 Retrieval Augmented Generation

5.2.1 What is RAG?

Large Language Models are trained on vast amounts of data and use billions of parameters to generate original output for tasks such as answering questions, translating languages, and completing sentences. However, they cannot access real-time information or domain-specific knowledge that falls outside their training data. Furthermore, LLMs are prone to generate inaccurate or fabricated content, commonly referred to as **Hallucinations**

Retrieval-Augmented Generation, also known as RAG, addresses these limitations. RAG enhances LLM output by retrieving relevant information from an external, authoritative knowledge base before generating a response.[4] This approach improves factual accuracy, reduces hallucinations, and enables domain-specific applications without the need for costly model retraining.

5.2.2 How it works

Without RAG, an LLM generates responses based solely on the data it was trained on, which may be outdated or lack domain-specific context. With RAG, however, an information retrieval component is introduced. This component uses the user's prompt to fetch relevant content from external sources—such as vector databases or the internet. The LLM then combines this retrieved information with its existing knowledge to generate a response that is more accurate, up-to-date, and contextually relevant than what it could produce using training data alone.

5.3 Fine Tuning

5.3.1 What is Fine-tuning?

Fine-tuning in large language models is the process of adapting an already trained model to perform a specific task with improved efficiency and precision. Rather than building a new model from scratch, fine-tuning leverages an existing, pre-trained LLM and refines its capabilities by training it on a task-specific dataset. This makes the model more suited to solve particular problems, while still utilizing the broad knowledge it learned from the original training.

5.3.2 How it works

Fine-tuning starts with a model that has already been trained on a large, general dataset - typically understanding grammar, sentence structure and general knowledge, but they may not excel at specialized tasks. A task-specific dataset containing examples of the specific task the model is being adapted for, is the introduced, adjusting the model's parameters based on this new data, allowing it to specialize in the target task.

This process adjusts only the necessary parts of the model, which enables it to learn how to solve the specific task more effectively.

5.4 Prompt Engineering

5.4.1 What is Prompt Engineering?

At its core, prompt engineering involves designing and refining prompts to elicit specific responses from AI models. These prompts can range from simple questions or keywords to complex instructions, code snippets, or even creative writing samples. The effectiveness of a prompt directly influences the quality and relevance of AI output.[5]

5.4.2 Why is it important?

Effective prompt engineering enhances the performance of AI models across various applications, including:

- Content Creation: Generating articles, summaries, or creative writing.
- Programming assistance: Debugging code or generating code snippets.
- Data analysis: Interpreting and summarizing complex datasets.
- Customer service: Automating responses to customer inquiries.

By providing clear and specific prompts, users can significantly improve the relevance and accuracy of AI-generated content .

5.4.3 How it works

Prompt engineering works by leveraging the underlying architecture of large language models (LLMs), which are trained on vast datasets comprising text from diverse sources. These models predict the next word or sequence of words based on the input prompt provided. The design of the prompt guides the model's focus, influencing how it interprets context and generates responses.

There are several strategies used in prompt engineering, such as:

- Zero-shot prompting
- Few-shot prompting
- Chain-of-thought (CoT) prompting
- Few-shot CoT prompting

- Zero-shot CoT prompting
- Self-consistency
- Generated Knowledge Prompting (GKP)
- Temperature tuning

Further along in this document, we will go over all some of these techniques by summarizing the main idea behind them and illustrate how they were useful to our use case.

6 Chosen Technique

Given the highly variable nature of the task, where the structure of target websites can change unpredictably and require complex adaptations to scraping logic, we determined that prompt engineering was the most appropriate technique for automating scraper repair.

Unlike approaches such as Traditional Retrieval-Augment Generation which, as explained above, are more effective in knowledge-intensive scenarios where external factual retrieval is essential, our use case demands a more flexible and context-drive solution. The primary challenge lies in dynamically understanding and adapting to the structural changes in websites, and subsequently generating accurate and functional code.

Prompt engineering allows us to tailor inputs to the language model with precision. By crafting structured prompts that include the failing scraper code and the modifications required to make the scraper functional again, we can guide the LLM to generate corrected code.

That being said, our use case is still subject to the limitations imposed by the model's context window. To address this, we will explore a simplified variation of Retrieval-Augmented Generation known as Context Pruning.

7 Previous work done on the matter

7.1 Kadoa

Kadoa is an AI-powered web scraping platform that automates data extraction and transformation from websites and documents, eliminating the need for manual coding or maintenance. By leveraging large language models like GPT-3.5 and GPT-4, Kadoa intelligently analyzes web pages to identify and extract relevant information. Users can input a URL, and the platform automatically captures the desired data, outputting it in structured formats such as JSON, CSV, or Excel files.

A standout feature of Kadoa is its self-healing capability. Traditional web scrapers often break when website structures change, requiring manual updates. Kadoa addresses this by employing AI agents that monitor for changes in website layouts. When a change is detected, these agents regenerate the necessary extraction logic on-the-fly, ensuring continuous and accurate data collection without human intervention.

The repair workflow involves several automated steps:

- Change Detection: Kadoa continuously monitors target websites for structural changes that could affect data extraction.
- Automated Code Regeneration: Upon detecting a change, AI agents regenerate the scraping code to adapt to the new structure.
- Validation: The system validates the newly extracted data against previous datasets to ensure consistency and accuracy.
- Deployment: Once validated, the updated scraper is deployed, resuming normal operation without downtime.

This self-healing mechanism significantly reduces maintenance efforts, making Kadoa a robust solution for dynamic web environments.

8 Practical Application of Adaptation Techniques

8.1 Element Alternative Gathering After Semantic Website Change

In the following tests, we evaluated the effectiveness of different large language models (LLMs) with varying parameter sizes, as well as the impact of various prompting techniques. The task involved identifying an appropriate alternative element, given a missing element and a list of newly gathered elements—effectively simulating a common challenge that arises when a website's structure changes.

Below are input objects representing the element that is presumed to be missing from the webpage, along with a list of newly identified elements.

Each object includes the element's unique locator (a CSS selector), its associated text content (if any), and its HTML tag.

```
oldElement: { locator: "#search-button", text: "
    Search", tag: "button" }
newElements: [
    { locator: "h1.item-title", text: "Item 1", tag:
        "h1" },
    { locator: "button.search-btn", text: "Pesquisar
        ", tag: "button" },
    { locator: "#checkbox", text: "", tag: "div" }
]
```

The LLMs are expected to respond with the following output.

```
{ locator: "button.search-btn", text: "Pesquisar",
    tag: "button" }
```

This is because, among all the options provided, the one above exhibits the highest degree of similarity to the missing element.

8.1.1 No Formal Prompting Techniques, Varying Models

Test purpose

In this test, our objective was to evaluate a group of LLMs—differing in both training datasets and parameter sizes—on how reliably they could return the appropriate element.

No formal prompting techniques were employed; instead, the only prompt that was used (a user prompt) was phrased in a natural, conversational manner, as a human might ask another person to perform the task.

User prompt

```
From the following input, containing an element which is no
    longer present in a webpage, and newly found elements:

    oldElement: { locator: "#search-button", text: "Search",
         tag: "button" },
    newElements: [
                { locator: "h1.item-title", text: "Item 1",
                    tag: "h1" },
                { locator: "button.search-btn", text: "
                    Pesquisar", tag: "button" },
                { locator: "#checkbox", text: "", tag: "div"
                     }
            ]

    Return only an alternative to the missing element, in
        the specified format, with no other text.
```

For reference, we used this prompt to query GPT-4o, a large scale LLM managed by OpenAI, and got the following answer.

GPT-4o

```
    { locator: "button.search-btn", text: "Pesquisar", tag:
        "button" }
```

This response is effectively correct and adheres precisely to the required output format. This result is expected, given that the model is a flagship LLM with a very large parameter count and an extensive training dataset.

Chosen Models

To conduct our local experiments, we selected four different models, each with a distinct training focus.

- **Mistral** - A general-purpose and efficient model, available exclusively in a 7B parameter configuration.
- **Gemma3** - Google's flagship open-use, general-purpose multimodal model, designed to run efficiently on a single GPU. It is available in 1B, 4B, 12B, and 27B parameter configurations.
- **CodeLlama** - A code-specialized variant of Meta's LLaMA model family, optimized for programming and code generation tasks. It is available in

multiple parameter sizes, including 7B, 13B, 34B and 70B, and supports a wide range of programming languages.

- **Llava** - A multimodal model that combines vision and language understanding, designed for tasks requiring image and text integration. It is available in multiple parameter sizes, including 7B, 13B, and 34B.

Smallest available parameter size results

We began by experimenting with the prompt described above, using the smallest available parameter configuration for each model. The results we gathered are presented below.

Mistral:7B

```
{ locator: "button.search-btn", text: "Pesquisar", tag:
    "button" }
```

Gemma3:1B

```
{ locator: "h1.item-title", text: "Item 1", tag: "h1" }
```

CodeLlama:7B

```
Here is the alternative to the missing element, in the
    specified format:

{ locator: "#checkbox", text: "", tag: "div" }
```

Llava:7B

```
 Based on the provided input and context, it appears
     that the oldElement
with the locator "#search-button" is no longer present
    in the webpage.
However, the newElements do not seem to be relevant
    replacements for the
oldElement as they are for different elements.

Therefore, there is no alternative element that can
    replace the missing
element based on the given input.
```

Conclusions and observations

These experiments yield several key insights. First, it becomes evident that a model's training dataset significantly shapes its response patterns and performance. This is particularly clear when examining the outputs of Mistral, Llava, and CodeLlama—models with similar parameter counts but trained on vastly different data. Mistral correctly identified the intended alternative and responded using the proper format. CodeLlama partially adhered to the expected structure: it began with a brief explanatory summary and selected an incorrect alternative, albeit presented in the correct format. Llava stood out as the clear outlier; it failed to follow the response format at all and incorrectly asserted that none of the alternatives were suitable replacements—an objectively false conclusion.

Gemma 3, despite having a much smaller parameter size (1B) compared to its peers, surprisingly followed the expected format but ultimately selected the wrong alternative. This raises an intriguing question: how might this model perform if scaled to a larger parameter size?

To answer this question, we prompted a larger version of Gemma3, this time with a 4B and 12B parameter set size, and got the following answer.

Gemma3:4B

```
```json
{
    "locator": "button.search-btn",
    "tag": "button"
}
```
```

Gemma3:12B

```
```json
{ "locator": "button.search-btn", "text": "Pesquisar", "tag
    ": "button" }
```
```

These experimental results suggest that model parameter size significantly influences both the content and correctness of responses. Notably, the Gemma3 model with a 12B configuration correctly identified the appropriate element and returned it in the expected format, albeit with additional text resembling JSON formatting tags.

Interestingly, the smaller 1B variant exhibited similar behavior. This tendency likely stems from the model's training data, which may include examples where JSON outputs are wrapped with triple backticks and a language identifier (e.g., ‘‘‘json) to denote code blocks. Such formatting is common in developer documentation and markdown files. Consequently, the model has learned to replicate this pattern in its outputs. While this inclusion deviates slightly from the desired output, it can be easily addressed by implementing a simple post-processing step to strip these tags from the final response.

8.1.2 Few-shot Prompting

Introduction

Few-shot prompting is a technique in natural language processing where large language models (LLMs) are guided to perform tasks by providing them with a few example input-output pairs within the prompt itself, eliminating the need for explicit fine-tuning. This approach leverages the in-context learning capabilities of LLMs, allowing them to generalize from limited examples to new, unseen inputs. The seminal work "Language Models are Few-Shot Learners" by Brown et al. (2020)[7] demonstrated that scaling up LLMs significantly enhances their ability to perform a wide range of tasks using few-shot prompts, achieving competitive results without task-specific training.

Test purpose

In this test, we aim to evaluate whether applying the few-shot prompting technique described above can improve the performance of models that previously produced partially incorrect responses, and whether it maintains consistent output in models that had already responded correctly.

To apply this technique, we simulated a conversation history between the prompter and the model using the **/api/chat** endpoint provided by Ollama's API.

Below are the prompts for the simulated conversation.

User

```
From the following input, containing an element which is no
    longer present in a webpage, and newly found elements:

oldElement: { locator: "#search-button", text: "Search", tag
    : "button" },
    newElements: [
            { locator: "h1.item-title", text: "Item 1", tag:
                "h1" },
            { locator: "button.search-btn", text: "Pesquisar
                ", tag: "button" },
            { locator: "#checkbox", text: "", tag: "div" }
            ]

Return only an alternative to the missing element, in the
    specified format, with no other text.
```

Model

```
{ locator: "button.search-btn", text: "Pesquisar", tag: "
    button" }
```

User

```
From the following input, containing an element which is no
    longer present in a webpage, and newly found elements:

oldElement: { locator: "button.login", text: "Log in", tag:
    "button" },
    newElements: [
            { locator: "h1.page-title", text: "Login Page",
                tag: "h1" },
            { locator: "input.username", text: "Username",
                tag: "input" },
            { locator: "#password-input", text: "Password",
                tag: "input" },
            { locator: "#login-btn", text: "Sign in", tag: "
                button" }
            ]

Return only an alternative to the missing element, in the
    specified format, with no other text.
```

Model

```
{ locator: "#login-btn", text: "Sign in", tag: "button" }
```

User

```
From the following input, containing an element which is no
    longer present in a webpage, and newly found elements:

oldElement: { locator: "#origin-input", text: "Origem", tag:
     "input" },
    newElements: [
            { locator: "h2.reservation-page-title", text: "
                Reservas", tag: "h2" },
            { locator: "#start-station", text: "Partida",
                tag: "input" },
            { locator: "#end-station", text: "Destino", tag:
                 "input" },
            { locator: "#search", text: "Procurar", tag: "
                button" }
            ]

Return only an alternative to the missing element, in the
    specified format, with no other text.
```

Based on the results from the previous experiment, we selected the most promising models: Mistral-7B, Gemma3-12B, and CodeLlama-7B. Llava was excluded from further testing, as its training focus and dataset are not well-aligned with the requirements of the task.

For this test, the expect result would be:

```
{ locator: "#start-station" , text: "Partida" ,tag: "input"
    }
```

Results

Mistral:7B

```
{ locator: "#start-station" , text: "" ,tag: "input" }
```

CodeLlama:7B

```
{ locator: "#start-station" , text: "Partida" ,tag: "input"
    }
```

Gemma3:4B

```
{ locator: "#search" , text: "Partida" ,tag: "button" }
```

Gemma3:12B

```
{ locator: "#search" , text: "Procurar" ,tag: "button" }
```

These results display an unexpected downgrade in model performance, but highlight a key observation.

Although we initially expected few-shot prompting to improve model performance in this task, this did not hold true. We believe this is because few-shot prompting tends to be more effective for tasks that follow clear, deterministic patterns and require minimal abstract reasoning. In contrast, our experiment demands that the model select an alternative to a missing element based on what a human would intuitively choose—a process that inherently involves domain knowledge and multi-step reasoning. For example, recognizing that ”Origem” and ”Partida” can refer to the same concept, despite their linguistic and contextual differences, is something humans do almost automatically. This kind of semantic association is difficult for a model to infer without explicit reasoning steps.

8.1.3 Chain of Thought Prompting and the Self Consistency principle

Introduction

Chain-of-thought (CoT) prompting, first introduced by Wei et al.(2022) [9], helps models tackle complex problems by breaking the solution into smaller reasoning steps. When used alongside few-shot prompting—which provides a few examples of how to approach a task—it can lead to better results on tasks that need thoughtful, step-by-step reasoning before reaching an answer.

As stated on the website *Prompt Engineering Guide* [8] and as proposed by *Wang et al. (2022), self-consistency aims to replace the naive greedy decoding used in chain-of-thought prompting". The idea is to sample multiple, diverse reasoning paths through few-shot CoT, and use the generations to select the most consistent answer. This helps to boost the performance of CoT prompting on tasks involving arithmetic and commonsense reasoning.*

Test purpose

In this test, we aim to assess how much model performance improves when using this technique to encourage more human-like reasoning in solving our abstract problem.

To apply the technique effectively, we combined the previously described few-shot prompting with the addition of intermediate reasoning steps in the model's response. As a result, the LLM now outputs both its reasoning process and the final answer. In practical use, this reasoning would need to be parsed separately after generation.

We also applied the self-consistency principle by selecting the most frequent answer among five model responses. This approach is expected to produce the most reliable result the models can generate.

To the previously described responses, we have added text such as the one below.

```
I need to identify which of the provided elements can
    replace the original element, which is an \texttt{input}
    with the text ``Origem'' and the locator \texttt{\#origin
    -input}. This element is most likely used for website
    users to enter the origin location of their trip.

Examining the first element, it is a header with the text ``
    Reservas'' and the locator \texttt{h2.reservation-page-
    title}. This appears to be a page title, and thus is
    unlikely to serve as a replacement for the input field.
```

```
The second element is also an \texttt{input}, matching the
    original element's tag. It contains the text ``Partida'',
     which in Portuguese refers to the starting point of a
    trip. This makes it a strong candidate to replace the
    original element.

The third element is a button labeled ``Procurar'', which
    likely functions as a submission control for the form.
    Therefore, it is not a suitable replacement for the input
     field.

Based on these observations, the most appropriate
    alternative is:

{ locator: "#start-station", text: "Partida", tag: "input" }
```

Results

For reference, when we provided GPT-4o with the exact same input used with our locally hosted models, it returned the correct answer:

```
{ locator: "#start-station", text: "Partida", tag: "
    input" }
```

Using the same models as before, but incorporating additional reasoning steps into the few-shot response examples, and choosing the most frequent answers, we achieved the following results.

Mistral:7B

```
Inconsistent replies across all test runs.
```

CodeLlama:7B

```
(2/5)
{ locator: "#origin-input", text:"Origem", tag:"input" }
```

Gemma3:4B

```
(2/5)
{ locator: "search", text: "Procurar", tag: "button" }
```

Gemma3:12B

```
(5/5)
{ locator: "#start-station", text: "Partida", tag: "
    input" }
```

Conclusions

Despite the addition of structured reasoning, the majority of models remained underperformant or exhibited inconsistent behavior. Notably, Gemma3:12B demonstrated robust and consistent accuracy across all test runs, correctly identifying the intended element in every instance while providing logically sound justifications for its selection.

This further underscores the importance of both model size and training focus. At the same time, it demonstrates that with the right model, this technique—enhanced prompting with structured reasoning—can deliver highly effective results.

Based on the results obtained, we identify Gemma3:12B as the most suitable replacement for a large-scale model like GPT-4o in this specific task domain. Despite being a medium-sized model, Gemma3:12B, when used in conjunction with a carefully crafted few-shot prompting strategy—augmented by chain-of-thought (CoT) reasoning and self-consistency techniques—demonstrated consistent accuracy and reliable reasoning. Its ability to match GPT-4o's performance in identifying the correct element across all test runs makes it a practical and efficient alternative, particularly in resource-constrained or latency-sensitive environments.

8.2 Website Change Recovery

8.2.1 The context window, a problem to overcome

As discussed earlier, an LLM's context window refers to the number of tokens it can process as input when generating a response. This limitation becomes especially important when working with HTML or other code-heavy formats, as these files often span hundreds of lines and contain thousands of tokens.

To illustrate this limitation, we conducted a simple experiment. We extracted the raw HTML from YouTube's homepage and fed it to Mistral:7B, a medium-sized model we've previously evaluated. After inputting the full HTML content, we asked the model to return the very first line of the HTML it received.

For reference, this was the first HTML line:

```
<html style="font-size: 10px;font-family: Roboto, Arial,
    sans-serif;" lang="en" darker-dark-theme="" darker-dark-
    theme-deprecate="" system-icons="" typography=""
    typography-spacing=""><head><script id="js-1065286052"
    src="https://www.gstatic.com/cv/js/sender/v1/cast_sender.
    js" nonce="">
```

Here's what it returned:

```
<html>
```

As we can see, the model responded with only the ¡html¿ tag—plausible and syntactically valid, but not the complete first line of the input. This simplification suggests that the model either failed to retain the original input due to context overflow or defaulted to a generic response.

This example clearly illustrates how quickly the context window can be exceeded, and how this impacts a model's ability to reason accurately over large inputs.

Overcoming this issue

To overcome context window limitations, several techniques can be employed. One of the most widely known is Retrieval-Augmented Generation (RAG)—a method we discussed earlier in this document. RAG works by retrieving relevant chunks of external information and dynamically feeding them into the model's context. While powerful, RAG typically requires additional infrastructure such as vector databases, indexing pipelines, and retrieval logic, making it relatively complex to implement.

However, for many use cases—especially those involving structured data like HTML or JSON—simplified alternatives to full RAG can be just as effective without the associated overhead. One such technique is Pruning.

Pruning: A Lightweight Alternative

Pruning refers to the process of intelligently reducing the size of the input by removing irrelevant or redundant sections before feeding it into the model. Unlike traditional truncation (which often just cuts from the beginning or end), pruning is context-aware: it retains only the parts of the document that are likely to be relevant for the task at hand.

In the context of HTML analysis, pruning might involve:

Stripping out **script**, **style**, **meta** and comment blocks that don't influence structural layout or semantic meaning. Filtering to keep only tags of interest (e.g. **input**, **button**, **form** and elements containing attached event listeners or child text). Removing repetitive or boilerplate elements (e.g. headers, footers) that occur across pages and don't vary meaningfully. Using heuristic rules or simple pattern-matching to retain sections with specific attributes, such as id, class, or name, that are commonly used in element identification tasks. By reducing the token load while preserving essential semantic content, pruning allows models to operate more effectively within their context window—without requiring changes to the model architecture or the addition of external systems.

For example, the homepage of the CP (Comboios de Portugal) website contains approximately 1,950 lines of raw HTML. Processing such a large document directly would easily exceed the context window of most language models.

However, by applying the pruning rules outlined earlier—such as filtering out scripts, styles, and non-semantic elements—we were able to significantly reduce the input size. Using a custom module we developed (which will be detailed in later sections), we extracted 263 relevant elements from the page. This means that the input ultimately passed to the model represented just **13 percent** of the original HTML content—a substantial reduction with minimal loss of task-relevant information.

This example highlights how effective pruning can be in practice: by compressing large HTML documents down to only their meaningful structural components, we enable models to operate within their context constraints while still reasoning accurately about the page's contents.

8.2.2 Simple Contained Website Change Recovery

Now that we have addressed the context window limitation, we are able to proceed with experiments focused on scraper recovery following changes to a website's structure.

In the following tests, we used a simple webpage developed and manipulated by ourselves in order to simulate a breaking change. Specifically, we altered a button's locator `id` from `"search-button"` to `"search-btn"`.

The intended modification to the scraper's code is to change the following line:

```
        webDriverWait.until(ExpectedConditions.
            elementToBeClickable(By.id("search-button"))).
            click()
```

To:

```
        webDriverWait.until(ExpectedConditions.
            elementToBeClickable(By.id("search-btn"))).click
            ()
```

To conduct these experiments, we selected the three best-performing models from our earlier element extraction evaluations: Gemma3:12B, Mistral:7B, and CodeLlama:7B. In addition, we included three new models that have been specifically trained for code generation and intervention tasks.

We hypothesize that the code-oriented models will perform particularly well in this scenario, as the task of scraper recovery involves less abstract reasoning and more direct, structural manipulation of code—making it well-suited to models optimized for programming tasks.

New model introductions

For our new model introductions, we selected a set of code-oriented models: CodeQwen:7B, CodeGemma:2B, DeepSeek-Coder:1.3B, Qwen2.5-Coder:1.5B, and Codestral:22B.

Most of these models fall into the smaller parameter range, with the exception of CodeQwen, which represents a medium-sized model, and Codestral, the latest flagship offering from Mistral AI. Codestral is currently available only in a 22B configuration, positioning it as a large, highly specialized model for advanced code generation and reasoning tasks.

Few-shot prompting

As in our previous experiments, we applied the few-shot prompting technique to this task. Given our earlier hypothesis—that few-shot prompting performs best in scenarios requiring minimal abstract reasoning—we anticipated favorable results.

This expectation is well aligned with the nature of the task at hand: this type of scraper recovery involves straightforward code substitution, where models are only required to replace outdated elements with already identified alternatives. The structured and deterministic nature of this task should, in theory, make it especially well-suited for prompt-based guidance.

We have also incorporated Chain of Thought (CoT) prompting into the assistant's simulated responses to help guide its reasoning and improve the success rate.

Below is an example of a simulated message included in the few-shot prompt.

```
# User input


   Original Code:
        package example

        fun main(wdw: WebDriverWait) {
            val driver = buildChromeDriver()

            driver.get("https://example.com")
            wdw.until(ExpectedConditions.
                elementToBeClickable(By.id("example"))).
                click()
        }
```

```
        Locators: [
          {old: {tag: button, locator: 'example'}, new:
              {tag: button, locator: 'new-example'}}
        ]



# Model output


    // Thought process:
    // 1. The original locator is `By.id("example")`,
        and the update specifies replacing it with `By.id
        ("new-example")`.
    // 2. The rest of the code remains unchanged, so
        only a single replacement is necessary.
    // 3. The updated locator is applied directly to the
         existing WebDriverWait expression.

    package example

    fun main(wdw: WebDriverWait) {
        val driver = buildChromeDriver()

        driver.get("https://example.com")
        wdw.until(ExpectedConditions.
            elementToBeClickable(By.id("new-example"))).
            click()
    }
```

Results

For these experiments, we included a simple and concise system prompt, applied only to models that support it. For example, Gemma models do not support system prompts.

For the mentioned models, and specified modification we got the following results.

For reference this was the response obtained from GPT-4o.

```
// Thought process:
// 1. The only locator to update is By.id("search-button
    ")     By.id("search-btn").
// 2. This locator appears in a WebDriverWait expression
    , so we replace that line only.
// 3. The rest of the code involving By.name("query")
    and By.id("submit") remains unchanged.
```

```
package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
     driver.get("https://search.example.com")
     webDriverWait.until(ExpectedConditions.
         elementToBeClickable(By.id("search-btn"))).click
         ()
     driver.findElement(By.name("query")).sendKeys("
         Kotlin")
     driver.findElement(By.id("submit")).click()
}
```

Gemma3:12B

```
// Thought process:
// 1. The code contains a WebDriverWait call using the
    ID "search-button".
// 2. The locator list indicates this should be updated
    to "search-btn".
// 3. Perform a direct replacement of the ID in the
    WebDriverWait expression.

package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
     driver.get("https://search.example.com")
     webDriverWait.until(ExpectedConditions.
         elementToBeClickable(By.id("search-btn"))).click
         ()
     driver.findElement(By.name("query")).sendKeys("
         Kotlin")
     driver.findElement(By.id("submit")).click()
}
```

Mistral:7B

```
// Thought process:
// 1. The code contains a single WebDriverWait call
    using ID "search-button".
// 2. The locator list indicates that this should be
    updated to "search-btn".
// 3. Perform a direct replacement of the respective ID
    in the code.

package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
     driver.get("https://search.example.com")
     webDriverWait.until(ExpectedConditions.
         elementToBeClickable(By.id("search-btn"))).click
         ()
     driver.findElement(By.name("query")).sendKeys("
         Kotlin")
     driver.findElement(By.id("submit")).click()
}
```

CodeLlama:7B

```
// Thought process:
// 1. The code contains one WebDriverWait call using an
    ID: "search-button".
// 2. The locator list indicates the ID should be
    updated from "search-button" to "search-btn".
// 3. Perform a direct replacement of the ID in the code
    .

package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
     driver.get("https://search.example.com")
     webDriverWait.until(ExpectedConditions.
         elementToBeClickable(By.id("search-btn"))).click
         ()
```

```
    driver.findElement(By.name("query")).sendKeys("
        Kotlin")
    driver.findElement(By.id("submit")).click()
}
```

CodeQwen:7B

```
// Thought process:
// 1. The original locator is `By.id("search-button")`,
    and the update specifies replacing it with `By.id("
    search-btn")`.
// 2. The rest of the code remains unchanged, so only a
    single replacement is necessary.
// 3. The updated locator is applied directly to the
    existing WebDriverWait expression.

package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
    driver.get("https://search.example.com")
    webDriverWait.until(ExpectedConditions.
        elementToBeClickable(By.id("search-btn"))).click
        ()
    driver.findElement(By.name("query")).sendKeys("
        Kotlin")
    driver.findElement(By.id("submit")).click()
}
```

CodeGemma:2B

```
// Thought process:\n// 1. The code contains a single
    WebDriverWait call usin
```

DeepSeek-Coder:1.3B

```
// Thought process:
// 1. The code contains three calls to `WebDriverWait`
    and uses IDs for the WebElements such as "
    search_button". Also, it is using a name attribute on
     an element named 'query'. These elements should be
    updated according to your locator list provided in
    this task description which has both old (using id)
    and new locations.
```

```
// 2. Replace each respective ID with its corresponding
    tag/locator as per the Locators array, ie., "search-
    btn" for search button's current Id location `{tag:
    button , locator:"id_name"}` in this case. This will
    update all references of that id to new locations and
     also replace any other occurrences within your code
    which have old ID names (in the given list).
// 3. Make sure you are replacing both clickable
    elements with their updated tag/locators `{tag:button
     , locator:"id_name"}` in this case, as ids might be
    referencing those buttons and not just 'search-btn'.
    If there're multiple places where such need to change
     replace them all.
// 4. The last call is using name attribute for the
    search query field which should have a locator of "
    query" now `{tag:input ,locator:"name_value"}` in
    this case, but it also uses an id 'submit', make sure
     you update that too as well if needed and there are
    multiple places where such thing is used.
// 5. Make careful changes to all occurrences of IDs
    within your code because updates should be done
    accurately reflecting the current locators provided
    in this task description for both button's (tag:
    button,locator:"id_name") as well search query field(
    input) with its name attribute "query" and submit id.
```

Qwen2.5-Coder:1.5B

```
// Thought process:
// 1. The original code contains a single WebDriverWait
    call using the ID: "search-button".
// 2. The locator list indicates that this should be
    updated.
// 3. Replace "search-button" with "search-btn".
// 4. Perform the update directly in the code.

package search

import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.support.ui.ExpectedConditions
import org.openqa.selenium.support.ui.WebDriverWait

fun performSearch(driver: WebDriver, webDriverWait:
    WebDriverWait) {
    driver.get("https://search.example.com")
    webDriverWait.until(ExpectedConditions.
        elementToBeClickable(By.id("search-btn"))).click
        ()
```

```
        driver.findElement(By.name("query")).sendKeys("
            Kotlin")
        driver.findElement(By.id("submit")).click()
    }
```

Codestral:22B

```
    We were not able to get a response in meaningful time.
        Model is too large for our test hardware. (Macbook M1
         Pro 16GB RAM, 16 GPU Cores)
```

Conclusions

The experiments conducted on simple scraper recovery demonstrate that model specialization and prompt design play a decisive role in the accuracy and reliability of code transformations. Despite differences in parameter sizes, code-focused models such as CodeQwen:7B, CodeLlama:7B, and Qwen2.5-Coder:1.5B consistently produced correct and well-reasoned outputs. These results highlight that for structured, low-ambiguity tasks such as element locator substitution, a smaller model trained specifically on code can outperform or match larger general-purpose models.

Among all tested models, Gemma3:12B stood out, once more, for its consistent correctness and logical clarity, even though it does not support system prompts. Its performance across different types of prompting (few-shot, CoT, self-consistency) and tasks (semantic element replacement and code repair) confirms its robustness and adaptability. Conversely, Mistral:7B, though a general-purpose model, also performed reliably—validating its utility in resource-constrained environments for simpler transformations when paired with good prompt structure.

In contrast, larger models like Codestral (22B) failed to run on typical hardware (e.g., MacBook M1 Pro with 16GB RAM), underlining the practical limitations of relying on heavyweight architectures in local environments. Similarly, smaller models like CodeGemma:2B and DeepSeek-Coder:1.3B underperformed, either truncating output or misinterpreting the task, suggesting that size alone does not guarantee reliability, and that insufficient capacity or unstable generation behavior may affect small models under even simple reasoning loads.

In summary, model size alone is not a definitive predictor of success. Instead, a combination of domain specialization, prompt clarity, and computational feasibility determines the best choice for a given scraping repair task. Models like CodeQwen:7B and Gemma3:12B demonstrate that it is possible to achieve near-GPT-4o levels of performance locally—without sacrificing reliability or precision—making them excellent candidates for production use in adaptive web scraping systems.

8.2.3 Context Driven Contained Website Change Recovery

After seeing models perform exceptionally well when given a simple change in a website's structure to recover from, we decided the next step would be to introduce a problem of the same domain but with introduced abstraction to see how well they would hold up.

This tests will involve requesting the models to perform code repairs, once again, but this time the input and problem at hand is slightly more complex. We are testing to see if the models can recover from a change in a table's presentation given that the table is now paged. We have told the models the following.

```
The following Kotlin code is now failing due to
    structural changes in the webpage:

        <CODE>
        val rows = driver.findElements(By.cssSelector("
            table#results tbody tr"))
        for (row in rows) {
            val columns = row.findElements(By.tagName("
                td"))
            println(columns[0].text)
        }
        </CODE>

        The table is now paginated. Only a subset of
            rows is visible at a time. The page includes
            the following new navigation controls:

        [
          { tag: "button", locator: "#prev-btn", text: "
              Previous" },
          { tag: "h1", locator: "current", text: "1" },
          { tag: "button", locator: "#next-btn", text: "
              Next" }
        ]

Please update the code to iterate through **all pages**
    of the table and maintain the original behavior (
    printing the first column of each row). Return only
    the corrected Kotlin code, with no extra commentary.
```

Although this raises a valid question—*"How would we detect the introduction of pagination without a human visually inspecting the page?"*—such changes can, in theory, be inferred automatically before being passed to the LLM. Two primary strategies can be employed:

Model-Assisted Snapshot Comparison

An auxiliary language model can analyze differences between previous and current DOM snapshots. By identifying structural changes—such as a drop in the number of visible rows or the appearance of new navigation controls—it can hypothesize that pagination was introduced.

DOM-Based Heuristics

Heuristics can be applied programmatically to flag likely pagination behavior. Examples include: Detecting fewer visible rows than an expected minimum Identifying the presence of elements with pagination-related keywords (e.g., "Next", "Previous") or navigation symbols (e.g., arrows or brackets) While these methods may not guarantee perfect accuracy, they can serve as general-purpose tools to surface the most common structural changes—enabling automated LLM prompting in the majority of cases.

Results

To perform these experiments, we selected a subset of the best-performing models from our previous tests—specifically, CodeQwen:7B, CodeLlama:7B, Qwen2.5-Coder:1.5B, and Gemma3:12B—based on their consistent accuracy and reliability in code repair and element-matching tasks.

Furthermore, have once again applied the techniques used previously (Few-shot and CoT prompting).

Below, and to keep consistency with previous runs is the output from GPT-4o.

GPT-4o

```
while (true) {
val rows = driver.findElements(By.cssSelector("table#
    results tbody tr"))
for (row in rows) {
    val columns = row.findElements(By.tagName("td"))
    if (columns.isNotEmpty()) {
        println(columns[0].text)
    }
}

val nextButton = driver.findElement(By.cssSelector("#
    next-btn"))
if (!nextButton.isEnabled) {
    break
}
nextButton.click()
```

```
    // Optionally wait for the next page to load
    WebDriverWait(driver, Duration.ofSeconds(5)).until {
        val currentPage = driver.findElement(By.cssSelector
            ("h1#current"))
        currentPage.text.toIntOrNull() != null // Ensures
            the page is updated
    }
}
```

The logic is sound and well-suited to the task. It correctly gathers all elements on the current page before proceeding to the next—only if the pagination button remains enabled. Additionally, it includes an intermediate step to wait for the page to load after each navigation action, ensuring that all relevant content is available before extraction continues.

CodeQwen:7B

```
    val rows = driver.findElements(By.cssSelector("table#
        results tbody tr"))
    for (row in rows) {
        val columns = row.findElements(By.tagName("td"))
        println(columns[0].text)
    }

    // Pagination control elements
    val prevBtn = driver.findElement(By.id("#prev-btn"))
    val currentPageDisplay = driver.findElement(By.id("
        current"))
    val nextBtn = driver.findElement(By.id("#next-btn"))

    while (true) {
        // Check if there's a Next page
        if (!nextBtn.isEnabled) break;

        // Iterate through the visible rows
        val rows = driver.findElements(By.cssSelector("table
            #results tbody tr"))
        for (row in rows) {
            val columns = row.findElements(By.tagName("td"))
            println(columns[0].text)
        }

        // Click Next button to move to the next page
        nextBtn.click()
        Thread.sleep(1000) // Wait for 1 second for the page
             to load
    }
```

CodeLlama:7B

```
driver.findElement(By.id("prev-btn")).click()
val rows = driver.findElements(By.cssSelector("table#
    results tbody tr"))
for (row in rows) {
    val columns = row.findElements(By.tagName("td"))
    println(columns[0].text)
}
driver.findElement(By.id("next-btn")).click()
```

Qwen2.5-Coder:1.5

```
driver.findElement(By.id("prev-btn")).click()
var current = driver.findElement(By.cssSelector("h1#
    current"))
var pageNumber = 0

while (!current.text.equals("Last")) {
    val options = driver.findElements(By.cssSelector("#
        next-btn"))
    for (option in options) {
        option.click()
        break
    }
    pageNumber++
    current = driver.findElement(By.cssSelector("h1#
        current"))
}

var rows = driver.findElements(By.cssSelector("table#
    results tbody tr"))
for (row in rows) {
    val columns = row.findElements(By.tagName("td"))
    println(columns[0].text)
}
```

Gemma3:12B

```
val pageCount = getPageCount()
var currentPage = 1
while (currentPage <= pageCount) {
    val rows = driver.findElements(By.cssSelector("table
        #results tbody tr"))
    for (row in rows) {
        val columns = row.findElements(By.tagName("td"))
        println(columns[0].text)
    }
    if (currentPage < pageCount) {
```

```
            driver.findElement(By.id("next-btn")).click()
        }
        currentPage++
    }

    fun getPageCount(): Int {
        val current = driver.findElement(By.cssSelector("h1.
            current")).text.toInt()
        val nextButton = driver.findElement(By.id("next-btn
            "))
        return if (nextButton.isDisplayed) {
            current
        } else {
            current
        }
    }
```

Conclusions

The final test marked a pivotal moment in evaluating the capabilities of LLMs for automated web scraper repair. While earlier tasks—such as straightforward locator substitutions—were handled effectively by several mid-sized models, the introduction of pagination as a structural abstraction exposed fundamental limitations in their multi-step reasoning and adaptability. Among all models tested, only GPT-4o consistently demonstrated the ability to generalize the task, construct appropriate control flow, and reason robustly across page transitions.

Most local or open-weight models struggled with issues such as incomplete logic, failure to re-fetch dynamic DOM elements, or incorrect interpretation of pagination controls. Code-oriented models like CodeQwen:7B and CodeLlama:7B showed partial success, but often failed to incorporate critical runtime behaviors, such as waiting for new pages to load. Even Gemma3:12B—despite its consistent performance in prior experiments—failed in this task by returning an incorrect abstraction based on the current page number rather than iterating through all pages.

These results reinforce a key insight from existing research: multi-step reasoning remains one of the most persistent limitations in small and medium-sized models. Even with advanced prompting techniques like Chain-of-Thought (CoT) and Few-Shot learning, certain forms of abstract reasoning and dynamic context handling appear to be emergent capabilities tied to larger-scale models. This suggests that, for complex automation tasks requiring procedural logic and contextual awareness, scaling model size is still a necessary condition for reliable performance.

Another important conclusion that emerges from these results aligns with a theory proposed in recent work by Yiming Cheng and Jintian Shao[10], which challenges the prevailing assumptions about Chain-of-Thought (CoT) prompting. The authors argue that CoT does not constitute genuine reasoning in the cognitive or symbolic sense, but rather reflects a sophisticated exploitation of the LLM's underlying token prediction capabilities. In this view, the model generates sequences of plausible intermediate steps not because it "understands" the reasoning process, but because it has learned patterns in how such steps typically appear in its training data.

Our findings reinforce this perspective. Despite the use of CoT and Few-Shot prompting, which even ourselves have demonstrated to be helpful techniques, most models failed to execute the correct logic when required to generalize beyond surface-level patterns—such as adapting control flow in response to newly introduced pagination. This suggests that while CoT can give the appearance of reasoning through structured and well-formed responses and undoubtedly enhances model performance in surface level tasks, it may lack the depth and robustness needed for tasks that require genuine causal inference, dynamic decision-making, or stateful environment interaction. Thus, the limitations observed in our experiments not only reflect model capacity but also the fundamental nature of how LLMs "reason"—a process that, as Cheng and Shao suggest, may be more superficial than previously assumed.

References

[1] Zixuan Ke, Adaptation of Large Language Models, arXiv:2504.03931 [cs.CL], 4 April 2025

[2] Nouha Dziri, Faith and Fate: Limits of Transformers on Compositionality, arXiv:2305.18654 [cs.CL], 29 May 2023

[3] Ziwei Xu, Hallucination is Inevitable: An Innate Limitation of Large Language Models, arXiv:2401.11817 [cs.CL], 22 Jan 2024

[4] Yunfan Gao, Retrieval-Augmented Generation for Large Language Models: A Survey, arXiv:2312.10997 [cs.CL], 18 Dec 2023

[5] Shubham Vatsal, A Survey of Prompt Engineering Methods in Large Language Models for Different NLP Tasks, arXiv:2407.12994 [cs.CL], 17 Jul 2024

[6] Ashish Vaswani et al, Attention Is All You Need, arXiv:1706.03762 [cs.CL], 12 Jun 2017

[7] Language Models are Few-Shot Learners, arXiv:2005.14165 [cs.CL], 28 May 2020

[8] The Prompt Engineering Guide, available at https://www.promptingguide.ai

[9] Jason Wei et al, Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, arXiv:2201.11903 [cs.CL], 28 Jan 2022

[10] Jintian Shao et Yiming Cheng, CoT is Not True Reasoning, It Is Just a Tight Constraint to Imitate: A Theory Perspective, arXiv:2506.02878 [cs.CL], 3 Jun 2025