SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY – INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Towards Automated Page Object Generation using Large Language Models

Betül Karagöz

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY – INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Towards Automated Page Object Generation using Large Language Models

Hin zu automatisierter Page-Object-Generierung mithilfe großer Sprachmodelle

Author:	Betül Karagöz
Supervisor:	Prof. Dr. Andrea Stocco
Advisor:	Prof. Dr. Andrea Stocco
Submission Date:	June 2, 2025

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, June 2, 2025 Betül Karagöz

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Prof. Dr. Andrea Stocco, for his invaluable support and guidance throughout the course of my thesis. His insights, encouragement, and expertise have been instrumental in shaping my work. I am profoundly thankful to my mother, who has always been a guiding light and a constant source of strength. Her unwavering belief in me, even during the times I doubted myself, has inspired me to always keep moving forward. Lastly, I am sincerely grateful to my dearest sisters and closest friends for their support throughout my master's degree, especially during the most challenging moments of this thesis. Their love and belief in me have been a deep well of strength and motivation.

Abstract

Automated test generation is a crucial aspect of software development because of the time-consuming nature of writing and maintaining test cases in the development cycle. Page Object Model (POM) is a widely used design pattern in web application testing used in many UI automation frameworks such as Selenium. However, manually creating Page Objects is labor intensive. Previous work, APOGEN has explored automated Page Object generation but without leveraging the recent advancements in Large Language Models (LLMs). In this thesis, we investigate the feasibility of using LLMs to automate the generation of Page Object Models, and aim to reduce the manual effort required to develop them. Our approach involves prompting LLMs on a structured manner with HTML elements to generate correct, complete and maintainable Page Objects. Based on empirical evaluation, we assess the effectiveness of LLM-generated Page Objects against manually written ones in terms of accuracy and coverage. The results demonstrate the potential of LLM-based Page Object automation in software testing.

Contents

1 Introduction

1.1 Motivation

In modern software development, End-to-End (E2E) testing plays an important role in ensuring the reliability of applications. It achieves this by validating the entire workflow from user interactions. E2E testing is different from other testing methods like unit testing or integration testing since it tests the whole system by simulating how real users would interact with it in a production-like setting. However, it is challenging to deal with the rapid evolution of the web application under test, because even tiny layout changes may result in several breakages which makes them hard to develop.

In order to enhance the efficiency of E2E testing, web automation frameworks like Selenium [1], Cypress [2] and Playwright [3] have become widely adopted. However, developing and maintaining automated test scripts remains a significant challenge, especially for large applications. It becomes even more difficult in case of dealing with dynamic and frequently changing user interfaces because the test cases become more fragile and break easily as the web application evolves.

The Page Object Model (POM) is a design pattern which improves the maintainability of UI tests by encapsulating web elements and their interactions within individual classes. With this abstraction of UI elements, POM provides decoupling to tests. It makes the management of test cases easier, reduces code duplication, and allows developers to create more robust test cases against UI changes. The classes created by POM can be reused among multiple test cases which improves the test maintenance. Moreover, this approach reduces code duplication meaning that if the UI changes, the fix needs to be only applied in one place. Despite how many benefits it provides, creating and maintaining Page Objects requires manual effort. As a result, their wide adoption is limited by the burden of this manual development effort.

To address this challenge, researchers previously have explored the potential of automating Page Object generation. For instance, Apogen [4] introduced a method by reverse engineering the target web application and creating Page Objects using static and dynamic analysis techniques. In the end, Apogen managed to achieve remarkable results in minimizing the effort required to write Page Objects with the use of clustering techniques [5].

There is a rising interest in exploring the potential of LLMs for automating repetitive and time-consuming development tasks, with the growing integration of Large Language Models (LLMs) into software engineering workflows. This research particularly investigates the feasibility and effectiveness of using LLMs to automate the generation of Page Object Models (POMs) which is known as another labor-intensive software development task. By analyzing how LLMs interpret given HTML structures, comprehend the relations between different page objects and translate them into functional page objects with correctly defined methods, this study aims to assess the quality, accuracy, and maintainability of LLM-generated POMs by comparing them to manually written ones.

The outcomes of this research are expected to be useful for defining best practices regarding Page Object generation using LLMs. Moreover, it may offer insights into prompt design and model capabilities. Ultimately, our objective is to advance the usability of LLMs in test automation and identify the conditions under which LLMs can effectively generate high-quality Page Object Models.

1.2 Problem Statement

The central focus of this research is to evaluate the capability of Large Language Models (LLMs) in generating Page Object Models (POMs). Previously, it has been proven many times that LLMs have great potential in various programming tasks [6]. However, their ability of producing syntactically correct, functionally complete, and maintainable POMs remains unexplored.

1.2.1 Research Question

How effective are LLMs at automating the generation of high-quality Page Object Models?

Our main research question explores how well Large Language Models (LLMs) can automate the creation of Page Object Models (POMs). In particular, it focuses on evaluating their ability to generate POMs which are correct, functionally complete and easy to maintain.

To assess this question, the study compares LLM-generated POMs with manually written POMs using a well-defined evaluation setup. This setup includes a comparison for consistency with user interface elements and adherence to design patterns and syntax rules. The evaluation evolves around some key points. First of all, we check whether elements and methods follow expected naming conventions. Secondly, we assess whether navigation methods correctly capture relationships between different pages by returning the appropriate Page Objects. Furthermore, we check whether the number of UI elements defined in the generated Page Object Models (POMs) is adequate. Our goal is to evaluate whether LLMs can replicate POM development practices, by analyzing all Page Objects generated by the LLM in a set of applications. We also aim to identify specific areas where LLMs are particularly successful or struggling in generating POs and uncover any underlying patterns, if any.

1.3 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 introduces the related background and terminology for our research, especially focusing on Page Object Models and Large Language Models. Chapter 3 reviews related work, centered on the previous works done regarding automating Page Object generation. Chapter 4 outlines the methodology of our experiments, including the selection of target applications and LLMs, the strategies used while developing the final prompt, and the evaluation metrics used. Chapter 5 reports the results obtained from applying LLMs to different applications by providing a detailed analysis of these results. Chapter 6 discusses the implication of these findings and threats to the validity of the study. Finally, Chapter 7 concludes the thesis with a summary of contributions and suggests directions for future work in this area.

2 Background and Terminology

2.1 Page Objects

The Page Object (PO) pattern in the context of web testing is used for abstracting the application's web pages in order to reduce the coupling between test cases and application under test (AUT). It has emerged to address the test maintenance difficulties by providing a layer of indirection which decouples test cases from the internals of the web page, in which web page elements are located and triggered by the web tests.

A page object is an object-oriented class which acts as an interface to one of the AUT's pages. By using page objects, whenever the tests need to communicate with the user interface of that page, they utilize the methods of this page object class. The page object contains the entire web page's information, which provides the test developer the advantage of operating at a higher degree of abstraction. The pattern is widely used in Selenium [1], Appium [7], and other automation frameworks.

2.1.1 Benefits of Using Page Objects

The main benefit of using page object pattern is that only the code inside the page object needs to change if the user interface for the page changes, not the tests themselves [8]. All locators and page-specific actions are stored in one place, which is the page class. Thus, all the modifications made to accommodate that new user interface are then centralized which reduces the maintenance effort [9].

Secondly, the same page objects can be reused across multiple test cases. For instance, a valid or invalid login test case can be a base action for multiple test case. It can be called from multiple test classes which reduces the code duplication and results in making the code base more modular.

Another important aspect could be encapsulation and abstraction. Page objects hide implementation details, ensuring that test scripts interact only only with high-level methods without dealing with UI locator details [9]. Moreover, Page Objects makes it easier to expand the test automation suites as new pages and functionalities are integrated into the AUT. New tests can be added with minimal changes to existing tests.

POs enhance readability and clarity as they use well-defined methods which define user actions in plain language [10]. Lastly, debugging becomes more straightforward as failures are localized to specific page objects.

2.1.2 Challenges in Using Page Objects

Despite the fact that it provides decoupling, reduced code duplication, and easier test maintenance, Leotta et al. [11] highlight that, building POs for a web application remains as a nontrivial task, which requires substantial development effort especially in the beginning.

Although the benefits associated with adoption of PO have been shown to be significant, it remains unclear whether such adoption justifies the initial effort needed to implement page objects (POs).

In case of small-scale test automation projects, implementing POM could potentially add unnecessary complexity. On the other hand, in case of bigger dynamic applications; maintaining page objects becomes cumbersome as discussed by Stocco et al. [5].

2.2 Large Language Models

Large Language Models (LLMs) are extremely large deep learning models which have been trained on vast volumes of text data in order to comprehend and generate human-like language. In contrast to conventional deep learning models, LLMs use unsupervised learning objectives and are pre-trained on large textual corpora. Some remarkable and widely used LLMs include GPT from OpenAI [12], Gemini from Google DeepMind [13], LLaMa models from Meta [14], and BERT [15].

Following their initial public use, they have become immensely popular in recent years and have made significant progress in many tasks, such as text generation, question answering, sentiment analysis, language translation, code generation and debugging. In addition, LLMs have been tested and proved to have strong capabilities in software development related tasks like code generation, code summarization, and test generation. With each passing day, new LLM models emerge, astonishing users and unlocking new possibilities for diverse applications.

2.2.1 Transformers Architecture in LLMs

The transformers architecture is the fundamental building block of LLMs. The idea first initially published by Vaswani in 2017 in the paper "Attention is all you need" [16]. Since then, it has transformed natural language processing by bringing in a novel idea: attention-based processing.

The transformers architecture outperforms other previous state-of-art architectures like Recurrent Neural Networks (RNNs) due to its attention mechanism that the transformers use. Transformers leverages the self-attention mechanism that empowers the model to focus on the most relevant parts of the input sequence when making each output token.

On the other hand, an attention mechanism is not a concern for the RNNs. RNNs process the input one word at a time while transformers can handle the entire input simultaneously and all at once. Hence, transformers are performant in handling more complicated connections between words in the input sequence and they can complete the tasks faster due to its simultaneous processing capabilities. After proving its versatility and effectiveness for many NLP tasks, it has become a benchmark architecture for various language modeling applications by extending beyond its initial use.

The memory limitations constraining RNN during training are resolved by this design decision, which has broad consequences for model efficiency, scalability, and performance since it permits the effective processing of lengthy sequences in parallel.

The Transformer architecture consists of two primary components: Encoder and Decoder, which also consist of multiple layers each. These two components work together in sequence-to-sequence tasks or separately in models like BERT [15] and GPT [17] to process and generate text.

The encoder's task is to process the input text and to convert it into a meaningful representation. On the other hand, the decoder is responsible for generating an output sequence. There are different models of transformers architecture using encoder only, decoder only or encoder-decoder which is explained in the next sections.

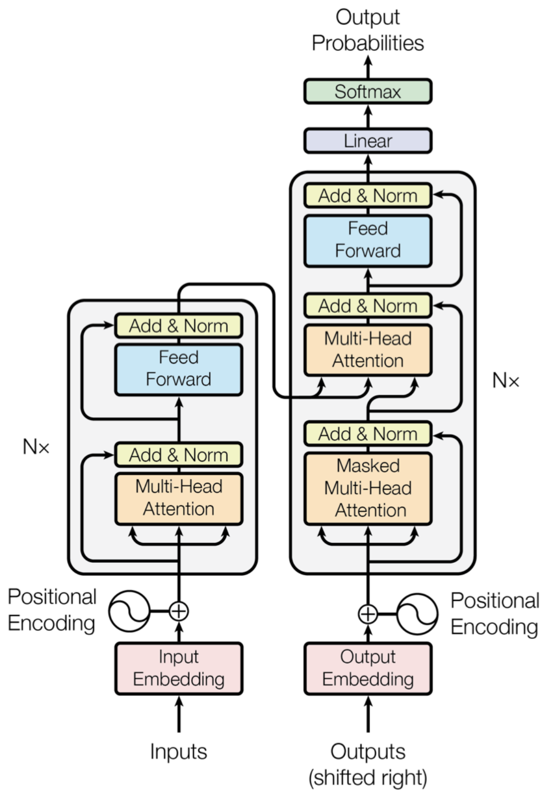

Figure 2.1: Transformers architecture in LLMs. Taken from [16]

2.2.2 Variations of the Transformers Architecture

There are three primary architectures for LLMs which are encoder-only, encoder-decoder, and decoder-only LLMs.

Encoder-only LLMs

Encoder-only models are based on solely on the encoder part of the Transformers architecture. The task of encoder is to process and encode the input sequence into a different representation that comprehends the relation between words and the overall context of a sentence. Hence, this model is appropriate for tasks that involve understanding the input text rather than generating new text. For instance, sentiment analysis and classification are two typical tasks for encoder-only models. The successful implementations of this model include BERT [15], RoBERTa [18] and DistilBERT [19].

Encoder-decoder LLMs

Encoder-decoder models, also known as sequence-to-sequence (Seq2Seq) models, utilizes both encoder and decoder components. The encoder deals with the input sequence, whereas the decoder is responsible for generating the output sequence. The primary aim of the encoder is to capture the most important context of input sequence and to generate the context vector. The decoder then uses this context vector to build the output sequence, one token at a time, while preserving the context vector and the previously generated tokens. Speech recognition and machine translation can be considered as successful applications utilizing the encoder-decoder model. This model is ideal for complex NLP tasks where both understanding and generation of new texts are required. The T5 [20], CodeT5 [21], and BART [22] models are a few examples.

Decoder-only LLMs

The last variation, decoder-only models, utilize the decoder component of Transformer architecture. Decoder only models are auto regressive, meaning that they generate text sequentially, predicting one token at a time. Unlike encoder-only models, they do not process the entire input simultaneously neither use encoder. Instead, they generate texts by predicting next tokens based on previous tokens. They first tokenize the input, uses self-attention mechanism and masked self-attention, in the end outputs the prediction by generating one token at a time.

Decoder only models excel in creative and open-ended tasks. Additionally, decoder-only models are more versatile than encoder-only and encoder-decoder LLMs because they may complete downstream tasks by providing straightforward suggestions without the need for fine-tuning or the addition of a prediction head. These benefits have drawn corporations and researchers to study and create decoder-only LLMs. The most popular decoder-only LLM is introduced by OpenAI GPT-modes as well as ChatGPT [23]. Some other examples are LLaMa series by Meta [14], Gemini by Google DeepMind [13].

2.2.3 Prompting Strategies for LLMs

As highlighted by Li et al. [24], prompting strategies significantly affects the behavior of Large Language Models. They define how input is structured to obtain the desired output. Different prompting methods differ in the amount of context or instruction they provide. Especially in tasks involving reasoning or structured code generation, prompting strategy can significantly impact the performance of the model.

Zero-Shot Prompting

Zero-shot prompting refers to the method of instructing an LLM to perform a task without any given examples of the desired output. In this prompting technique, the model is expected to only rely on the given input and its pretrained knowledge to understand the task.

Few-Shot Prompting

Few-shot prompting provides the LLM a few input-output pairs within the prompt to guide its understanding of the task. With given examples, the model can infer patterns to produce the desired output.

Chain-of-Thought Prompting

Chain-of-thought (CoT) prompting encourages the LLM to create intermediate reasoning steps before producing a final output. The model is prompted to follow a more structured and transparent reasoning process. This prompting strategy is usually followed for more complex tasks.

3 Related Work

This section describes the existing studies in automatic generation of Page Object for web applications, and possible utilization of LLMs in this topic. There are not many studies about this topic, and the most comprehensive study is called Apogen [4]. Apogen applies clustering and static analysis to derive structured and meaningful page objects, however, it has no implementation of LLMs. Therefore, our work is the first that we are aware of that uses LLMs to generate Page Object models.

The main motivation behind Apogen is to help tester to save precious testing time in the creating of test suites for web application, because it is known that creating Page Objects manually requires substantial effort. Apogen aims to automate the creation of a considerable amount of code for Page Objects that should be otherwise written manually.

Apogen enhances the level of automation beyond the creation of a class skeleton containing web elements, exploiting the knowledge present in the application itself. The page objects generated fully automated by Apogen contains the following features: WebElement instances for each clickable element like links and buttons, methods to navigate through pages, methods to fill and submit forms.

The automatic generation of page objects consists of several steps. First of all, Apogen infers a model of the AUT by reverse engineering using an event-based crawler. Then, similar web pages are clustered into syntactically and semantically meaningful groups. After that, a state object-based model is generated by statically analyzing the event-based model (Graph) and the additional information (DOMs and clusters). Lastly, the state object-based model is converted into a meaningful Java page objects via model to text transformations. The output of this entire process is a set of Java Page Objects, organized using Page Factory design pattern as supported by the Selenium Web Driver framework. In summary, Apogen provides a structured and effective approach to automating Page Object generation, reducing manual effort and code duplication.

Moreover, Leotta et al. [25] investigated the impact of using the Page Object pattern on the maintainability of automated test suites. The study provides empirical results that using the Page Object pattern reduces the amount of maintenance effort. Each Page Object class encapsulates the UI interaction logic which provides isolation of the test scripts from structural changes in the application. Therefore, this encapsulation improves the robustness of tests. This work not only confirms the practical value of the Page Object pattern but also strengthens the case for using well-structured design patterns in test automation.

Leotta et al. [26], conduct a series of experiments in another research. In these experiments, various test suites, developed with and without the Page Object (PO) pattern, were compared.

The study first investigates the initial effort required for designing the PO classes. Secondly, it assesses the possible long-term benefits of POs in terms of speed and robustness. As a result, it has been shown that introducing POs in test suites requires a substantial initial effort. However, the results were able to identify a break-even point where using POs becomes more reasonable despite of this initial effort. Specifically, when the test suite grows to approximately ten times the size of the initial benchmark, then using Page Objects becomes way more efficient than manually writing all Selenium scripts. Overall, the results suggest that, although PO pattern may require more work initially, for larger test suites, it offers advantages in scalability and maintainability.

While these works highlight the importance of POs in web test suites, their development is still mostly carried out manually, which motivates our research towards LLM-assisted POs.

4 Methodology

4.1 Research Approach

This section explains the methodology we followed to complete our research. The pipeline we followed was developed around the main goal of evaluating if LLMs can generate Page Objects from given HTML elements. We started by designing and refining prompts which are tailored to this task. Next, we selected a set of applications suitable for experimentation. While selecting them we ensured the availability of ground truth POs which will be later used to evaluate the correctness of LLM-generated POs. We then deployed various LLMs to assess their ability to automate PO generation. Finally, we post-process the output files produced by each model by categorizing generated elements and methods inside them. We used a predefined criteria to assess the accuracy, coverage, and overall effectiveness of the results. The following sections describes the components of our experimental pipeline in detail.

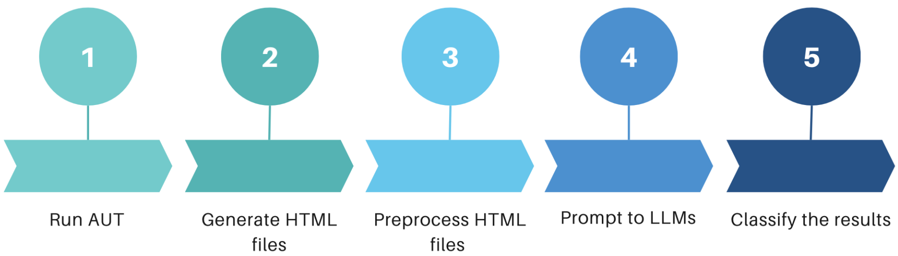

Figure 4.1: Overall Pipeline

4.1.1 Selected Apps

In our research, results are obtained by comparing LLM-generated POs against manually generated POs. Therefore, when selecting our Applications Under Test (AUTs), it is necessary to check whether existing research has already manually created POs for them. These POs serve as our ground truth, and they enable consistent and reliable evaluation.

To facilitate this evaluation, we utilized the a benchmark of web applications, which includes manually written Page Objects by a developer. Along with the POs, it provides docker containers to run the applications and associated test cases. This benchmark covers a set of applications, and we selected a total of five apps from various categories. By integrating a variety of applications, we aim to better evaluate the performance of LLMs in different scenarios.

Bludit-3.13.1

Bludit is lightweight content management system (CMS) that allows users to create and manage websites. It does not require a database, but instead it relies on flat files [27]. Its simplicity and fast performance makes it a possible useful test application for evaluating LLMs in content-oriented applications.

ExpressCart-1.19

ExpressCart is an open-source e-commerce application built with Node.js. It includes main features like listing products, shopping cart functionality, and checkout processes [28].

Kanboard-1.2.15

Kanboard is a project management tool which uses the Kanban methodology to help users visualize tasks, manage workflows, and track project progress [29].

MediaWiki-1.40.0

MediaWiki is a collaborative wiki platform best known for powering Wikipedia [30]. It supports structured content, revision history, and user permissions [31].

PrestaShop-1.7.8.5

PrestaShop is an open source e-commerce platform that provides a set of functionalities for managing online stores. These functions include product management, order processing and customer interactions [32].

4.1.2 Generating HTML Files

Benchmark we use includes page objects, Docker containers for running web applications, and test specifications. However, it does not provide the actual HTML content, which is the main input needed for prompting LLMs in our research. To obtain HTML content for screens, we ran the applications locally using the provided Docker containers, then we navigated through the relevant screens, and manually captured the corresponding HTML files.

4.1.3 Preprocessing HTML Files

These raw HTML documents, however, often contained large amounts of irrelevant or redundant information such as scripts and embedded resources. They were known to be unnecessary for LLMs and could decrease their performance.

To address this potential problem, we applied a preprocessing step to truncate and clean the HTML content before passing it to the model. This approach was guided by findings from the paper *"What's Important Here? Opportunities and Challenges of Using LLMs in Retrieving Information from Web Interfaces,"* [33]. In this paper, Huq et al. demonstrate that raw HTML files can overwhelm LLMs because of their size and noise. They might lead LLMs to hallucinate and to fail following the given instructions which may result in a degraded performance. In particular, the study found that applying an effective truncation strategy could improve model performance by up to 11%.

Truncating the HTML not only helped us stay within the model's context limits but also reduced irrelevant data that could distract or confuse the model. By focusing only on the most informative elements, we were able to enhance the clarity, and overall effectiveness of the LLM's outputs.

4.1.4 Selecting LLMs

In order to conduct our experiment for automated page object generation and to answer our RQ, we opted for two cutting edge LLMs: GPT-4o [34] and DeepSeek [35]. The decision was made by taking into consideration their state-of-art performance and specialization in code-related tasks accessibility. By selecting these models, we aim to evaluate the capabilities of leveraging LLMs to generate page objects.

Selection Criterias

The computational and financial demands associated with training, deploying, and maintaining LLMs has grown with the popularity of them in recent years. General purpose LLMs with over 70 billion parameters require substantial amount of GPU resources which results in high operational costs.

Although these large models often deliver superior performance, they come with the problem of consuming huge amount of energy and having a significant footprint. From an environmental standpoint, there are lots of concerns about their carbon impact and sustainability. In response to this, there has been growing interest in developing more efficient architectures. These smaller and task-specific models aim to provide a better balance between performance and environmental impact.

In line with this, our initial research goal was to explore small, open-source models with fewer parameters because of their accessibility and lower resource requirements. However, after conducting preliminary experiments, we found that these smaller models failed to deliver accurate results for our task. Larger open-source models with 7 billion parameters or more, demonstrated stronger performance but exceeded the limits of our available computational resources. Consequently, we changed our focus to high-performing models accessible via external APIs. We limited our study to GPT-4 and DeepSeek, and used their APIs to evaluate advanced capabilities without the need for local deployment.

1. **GPT-4o:** The GPT-4o model was released by OpenAI in May 2024 as a new version in the GPT-4 series. It is designed to improve both performance and efficiency. This model can handle more than one type of input, such as text, images, and audio. All of them can be processed in the same model architecture. Unlike earlier versions that used separate systems for different types of input, GPT-4o combines them into one. This speeds up extraction, lowers latency, and keeps output quality high [34].

2. **Deepseek AI:** DeepSeek AI is a large language model developed by DeepSeek. DeepSeek models are designed to compete with other leading LLMs. They are trained on a large-scale corpus of variety of domains, including programming, scientific literature, and natural language. In this project, we made particular use of **DeepSeek Coder** [36], a variant of the DeepSeek model optimized for software development tasks. DeepSeek Coder excels at generating, completing and understanding code. It achieves this through training on extensive programming datasets.

Setup for LLMs

LLMs contain various hyper parameters that change the way they behave, which ends up affecting the output quality. In this research, we are specifically interested in the temperature parameter which controls the level of creativeness in the generated responses. This parameter may vary from 0 to 1: a higher value encourages the model to produce more diverse and exploratory outputs, while a lower value leads to more focused, and deterministic responses.

Given that our task does not require the LLM to be more creative, and instead to be more consistent and deterministic, we opted for setting the temperature value to 0.1 across all LLMs used in our experiments. This helps to ensure that the output remains deterministic.

Another important hyper parameter in our research is max tokens. It defines the maximum number of tokens (words, subwords, or characters) which can be generated in a single response or input-output sequence. Since our task requires a complex output, our decision on this matter was to set it the default one and not limit it.

4.1.5 Prompt Strategies

After reviewing previous studies on testing with LLMs, it becomes clear that the prompt plays an undeniably critical role in guiding an LLM's performance on a given task. LLMs rely on context given in the prompt to figure out what we want them to achieve. A well-written prompt can clearly define the task and achieve good results whereas a vague or poorly crafted prompt might lead the model to get confused and produce off-target outputs. Thus, the initial prompt tells the LLM what to do, how to do it and what would be the expected result, which makes a great impact on the research.

The prompts were divided into two main types: system prompt and user prompt. Below is a brief introduction of both prompt types. Furthermore, Listing 4.3 shows the sample prompts for both system and user prompts.

```python
def generate_pom_code_with_open_ai():
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        temperature=0.1
    )

    return response.choices[0].message.content.strip()
```

In our research, we assume that no existing test suites or pre-defined Page Objects (POs) are available for the selected application. Therefore, it leads us to follow a zero-shot prompting approach. Since zero-shot prompting does not rely on prior examples or predefined structures, it enables the model to generate Page Objects and test cases based solely on the provided requirements or task description.

By choosing this approach, we can assess the LLM's ability to create correct POs without any prior knowledge of the AUT. This evaluation can also be considered as a powerful method for assessing the model's generalization capabilities in real-world scenarios and the performance achieved without the need for prior setup or input from existing resources.

Thus, as illustrated in the prompt below, we explicitly define the rules for generating Page Objects in detail and provide the corresponding HTML content for the given page. We expect the LLM to generate a Page Object in the form of a Java class and this Java class must follow a certain structure as given in the prompt. This includes following naming conventions, correctly setting up the constructor, and creating WebElements for each input field, button, and link in the given HTML page. The resulting Page Object must also include the right imports from Selenium, methods for navigating to other pages when a button or link is clicked,, and methods for interacting with the page, like clicking or entering text.

While conducting the experiments, we observed that certain requirements were often not implemented even though they were explicitly stated in the prompt. Afterwards, we noticed that using more assertive language especially through the use of imperative patterns such as "must" significantly increased the likelihood of those requirements being correctly implemented. As a result, we adopted the use of "must" statements to clearly emphasize critical instructions and eventually improve model compliance.

An Example Prompt

Listing 4.1: Page Object Generation Prompt

```
You are generating a Selenium Page Object Model (POM) class in Java for a web
application.

    ### File Naming Rules:
    - The Java class must be named **exactly**: {class_name}
        - This name must be used:
            - As the Java **filename** (‘{class_name}.java‘) **(must match exactly)**
            - As the Java **class name** (‘public class {class_name}‘)
            **(must match exactly)**
        - Do not infer or modify the class name based on page content.

    ### Generation Instructions:
    1. You must define all key elements (inputs, buttons, links, text fields, etc.)
    as ‘WebElement‘ fields using ‘@FindBy‘ annotations.
    2. At the top of the class, you must include this field:
        ‘public WebDriver driver;‘
    3. You must use **only generic, structural field names**, based on structure
    or position:
        - Examples: ‘firstButton‘, ‘mainInputField‘, ‘headerLabel‘,
        ‘secondaryLink‘
        - You must **not** use domain-specific terms like "product",
        "cart", "price", etc.
    4. If multiple similar elements are present (like multiple buttons or links),
    you should use **ordinal names**: ‘firstButton‘, ‘secondButton‘, etc.
    (optional unless needed for clarity).
    5. You must use **camelCase** for all field and method names.
    6. Create interaction methods that:
        - Perform actions like clicking, typing, or retrieving text.
        - If clicking an element clearly leads to another page
        (e.g., a link, a save button after form submission),
        **the method must return the appropriate POM class** (assume it already exists).
            - Example: after clicking a save button, return
            ‘new ProjectSummaryPage(driver);‘
        - If the action does not cause page navigation
        (typing into inputs, basic clicks), the method must return ‘void‘ or
        the appropriate text (‘String‘) if retrieving values.
    7. You must add a constructor with:
        ‘PageFactory.initElements(driver, this);‘
    8. You must not generate any other classes unless explicitly instructed.
```

```
9. You must only use information available in the provided HTML,
do not assume additional logic or elements.

### Error Message Handling Rules:
- If the HTML contains elements indicating errors, validation messages,
or warnings (e.g., classes or IDs like ‘error‘, ‘form-errors‘,
‘validation-error‘, ‘alert-danger‘):
    - You must create a ‘WebElement‘ field for each error message.
    - You must create a method named ‘getErrorMessage()‘ that returns the
    visible error text (‘element.getText()‘).
    - If multiple error messages exist, you must create methods
    like ‘getFirstErrorMessage()‘, ‘getSecondErrorMessage()‘, etc.,
    or alternatively return a list of error texts if appropriate.



### HTML Content:
{html_content}
    """
```

4.1.6 Evaluation Setup

In the evaluation phase of our study, we compared the ground truth page objects provided in the benchmark with the outputs generated by our script across five different applications, using two distinct LLMs. Since this comparison was performed manually, we went over each generated file individually by comparing the defined WebElement objects and corresponding methods against the expected outputs. In order to ensure consistency and clarity in our analysis, we categorized each identified element and method into three main groups:

1. **Correct:** Elements are classified as correct if they are defined using the correct type and if they have a meaningful name. For methods, it requires the method name to be accurate, the implementation to align with the intended behavior, and the return type to match the one defined in the ground truth version, to be classified in this category.

2. **To modify:** If an element is present but does not fully follow to the naming conventions or structural guidelines given in the prompt, it is categorized as an element which requires modification. These elements are partially correct and can still be used in the Page Objects with some manual adjustments. An important part of this classification was identifying the specific shortcomings, which helped us detect recurring patterns in how the LLM misinterpreted or failed to capture the complete structure of certain elements. If the element marked for modification is not a method, we check its naming and type to determine in which part adjustments are needed. However, if the element is a method, we perform a more detailed analysis. We examine its return type, implementation, and naming. Based on these criteria, we classify such methods into five distinct categories:

- **Missing return type** – The method lacks an explicitly defined return type, most of the cases the return type is defined as **void**
- **Wrong return type** – The method includes a return type, but it does not match the expected one.
- **Missing return type (returns same page)** – The method returns void, but according to the ground truth, it should have returned the same page object to support chaining in test cases.
- **Wrong naming** – The method name does not match the naming rules or does not reflect its intended functionality.
- **Wrong implementation** – The method logic is incorrect or differs from the expected behavior defined in the ground truth.
- **Missing implementation** – The method signature is defined but the actual implementation is left empty, which makes the method non-functional.

3. **Missing:** Elements are placed in the missing category if they are entirely absent from the generated output. This included both WebElements and methods that were present in the ground truth but re not produced by the LLM.

4. **Extra:** Elements are placed in the extra category if they are present in the LLM generated page objects but had no corresponding version in the ground truth page objects. Since it is expected that some manually generated page objects are not complete and do not represent the entire functionality of the page, this case it also expected. Extra defined elements by LLMs represents how much better LLMs in generating broad page objects.

When we began our study, we anticipated that correctly defining the return type would be a non-trivial challenge for LLMs. This task requires more than simply identifying the element type. Instead it requires a deeper understanding of the element's role within the broader application flow, especially for navigational elements which transition from one page to another. When only a single HTML file is provided as input, inferring the correct return type becomes even more difficult because of the dependency on recognizing the target page object correctly. LLMs struggle to correctly reason relationships between different pages without access to the surrounding context like the structure of other page objects, routing logic, or application state.

Our study particularly interested in exploring whether LLMs could generalize page object relationships correctly based solely on information from isolated HTML snapshots. In large applications, it is not possible to include the full content of the application in a single prompt because of the limitations of the context length. Therefore, this challenge becomes even more important to address the capabilities of LLMs under these limitations. Evaluating how much LLMs can infer from these partial views of the entire application is critical for ensuring practical and scalable usage. That is precisely why we chose to classify the elements requiring modification in a more detailed manner. By doing so, we can better understand where and why the model's reasoning falls short. This topic will be discussed in further depth in the upcoming sections.

4.1.7 Metrics & Analysis for Evaluation

Our study employs some standard metrics to evaluate the performance of LLMs in automating page object creation. These metrics allow us to quantitatively measure how effectively each LLM fulfills the assigned task. Each metric used in evaluation is explained in detail below.

1. **Accuracy:** The accuracy metric is used to measure the percentage of elements what were correctly classified as "Correct". This metric provides us a result to understand how well LLMs are performing in terms of correctly identifying elements.

$$\text{Accuracy} = \frac{\text{Number of Correct Elements}}{\text{Total Number of Elements}} \times 100$$

2. **Error Analysis / Breakdown by Category:** For the "To Modify" category, the results can be broken down into specific subcategories. These subcategories represent specific types of issues encountered. We can calculate each percentage of these subcategories over all and try to pinpoint in which aspects the LLMs specifically struggle with.

$$\text{Subcategory Percentage} = \frac{\text{Number of Elements in Subcategory}}{\text{Total Number of Elements to Modify}} \times 100$$

3. **Coverage:** This metric measures the proportion of ground truth elements that are successfully classified as "Correct" or "To modify". This will give us an idea of how much of the ground truth LLMs are able to accurately process and identify for potential change.

$$\text{Coverage} = \frac{\text{Total Correct + Requires Modification Elements}}{\text{Total Number of Elements}} \times 100$$

5 Results

This section presents the results obtained following the experimental setup mentioned in the previous section. The experiments were performed for five different applications and the detailed results are discussed below.

App	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
Bludit - GPT-4o	46	3	0	1	0	0	26	24	12	15	5	7	106	23	47	22
Bludit - Deepseek	48	9	0	3	0	1	23	19	10	8	7	11	110	51	46	26
Kanboard - GPT-4o	46	2	1	6	4	3	20	15	19	19	3	5	178	16	70	91
Kanboard - Deepseek	48	10	2	4	7	3	17	20	14	11	5	2	253	44	82	95
MediaWiki - GPT-4o	30	2	2	0	3	0	22	11	8	7	6	7	88	6	44	30
MediaWiki - Deepseek	28	3	0	1	2	0	23	4	11	6	7	13	142	15	86	14
PrestaShop - GPT-4o	73	5	0	3	4	0	4	25	30	14	9	11	212	6	145	74
PrestaShop - Deepseek	69	8	1	1	4	0	3	23	34	11	9	15	235	26	178	36
ExpressCart - GPT-4o	20	1	2	2	0	3	6	11	2	1	0	2	69	9	41	13
ExpressCart - DeepSeek	21	3	2	1	0	0	8	12	1	0	0	2	77	19	39	9
Total	**429**	**46**	**10**	**22**	**24**	**10**	**152**	**164**	**141**	**92**	**51**	**75**	**1470**	**215**	**778**	**410**

Table 5.1: Detailed classification of Page Object review using GPT-4o and DeepSeek, for all apps

Table 5.1 provides a comprehensive overview of the results from a large number of experiments performed in five different applications using two different LLMs. The results are divided into specific sections according to the accuracy and type of elements detected. Each element is labeled as Correct, To Modify, Missing or Extra depending on the performance of the LLM. Furthermore, each of these categories is divided into four sub-categories: Web Elements, Getter Methods, Action Methods and Navigation Methods. The total values obtained for each application in both LLMs are presented in the table with their respective breakdowns.

The overall results show that across all implementations and LLM combinations, the majority of correctly identified elements were web elements, accounting for 429 out of 507 instances. Action methods and navigation methods were the most common elements which requires modification. In contrast, the distribution of missing elements was more balanced: 141 web elements, 91 getter methods, 75 navigation methods and 51 action methods out of a total of 359 missing instances. Across all applications, DeepSeek consistently showed a tendency to generate extra items compared to GPT-4o. Furthermore, DeepSeek showed higher success in correctly identifying getter methods and outperformed GPT-4o in this category.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
GPT-4o	54	14	74	33	2
DeepSeek	54	12	67	28	8

Table 5.2: Breakdown of "To Modify" Subcategories

Table 5.2 shows the total number of instances across all applications for issues that occurred in the "to modify" category. It can be seen that the largest proportion of these issues were related to missing same page return type in both LLMs, with 74 out of 181 instances in GPT-4o, and 67 out of 169 instances in DeepSeek, with corresponding percentages of 40.9% and 39.6% respectively. Both LLMs had the same number of completely missing return type issues with 54 instances each. GPT-4o showed slightly more incorrect return types (14) and a higher number of naming issues (33 vs. 28) compared to DeepSeek (12), indicating a slightly higher tendency towards semantic inconsistencies in method identification. Finally, DeepSeek showed a higher number of missing implementations (8 compared to GPT-4o's 2), suggesting that it occasionally omits parts of the expected method logic.

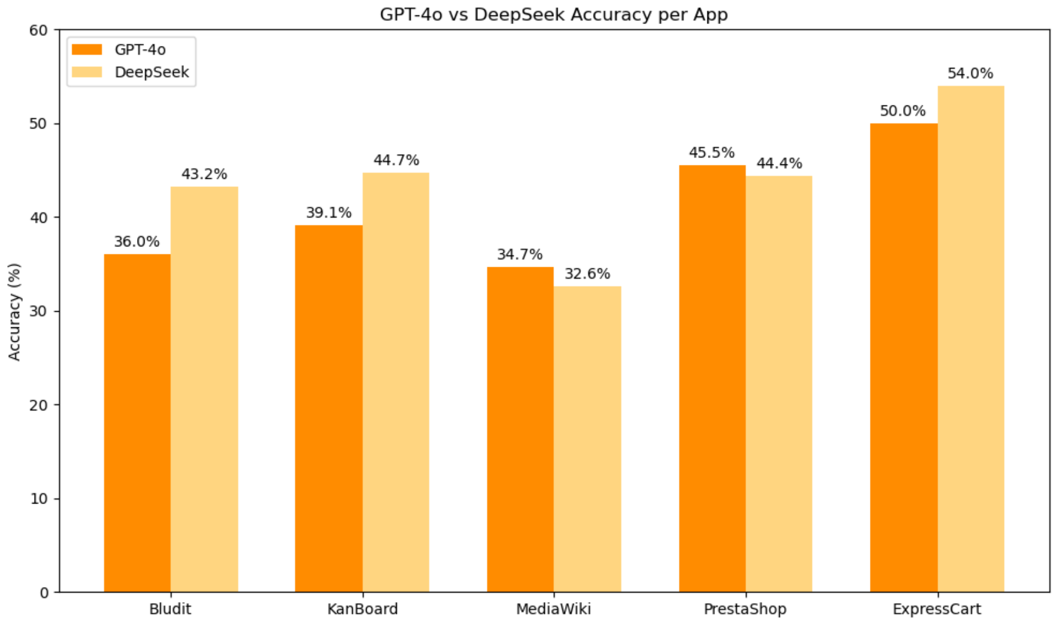

Figure 5.1: Accuracy Comparison of DeepSeek and GPT-4o Across All Apps

Figure 5.1 presents the overall accuracy comparison across all applications for both LLMs. In three of the five applications, Bludit, KanBoard and ExpressCart, DeepSeek outperformed GPT-4o and achieved higher accuracy scores. On the other hand, GPT-4o performed slightly better than DeepSeek in MediaWiki and PrestaShop applications. The overall results indicate that DeepSeek achieves better accuracy in page object creation.

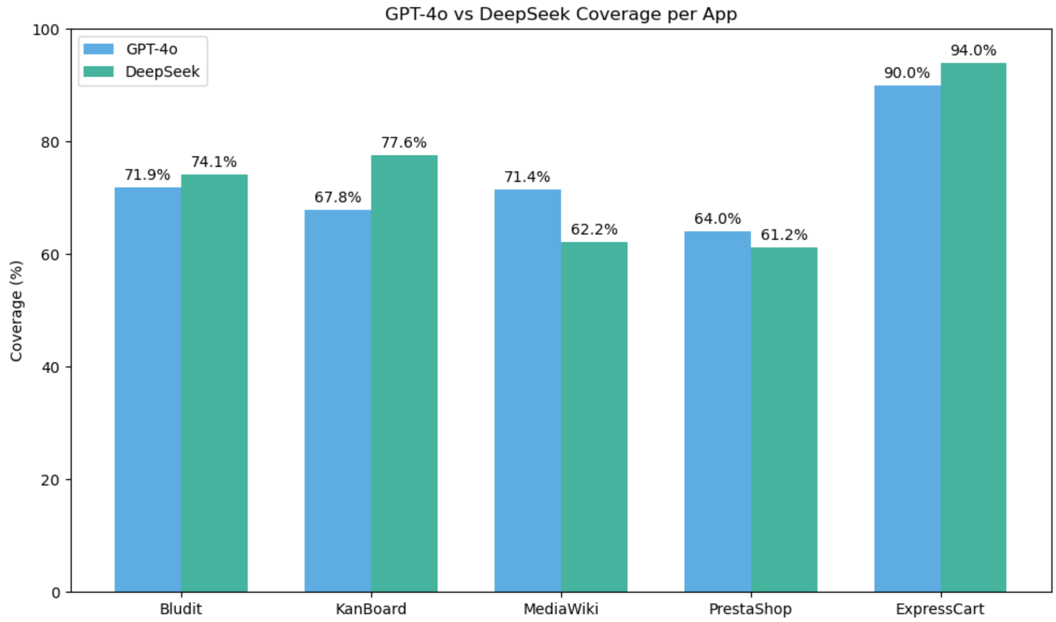

Figure 5.2: Coverage Comparison of DeepSeek and GPT-4o Across All Apps

Figure 5.2 shows the overall coverage comparison across all applications for both LLMs. Similar to the accuracy results, DeepSeek outperformed GPT-4o by achieving higher coverage values in three out of five applications (Bludit, KanBoard and ExpressCart). In contrast, GPT-4o performed slightly better than DeepSeek on MediaWiki and PrestaShop. In particular, the coverage achieved for ExpressCart was significantly higher than for the other applications. In this particular application, DeepSeek achieved 94% coverage while GPT-4o reached 90%. They performed both significantly better than the average coverage which was reported to be 73.4% across all applications.

The lowest results achieved in MediaWiki application with accuracy values 34.7% and 32.6% for GPT-4o and DeepSeek, respectively. On the other hand, the highest accuracy values were obtained in ExpressCart application with 50% in GPT-4o and 54% in DeepSeek.

Comparing the coverage values, PrestaShop showed the lowest performance with 64% in GPT-4o and 61.2% in DeepSeek. On the other hand, GPT-4o and DeepSeek presented the highest coverage values in ExpressCart application, reaching 90% and 94% respectively.

It is important to emphasize that all Page Objects produced by both LLMs exhibit proper class naming conventions, syntactically correct Java code with all necessary imports, and correctly defined constructors.

The following listings illustrate an example Page Object (PO) class with various implementations. Listing 5.1 shows a manually created PO class from the benchmark. This class includes core action methods for interacting with input fields, such as setUsername and setPassword, as well as getter methods like getAlertText. Two different login methods can also be seen in this PO. These methods simulates different login scenarios. The first method performs a standard login and goes to the AdminHome page when successful. The second method, badLogin, does a failed login attempt, which is expected to trigger and display a warning message.

Listing 5.1: Manually implemented AdminLogin page from Bludit application

```java
package po;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import po.excluded.AdminHome;
import po.excluded.PageObject;

public class AdminLogin extends PageObject {

	@FindBy(id = "jsusername")
	protected WebElement username;

	@FindBy(id = "jspassword")
	protected WebElement password;

	@FindBy(name = "save")
	protected WebElement loginBtn;

	@FindBy(id = "alert")
	protected WebElement alert;

	public AdminLogin(WebDriver driver) {
		super(driver);
	}

	public AdminLogin setUsername(String usr) {
		username.sendKeys(usr);
		return this;
	}

	public AdminLogin setPassword(String psw) {
		password.sendKeys(psw);
		return this;
```

```
        }

        public AdminHome login() {
                loginBtn.click();
                return new AdminHome(driver);
        }

        public AdminLogin badLogin() {
                loginBtn.click();
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                return new AdminLogin(driver);
        }

        public boolean isAlertVisible() {
                return alert.isDisplayed();
        }

        public String getAlertText() {
                return alert.getText();
        }
}
```

Listing 5.2 presents the GPT-4o generated implementation of AdminLogin, the same Page Object (PO) class. As shown in the code, it contains proper import statements and follows the correct Java syntax. The class name is correct and the constructor is correctly identified, initializes the WebDriver and sets up elements using Selenium's PageFactory. Compared to the manually implemented PO, this version includes an additional WebElement for the Remember Me checkbox and a corresponding action method to set its value. However, there are a few limitations: the clickLoginButton method lacks a proper return type and returns void instead of going to the appropriate page. Moreover, it does not have a custom implementation for a failed login attempt and it does not include getter methods to retrieve warning messages. Despite these problems, the basic functionality and overall structure of the code is reasonable and in line with standard Page Object design principles.

Listing 5.2: AdminLogin Page Object Model implemented by GPT-4o

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
```

```java
public class AdminLogin {

    public WebDriver driver;

    @FindBy(id = "jsusername")
    private WebElement usernameInput;

    @FindBy(id = "jspassword")
    private WebElement passwordInput;

    @FindBy(id = "jsremember")
    private WebElement rememberCheckbox;

    @FindBy(xpath = "//button[@type='submit']")
    private WebElement loginButton;

    public AdminLogin(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    public void enterUsername(String username) {
        usernameInput.clear();
        usernameInput.sendKeys(username);
    }

    public void enterPassword(String password) {
        passwordInput.clear();
        passwordInput.sendKeys(password);
    }

    public void clickRememberMe() {
        if (!rememberCheckbox.isSelected()) {
            rememberCheckbox.click();
        }
    }

    public void clickLoginButton() {
        loginButton.click();
    }
}
```

Listing 5.3 shows the implementation of the AdminLogin Page Object created by DeepSeek. Similar to GPT-4o, it includes the required import statements, a properly defined constructor and provides an additional WebElement for the Remember Me checkbox. However, unlike GPT-4o, DeepSeek correctly infer that a login method may need to go to another page. As a result, it provides two different implementations: one version of the login method stays on the same page and returns void, while the other goes to DashboardPage and lets the developer decide which one to use. Furthermore, DeepSeek organizes the defined elements by type, grouping them under clearly marked comment sections such as Login Fields, Interaction Methods, Constructor and Navigation Methods. This structured approach improves readability and demonstrates an awareness of clean coding practices.

Listing 5.3: AdminLogin Page Object Model implemented by DeepSeek

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class Adminlogin {
    public WebDriver driver;

    // Input fields
    @FindBy(id = "jsusername")
    private WebElement usernameInput;

    @FindBy(id = "jspassword")
    private WebElement passwordInput;

    // Checkbox
    @FindBy(id = "jsremember")
    private WebElement rememberCheckbox;

    // Button
    @FindBy(name = "save")
    private WebElement loginButton;

    // Constructor
    public AdminLogin(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    // Interaction methods
    public void setUsername(String username) {
```

```java
        usernameInput.clear();
        usernameInput.sendKeys(username);
    }

    public void setPassword(String password) {
        passwordInput.clear();
        passwordInput.sendKeys(password);
    }

    public void toggleRememberMe() {
        rememberCheckbox.click();
    }

    // Assuming login leads to another page - adjust return type as needed
    public DashboardPage clickLogin() {
        loginButton.click();
        return new DashboardPage(driver);
    }

    // If login stays on same page, use this instead:
    // public void clickLogin() {
    // loginButton.click();
    // }
}
```

Another strength of DeepSeek's implementation is the addition of a Notes section at the end of each Page Object class, as shown in Listing 5.4. This section provides as a summary of the decisions taken during code generation by DeepSeek. In this implementation in particular, it highlights specific implementation details of the navigational login methods and advises the developer to choose the appropriate version depending on the actual behavior of the application. It is a useful section that increases transparency and helps developers to more effectively integrate the generated code and understand the implementation decisions made by LLM.

Moreover, it adds a section to remind the developer of possible adjustments which might be necessary. For instance, it suggests creating an additional Page Object class for the target page (e.g. DashboardPage) if it does not already exist, and also adjusting the return type of the navigation input method accordingly. These suggestions can help bridge the gap between the generated code and the real-world application structure. It might also make the integration process more intuitive and reduce the chance of runtime errors.

Listing 5.4: AdminLogin PO Additional Notes by DeepSeek

```
Notes:
1. I've named all elements based on their structural position
(usernameInput, passwordInput, etc.) as required.
2. The login button method returns a 'DashboardPage' assuming that's where the login
leads - you should replace this with the actual page class name that comes after login.
3. If there's no page navigation after login, you can use the commented void version
instead.
4. I didn't find any error message elements in the provided HTML,
so no error handling methods were included.
5. All naming follows camelCase convention.
6. The class name and file name match exactly as required ('AdminLogin').

Remember to:
1. Create the 'DashboardPage' class (or whatever your actual post-login page is)
2. Import the appropriate packages
3. Adjust the return type of 'clickLogin()' based on your actual application flow
```

Following the general overview of the results, the following sections provide detailed analysis of each individual application. The results highlight the specific values obtained during experiments, along with accuracy and coverage values achieved.

5.1 Results for Bludit

The first application we will examine in detail is Bludit which is a content management tool. Tables 5.3 and 5.4 present the detailed classification of each page object tested for this application using GPT-4o and DeepSeek respectively.

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AddUserPage	5	0	0	0	0	0	5	1	0	0	0	0	2	0	0	0
AdminLogin	3	0	0	0	0	0	2	1	0	0	0	0	1	0	1	0
AdminSidebar	4	0	0	0	0	0	0	4	0	0	0	0	8	0	0	8
CategoriesPage	2	0	0	0	0	0	0	1	2	3	0	2	0	0	1	0
ContentPage	4	0	0	0	0	0	2	2	1	3	0	0	10	0	3	7
EditCategoryPage	2	0	0	0	0	0	2	0	1	0	0	1	5	0	4	1
EditUserPage	2	0	0	0	0	0	0	2	0	0	0	0	2	0	0	2
GeneralSettingsAbstractPage	5	0	0	0	0	0	0	4	0	0	0	0	3	0	0	4
GeneralSettingsPage	1	1	0	1	0	0	1	0	0	0	0	0	3	3	3	0
LanguagePage	0	1	0	0	0	0	1	1	0	0	0	0	4	3	3	0
NewCategoryPage	3	0	0	0	0	0	2	1	0	0	0	0	1	0	1	0
NewContentPage	5	0	0	0	0	0	7	1	4	4	3	2	18	0	15	0
SiteSocialsPage	1	1	0	0	0	0	1	1	0	0	0	0	9	9	9	0
SocialNetworksPage	3	0	0	0	0	0	2	1	0	2	0	0	6	0	7	0
UserSecurityPage	4	0	0	0	0	0	1	3	2	0	2	0	4	1	3	0
UsersPage	2	0	0	0	0	0	0	1	2	3	0	2	12	7	0	1
Total	**46**	**3**	**0**	**1**	**0**	**0**	**26**	**24**	**12**	**15**	**5**	**7**	**106**	**23**	**47**	**22**

Table 5.3: Detailed classification of Page Object review using GPT-4o

To begin with, when the items correctly identified by the LLMs are analyzed, it is seen that the vast majority are WebElements, accounting for 46 of the 50 correctly identified items. Only 4 of these correctly identified elements are methods: 3 are getter methods and 1 is a navigation method. Furthermore, when analyzing the elements that need to be modified, it can be observed that there are no web elements or getter methods classified in this group, instead only action and navigation methods need to be modified. In the category of missing elements, the results are spread throughout all four categories, with getter methods making up the largest portion. This is followed by WebElements, then navigation methods and finally action methods. Lastly, elements created by LLMs that are not existent in the ground truth are grouped under the Extra column. The vast majority of these extra elements are WebElements. In particular, the number of extra elements created by LLMs is significantly higher compared to the actual page objects defined in the ground truth.

A closer examination of Table 5.3 reveals the presence of several outliers. As can be seen from the table, most page objects have relatively few missing elements, or at least missing elements are not seem to be the dominant classification. However, some page objects stand out with a significant proportion of their ground truth elements marked as missing.

For example, the NewContentPage ground truth PO actually contains 26 elements, of which 13 elements are labeled as missing. Similarly, CategoriesPage contains only 2 correctly identified elements, 1 element that requires modification, and 7 elements that are missing. These special cases particularly reduced the overall accuracy and potentially point to limitations in the

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AddUserPage	5	0	0	0	0	0	5	1	0	0	0	0	2	0	0	1
AdminLogin	3	0	0	0	0	0	2	1	0	0	0	0	1	0	1	0
AdminSidebar	4	0	0	0	0	0	0	4	0	0	0	0	9	1	0	8
CategoriesPage	3	2	0	0	0	0	0	1	1	1	0	2	7	7	0	3
ContentPage	5	2	0	0	0	0	2	2	0	1	0	0	21	9	4	7
EditCategoryPage	2	0	0	0	0	0	2	0	1	0	0	1	7	4	5	0
EditUserPage	2	0	0	0	0	0	0	2	0	0	0	0	2	0	0	2
GeneralSettingsAbstractPage	5	0	0	0	0	0	0	4	0	0	0	0	4	0	0	4
GeneralSettingsPage	1	1	0	1	0	0	1	0	0	0	0	0	3	3	4	0
LanguagePage	0	0	0	1	0	1	1	0	0	0	0	0	4	3	3	0
NewCategoryPage	3	0	0	1	0	0	2	0	0	0	0	0	2	0	1	0
NewContentPage	5	0	0	0	0	0	3	1	4	4	6	3	19	2	10	0
SiteSocialsPage	1	1	0	0	0	0	1	0	0	0	0	1	9	9	9	0
SocialNetworksPage	2	2	0	0	0	0	2	0	1	0	0	1	7	7	7	0
UserSecurityPage	4	0	0	0	0	0	2	2	2	0	1	1	2	2	1	0
UsersPage	3	1	0	0	0	0	0	1	1	2	0	2	11	4	1	1
Total	**48**	**9**	**0**	**3**	**0**	**1**	**23**	**19**	**10**	**8**	**7**	**11**	**110**	**51**	**46**	**26**

Table 5.4: Detailed classification of Page Object review using Deepseek

LLM's understanding of these page objects or possible inconsistencies in their ground truth definitions.

Table 5.4 presents the results obtained for Bludit using DeepSeek and the results shows a similar pattern to what observed with GPT-4o. Similar to GPT-4o, DeepSeek correctly identifies most of the WebElements, while the elements that require modification are primarily actions and navigation methods. Missing elements are also more evenly distributed across all categories.

However, a key difference is that DeepSeek shows improved performance by accurately identifying a greater number of items. It also produces fewer items that require modification or are completely missing.

DeepSeek defines 233 extra elements/methods over the entire app, whereas this amount is reported as 198 in GPT-4o, which implies 17.68% increase in this extra generated content.

LLM	#Correct	#To Modify	#Missing	#Extra	Accuracy (%)	Coverage (%)
OpenAI	50	50	39	198	36.0	71.9
DeepSeek	60	43	36	233	43.2	74.1

Table 5.5: Summary of PO classification results for Bludit. Counts (Correct, To Modify, Missing, Extra) and total are shown alongside derived metrics: Accuracy = Correct / Total, Coverage = (Correct + To Modify) / Total.

Table 5.5 provides the comparison of the performance of both GPT-4o and DeepSeek in Bludit application. Each page object element is categorized according to predefined categories, and with these values both accuracy and coverage metrics are reported. As explained in the previous section, accuracy in this context refers to the proportion of correctly identified items among all of the generated items, while coverage refers to the percentage of correctly identified items and items requiring changes over all items. DeepSeek achieved a slightly better coverage value of 74.1%. DeepSeek also shows a higher accuracy rate of 43.2% compared to GPT-4o with 36%, indicating a better precision in producing accurate outputs.

LLM	#Correct	#To Modify						#Missing
		Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.	Total	
OpenAI	50	21	0	28	1	0	50	39
DeepSeek	60	13	3	22	1	4	43	36

Table 5.6: Detailed classification of Bludit PO modifications with comparison of ChatGPT-4o and DeepSeek. Subcategories of "To Modify" include various return type and naming issues.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
OpenAI	42.0 %	0.0%	56.0%	2.0%	0.0%
DeepSeek	30.2%	7.0%	51.2%	2.3%	9.3%

Table 5.7: Percentage breakdown of "To Modify" subcategories for Bludit PO classification by ChatGPT-4o and DeepSeek. Percentages are relative to total modifications.

Table 5.6 and 5.7 demonstrate the detailed breakdown of the classification for Bludit, focusing specifically on the elements that require modification. From the results, it can be seen that the largest percentage of elements requiring changes are linked to return type issues, a finding that is consistent with the complexity involved in properly understanding and creating page object structures. Among return type issues, the most common categories are completely missing return types and cases where the return type is missing but is typically expected to return to the same page object for continuity of test flow.

Moreover, GPT-4o did not produce any instances where the return type was incorrectly linked to a different page object. In contrast, DeepSeek exhibited a 7% error rate in this category, with elements returning the wrong page object when another was expected. Both LLMs had small error rates regarding the wrong naming. The values reported were 2.1% in ChatGPT-4o, and 2.3% in DeepSeek.

Lastly, there were a few instances in DeepSeek where methods were generated with empty implementations, a pattern that was not observed in ChatGPT-4o, which consistently provided non-empty method bodies.

Listing 5.5: DeepSeek missing implementation methods example

```
// Language dropdown methods
public void selectLanguage(String languageValue) {
    // Implementation to select language from dropdown
}

public String getSelectedLanguage() {
    return ""; // Implementation to get selected language
}

// Timezone dropdown methods
public void selectTimezone(String timezoneValue) {
    // Implementation to select timezone from dropdown
}
```

A distinctive implementation pattern exhibited only by DeepSeek is the inclusion of multiple implementation options when the model is uncertain about the expected behavior. This approach allows developers to manually make a final decision based on the specific application context.

For example, in the following code fragment, DeepSeek is unsure whether clicking the login button will lead to a navigation event (i.e. a redirect to a new page) or simply perform an in-place action. It offers two variants of methods to handle both possibilities. The first one assumes that clicking the login button navigates to a new page and therefore returns a DashboardPage object. The second assumes no navigation is required and it returns void from the method. In the PO, the second method is left commented out and asks the developer to choose whichever is correct.

This dual implementation strategy can be useful for developers as it clearly presents alternative implementation choices. It can also reduce uncertainty and allow the developer to choose the right one based on the test and application flow.

Listing 5.6: DeepSeek multiple implementation options

```
// Assuming login leads to another page - adjust return type as needed
public DashboardPage clickLogin() {
      loginButton.click();
     return new DashboardPage(driver);
}
// If login stays on same page, use this instead:
// public void clickLogin() {
// loginButton.click();
// }
```

5.2 Results for Kanboard

The second application we will examine in detail is KanBoard, a project management tool. Tables 5.8 and 5.9 present the detailed classification of each page object tested for this application using GPT-4o and DeepSeek.

PO Class	**Correct**				**To Modify**				**Missing**				**Extra**			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AddNewProjectPage	2	0	0	0	0	0	0	1	1	1	1	0	6	0	6	0
ApplicationSettingsPage	2	0	0	0	0	0	2	0	0	2	0	0	17	0	5	10
BoardSettingsPage	2	0	0	0	0	0	2	0	0	1	0	0	2	0	2	0
CurrencyRatesPage	5	0	1	0	0	0	4	0	2	3	0	0	2	0	2	0
EditProjectPage	2	0	0	0	0	0	2	0	0	0	0	0	12	0	12	0
EditUserProfilePage	2	0	0	0	0	0	1	1	0	0	0	0	6	0	6	0
KanboardHomePage	3	0	0	0	0	0	0	1	3	0	0	5	8	1	2	7
LoginPage	3	0	0	0	0	0	0	1	0	0	0	0	2	0	3	1
NewUserPage	6	0	0	0	0	0	6	1	3	3	1	0	10	0	8	0
ProjectSidebar	6	0	0	5	0	0	0	2	0	0	0	0	11	0	0	10
ProjectSummaryPage	0	1	0	0	3	3	0	0	2	1	0	0	4	4	0	0
SettingsSidebar	3	0	0	0	0	0	0	3	0	0	0	0	8	0	0	8
SwimlanesPage	3	0	0	0	0	0	3	0	1	2	0	0	5	0	5	0
TaskSidebar	1	0	0	0	0	0	0	1	0	0	0	0	18	0	0	18
TaskPage	1	1	0	0	0	0	0	0	0	0	0	0	14	9	5	0
TopNavBar	2	0	0	1	0	0	0	1	1	0	0	0	7	0	2	5
UsersManagementPage	1	0	0	0	1	0	0	2	2	1	1	0	26	0	8	17
UserSummaryPage	1	0	0	0	0	0	0	1	4	4	0	0	15	0	0	15
PermissionsPage	1	0	0	0	0	0	0	0	0	1	0	0	5	2	4	0
Total	**46**	**2**	**1**	**6**	**4**	**3**	**20**	**15**	**19**	**19**	**3**	**5**	**178**	**16**	**70**	**91**

Table 5.8: Detailed classification of Page Object review of Kanboard using GPT-4o

First of all, when analyzing the correctly identified elements in the tables generated by the LLMs, it is clear that most of them belong to the WebElements category. This is to be expected, because WebElements are usually simpler to define, since they do not involve inter-page relationships or complex return type logic. A WebElement usually only needs a reasonable name and the correct type declaration (WebElement) to be classified correctly. Therefore, the proportion of correctly identified WebElements is relatively high, with only 4 out of 42 items requiring modification generated by GPT-4o being WebElements.

In contrast, navigation methods pose a greater challenge. Of the 26 navigation methods created by GPT-4o, only 6 were defined completely correctly. Most of the elements that required modification were either action methods or navigation methods, with only a small number of WebElements and getter methods in this category.

Looking at the missing elements category, it can be seen that GPT-4o struggled to identify getter methods and WebElements, with 19 missing instances in each category. On the other hand, it was not able to identify only 3 action methods and 5 navigational methods which indicates that GPT-4o was relatively more successful in capturing higher-level interactions than lower-level UI components.

Lastly, elements generated by the LLMs that do not exist in the manually defined Page

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AddNewProjectPage	3	1	0	1	0	0	0	0	0	0	1	0	14	0	15	0
ApplicationSettingsPage	2	0	1	0	0	0	1	0	0	2	0	0	18	1	6	10
BoardSettingsPage	2	1	1	0	0	0	1	0	0	0	0	0	5	2	2	0
CurrencyRatesPage	4	1	0	0	0	0	3	0	3	2	2	0	5	1	4	0
EditProjectPage	2	0	0	0	0	0	2	0	0	0	0	0	12	0	12	0
EditUserProfilePage	2	0	0	0	0	0	1	1	0	0	0	0	9	5	6	0
KanboardHomePage	4	0	0	0	0	0	0	5	2	0	0	1	9	0	2	6
LoginPage	3	0	0	0	0	0	0	1	0	0	0	0	2	0	3	1
NewUserPage	7	1	0	1	0	0	6	0	2	2	1	0	12	2	9	0
ProjectSidebar	6	0	0	0	0	0	0	7	0	0	0	0	11	0	0	10
ProjectSummaryPage	0	1	0	0	3	3	0	0	2	1	0	0	6	5	0	0
SettingsSidebar	0	0	0	0	3	0	0	3	0	0	0	0	8	0	0	8
SwimlanesPage	4	1	0	0	0	0	3	0	0	1	0	0	12	2	7	0
TaskSidebar	1	0	0	0	0	0	0	1	0	0	0	0	19	1	0	18
TaskPage	1	1	0	0	0	0	0	0	0	0	0	0	21	10	1	10
TopNavBar	2	0	0	2	0	0	0	0	1	0	0	0	8	1	0	7
UsersManagementPage	1	0	0	0	1	0	0	2	2	1	1	0	30	2	12	16
UserSummaryPage	4	3	0	0	0	0	0	0	1	1	0	1	24	9	0	15
PermissionsPage	0	0	0	0	0	0	0	0	1	1	0	0	10	3	5	0
Total	**48**	**10**	**2**	**4**	**7**	**3**	**17**	**20**	**14**	**11**	**5**	**2**	**253**	**44**	**82**	**95**

Table 5.9: Detailed classification of Page Object review of Kanboard using DeepSeek

Objects are categorized under the Extra column. The overwhelming majority of these were WebElements, totaling 178, followed by 91 navigational methods, 70 action methods, and only 16 getter methods. The total number of extra elements generated was significantly higher than the actual number of elements in the ground truth.

Table 5.9, which presents the results obtained for the Kanboard application using DeepSeek, reveals a pattern similar to GPT-4o. As in GPT-4o, the largest proportion of correctly identified items are WebElements. However, one notable difference DeepSeek showed is that it was able to capture more getter methods correctly and had fewer missing ones compared to GPT-4o. This difference contributed to the overall accuracy of DeepSeek in the Kanboard application.

LLM	#Correct	#To Modify	#Missing	#Extra	Accuracy (%)	Coverage (%)
OpenAI	55	42	46	355	39.1	67.8
DeepSeek	64	47	32	474	44.7	77.6

Table 5.10: Summary of PO classification results for Bludit. Counts (Correct, To Modify, Missing, Extra) and total are shown alongside derived metrics: Accuracy = Correct / Total, Coverage = (Correct + To Modify) / Total.

The number of elements requiring modification is comparable between the two models. DeepSeek presented 20 navigation method, followed by 17 action methods, 7 WebElements and 3 getter methods. While DeepSeek has fewer missing elements overall, the distribution of element types is similar to what was observed in GPT-4o.

LLM	#Correct	#To Modify						#Missing
		Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.	Total	
OpenAI	55	6	1	18	17	0	42	46
DeepSeek	64	12	5	13	15	2	47	32

Table 5.11: Detailed classification of Kanboard PO modifications with comparison of ChatGPT-4o and DeepSeek. Subcategories of "To Modify" include various return type and naming issues.

Consistent with the findings from the Bludit application, DeepSeek generated a significantly higher number of extra elements, totaling 474 which represents a 33.52% increase compared to GPT-4o's 355 extra elements.

Inspecting the table of total values obtained using both LLMs for the Kanboard application shows a clear difference between the performances of both LLMs. DeepSeek correctly identifies a total of 64 elements, outperforming GPT-4o, which correctly identifies only 55 elements. Furthermore, DeepSeek presents 32 missing elements while GPT-4o presentes 46 missing elements. These differences in the number of both missing and complete elements results in a performance gap in the completeness of the generated Page Objects.

These figures translate into better accuracy and coverage results for DeepSeek. In terms of accuracy, DeepSeek outperforms GPT-4o with 44.5%, compared to 39.1%. In terms of coverage, DeepSeek again outperforms GPT-4o, reaching 77.6%, whereas GPT-4o was able to get only 67.8%. Overall, these findings show that DeepSeek produces more accurate Page Objects, specifically for the Kanboard application.

Furthermore, another difference lies in the number of extra elements generated by the LLMs. DeepSeek provided 474 more elements which were not present in the manually defined Page Objects, whereas GPT-4o provided only 355. While both models performed similarly in terms of generating extra navigation and action items, a deeper examination of the detailed tables reveals that DeepSeek generated more getter methods and item definitions. In several instances within the generated Page Objects, DeepSeek consistently added getter methods for error and success messages as well as textual content, but GPT-4o frequently failed to discover or generate these elements.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
OpenAI	14.3%	2.4%	42.9%	40.5%	0.0%
DeepSeek	25.5%	10.6%	27.7%	31.9%	4.3%

Table 5.12: Percentage breakdown of "To Modify" subcategories for Kanboard PO classification by ChatGPT-4o and DeepSeek. Percentages are relative to total modifications.

Tables 5.11 and 5.12 provide a detailed breakdown of the changes required to the Page Object

(PO) definitions for the Kanboard application, comparing GPT-4o and DeepSeek outputs. As can be seen, both models produced a significant number of items requiring manual correction, with 42 correction for GPT-4o and 47 correction for DeepSeek. However, the distribution across subcategories highlights that there are ket differences in the types of problems encountered by each model.

A substantial portion of the elements requiring modification emerged from the return type issues which reflects a common challenge in the automating Page Object generation. In particular, GPT-4o exhibited a high rate of "Same Page RT" issues (42.9%), but was missing or incorrect. DeepSeek was also affected by this issue with 27.7%. However, it had more common issues with "Missing RT" (25.5%) and "Incorrect RT" (10.6%).

Both models resulted in struggling with the naming issues. In this specific category, GPT-4o requires 17 name corrections (40.5%) and DeepSeek requires 15 (31.9%). T These problems show that there are limitations in the semantic understanding of both LLMs, especially in this application.

DeepSeek produced two methods with missing implementations, whereas GPT-4o did not show this behavior. This may suggest that although DeepSeek tries to define a broader set of elements, it sometimes does so without completing the method logic.

Overall, it can be seen that while DeepSeek generated more correctly defined elements (64 vs. 55) and had less missing (32 vs. 46), it also introduced a slightly higher number of elements that would require modification.

In Page Object generated by each LLM, a few remarkable patterns were observed. For example, in GPT-4o outputs, even though it was clear that the navigation method should transition to another page, they were defined with a void return type. In those cases, GPT-4o often included a commented line to tell the developer there might be another PageObject expected to be returned from the method. The model usually explained that it assumed the method redirected to another page, but it has not been successful to determine the exact target Page Object. As a result, it assigned a placeholder name based on the functionality itself. For instance, calling it MoveToAnotherProjectPage to prompt the developer to manually replace it with the correct return type.

Listing 5.7: GPT-4o comments about return type of navigational methods

```
public void clickMoveToAnotherProjectLink() {
    moveToAnotherProjectLink.click();
    // Assuming navigation to a new page, return the appropriate POM class
    // return new MoveToAnotherProjectPage(driver);
}

public void clickSendByEmailLink() {
    sendByEmailLink.click();
    // Assuming navigation to a new page, return the appropriate POM class
    // return new SendByEmailPage(driver);
}

public void clickMovePositionLink() {
    movePositionLink.click();
    // Assuming navigation to a new page, return the appropriate POM class
    // return new MovePositionPage(driver);
}
```

Moreover, there was a specific Page Object in which DeepSeek consistently failed to generate meaningful return types for its navigational methods. Instead of returning the appropriate target Page Object, the model simply returned the current Page Object itself by neglecting the navigation intent of the methods. This behavior is illustrated in Listing 5.8.

Listing 5.8: DeepSeek failing to return correct Page Object in KanboardHomePage

```
 public Kanboardhomepage clickNewProjectLink() {
    newProjectLink.click();
    return this;
}

public Kanboardhomepage clickNewPersonalProjectLink() {
    newPersonalProjectLink.click();
    return this;
}

public Kanboardhomepage clickProjectManagementLink() {
    projectManagementLink.click();
    return this;
}

// Interaction methods for projects
```

```
public Kanboardhomepage clickFirstProjectLink() {
    firstProjectLink.click();
    return this;
}

public Kanboardhomepage clickSecondProjectLink() {
    secondProjectLink.click();
    return this;
}

// Interaction methods for tasks
public Kanboardhomepage clickTaskLink() {
    taskLink.click();
    return this;
}
```

In this application, in addition to incorrect return type issues, we also observed several cases of inconsistent or incorrect method naming, particularly in navigation methods. While LLMs generally correctly understood the intent and returned a reasonably appropriate Page Object, the method names themselves did not align with the ground truth rules and it resulted in inconsistencies from expected behavior. For example, in Listing 5.9, the ground truth defines a save() method that returns a UserSummaryPage that accurately reflects navigation following a save action. However, the method created by LLM is called clickSaveButton() and returns SummaryPage, which, although semantically similar, does not exactly match.

Other examples include: newPersonalProject() vs. clickNewPersonalProjectLink() returning AddNewProjectPage vs. NewPersonalProjectPage loginToKanboard(String username, String psw) vs. clickSignInButton() returning KanboardHomePage vs. HomePage

Listing 5.9: OpenAI failing to return correct Page Object in several pages with wrong naming

```
public UserSummaryPage save()
public SummaryPage clickSaveButton()

public AddNewProjectPage newPersonalProject()
public NewPersonalProjectPage clickNewPersonalProjectLink()

public KanboardHomePage loginToKanboard(String username, String psw)
public HomePage clickSignInButton()
```

5.3 Results for MediaWiki

The next application we could like to assess in detail is MediaWiki which is a collaborative wiki platform. Tables 5.13 and 5.14 demonstrates the detailed results of each page object class which was created with LLMs, GPT-4o and DeepSeek in MediaWiki.

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
BlockUserPage	3	1	0	0	0	0	4	0	1	0	0	0	10	0	8	0
ChangeCredentialsPage	3	0	0	0	0	0	3	0	0	2	1	0	2	1	1	0
CreateAccountPage	5	0	0	0	0	0	5	0	0	2	0	0	1	0	1	0
LoginPage	3	0	0	0	0	0	2	1	0	0	0	0	2	0	2	0
DeletePage	1	0	0	0	0	0	1	0	0	1	0	0	4	0	3	1
EditSourcePage	2	0	0	0	0	0	1	1	0	0	1	0	8	0	6	2
PageProtectPage	1	0	0	0	0	0	1	1	0	1	0	0	11	0	10	0
SearchResultsPage	1	0	0	0	0	0	0	0	0	0	0	1	7	2	6	0
SpecialPagesPage	2	0	0	0	0	0	0	2	2	0	0	2	15	1	0	13
UserRightsPage	4	1	1	0	0	0	2	0	1	1	1	0	7	0	5	0
MainPage	3	0	0	0	0	0	0	4	1	0	0	1	19	2	0	14
PageCreationPage	2	0	1	0	3	0	3	2	3	0	3	3	2	0	2	0
Total	**30**	**2**	**2**	**0**	**3**	**0**	**22**	**11**	**8**	**7**	**6**	**7**	**88**	**6**	**44**	**30**

Table 5.13: Detailed classification of Page Object review of MediaWiki using GPT-4o

When examining Table 5.13, it can be seen that MediaWiki has a relatively low number of elements correctly defined in the Page Objects. A total of 34 elements matched those found in the manually generated files. As with other applications, the majority of correctly defined elements were WebElements. Additionally, there were 2 correctly implemented getter methods and 2 action methods. However, no navigational methods were fully and correctly defined by GPT-4o.

For the first time, the number of elements requiring modification (36) exceeded the number of correctly defined elements. Most of these needed changes were action methods, followed by navigational methods and WebElements. There was a uniform distribution in the types of elements missing from the Page Objects generated by the language model. Lastly, the total number of extra elements was 168, presenting a significant reduction compared to previous applications.

Subsequently, examining Table 5.14 presents a detailed classification of MediaWiki Page Objects generated by DeepSeek. A total of 32 elements are classified as correct, of these, 28 are WebElements. Furthermore, 3 getter methods and 1 navigation method are correctly identified, while no action methods are found to be correctly implemented. In contras, 23 of the elements requiring changes were action methods. DeepSeek had only 4 navigation methods that required changes, compared to 11 methods in GPT-4o.

Table 5.14 also reveals an increase in the total number of missing elements compared to Table 5.13. The largest number of missing elements are navigation methods (13), followed by WebElements (11), action methods (7) and getter methods (6). Finally, there has been a definite increase in the number of extra elements produced by DeepSeek, reaching a total of

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
BlockUserPage	3	1	0	0	0	0	4	0	1	0	0	0	11	1	8	0
ChangeCredentialsPage	3	1	0	0	0	0	3	0	0	1	1	0	3	1	1	0
CreateAccountPage	5	0	0	0	0	0	5	0	0	2	0	0	2	1	1	0
LoginPage	3	0	0	0	0	0	2	1	0	0	0	0	5	1	2	0
DeletePage	1	0	0	0	0	0	1	0	0	1	0	0	5	2	5	0
EditSourcePage	2	0	0	0	0	0	1	1	0	0	1	0	17	1	13	3
PageProtectPage	1	0	0	0	0	0	1	1	0	1	0	0	12	1	10	0
SearchResultsPage	1	0	0	0	0	0	0	0	0	0	0	1	9	1	6	2
SpecialPagesPage	2	0	0	0	0	0	0	0	2	0	0	4	22	3	3	5
UserRightsPage	3	1	0	0	1	0	4	0	1	1	0	0	19	0	17	0
MainPage	2	0	0	1	1	0	0	1	1	0	0	3	17	2	3	4
PageCreationPage	2	0	0	0	0	0	2	0	6	0	5	5	20	2	17	0
Total	**28**	**3**	**0**	**1**	**2**	**0**	**23**	**4**	**11**	**6**	**7**	**13**	**142**	**15**	**86**	**14**

Table 5.14: Detailed classification of Page Object review of MediaWiki using DeepSeek

257. Of these, 142 were WebElements, 86 were action methods, 15 were getter methods and 14 were navigation methods.

LLM	#Correct	#To Modify	#Missing	#Extra	Accuracy (%)	Coverage (%)
OpenAI	34	36	28	168	34.7	71.4
DeepSeek	32	29	37	257	32.6	62.2

Table 5.15: Summary of PO classification results for MediaWiki. Counts (Correct, To Modify, Missing, Extra) and total are shown alongside derived metrics: Accuracy = Correct / Total, Coverage = (Correct + To Modify) / Total.

Comparing the overall results obtained by the two LLMs, namely GPT-4o and DeepSeek in Table 5.15, the number of correctly identified items was relatively close, with GPT-4o identifying 34 and DeepSeek 32. This is the first instance in all applications where GPT-4o outperforms DeepSeek in terms of correctly identified items, resulting in better accuracy results.

Regarding elements requiring modification, GPT-4o had 36, while DeepSeek had 29. As indicated in the tables above, this difference is primarily due to the lower number of navigational methods flagged for modification in DeepSeek. Instead of being marked for modification, these navigational methods were categorized as missing, which negatively impacted DeepSeek's overall coverage score.

However, consistent with the results from other applications, DeepSeek again generated more extra elements that were not present in the ground truth files. Compared to GPT-4o, DeepSeek showed a 52.98% increase in the number of extra elements defined.

From the table, it is evident that the number of extra generated elements exceeds those found in the manually created POs. GPT-4o generated 168 extra elements, while DeepSeek generated

LLM	#Correct	#To Modify						#Missing
		Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.	Total	
OpenAI	34	7	7	18	0	0	36	28
DeepSeek	32	5	0	20	2	2	29	37

Table 5.16: Detailed classification of MediaWiki PO modifications with comparison of ChatGPT-4o and DeepSeek. Subcategories of “To Modify” include various return type and naming issues.

257 for the MediaWiki application. Considering that the manually constructed page objects consisted of only 98 elements in total, this means both LLMs produced significantly more extra elements. Approximately 1.7× and 2.6× the number of existing elements for GPT-4o and DeepSeek, respectively.

For this application, GPT-4o outperformed DeepSeek in both accuracy and coverage values. GPT-4o reached 34.7% accuracy, whereas DeepSeek could reach 32.6%. Moreover, there can be seen a significant difference in the coverage values for both LLMs. GPT-4o resulted in having 71.4% coverage value, outperforming DeepSeek, which achieved 62.2% overall score.

Table 5.16 provides a detailed breakdown of the “to modify” category and along with total number of elements. It can be observed that, in GPT-4o, 18 of the 36 elements requiring modification are due to missing “same page” return type problem. This accounts for half of the entire “to modify” category. Similarly, in the DeepSeek results, 20 out of 29 instances reported problems with the same page return type being missing, which corresponds to 69% of the entire category.

This issue has been consistently reported across all applications and LLMs as one of the main reasons which leads elements to be classified as requiring modification. However, it is important to note that returning the same page object from action methods is a design choice rather than a strict requirement in Page Object Models. Whether this is necessary or not depends largely on how the test suites are written.

For instance, if we examine the test structure in the manually generates test cases, we can observe a method chaining as shown in the example below Listing 5.10. In such scenarios, returning void instead of the same page object breaks the method chain, and results in failing test cases. Therefore, we classify such elements as requiring modification. However, it is important to clarify that returning the same page object is not a mandatory rule in page object design, but rather a rule that supports fluent interface patterns in test automation.

Listing 5.10: An example test case AddEmptyTask from benchmark

```
public class AddEmptyTask extends BaseTest {
    @Test()
    public void addEmptyTask() {
        ProjectManagementPage project = new LoginPage(driver)
```

```
                .loginToKanboard("admin", "admin")
                .goToFirstProject()
                .newTask()
                .setNewTaskName("")
                .saveNewTask();
        assertEquals("The␣title␣is␣required", project.getErrorMessage());
    }
}
```

The second most common return type issues was reported to be missing return type, meaning that the LLM returned just void but it was expected to return a page object instead, with percentages of 19.4% and 17.2% in GPT-4o and DeepSeek, respectively. Additionally, DeepSeek reported 2 cases where the naming was wrong and 2 more cases where the implementation was missing, which were not observed in GPT-4o results.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
OpenAI	19.4%	19.4%	50.0%	0.0%	0.0%
DeepSeek	17.2%	0.0%	69.0%	6.9%	6.9%

Table 5.17: Percentage breakdown of “To Modify” subcategories for MediaWiki PO classification by ChatGPT-4o and DeepSeek. Percentages are relative to total modifications.

Figure 5.3 shows a part of an example page from the MediaWiki application which is called SpecialPages. This page consists of a lot of links to various pages inside the application. Therefore, it is expected to have a lot of defined web elements for links and a lot of navigational methods inside a complete page object created for his page. However, the manually generated Page Object used for testing included only 4 WebElements and 4 navigational methods. Obviously, the manually page object generated was not complete since it was used to test only a few functionality, but it was expected from the LLM to generate a more extensive page object. In reality, though, the model generated only 5 additional navigational methods.

This limited output could be related to the length and complexity of the HTML input. The HTML file provided in the prompt consisted of approximately 500 lines, which is remarkably longer than the average HTML files used in other experiments. As mentioned in previous sections, large input files can potentially reduce the model's ability to comprehensively parse and understand all relevant parts of the page. This may lead the LLM to focus only on the most prominent elements or shorten its output due to internal constraints. It may result in an incomplete representation of all available navigation paths.

This was one of the reasons why we tried to truncate the HTML files and preprocess them manually before prompting them directly to the LLMs.

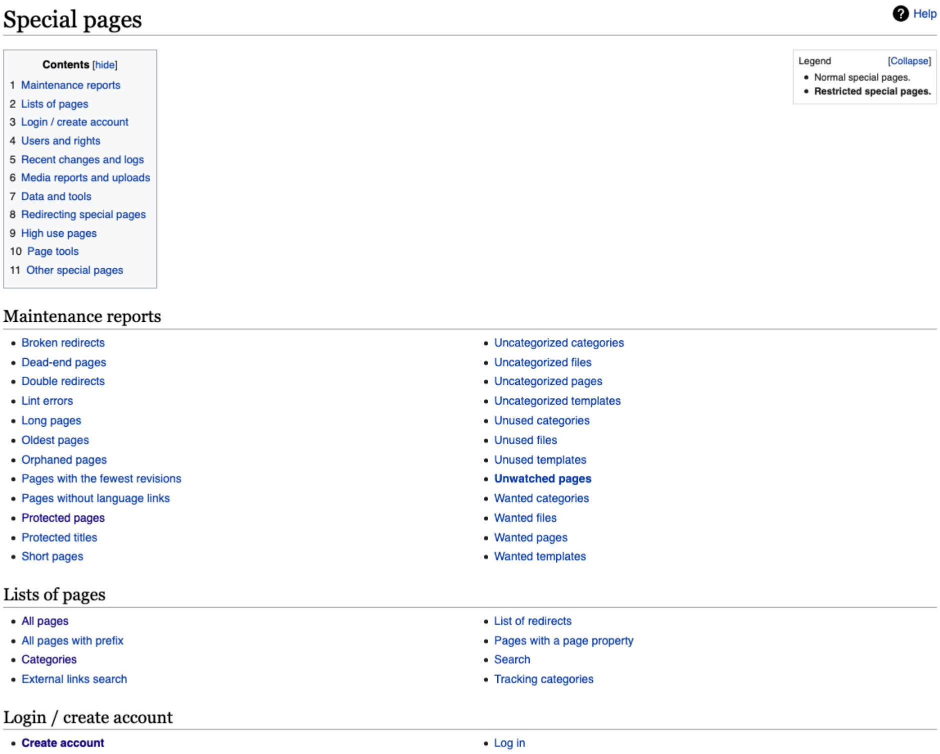

Figure 5.3: Special Pages page from MediaWiki application

5.4 Results for Prestashop

The fourth application we would like the discuss the results about is Prestashop which is an e-commerce platform offering a wide range of features. There were total of 24 page objects were compared for this application. Tables 5.18 and 5.19 represent the detailed classification results for this application for both LLMs used.

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AdminSidebar	15	0	0	0	1	0	0	7	5	0	0	4	55	0	0	64
AttributesPage	1	0	0	0	0	0	0	1	1	1	0	0	3	0	2	1
CategoriesPage	1	0	0	1	0	0	0	1	1	0	0	0	20	4	15	0
CurrencyPage	1	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0
FeaturesPage	1	0	0	1	1	0	0	0	0	1	0	0	4	1	2	2
OrdersPage	1	0	0	0	0	0	0	1	0	0	0	0	5	0	3	2
PrestaLogin	3	0	0	0	0	0	0	1	0	0	0	0	2	0	3	1
SupplierPage	1	0	0	1	0	0	0	0	1	0	1	0	16	1	12	3
AddCategoryPage	2	0	0	0	0	0	0	1	2	1	1	0	14	0	8	0
AddAttributePage	3	0	0	0	0	0	2	1	2	1	1	0	6	0	5	0
AddSupplierPage	5	0	0	0	0	0	0	1	2	2	1	0	14	0	15	0
EmployeesPage	1	0	0	0	0	0	0	1	1	0	0	1	14	0	13	0
EditEmployeePage	2	0	0	0	0	0	1	0	3	0	0	0	16	0	15	0
RegisterEmployeePage	5	0	0	0	2	0	0	1	3	4	0	5	8	0	11	0
AddAddressPage	11	1	0	0	0	0	0	1	1	0	1	1	7	0	16	0
AddFeaturesPage	2	0	0	0	0	0	1	1	2	1	1	0	5	0	4	0
StatesPage	2	0	0	0	0	0	0	2	1	1	0	0	4	0	2	1
TagsPage	2	1	0	0	0	0	0	1	0	0	0	0	2	0	2	0
AddCurrencyPage	5	1	0	0	0	0	0	1	5	2	1	0	7	0	7	0
AddNewStatePage	6	1	0	0	0	0	0	1	0	0	1	0	3	0	5	0
AddTagPage	3	1	0	0	0	0	0	1	0	0	1	0	6	0	4	0
Total	**73**	**5**	**0**	**3**	**4**	**0**	**4**	**25**	**30**	**14**	**9**	**11**	**212**	**6**	**145**	**74**

Table 5.18: Detailed classification of Page Object review of PrestaShop using GPT-4o

The table 5.18 for GPT-4o's classification of PrestaShop Page Objects shows a total of 81 correctly defined elements, consisting of 73 WebElements, 5 getter methods, and 3 navigation methods. Notably, there were no action methods correctly identified. In terms of elements which require modifications, GPT-4o identified 33 elements in total. Of these, 25 were navigation methods, which appears to be the most frequent category requiring adjustment. This is followed by a small number of action methods and web element modifications. Looking at missing elements, GPT-4o fell short in 64 cases, distributed among WebElements (30), getter methods (14), action methods (9), and navigation methods (11). Taking a look at the extra defined elements column, GPT-4o generated a total of 437 extra elements. The majority of these (212) were WebElements, followed by 145 action methods, 74 navigation methods, and 6 getter methods.

The AdminSidebar class in this application stands out with higher number of generated elements, contributing the total values in each category. For instance, the AdminSidebar PO alone accounts for 119 extra defined elements, representing 27.2% of all extra generated elements by itself.

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
AdminSidebar	10	0	0	0	0	0	0	4	11	0	0	7	20	0	0	23
AttributesPage	1	0	0	0	0	0	0	1	1	1	0	0	2	0	2	1
CategoriesPage	1	0	0	0	0	0	0	1	1	0	0	1	32	4	26	0
CurrencyPage	0	0	0	0	1	0	0	1	0	0	0	0	2	1	0	1
FeaturesPage	1	1	0	0	1	0	0	1	0	0	0	0	20	6	9	2
OrdersPage	0	0	0	0	1	0	0	1	0	0	0	0	6	1	2	3
PrestaLogin	3	0	0	0	0	0	0	1	0	0	0	0	3	1	3	1
SupplierPage	1	0	0	0	0	0	0	1	1	0	1	0	15	2	8	1
AddCategoryPage	2	0	0	0	0	0	0	1	2	1	1	0	22	0	13	0
AddAttributePage	3	0	0	0	0	0	2	1	2	1	1	0	8	0	6	0
AddSupplierPage	6	2	0	0	0	0	0	1	1	0	1	0	14	0	15	1
EmployeesPage	1	0	0	0	0	0	0	1	1	0	0	1	25	0	21	0
EditEmployeePage	2	0	1	0	0	0	0	0	3	0	0	0	21	8	13	0
RegisterEmployeePage	6	0	0	0	1	0	0	1	3	4	0	5	8	0	11	0
AddAddressPage	11	0	0	0	0	0	0	1	1	1	1	1	7	0	16	0
AddFeaturesPage	2	0	0	0	0	0	1	1	2	1	1	0	7	3	8	0
StatesPage	2	0	0	1	0	0	0	1	1	1	0	0	4	0	2	2
TagsPage	2	1	0	0	0	0	0	1	0	0	0	0	3	0	3	0
AddCurrencyPage	6	2	0	0	0	0	0	1	4	1	1	0	7	0	10	0
AddNewStatePage	6	1	0	0	0	0	0	1	0	0	1	0	3	0	6	0
AddTagPage	3	1	0	0	0	0	0	1	0	0	1	0	6	0	4	1
Total	**69**	**8**	**1**	**1**	**4**	**0**	**3**	**23**	**34**	**11**	**9**	**15**	**235**	**26**	**178**	**36**

Table 5.19: Detailed classification of Page Object review of PrestaShop using DeepSeek

Table 5.19 demonstrates the detailed results for PrestaShop using DeepSeek. Consistent with the findings from GPT-4o, the largest proportion of correctly defined elements are WebElements (69 out of 79). The majority of elements requiring modification are navigational methods (23 out of 30). Although the missing elements show a more balanced distribution across types, WebElements still make up the largest portion with 34 out of 69, followed by 15 navigational methods, 11 getter methods, and 9 action methods.

LLM	#Correct	#To Modify	#Missing	#Extra	Accuracy (%)	Coverage (%)
OpenAI	81	33	64	437	45.5	64.0
DeepSeek	79	30	69	475	44.4	61.2

Table 5.20: Summary of PO classification results for PrestaShop. Counts (Correct, To Modify, Missing, Extra) and total are shown alongside derived metrics: Accuracy = Correct / Total, Coverage = (Correct + To Modify) / Total.

From the table, it can be seen that the number of extra generated elements are more than the ones that manually generated POs have. GPT-4o generated extra 437 elements and DeepSeek generated extra 475 elements for PrestaShop application. Given that the manually generated page objects only had total of 178 elements, it can be concluded that both LLMs produced substantially more extra elements, over 2.4x and 2.6x the number of existing elements for OpenAI and DeepSeek, respectively.

LLM	#Correct	#To Modify						#Missing
		Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.	Total	
OpenAI	81	10	6	3	12	2	33	64
DeepSeek	79	17	3	3	7	0	30	69

Table 5.21: Detailed classification of Prestashop PO modifications with comparison of ChatGPT-4o and DeepSeek. Subcategories of "To Modify" include various return type and naming issues.

Table 5.20 compares the two LLMs, GPT-4o and DeepSeek, along with the coverage and accuracy metrics and presents the total values for each category. The results reveal that both models perform quite similarly in this application. GPT-4o was able to correctly identify 81 items, while DeepSeek was able to correctly identify 79 items. For elements requiring modification, GPT-4o demonstrated 33 total compared to DeepSeek's 30. In the missing elements category, GPT-4o had 64, and DeepSeek had 69. The numbers of extra generated elements were also close, with 437 for GPT-4o and 475 for DeepSeek. So far DeepSeek tends to generate more extra elements than GPT-4o with a significant gap between the values, but this application marks the first instance where their numbers are this close. GPT-4o achieved a slightly higher accuracy of 45.5%, and also outperformed DeepSeek in coverage, with 64.0% compared to DeepSeek's 61.2%.

Specifically for this application, the results tended to have fewer elements requiring modification but instead a higher number of missing elements, which contributed to a lower overall coverage value.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
OpenAI	30.3%	18.2%	9.1%	36.4%	6.1%
DeepSeek	56.7%	10.0%	10.0%	23.3%	0.0%

Table 5.22: Percentage breakdown of "To Modify" subcategories for PrestaShop PO classification by ChatGPT-4o and DeepSeek. Percentages are relative to total modifications.

The results regarding the detailed breakdown of "to modify" category can be seen in tables 5.21 and 5.22. GPT-4o's most common issue was incorrect element naming, accounting for 12 out of 33 instances (36.4%), followed by completely missing return types at 30.3%. There were also 6 instances of incorrect return types, 3 instances of missing same-page return types and 2 instances of missing implementation. On the other hand, DeepSeek struggled with missing return types the most, which occurred in 17 out of 30 instances (56.7%). It was followed by incorrect naming issues at 23.3%, and 10% each for incorrect return types and missing same-page return types.

This application's POs contained numerous action methods specifically designed to test empty inputs and capture error messages. Since they represent custom behavior of the application, there can be a lot of of these cases. Therefore, it is unreasonable to expect the LLM to cover all such scenarios. If it is required to address this challenge, the prompt should be refined to include methods for empty or invalid input cases. In particulat, there were 10 methods created for this purpose, all of which were missing in the page objects generated by the LLMs. This resulted in affecting the total accuracy values achieved by both LLMs.

Listing 5.11: Some custom implementation methods from PrestaShop

```
public AddTagPage addEmptyTag()
public AddNewStatePage addEmptyState()
public AddCurrencyPage addEmptyCurrency()
public AddAddressPage addEmptyAddress()
public RegisterEmployeePage addEmployeeNoName(String emailStr, String passStr)
public RegisterEmployeePage addEmployeeNoPassword
(String nameStr, String lastnameStr, String emailStr)
public RegisterEmployeePage addEmployeeNoEmail(String nameStr,
String lastnameStr, String passStr)
public RegisterEmployeePage addEmployeeNoPermission(String nameStr,
String lastnameStr, String emailStr, String passStr)
public AddSupplierPage addEmptySupplier()
public AddSupplierPage addEmptySupplier()
```

5.5 Results for ExpressCart

The last application which will be discussed in this section is ExpressCart, which is an e-commerce application. Total of 11 Page Objects were included in the results for this application. The detailed results are discussed below.

Tables 5.23 and 5.24 represents the values achieved for both applications, from the results it can be seen that the Page Objects used in this application are relatively smaller than the previous application's Page Objects. Focusing on Table 5.23, which displays the results for GPT-4o, a total of 25 elements were correctly defined. Of these, 20 were web elements, 2 were action methods, 2 were navigational methods, and 1 was a getter method. In the "to modify" category, the majority were navigational methods (11 out of 20), followed by 6 action methods and 3 getter methods. The number of missing elements in this application was relatively low. There were only 5 elements marked as missing. out of a total of 55. These elements included 2 web elements, 2 navigational methods, and 1 getter method. These elements included 2 web elements, 2 navigational methods, and 1 getter method.

The second table showing the results for DeepSeek demonstrates results closely in line with GPT-4o. However, a few minor differences are noticeable. DeepSeek was able to correctly identify a slightly larger number of items, many of which corresponded to those that

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
CartSidebar	2	0	0	1	0	0	0	1	0	0	0	0	7	2	5	0
CheckoutPage	1	0	0	0	0	0	0	1	0	0	0	0	20	2	16	1
CustomerLoginPage	3	0	2	0	0	0	0	1	0	0	0	0	0	0	0	0
PaymentPage	2	0	0	0	0	2	0	0	0	0	0	0	5	1	4	0
ProductPage	3	0	0	0	0	0	1	2	0	0	0	0	4	1	3	0
ReviewForm	2	0	0	0	0	0	3	1	0	0	0	0	3	0	1	0
ShippingPage	1	0	0	1	0	0	0	0	0	0	0	0	3	2	0	1
TopNavBar	2	0	0	0	0	0	0	1	2	1	0	2	7	0	5	3
SearchResults	1	1	0	0	0	1	0	0	0	0	0	0	4	1	0	2
MainNavbar	1	0	0	0	0	0	0	1	0	0	0	0	4	0	1	3
Home	2	0	0	0	0	0	2	3	0	0	0	0	12	0	6	3
Total	**20**	**1**	**2**	**2**	**0**	**3**	**6**	**11**	**2**	**1**	**0**	**2**	**69**	**9**	**41**	**13**

Table 5.23: Detailed classification of Page Object review of ExpressCart using GPT-4o

required modification in GPT-4o's results. Also, consistent with its typical behavior, DeepSeek produced more extra items than GPT-4o, especially in the category of getter methods.

PO Class	Correct				To Modify				Missing				Extra			
	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav	Elem	Get	Act	Nav
CartSidebar	2	0	0	0	0	0	0	2	0	0	0	0	9	4	6	1
CheckoutPage	1	0	0	1	0	0	0	0	0	0	0	0	20	3	16	1
CustomerLoginPage	3	0	2	0	0	0	0	1	0	0	0	0	0	0	0	0
PaymentPage	2	0	0	0	0	0	2	0	0	0	0	0	5	1	4	0
ProductPage	3	0	0	0	0	0	1	2	0	0	0	0	5	3	2	1
ReviewForm	2	0	0	0	0	0	3	1	0	0	0	0	6	1	2	0
ShippingPage	1	0	0	0	0	0	0	1	0	0	0	0	3	2	0	1
TopNavBar	3	1	0	0	0	0	0	1	1	0	0	2	7	0	6	1
SearchResults	1	2	0	0	0	0	0	0	0	0	0	0	4	1	2	0
MainNavbar	1	0	0	0	0	0	0	1	0	0	0	0	4	1	1	2
Home	2	0	0	0	0	0	2	3	0	0	0	0	14	3	0	2
Total	**21**	**3**	**2**	**1**	**0**	**0**	**8**	**12**	**1**	**0**	**0**	**2**	**77**	**19**	**39**	**9**

Table 5.24: Detailed classification of Page Object review of ExpressCart using DeepSeek

LLM	#Correct	#To Modify	#Missing	#Extra	Accuracy (%)	Coverage (%)
OpenAI	25	20	5	132	50.0	90.0
DeepSeek	27	20	3	144	54.0	94.0

Table 5.25: Summary of PO classification results for ExpressCart. Counts (Correct, To Modify, Missing, Extra) and total are shown alongside derived metrics: Accuracy = Correct / Total, Coverage = (Correct + To Modify) / Total.

Table 5.25 shows the total numbers for each category along with the accuracy and coverage values. GPT-4o reached 50% and 90% coverage for ExpressCart, whereas DeepSeek obtained 54% of accuracy and 94% of coverage. It is important to point out here that although there is not a huge difference in the accuracy values compared to other applications, the coverage value difference is quite significant. This difference is primarily due to the fact that there were few missing elements overall, but many were classified under the "to modify" category.

This could be attributed to the simplicity of the Page Objects defined for this application. The Page Objects were more focused on core functionalities, and they typically included only one or two per page. Additionally, the HTML files used as input in the prompts were relatively small. This is likely to have contributed to producing more concise and focused Page Objects.

LLM	#Correct	#To Modify						#Missing
		Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.	Total	
OpenAI	25	10	0	7	3	0	20	5
DeepSeek	27	7	1	9	3	0	20	3

Table 5.26: Detailed classification of ExpressCart PO with comparison of ChatGPT-4o and DeepSeek.

LLM	Missing RT	Wrong RT	Same Page RT	Wrong Name	Missing Impl.
OpenAI	50.0%	0.0%	35.0%	15.0%	0.0%
DeepSeek	35.0%	5.0%	45.0%	15.0%	0.0%

Table 5.27: Percentage breakdown of “To Modify” subcategories for ExpressCart PO classification by ChatGPT-4o and DeepSeek. Percentages are relative to total modifications.

Table 5.26 provides a detailed classification of the "to modify" category, while Table 5.27 presents the corresponding percentages. Both LLMs produced the same total number of elements in the "to modify" category, 20 elements each.

For GPT-4o, 10 of these elements (50%) were entirely due to missing return types. In addition, 7 elements contained missing return types for the same Page Object and 3 were categorized as naming issues. GPT-4o did not have any cases with missing implementations or incorrect return types. In contrast, DeepSeek’s largest subcategory in the “to modify” group was missing return types for the same Page Object, with 9 instances (45%). This was followed by 7 items with completely missing return types, 3 naming issues and 2 items with incorrect return types.

6 Discussion

The results obtained from our empirical study on the effectiveness of large language models (LLMs) in automating Page Object generation demonstrate that both ChatGPT-4o and DeepSeek perform comparably. They both resulted in creating PO with closely aligned accuracy and coverage values. The following section discusses the findings, their implications and possible limitations of our results.

6.1 Effectiveness of LLMs in Automating Page Object Creation

By employing GPT-4o and DeepSeek in five distinct applications, it can be concluded that LLMs are capable of generating effective page objects. In most cases, the generated page objects successfully captured the majority of UI elements necessary for testing the application's functionality, with coverage values frequently exceeding 70%.

It has been observed that both models face similar challenges. Especially in cases where they were unable to correctly identify return types from navigational method. This issue highlights the lack of understanding the relations between different page objects in case of relying only on given instructions and static HTML input files with a zero-shot prompting setup.

Without access to dynamic application behavior, such as user flow or navigation logic, LLMs struggle to understand which actions transition to new pages or which remain on the same screen. As a result of the incorrect assumptions about method return types, generated page objects are often turned out to be having several inconsistencies compared to the ground truth POs.

This limitation is especially apparent in the naming of return types for navigational methods. In some cases LLMs were not able to identify that the method should navigate to another page, and instead they generated those methods with void return types. In some other cases, they recognized that a navigational action exists and attempt to redirect to another page object class. To be able to accurately define these methods, they must infer the target page object class name solely based on cues from the given HTML file, such as link text or attribute values. If given HTML uses vague or not explanatory link names, the model can be misled which results in incorrect or generic class names that do not match the actual ones. Even in cases where LLM managed to produce contextually close and reasonable naming, there were cases which LLM failed to exactly match the expected class name. This most likely to occur due to subtle naming conventions or nuances which LLM could not infer without the awareness of the broader page object system.

Therefore, in the cases where LLMs were able to generate completely correct navigational methods, the links inside the HTML files were named almost the same as the navigated PO class name, since it is the only source of information which LLM could use to infer the naming of the PO. Thus, it can be concluded that HTML input files, their structure and semantics, plays a crucial role in this matter.

Despite these shared shortcomings, each LLM exhibited distinct coding behaviors and stylistic differences in page object generation. These differences most likely to have occurred due to their underlying training data, architectural biases, and prompt interpretation strategies.

In majority of the cases DeepSeek showed better performance in both accuracy and coverage comparisons. However, usually there was not a huge difference between two LLMs. In all applications, DeepSeek presented a tendency over creating more extensive page objects compared to GPT-4o, by incorporating more elements and methods. Another characteristic observed in DeepSeek was to generate some methods only with the signature but without implementation, which also seemed different than GPT-4o.

It is important to point out that, in this research we specifically employed DeepSeek Coder [36], which is a variant of the DeepSeek model. This model has been fine-tuned and optimized for software development and code generation tasks. Thus, using it likely contributed to its stronger performance in both accuracy and coverage. Consequently, the choice of this particular model may have strengthened DeepSeek's tendency to produce more comprehensive page objects compared to the more general-purpose LLM GPT-4o.

Although DeepSeek showed a characteristic for generating extra elements compared to GPT-4o, these elements are not necessarily meaningful or required for testing purposes. Their value for testing the functionality of the AUT should be evaluated in detail. In our experiments, we closely examined the extra methods generated by reviewing each Page Object (PO) individually. Most of them appear to be valuable and reasonable for the basic functionality of pages. However, in order to truly assess the value of these methods, more comprehensive test suites need to be created. These test suites should be built on more comprehensive Page Objects that go beyond the core functionality to cover a wider range of behaviors and interactions.

The relevance comparison between the prompted HTML files and the completeness of generated page objects revealed that when a prompted HTML file is long, even if not structurally complex, the LLMs are not able to generate the related pages in its full context. For instance, on pages like a sidebar or similar which contain many similar links, the models would produce only a subset of the page object and omit the remaining elements. This behavior of LLM affected the results in a negative way in many cases.

Some possible reasons for this behavior could be:

- **Context-window (token) limits:** when the combined length of the prompt and the generated output approaches the model's maximum context size, the LLM decides to truncate or stop generating further content. Long pages with many repeated elements like in our example may easily exhaust this limit.

- **Repetition and deduplication bias:** transformer models often detect repeated structures and, rather than reproducing each instance, they prefer to generate only a few examples and implicitly assume the rest. This behavior leads to omitted elements in long pages with a repeated structure.
- **Prompt phrasing and stopping criteria:** if the prompt does not strongly emphasize *include every element*, or if a low max-tokens limit is set, the model might think that truncation is acceptable and terminate generation once that threshold is reached.

That also shows the importance of our preprocessing step added to HTML files, which included removing unnecessary headers, scripts and obtaining a simplified HTML file which contains only the relevant sections for page object generation.

When we analyzed our results across the five test applications from different domains, we did not observe any consistent correlation between the type of application and the accuracy or coverage metrics achieved. This may suggest that it was structural complexity, repetition patterns, and prompt framing, and not the application domain itself that had the greatest impact on the LLMs' ability to generate accurate page objects. However, to verify this assumption, a larger set of applications within similar categories should be used to evaluate the performance of LLMs, which was outside of the scope of this study.

6.2 Threats to Validity

Although the experiments yielded promising results, it is important to review potential threats to validity in order to correctly interpret the findings. These threats are discussed in the section below.

Limited Selection of LLMs: The experiments were conducted using only with two LLMs because of the limitation of our local processing power to use open source LLMs with bigger number of parameters. Additionally, smaller open-source models failed to complete the task. As a result, we limited the our research focus to GPT-4o and DeepSeek which we could access via their API. While these models are recognized as the state-of-art LLMs, their performance might not represent all available LLMs. It is known that LLMs performances differ based on the data they have been trained and the architectures they follow. Therefore, it is possible that there might be other LLMs available which can excel the automating page object creation and achieve better results than the ones we chose. This poses a potential threat to the external validity of our findings.

Correctness of Manually Defined Page Objects: As the evaluation phase of the study involved manual comparison of page objects, we relied on a dataset of manually implemented page objects. These were considered as ground truth to assess the accuracy of the outputs produced by the LLM. However, this poses a potential threat to validity, as the reference implementations may contain omissions, inaccuracies or incorrectly included methods. Such problems in the reference implementations could affect the reliability of our assessment and thus the validity of our findings.

Limited Prompting Strategies: The performance of the LLMs in automating page object creation has shown to be relied on the prompt provided. The keywords chosen and the structure of the input prompt affected the result created. During prompt development, we observed how variations in prompt design affected model output. Although we investigated some configurations that might affect LLM's performance and tried to improve our prompt to achieve best results, there are many other possibilities have yet to be explored. These untested alternatives could potentially provide improvements and result in producing better accuracy. Thus, the limited number of prompt configurations tested in shaping our prompt is another threat to the validity of our work.

Potential Data Leakage in LLMs: A possible threat to the validity of this work is data leakage due to the pre-training of large language models. For the reason that LLMs are trained on a lot of publicly available data, it's probable that some of the UI code structures, websites, or page object patterns we utilized in our review were included in the training data for the models. If this is the case, it is possible that LLMs are reproducing patterns they have seen before, instead of generating solutions based on their reasoning mechanism. This might result in overestimation of the model's real capability to automate page object generation in untested applications. Since we cannot verify the exact training data for models we employed, it becomes another potential threat to the validity of our research.

7 Conclusion and Future Work

7.1 Conclusion

This thesis investigated the ability of Large Language Models to generate Page Object Models, which are known to be an important but labor-intensive component in UI testing. By comparing LLM-generated Page Objects with manually generated ones across a range of applications, we evaluated the accuracy and functional integrity of the POs produced.

The results obtained from our empirical evaluation demonstrate that LLMs, particularly GPT-4o and DeepSeek Coder, have promising abilities in automating Page Object generation. Both models were able to generate syntactically correct Java classes which include proper imports and follow appropriate use of Selenium constructs. These Page Object classes included WebElements and methods which represent the interactions of related page. Notably, DeepSeek consistently achieved higher accuracy and coverage in three out of the five tested applications whereas GPT-4o performed better in more complex applications such as MediaWiki and PrestaShop.

However, the evaluation also highlighted certain limitations. A significant proportion of action and navigational methods required manual modification. This modifications were mainly needed because of incorrect or missing return types issues, which are known to be difficult to comprehend merely from static HTML files in zero shot prompting settings. Additionally, LLMs often generated extra elements not present in the ground truth. These additional methods could enhance test coverage in some cases, which also might suggest that LLMs can better capture useful patterns compared to manual development.

Despite these challenges, the results conclude that, especially guided by a well-tailored prompt, LLMs can significantly reduce the manual burden required for creating Page Objects. Our study also emphasizes how much prompt design can have impact on achieving high quality outputs. Furthermore, our findings prove the potential of LLMs in test automation tasks, specifically page object generation. By leveraging LLMs in software development, we can improve productiveness with relatively small amount of human intervention.

In summary, our study validates that LLMs can be useful in software testing tasks, especially for accelerating and scaling test development in dynamic web apps.

7.2 Future Work

Our research has shown the potential to leverage large language models (LLMs) to automate the creation of page objects and reduce the amount of work that has to be done manually otherwise. However, some aspects could not be explored due to the limited scope of this thesis, and left open for future work.

Upon concluding our research, we concluded that one main challenge LLMs encounter in generating page objects is correctly defining navigational methods. Specifically, when returning the correct page object after a navigational action is taken. This limitation stems from the lack of information about the relationships between different pages and application's structure, LLMs have. A possible direction for future work which we believe that would improve the performance of LLM models is to enhance the prompting strategy by explicitly providing inter-page relationships. This could help LLMs better comprehend navigation flows and improve the overall accuracy of generated page objects.

Due to the limited processing power we have, we could not test powerful open source LLMs with higher number of parameters. In the next steps of this research, the performance of such LLMs could be tested. Furthermore, after doing some experiments on two LLMs we employed, we agreed on a uniform set of hyper parameters across all experiments. However, these settings can also be explored with using different values in different models to optimize the performance of the models. In our study, we opted for a lower temperature to prioritize determinism over creativity but, it is possible that alternative configurations in different models produce better results. Therefore, investigating those configurations further would be helpful.

Another area for future work would be to employ another evaluation methodology by involving real developers in the study, and collecting their feedback. This could include having developers to work with LLM generated page objects. Working with these developers can help us to measure the time and effort required to adjust and complete those page objects to use in actual test cases. The results of this research would provide more practical insights into the usability and effectiveness of the generated artifacts, as well as how much they would reduce the manual effort needed, and help assess their real impact on development efficiency.

Furthermore, it can be verified whether the extra content generated by LLMs truly enhances test coverage and effectiveness by engaging these developers to extend test suites using LLM-generated page objects. Concretely, existing test cases could be expanded with extra methods generated by the model and then measure the overall impacts on them in terms of code coverage. Such an experimental addition would provide quantitative evidence of practical benefits of integrating LLM-generated artifacts into real testing scenarios.

Overall, our study has shown the potential of LLMs in another code generation task, namely page object generation. However, there are still points that can are open for further development and evaluation of the research topic. We can address the challenges encountered in this research and ultimately unlock new opportunities in automating page object generation using LLMs by exploring them in the next steps.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Selenium Project. *Selenium*. https://www.selenium.dev/. Accessed: 2025-05-21. 2025.

[2] Cypress.io. *Cypress*. https://www.cypress.io/. Accessed: 2025-05-21. 2025.

[3] Microsoft. *Playwright*. https://playwright.dev/. Accessed: 2025-05-21. 2020.

[4] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. "APOGEN: Automatic Page Object Generator for Web Testing". In: *Software Quality Journal* 25.3 (2017), pp. 1007–1039. DOI: 10.1007/s11219-016-9331-9. URL: https://doi.org/10.1007/s11219-016-9331-9.

[5] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. "Clustering-Aided Page Object Generation for Web Testing". In: *Proceedings of the 16th International Conference on Web Engineering (ICWE 2016)*. Vol. 9671. Lecture Notes in Computer Science. Springer, 2016, pp. 132–151. DOI: 10.1007/978-3-319-38791-8_10. URL: https://doi.org/10.1007/978-3-319-38791-8_10.

[6] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Ed. by H. Bouamor, J. Pino, and K. Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1069–1088. DOI: 10.18653/v1/2023.emnlp-main.68. URL: https://aclanthology.org/2023.emnlp-main.68/.

[7] Appium Contributors. *Appium: Open Source Test Automation Framework for Use with Native, Hybrid and Mobile Web Apps*. https://appium.io/. Accessed: 2025-05-25. 2024.

[8] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. "Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2013, pp. 108–113. DOI: 10.1109/ICSTW.2013.19.

[9] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, and P. Tonella. "A Family of Experiments to Assess the Impact of Page Object Pattern in Web Test Suite Development". In: *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020, pp. 263–273. DOI: 10.1109/ICST46399.2020.00035.

[10] M. Neelapu. "Enhancing Test Automation with an Advanced Page Object Model for Scalable and Maintainable Web Applications". In: *International Journal of Leading Research Publication (IJLRP)* 5.11 (2024), pp. 1–13.

[11] M. Leotta, B. Garcia, F. Ricca, and J. Whitehead. "Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey". In: *Proceedings of the 16th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 339–350. DOI: 10.1109/ICST57152.2023.00039.

[12] OpenAI. "GPT-4 Technical Report". In: *arXiv preprint arXiv:2303.08774* (2023). URL: https://arxiv.org/abs/2303.08774.

[13] G. DeepMind. "Gemini: A Family of Highly Capable Multimodal Models". In: *arXiv preprint arXiv:2312.11805* (2023). URL: https://arxiv.org/abs/2312.11805.

[14] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971* (2023). URL: https://arxiv.org/abs/2302.13971.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2019, pp. 4171–4186. URL: https://arxiv.org/abs/1810.04805.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008. URL: https://arxiv.org/abs/1706.03762.

[17] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. "Improving Language Understanding by Generative Pre-Training". In: *OpenAI Blog* (2018). https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[19] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67.

[21] Y. Wang, W. Jiang, H. Peng, X. Hu, S. He, L. Zhou, and P. S. Yu. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685.

[22] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2020, pp. 7871–7880. DOI: `10.18653/v1/2020.acl-main.703`.

[23] OpenAI. *ChatGPT: Optimizing Language Models for Dialogue*. `https://openai.com/blog/chatgpt`. Accessed: 2025-05-25. 2022.

[24] J. Li, G. Li, Y. Li, and Z. Jin. "Structured Chain-of-Thought Prompting for Code Generation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34.2 (2025), pp. 1–23. DOI: `10.1145/3690635`.

[25] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. "Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2016, pp. 1120–1125. DOI: `10.1145/2851613.2851817`.

[26] M. Leotta, M. Biagiola, F. Ricca, M. Ceccato, and P. Tonella. "A Family of Experiments to Assess the Impact of Page Object Pattern in Web Test Suite Development". In: *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST 2020)*. IEEE, 2020, pp. 263–273. DOI: `10.1109/ICST46399.2020.00035`.

[27] B. Team. *Bludit — Flat-File CMS*. `https://www.bludit.com`. Version 3.13.1. 2024.

[28] M. Vautin. *ExpressCart: A Node.js Shopping Cart*. `https://github.com/mrvautin/expressCart`. Version 1.19. 2024.

[29] F. Guillot. *Kanboard: Project Management Software*. `https://github.com/kanboard/kanboard`. Version 1.2.15. 2024.

[30] Wikipedia contributors. *Wikipedia, The Free Encyclopedia*. `https://www.wikipedia.org/`. Accessed: 2025-05-27. 2025.

[31] W. Foundation. *MediaWiki*. `https://www.mediawiki.org/wiki/MediaWiki`. Version 1.40.0. 2024.

[32] P. Contributors. *PrestaShop: Open Source e-Commerce Solution*. `https://www.prestashop-project.org/`. Version 1.7.8.5. 2024.

[33] F. Huq, J. P. Bigham, and N. Martelaro. *"What's important here?": Opportunities and Challenges of Using LLMs in Retrieving Information from Web Interfaces*. `https://arxiv.org/abs/2312.06147`. Accepted at NeurIPS 2023 Workshop on Robustness of Zero/Few-Shot Learning in Foundation Models (R0-FoMo). 2023.

[34] OpenAI. *GPT-4o System Card*. `https://openai.com/index/gpt-4o-system-card/`. Accessed: 2025-05-25. 2024.

[35] DeepSeek-AI. "DeepSeek LLM: Scaling Open-Source Language Models with Longtermism". In: *arXiv preprint arXiv:2401.02954* (2024). URL: `https://arxiv.org/abs/2401.02954`.

[36] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: `2401.14196 [cs.CL]`. URL: `https://arxiv.org/abs/2401.14196`.