Towards Automated Page Object Generation for Web Testing using Large Language Models

Betül Karagöz[1], Filippo Ricca[2], Matteo Biagiola[3], Andrea Stocco[1,4]
[1]*Technical University of Munich*, Munich, Germany, betul.karagoez, andrea.stocco@tum.de
[2]*Università degli Studi di Genova*, Genova, Italy, filippo.ricca@unige.it
[3]*University of St. Gallen and Università della Svizzera italiana*, St. Gallen / Lugano, Switzerland, matteo.biagiola@{unisg,usi}.ch
[4]*fortiss GmbH*, Munich, Germany, stocco@fortiss.org

***Abstract*—Page Objects (POs) are a widely adopted design pattern for improving the maintainability and scalability of automated end-to-end web tests. However, creating and maintaining POs is still largely a manual, labor-intensive activity, while automated solutions have seen limited practical adoption. In this context, the potential of Large Language Models (LLMs) for these tasks has remained largely unexplored. This paper presents an empirical study on the feasibility of using LLMs, specifically GPT-4o and DeepSeek Coder, to automatically generate POs for web testing. We evaluate the generated artifacts on an existing benchmark of five web applications for which manually written POs are available. Our results show that LLMs can generate syntactically correct and functionally useful POs with accuracy values ranging from 32.6% to 54.0% and element recognition rate exceeding 70% in most cases. Our study contributes the first systematic evaluation of LLMs strengths and open challenges for automated PO generation, and provides directions for further research on integrating LLMs into practical testing workflows.**

***Index Terms*—web testing, page objects, large language models**

I. Introduction

Web automation frameworks such as Selenium [1], Cypress [2], and Playwright [3] have been developed to facilitate automated interactions between test scripts and web applications [4], [5]. However, maintaining these tests remains a significant challenge: even minor modifications to the page layout can cause tests to fail [4], [6], [7].

The Page Object (PO) design pattern is a common solution to improve test maintainability. By encapsulating web elements and their interactions in dedicated classes, the PO model decouples the test logic from the underlying GUI structure. This abstraction simplifies test management, reduces duplication, and allows for more robust scripts: if the GUI changes, only the corresponding PO needs to be updated. Moreover, POs can be reused across test suites, further improving maintainability [8]. Despite these advantages, building and maintaining POs is labor-intensive, which hinders their widespread adoption.

To reduce this manual effort, prior research explored automated PO generation [9], [10], [11], [12], [13]. While effective, these approaches do not use recent advances in Large Language Models (LLMs) in code generation and information processing, which are increasingly being integrated into software engineering workflows to automate coding tasks. Unlike existing structural or clustering-based PO generators [9], [10], [11], [12], which rely on static DOM parsing or handcrafted rules [14], this work explores the use of LLMs as generalizable PO synthesizers that can interpret markup, infer semantics, and produce executable Selenium code.

This paper investigates specifically the effectiveness of GPT-4o and DeepSeek Coder to automate PO generation, selected for their complementary design goals and public accessibility. GPT-4o represents a general-purpose, instruction-tuned multimodal model optimized for reasoning and code synthesis across diverse programming contexts, while DeepSeek Coder is a specialized model explicitly trained on large-scale code corpora with strong emphasis on software engineering tasks. Together, they provide a balanced evaluation of both generalist and domain-specialized LLMs.

We study how LLMs interpret HTML structures, infer relationships between pages, and translate them into functional POs with syntactically correct and semantically meaningful methods. Using a benchmark of five web applications with manually written POs as ground truth, we evaluate the generated artifacts in terms of accuracy and element recognition rate.

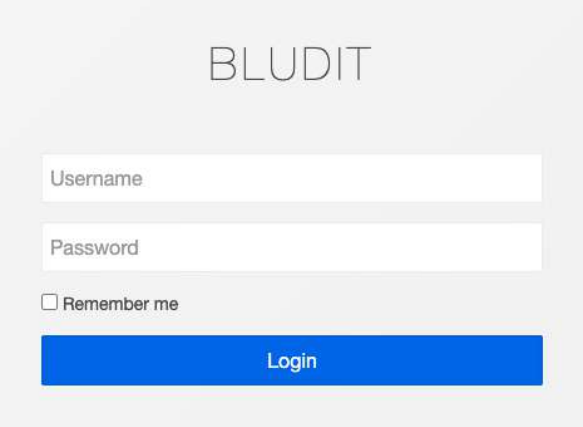
Our results show that both models can generate valid and reusable POs, with accuracy ranging from 32.6% to 54.0% and element recognition rate exceeding 70% in most cases. While DeepSeek achieved slightly higher average accuracy and element recognition rate, differences with GPT-4o were not statistically significant. Both models consistently produced syntactically valid Java classes, but struggled with navigation methods and return types, where missing or incorrect definitions were the most frequent source of errors. Conversely, LLMs often generated extra elements and methods beyond the ground truth, some of which could potentially enrich test suites.
This paper makes the following contributions:

- The first systematic empirical evidence on the effectiveness of LLMs for PO generation. Our empirical study evaluates the effectiveness of GPT-4o and DeepSeek Coder in generating POs on a benchmark of five web applications.
- A replication package including the experimental library and dataset to support open research [15].

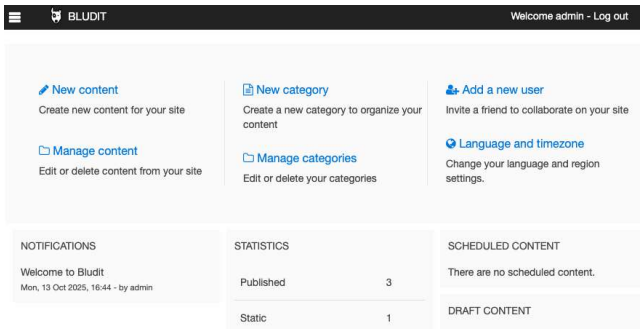
Andrea Stocco was partially supported by the Bavarian Ministry of Economic Affairs, Regional Development, and Energy. Matteo Biagiola is partially supported by Fondo Istituzionale per la Ricerca granted by Università della Svizzera italiana (USI).

II. Background

Figure 1 shows our running example, i.e., an exceprt of the `Bludit` web application (v. 2.3.4). We consider a scenario in which a user inserts `username` and `password` in the login form (Figure 1 **(a)**); if these credentials are correct, the `username` (in our example "admin") is displayed on the top right corner of the homepage (Figure 1 **(b)**).



(a) *Login page.*



(b) *Home page.*

Fig. 1: Bludit web application.

A. Programmable Web Application Testing Tools

With programmable web application testing tools, testers develop executable test cases using a high-level programming language such as Java, Python, or Ruby. The tool APIs provide ways to interface with the browser; thus, test cases are composed of commands that either control the browser or provide input data, set the values of GUI components, and assert whether the program behaves correctly (e.g., by means of JUnit or TestNG assertions).

Listing 1 displays a programmable version of a test case for the login functionality, created using Selenium WebDriver. In Line 3, an instance of WebDriver is created to control the Firefox browser. In Line 4, the executable test case instruments WebDriver to navigate to the URL of the target web application. Lines 5–10 fill in the username and password input boxes with the administrator credentials, and Lines 11–13 submit the form. Lines 14–16 verifies, by means of an assertion, the presence of an HTML element containing the username in the home page, and asserts that it equals the string "admin". Finally, Line 17 shuts down the WebDriver instance and closes the browser.

```
1  @Test
2  public void testLogin(){
3      WebDriver driver = new FirefoxDriver();
4      driver.get("localhost/Bludit/index.php");
5      driver.findElement(By
6          .xpath("//*[@id=`LoginForm']/input[1]"))
7          .sendKeys("admin");
8      driver.findElement(By
9          .xpath("//*[@id=`LoginForm']/input[2]"))
10         .sendKeys("secret");
11     driver.findElement(By
12         .xpath("//input[@value=`Login']"))
13         .click();
14     AssertThat(driver
15         .findElement(By.id("loggedUser"))
16         .getText(), is("admin"));
17     driver.quit();
18 }
```

Listing 1: An example of a programmable automated test case.

The implementation of advanced programmable test cases may require development skills comparable to those required for the development of production code. However, all of the benefits of modular programming can be brought to the creation of test suites, such as parametric and conditional execution, reuse of common functionalities across test cases (e.g., applying design patterns), and robust mechanisms for referencing the HTML elements in a web page [6].

B. Page Objects

In web test automation, the "Page Object" design pattern [16] has emerged as the leading pattern for enhancing test case maintenance, reducing code duplication and lowering the coupling between test cases and web applications [10]. Page objects apply naturally to the programmable approach to web testing, because they are object-oriented classes that serve as interfaces to the web pages of the application under test. Test cases use the methods of the page object class whenever they need to interact with an HTML element of the GUI, which allows them to do anything that an end user can see and do on a page. A direct benefit of this is that test cases do not need to be modified if the application underneath undergoes structural changes [5], i.e., changes involving only the web page layout/structure. On the other hand, test cases still require maintenance in the presence of logical changes [5], i.e., those involving a web application's functionality. There is empirical evidence of the benefits associated with the adoption of the Page Object pattern in the maintenance of test suites for web applications, in both industry [17] and academia [18], [8].

Listing 2 and Listing 3 present simplified page objects for `Bludit`. The classes are written in Java and implemented within the Selenium WebDriver framework. Each HTML element becomes a `WebElement` class instance[1], properly named and with a `@FindBy` annotation containing the locator.

[1]https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebElement.html

```
1  public class IndexPage{
2    private WebDriver driver;
3    @FindBy(name="user")
4    private WebElement username;
5    @FindBy(name="pass")
6    private WebElement password;
7    @FindBy(id="submitLogin")
8    private WebElement login;
9
10   public IndexPage(WebDriver driver){
11       this.driver = driver;
12       PageFactory.initElements(driver, this);
13   }
14
15   public HomePage login(String usr, String pwd){
16       username.sendKeys(usr);
17       password.sendKeys(pwd);
18       login.click();
19       return new HomePage(driver);
20   }
```

Listing 2: Java page object for the Login page in Figure 1 **(a)**.

```
1  public class HomePage{
2    private WebDriver driver;
3    @FindBy(id="loggedUser")
4    private WebElement loggedUser;
5
6    public HomePage(WebDriver driver){
7        this.driver = driver;
8        PageFactory.initElements(driver,  this);
9    }
10
11   public String getLoggedUser(){
12       return loggedUser.getText();
13   }
```

Listing 3: Java page object for the Home page of Figure 1 **(b)**.

The page objects expose web page functionalities as methods; for simplicity, we limit the example to the methods of interest to our sample login test case. `IndexPage` contains a method to perform the login, which wraps the calls to the various HTML elements in a single shareable unit (we assume the scenario in which credentials are correct, and that the result of the login operation is a page modeled with the `HomePage` page object). The constructors of the page objects of Listing 2 and Listing 3 contain calls to a support library named *Page Factory* (Line 12 and Line 8, respectively). The Page Factory instantiates the HTML elements of the page object and pre-populates them based on the annotations (e.g., Lines 3, 5 and 7 of Listing 2). In the `HomePage` page object, the `getLoggedUser()` method (Lines 11–13 of Listing 3) retrieves the textual content associated with the HTML element having id equal to "loggedUser".

Listing 4 shows a version of the login test case originally presented in Listing 1, modified to use the page objects of Listing 2 and Listing 3. The code has become more readable because the test case steps now correspond more directly to the steps of the test case specification (e.g., open the index page, log in to the application, and assert that the user is logged in).

```
1  @Test
2  public void testLogin(){
3      IndexPage ip = new IndexPage(driver);
4      HomePage hp = ip.login("admin", "secret");
5      assertEquals(hp.getLoggedUser(), "admin");
6  }
```

Listing 4: Page object-based programmable automated test case (Selenium WebDriver)

C. Benefits of Using Page Objects

The main advantage of using the Page Object (PO) pattern is that changes to the user interface (UI) only require updates within the page object itself, not in the test scripts [19]. All UI locators and page-specific actions are centralized in a dedicated page class, making maintenance easier when the UI evolves [8].

Another benefit is reusability. The same page objects can be used across multiple test cases. For example, a login operation (valid or invalid) can serve as a reusable action across various tests. This reduces code duplication and promotes a more modular, maintainable codebase.

The PO pattern also provides encapsulation and abstraction. Test scripts interact only with high-level methods, without needing to manage low-level UI details like locators [8]. This makes it easier to expand the test suite as new pages or features are added to the application under test (AUT), requiring minimal changes to existing tests. Additionally, POs enhance readability and clarity by exposing user actions through clearly named methods [20]. This makes the tests easier to understand and maintain. Finally, debugging is simplified, as failures are isolated within the relevant page objects

D. Challenges in Using Page Objects

While POs offer clear benefits, such as decoupling, reduced code duplication, and easier test maintenance, Leotta et al. [19] emphasize that creating POs for a web application is a nontrivial task. It requires significant development effort, particularly during the early stages of testing when POs must be built from scratch. Although the advantages of adopting the PO pattern are well-documented, it remains uncertain whether these benefits always justify the upfront investment, especially for smaller-scale automation projects where POs may introduce unnecessary complexity. Conversely, in large and dynamic applications, maintaining a growing set of page objects can become a burden, as highlighted by Stocco et al. [11].

III. Study Design

This section describes the definition, the design, and the settings of the experiment we conducted in a structured way, following the guidelines by Wohlin et al. [21].

A. Goal and Research Questions

The goal of this study is to evaluate the effectiveness of Large Language Models in generating Page Objects (POs) for web testing, with a focus on their correctness and completeness, and to compare these results against manually written POs,

which serve as the ground truth. Our evaluation includes the following three research questions:

RQ_1 (Accuracy): *How effective are Large Language Models in generating Page Objects with high accuracy?*

RQ_2 (Element Recognition Rate): *To what extent are Large Language Models able to recognize ground-truth elements when generating Page Objects?*

RQ_3 (Issue categories): *In which specific categories of issues do Large Language Models struggle the most when generating Page Objects?*

The first research question investigates the overall accuracy of LLMs in generating POs, i.e., their ability to produce correct elements that match the ground truth. The second research question focuses on the element recognition rate, measuring to what extent LLMs are able to identify and reproduce existing elements in the target application. The third research question analyzes the types of issues that arise in LLM-generated POs, with the goal of understanding where these models most frequently fail.

B. Objects of the Study

In our study, we evaluate LLM-generated POs by comparing them with manually created POs. For this reason, the Applications Under Test (AUTs) must already have manually written POs available. To ensure consistency and reliability, we used the `BEWT` benchmark of web applications [22], which provides developer-written POs, Docker containers to run the applications, and associated test cases. From this benchmark, we selected five applications from different categories, allowing us to assess LLMs performance across diverse scenarios, detailed in Table I and described next.

`Bludit` (v. 3.13.1) 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 [23]. Its simplicity and fast performance makes it a useful test application for evaluating LLMs in content-oriented applications. `ExpressCart` (v. 1.19) is an open-source e-commerce application built with Node.js. It includes main features like listing products, shopping cart functionality, and checkout processes [24]. `Kanboard` (v. 1.2.15) is a project management tool which uses the Kanban methodology to help users visualize tasks, manage workflows, and track project progress [25]. `MediaWiki` (v. 1.40.0) is a collaborative wiki platform best known for powering Wikipedia [26]. It supports structured content, revision history, and user permissions [27]. `PrestaShop` (v. 1.7.8.5) 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 [28].

C. Benchmark Setup

The `BEWT` benchmark includes POs, Docker containers for running web applications, and test scripts. However, it does not provide the HTML content, which is the primary input required for prompting LLMs. To gather the necessary HTML for the screens, we ran the applications locally using the provided Docker containers, navigated to the relevant pages, and saved the corresponding HTML files. These raw HTML files, however, often contain excessive or redundant data, such as scripts and embedded resources, that could easily overload the context for LLMs [29], [30]. To address this issue, we implemented a pre-processing step to truncate and clean the HTML content before feeding it into the model. This approach was informed by the findings of Huq et al. [29], which showed that feeding raw HTML files to LLMs often leads to hallucinations and failure to follow instructions. Specifically, the study found that applying an effective truncation strategy could improve model performance by up to 11%.

TABLE I: Statistics of the Benchmark Applications.

Application	Source LoC	Test LoC	#Test Cases/POs	Version
`Bludit`	∼12,000	∼500	∼50	3.13.1
`ExpressCart`	∼8,500	∼300	∼40	1.19
`Kanboard`	∼25,000	∼3,000	∼120	1.2.15
`MediaWiki`	∼150,000	∼10,000	∼300	1.40.0
`PrestaShop`	∼220,000	∼15,000	∼400	1.7.8.5

D. Selection of Large Language Models

We selected GPT-4o [31] and DeepSeek Coder [32] (DeepSeek, hereafter) for our experiment on automated PO generation due to their state-of-the-art performance and strong specialization in code-related tasks. Initially, our goal was to rely on smaller, open-source models for accessibility and lower resource demands, but preliminary tests showed they lacked sufficient accuracy. Larger open-source models performed better but exceeded our computational capacity. Therefore, we focused on high-performing models available via external APIs, which allowed us to evaluate advanced capabilities without the burden of running large models locally.

E. Large Language Model Setup

1) Hyperparameters: LLMs contain various hyperparameters that change the way they behave, which ends up affecting the output quality. Among these, there is the *temperature parameter*, which controls the creativity of the output. Higher values (closer to 1) generate more diverse responses, while lower values (closer to 0) produce more consistent and deterministic results. Since the task requires consistency rather than creativity, we set the temperature to 0.1 in all experiments, mitigating the impact of randomness in the results [33].

Another important hyperparameter is *max tokens*, which defines the maximum number of tokens that can be generated in a single interaction by the LLM. Since our task requires a complex output, we chose to use the default value (i.e., 16,384 for GPT-4o and 2,048 for DeepSeek Coder).

2) Prompt Strategies: Previous studies show that prompts are crucial in guiding LLM performance [34]. Clear, well-structured prompts define the task and lead to good results, while vague prompts cause confusion and off-target outputs.

In our approach, we assume that no existing test suites or pre-defined POs are available for the selected AUT. This leads us to adopt a zero-shot prompting approach, which does not

rely on prior examples or predefined structures. Instead, it allows the model to generate POs and test cases based solely on the provided requirements or task description. This approach enables us to evaluate an LLM's ability to create correct POs without any prior knowledge of the AUT. It also serves as a means to assess the model's capabilities in real-world scenarios, where no setup or input from existing resources is available. Being the first study of this kind, we adopted a zero-shot setup to establish a reproducible lower bound of model capability, isolating intrinsic LLM reasoning from external aids such as few-shot examples or retrieval.

Listing 1 shows the prompt structure we used to specify the rules for generating POs together with the corresponding HTML content for the page. The LLM is expected to generate a PO in Java, adhering to a specific structure outlined in the prompt. This includes using appropriate naming conventions, setting up the constructor, and creating `WebElements` for each input field, button, and link on the page. The generated PO must also import necessary libraries from Selenium and JUnit, which we use in this work. Additionally, it should include methods for navigation (e.g., clicking buttons or links) and interactions (e.g., entering text).

During our experiments, we noticed that some requirements were not implemented, even though they were clearly stated in the prompt. Upon further analysis, we found that using more assertive language—specifically imperative statements like "must", increased the likelihood of those requirements being correctly implemented. Consequently, we refined the prompt by using "must" statements to clearly highlight critical instructions, improving model compliance.

```
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}
        """
```

Listing 1: Page Object Generation Prompt.

F. Evaluation Setup

In the evaluation phase of our study, we manually compared the ground-truth POs provided in the benchmark with the outputs generated by both GPT4-o and DeepSeek across five different applications. For each generated POs, we manually compared the `WebElement` objects (e.g., username and password `WebElement`) and corresponding methods (e.g., login method) against the ground truth. In order to ensure consistency and clarity in our analysis, we categorized each identified element and method into four main groups, namely correct, to modify, missing, and extra, following prior work [12].

TABLE II: Detailed classification of Page Object review using GPT-4o and DeepSeek, for all apps.

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**

Abbreviations: Elem = Web Elements; Get = Getter Methods; Act = Action Methods; Nav = Navigation Methods.

Correct. Web elements are classified as *correct* if they use the appropriate element type and if their assigned name is meaningful and matches, or closely aligns with, the ground truth. For methods, we require 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.

To modify. If an element is present but does not fully follow the naming conventions or structural guidelines given in the prompt, it is categorized as an element which requires modification by a developer. These elements are partially correct, and can still be used in the POs with manual adjustments. An important part of this classification was identifying 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 based on the requested fix.

Missing. Elements are classified as *missing* if they are completely absent from the generated output. This applies to both web elements and methods that exist in the ground truth but are not generated by the LLM;

Extra. Elements are categorized as *extra* if they appear in the LLM-generated POs but have no corresponding match in the ground-truth POs. Since manually created POs may not always be complete or fully representative of the page's functionality, extra elements generated by LLMs reflect the model's ability to generate more comprehensive POs, potentially capturing additional functionality that may have been missed in the manually created ones.

The task was performed by the first author and validated through discussions with the other authors to resolve disagreements in ambiguous cases. No method was left unassigned.

G. Metrics and Analysis for Evaluation

To evaluate the performance of LLMs in automating PO creation, we used the following metrics. For RQ_1, we computed the accuracy metric is used to measure the percentage of methods that are classified as "Correct". In particular, accuracy is defined as follows:

$$\text{Accuracy} = \frac{\text{No. of Correct Methods}}{\text{Total Number of Methods}} \times 100$$

The corresponding metric for web elements is defined analogously. For RQ_2, 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:

$$\text{Elem Rec. Rate} = \frac{\text{Total Correct + Requires Mod. Elems}}{\text{Total Elements}} \times 100$$

The corresponding metric for methods is defined analogously. For RQ_3, we broke down the results of the "To Modify" category into specific subcategories. These subcategories represent specific types of issues where the LLM output does not match the ground-truth POs. For each subcategory me measure:

$$\text{Subcategory } \% = \frac{\text{No. of Elements in Subcategory}}{\text{Total Number of Elements to Modify}} \times 100$$

The task was performed by the first author and validated through discussions with the other authors with prior experience in web testing to resolve disagreements in ambiguous cases, following the qualitative analysis protocol of Wohlin et al. [21]. No case was left unassigned. The corresponding metrics for web elements/methods are defined analogously to those used for methods/web elements. This triangulation is aimed at mitigating subjectivity in class assignments.

IV. Results

A. RQ_1 (Accuracy)

Table II provides an overview of how accurately each LLM generated POs compared to the ground truth. We focus here on elements classified as *Correct*, which reflect full alignment in type, naming, and behavioural semantics.

Across all applications, the majority of correctly generated items were WebElements (429 in total), indicating that LLMs

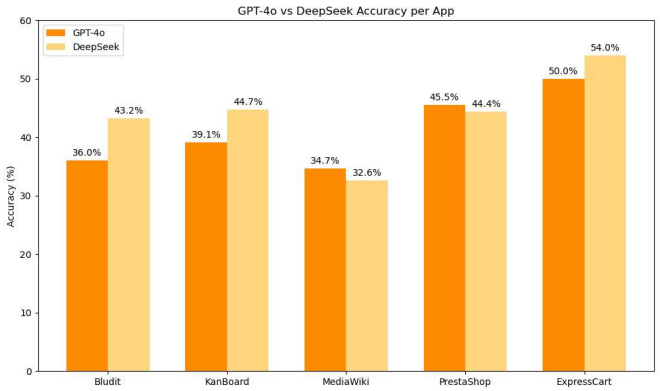


Fig. 2: Accuracy of GPT-4o and DeepSeek.

are fairly reliable in detecting static page components. In contrast, methods, particularly Action and Navigation methods, were identified less accurately, confirming that behavioural elements remain challenging due to their dependency on application flow [35]. Between the two models, DeepSeek showed marginally better performance than GPT-4o, especially in correctly generating Getter methods. However, neither model exceeded an overall accuracy threshold sufficient for full automation without developer intervention.

Figure 2 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. On our data, the results suggest that DeepSeek performs slightly better in terms of accuracy for PO creation. The lowest results were achieved in the MediaWiki application with accuracy values of 34.7% and 32.6% for GPT-4o and DeepSeek, respectively. On the other hand, the highest accuracy values were obtained in the ExpressCart application with 50% in GPT-4o and 54% in DeepSeek.

RQ_1 (Accuracy): LLMs can generate syntactically valid and partially functional POs, achieving accuracy between 32.7% and 54.0%. DeepSeek slightly outperforms GPT-4o in method-level correctness.

B. RQ_2 (Element Recognition Rate)

For RQ_2, we analyse the extent to which LLMs correctly detect relevant elements, regardless of whether they require refinement. Here, we consider both *Correct* and *To Modify* instances as successful recognitions.

The element recognition rate (Figure 3) ranging from 61.2% to 94%, indicates that, even when LLMs do not produce fully correct implementations, they are generally capable of identifying the presence and role of page components. DeepSeek again demonstrates slightly higher recognition capabilities, particularly in complex applications such as Kanboard and MediaWiki, where structural depth poses additional challenges.

TABLE III: Breakdown of "To Modify" Subcategories.

LLM	Miss. Ret.	Wrong Ret.	SamePg Ret.	Wrong Nm.	Miss. Im.
GPT-4o	54	14	74	33	2
DeepSeek	54	12	67	28	8

Abbreviations: Miss. Ret. = Missing Return Type; Wrong Ret. = Wrong Return Type; SamePg Ret. = Same Page Return Type; Wrong Nm. = Wrong Name; Miss. Im. = Missing Implementation.

However, both models frequently introduce *Extra* elements, suggesting a tendency to overgenerate or infer functionality beyond the ground truth.

RQ_2 (Element Recognition Rate): LLMs reliably detect most relevant page elements, with element recognition rates between 61.2% and 94%. DeepSeek shows a modest advantage in recognising method-level components.

C. RQ_3 (Issue categories)

Table III shows the total number of instances across all applications for issues that occurred in the "To modify" category. Our analysis revealed the following five categories:

- **Missing return type**: The method lacks an explicitly defined return type;
- **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 PO to support chaining in test cases [36];
- **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.

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 3 shows the overall element recognition rate comparison across all applications for both LLMs. Similar to the accuracy results, DeepSeek outperformed GPT-4o by achieving higher element recognition rate values in three out of five

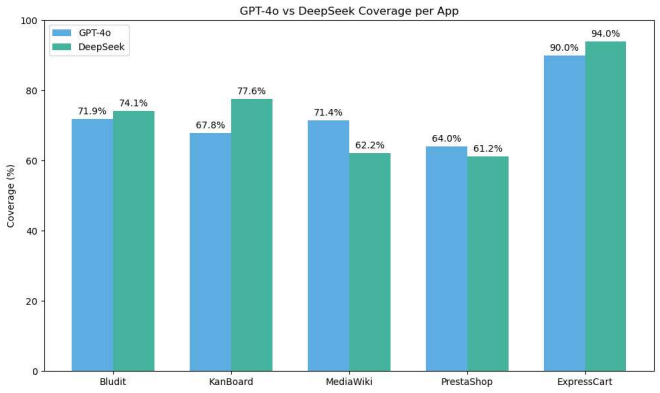


Fig. 3: Element Recognition Rate of GPT-4o and DeepSeek.

applications (Bludit, KanBoard and ExpressCart). In contrast, GPT-4o performed slightly better than DeepSeek on MediaWiki and PrestaShop. In particular, the element recognition rate achieved for ExpressCart was significantly higher than for the other applications. In this particular application, DeepSeek achieved 94% element recognition Rate while GPT-4o reached 90%. They performed both significantly better than the average element recognition Rate which was reported to be 73.4% across all applications. GPT-4o and DeepSeek showed the lowest performance in PrestaShop, with element recognition rate values of 64% and 61.2% respectively. On the other hand, GPT-4o and DeepSeek performed best in ExpressCart application, reaching 90% and 94% element recognition rate.

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

RQ_3 (Issue categories): The main difficulties arise in identifying method return types, with most errors due to missing return types (often defaulting to void) or methods that should return the same PO to enable chaining; the two LLMs behave similarly in this respect.

D. 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.

1) Internal Validity: Since the evaluation involved manual comparison of POs, we relied on a dataset of manually implemented ones that had been previously developed by other researchers and made publicly available. We consider these implementations reliable and used them the ground truth to assess the LLM's accuracy. However, this still introduces a potential validity threat, as the reference implementations may contain omissions or inaccuracies that could affect the reliability of our assessment and, consequently, the validity of our findings. Moreover, the manual validation process introduces a subjective component. The manual classification of LLM-generated outputs introduces subjectivity, as evaluators may interpret naming conventions differently. To mitigate this, multiple authors reviewed and discussed ambiguous cases until consensus was reached. All evaluators have substantial prior experience with PO design and implementation, which reduced the risk of systematic bias. Since the readability of POs is not an objective of this study, disagreements related to naming conventions do not affect our findings and are therefore not considered problematic.

Since the ground-truth POs represent one of multiple possible valid designs, structurally different yet functionally correct generations may be classified as deviations. We mitigated this by reviewing extra elements manually; several were found to implement valid, complementary interactions. Hence, our reported accuracy figures should be interpreted as a conservative lower bound.

Another potential threat to validity is data leakage from the pre-training of LLMs. Since LLMs are trained on extensive public data, some code structures or PO patterns used in our study may already exist in their training corpus. In such cases, models might reproduce patterns they have seen before rather than generate novel solutions, potentially leading to an overestimation of their actual ability to automate PO generation in unseen applications. As the exact training data are unknown, this remains a potential threat to our study's validity.

2) External Validity: The experiments were conducted using only two LLMs due to local processing limitations constraints and the inability of smaller open-source models to complete the task. Consequently, our study focused on GPT-4o and DeepSeek, accessed via their APIs. While both are state-of-the-art models, their performance may not represent all existing LLMs, whose results can vary based on training data and architecture. Therefore, other models might outperform them.

The performance of LLMs in automating PO creation strongly depends on the provided prompt. We observed that both keyword choice and input structure significantly affected the output. Although we explored several configurations to optimize results, many other prompting strategies remain untested. These unexplored alternatives could yield better performance, making the limited set of prompts evaluated a potential threat to the validity of our work.

3) Conclusion Validity: Conclusion validity may be affected by the small number of LLMs and applications tested. The observed differences between GPT-4o and DeepSeek might be influenced by model-specific biases or the characteristics of the selected benchmarks rather than general trends. As a result, while the findings provide valuable insights into LLM performance on PO generation, they should be interpreted as indicative rather than conclusive. When we began the study, we expected LLMs to struggle particularly with determining the correct return type. Unlike simple element classification, this requires understanding an element's role within the application flow, especially for navigational components that trigger page transitions. With only a single HTML file as input, the model must infer the correct target PO without access to the broader

context, such as routing logic or related POs. A central question in our evaluation is whether LLMs can generalize these relationships from isolated HTML snapshots. In large applications, it is not feasible to include the full codebase in a single prompt due to context limitations, making it essential to assess how well LLMs perform under partial access to the web app. Understanding these limitations is key to judging their scalability and practical use, but is left for future work.

V. Discussion

The results obtained from our empirical study on the effectiveness of LLMs in automating page object (PO) generation indicate that ChatGPT-4o and DeepSeek perform comparably well. Both models achieved similar levels of accuracy and element recognition rates in the generated POs. Our study establishes an initial benchmark for automated PO generation with LLMs. While accuracy remains moderate, the consistency of syntactic correctness and high element recognition demonstrate tangible progress toward replacing template-based approaches with semantic generation. The following section discusses these findings, along with their implications.

Comparison of General-Purpose and Code-Oriented LLMs. By employing GPT-4o and DeepSeek in five distinct applications, it can be concluded that LLMs are capable of generating effective POs. In most cases, the generated POs successfully captured the majority of GUI elements necessary for testing the application's functionality, with Element Recognition Rate values frequently exceeding 70%.

In majority of the cases DeepSeek showed better performance in both accuracy and element recognition comparisons. However, usually there was not a large difference between two LLMs. Across all applications, DeepSeek demonstrated a tendency to produce more extensive POs than GPT-4o, incorporating a greater number of 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 employed DeepSeek Coder [37], 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 Element Recognition Rate. Consequently, the choice of this particular model may have strengthened DeepSeek's tendency to produce more comprehensive POs compared to the more general-purpose LLM GPT-4o.

Although DeepSeek tended to generate a larger number of extra elements compared to GPT-4o, it is not yet clear whether these additions are truly beneficial for testing. In this study, we manually examined the extra methods by reviewing each PO individually. Many of them appeared to be reasonable and aligned with the basic functionality of the pages. However, to properly assess their practical value, more comprehensive test suites would be required, built upon POs that extend beyond core functionality to capture a broader range of behaviors and interactions. The contribution of these extra elements to functional coverage and fault detection remains an open question and should be investigated in future work.

Open Challenges in Structural Reasoning and Navigation Inference. Both models exhibited similar challenges, particularly in correctly inferring the return types of navigation methods. This suggests that relying solely on static HTML and task instructions within a zero-shot prompting setup limits the models' ability to infer inter-object relationships. Without access to dynamic behaviors, such as user flows or page transitions, LLMs struggle to distinguish actions that lead to new pages from those that remain within the same view. As a result, incorrect assumptions about method return types frequently caused inconsistencies between the generated POs and the ground truth.

This limitation is especially evident 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 PO class. To be able to accurately define these methods, they must infer the target PO class name solely based on cues from the given HTML file, such as link text or attribute values. If the provided HTML contains vague or non-descriptive link names, the model can be misled, resulting in incorrect or overly generic class names that do not correspond to the actual ones. Even in cases where LLMs managed to produce contextually close and reasonable naming, there were cases in which the LLM failed to exactly match the expected class name. This most likely occurs due to subtle naming conventions or nuances that LLMs could not infer without awareness of the broader web app context.

On the other hand, in cases where the LLMs successfully generated entirely correct navigational methods, the links within the HTML files were typically named almost identically to the target PO class names. This suggests that such naming consistency served as the primary cue the models relied upon to infer the correct PO naming. Consequently, the structure and semantics of the HTML input files play a crucial role in determining model performance. Despite these shared shortcomings, each LLM exhibited distinct coding behaviors and stylistic variations in PO generation. These differences are likely attributable to disparities in their underlying training data, architectural biases, and prompt interpretation strategies. Overall, our findings highlight the limitations of zero-shot prompting; more sophisticated prompting techniques, such as chain-of-thought reasoning or few-shot examples, may significantly influence outcomes and warrant further investigation in future work.

Impact of Input Size and Context Limitations. The comparison between prompted HTML files and the completeness of the generated POs shows that lengthy HTML pages, even if not structurally complex, often challenge LLMs in capturing the full contextual relationships needed for complete page representations. For instance, in pages containing sidebars or menus with many similar links, the models frequently generated

only a subset of elements, omitting others and thus lowering overall PO accuracy.

This behavior can be attributed to several factors. First, when the combined size of the prompt and output nears the model's context window limit, generation may be truncated before all elements are produced—particularly for long pages with repetitive structures. Second, transformer-based models naturally avoid excessive repetition, often generating a few representative instances of recurring patterns rather than reproducing every occurrence explicitly.

Additionally, the phrasing of the prompt and the imposed generation constraints likely influenced these outcomes. If the instructions did not explicitly require exhaustive element coverage or the token budget was insufficient, the model may have prioritized conveying the overall structure over full completeness. This behavior highlights the importance of our preprocessing step, where HTML files were cleaned by removing headers and scripts, enabling the model to focus on relevant content and reducing premature truncation.

Across the five applications from different domains, we found no consistent correlation between application type and the accuracy or element recognition rate achieved by the LLMs. This indicates that structural complexity, repetition, and prompt design had greater impact on performance than domain characteristics. A larger and more domain-balanced dataset would be needed to confirm this observation. Future work should also explore iterative or conversational refinement, allowing the LLM to incrementally adjust generated POs based on developer feedback.

VI. Related work

Several works in the literature, including recent ones based on LLMs, have explored various aspects of web testing automation [38], [39], [35], [40], [41], [42]. However, when it comes to POs, most studies primarily focus on assessing their effectiveness or maintainability [8], [43], while only a very limited number of approaches have explicitly addressed the problem of automatically generating them [10], [44].

The empirical study conducted by Leotta et al. [18] indicates that test suites developed using a programmable approach (i.e., Selenium WebDriver) require higher initial development effort but results in lower maintenance effort compared to those created with a capture-and-replay approach (i.e., Selenium IDE). This outcome can be largely attributed to the adoption of the PO design pattern, which effectively decouples test logic from application logic.

Subsequently, Leotta et al. [43] further investigated the impact of applying the PO pattern on the maintainability of automated test suites. Their empirical results show that encapsulating GUI interaction logic within PO classes reduces overall maintenance effort by decoupling test scripts from structural changes in the application. This study reinforces the practical value of the PO pattern and underlines the importance of adopting structured design principles in test automation.

More recently, the same authors conducted controlled experiments [8] to compare test suites developed with and without the PO pattern. Their analysis considered both the initial design effort and the long-term benefits associated with POs. The results show that, although introducing POs entails a significant upfront cost, a clear break-even point emerges as the test suite scales: when the suite grows to approximately ten times its original size, the use of POs becomes substantially more advantageous in terms of maintainability and robustness. Overall, these studies highlight that while POs require greater initial effort, they provide considerable benefits in scalability and maintainability for large and evolving test suites.

To the best of our knowledge, only two studies have addressed the automatic generation of POs. The first, Apogen [10], automates PO creation by combining dynamic crawling, clustering, and static analysis. It reverse-engineers the application under test, groups similar pages, builds a state-based model, and converts it into Java Page Objects following the Selenium Page Factory pattern. Unlike approaches that generate only class skeletons, Apogen produces complete POs with WebElement instances and methods for navigation and form submission, but does not employ LLMs. The second work [44] introduces a prototype for automatically generating POs and PO-based test suites through feedback-directed random testing. It decouples test code from web pages by generating POs and was evaluated on seven real-world applications, achieving code coverage between 23.3% and 90.8% while revealing numerous HTML and runtime errors. Despite their promise, existing approaches [10], [44] remain research prototypes that often require manual correction of generated abstractions. In contrast, this work is the first to explore the use of LLMs for PO generation, aiming to improve automation and quality while reducing the need for human intervention.

VII. Conclusions and Future Work

This paper investigates the potential of LLMs to automate Page Object (PO) creation, an essential yet time-consuming task in web testing. By comparing LLM-generated POs with manually developed ones across multiple applications, we assessed their accuracy and element recognition rate against a curated ground truth. Our findings show that models such as GPT-4o and DeepSeek Coder can substantially reduce the manual effort required for PO development, though challenges persist in handling return types, navigation methods, and extraneous element generation. Notably, some of these additional methods may enhance GUI coverage, indicating that LLMs can capture meaningful patterns overlooked in manual design and potentially support test suite augmentation.

Future work will explore enhanced prompting strategies that explicitly encode page relationships, evaluate additional open-source models, extend this baseline study with few-shot and chain-of-thought prompting to assess contextual improvements, and using visual information [13], [7] and multi-modal LLMs. In addition, we plan to involve practitioners in user studies to provide more realistic insights into code quality [45], [46], [47], the usability of LLM-generated artifacts [48], the required effort for adjustments [8], and their actual impact on coverage and productivity.

REFERENCES

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

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

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

[4] F. Ricca, M. Leotta, and A. Stocco, "Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE," *Advances in Computers*, vol. 113, jan 2019.

[5] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Approaches and tools for automated end-to-end web testing," *Advances in Computers*, vol. 101, 2016.

[6] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "ROBULA+: An algorithm for generating robust XPath locators for web testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, 2016.

[7] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, ACM, nov 2018.

[8] 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.

[9] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015.

[10] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "APOGEN: Automatic page object generator for web testing," *Software Quality Journal*, vol. 25, no. 3, 2016.

[11] 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 of *Lecture Notes in Computer Science*, Springer, 2016.

[12] A. Stocco, M. Leotta, F. Ricca, and P. Tonella, "Why creating web page objects manually if it can be done automatically?," in *Proceedings of the 10th IEEE/ACM International Workshop on Automation of Software Test (AST 2015)*, IEEE, 2015.

[13] M. Bajammal, A. Stocco, D. Mazinanian, and A. Mesbah, "A Survey on the Use of Computer Vision to Improve Software Engineering Tasks," *IEEE Transactions on Software Engineering*, vol. 48, oct 2020.

[14] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the 42nd International Conference on Software Engineering*, ICSE '20, ACM, jun 2020.

[15] "Replication Package." https://github.com/ast-fortiss-tum/page-objects-generation-with-llms, 2026.

[16] M. Fowler, "Pageobject." http://martinfowler.com/bliki/PageObject.html, 2013.

[17] 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 2013)*, IEEE, 2013.

[18] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, 2013.

[19] 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.

[20] M. Neelapu, "Enhancing test automation with an advanced page object model for scalable and maintainable web applications," *International Journal of Leading Research Publication (IJLRP)*, vol. 5, no. 11, 2024.

[21] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[22] D. Olianas, M. Leotta, and F. Ricca, "BEWT: A Benchmark for End-to-End Web Testing," in *Proceedings of the 51st Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA 2025)*, Lecture Notes in Computer Science, Springer, Sept. 2025.

[23] B. Team, "Bludit — flat-file cms." https://www.bludit.com, 2024. Version 3.13.1.

[24] M. Vautin, "Expresscart: A node.js shopping cart." https://github.com/mrvautin/expressCart, 2024. Version 1.19.

[25] F. Guillot, "Kanboard: Project management software." https://github.com/kanboard/kanboard, 2024. Version 1.2.15.

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

[27] W. Foundation, "Mediawiki." https://www.mediawiki.org/wiki/MediaWiki, 2024. Version 1.40.0.

[28] P. Contributors, "Prestashop: Open source e-commerce solution." https://www.prestashop-project.org/, 2024. Version 1.7.8.5.

[29] 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, 2023. Accepted at NeurIPS 2023 Workshop on Robustness of Zero/Few-Shot Learning in Foundation Models (R0-FoMo).

[30] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, 2024.

[31] OpenAI, "Gpt-4o system card." https://openai.com/index/gpt-4o-system-card/, 2024. Accessed: 2025-05-25.

[32] DeepSeek-AI, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024.

[33] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 2, 2025.

[34] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[35] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Dependency-aware web test generation," in *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation*, ICST '20, IEEE, oct 2020.

[36] M. Biagiola, F. Ricca, and P. Tonella, "Search based path and input data generation for web application testing," in *Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings* (T. Menzies and J. Petke, eds.), vol. 10452 of *Lecture Notes in Computer Science*, Springer, 2017.

[37] 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.

[38] T. Li, C. Cui, R. Huang, D. Towey, and L. Ma, "Large language models for automated web-form-test generation: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, May 2025. Just Accepted.

[39] Y. Sasazawa and Y. Sogawa, "Web page classification using llms for crawling support," 2025.

[40] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, ACM, aug 2019.

[41] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, ACM, aug 2019.

[42] K. Kanaththage, L. L. L. Starace, M. Biagiola, P. Tonella, and A. Stocco, "Neural embeddings for web testing," in *Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation*, ICST '26, p. 12 pages, IEEE, 2026.

[43] 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.

[44] B. Yu, L. Ma, and C. Zhang, "Incremental web application testing using page object," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015.

[45] F. Ricca, A. Marchetto, and A. Stocco, "Ai-based test automation: A grey literature analysis," in *Proceedings of the 14th IEEE International*

Conference on Software Testing, Verification and Validation Workshops, ICSTW 2021, Springer, 2021. [Best Presentation Award].

[46] F. Ricca and A. Stocco, "Web test automation: Insights from the grey literature," in *Proceedings of the 47th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM 2021, Springer, 2021. [Best Paper Nominee].

[47] F. Ricca, A. Marchetto, and A. Stocco, "A retrospective analysis of grey literature for ai-supported test automation," in *Proceedings of the 16th International Conference on the Quality of Information and Communications Technology*, QUATIC 2023, Springer, 2023.

[48] F. Ricca, A. Marchetto, and A. Stocco, "A multi-year grey literature review on ai-assisted test automation," *Information and Software Technology*, 2025.