IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 17 November 2024, accepted 17 December 2024, date of publication 23 December 2024,
date of current version 7 January 2025.

Digital Object Identifier 10.1109/ACCESS.2024.3521407

RESEARCH ARTICLE

Cypress Copilot: Development of an AI Assistant for Boosting Productivity and Transforming Web Application Testing

SURESH BABU NETTUR[1,*], SHANTHI KARPURAPU[1,*], UNNATI NETTUR[2], AND LIKHIT SAGAR GAJJA[3]

[1]Independent Researcher, Virginia Beach, VA 23456, USA
[2]Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
[3]Department of Computer Science, BML Munjal University, Gurugram, Haryana 122413, India

Corresponding authors: Shanthi Karpurapu (shanthi.karpurapu@gmail.com) and Suresh Babu Nettur (nettursuresh@gmail.com)

*Suresh Babu Nettur and Shanthi Karpurapu are co-first authors.

ABSTRACT In today's fast-paced software development environment, Agile methodologies demand rapid delivery and continuous improvement, making automated testing essential for maintaining quality and accelerating feedback loops. Our study addresses the challenges of developing and maintaining automation code for web-based application testing. In this paper, we propose a novel approach that leverages large language models (LLMs) and a novel prompt technique, few-shot chain, to automate code generation for web application testing. We chose the Behavior-Driven Development (BDD) methodology owing to its advantages and selected the Cypress tool for automating web application testing, as it is one of the most popular and rapidly growing frameworks in this domain. We comprehensively evaluated various OpenAI models, including GPT-4-Turbo, GPT-4o, and GitHub Copilot, using zero-shot and few-shot chain prompt techniques. Furthermore, we extensively validated with a vast set of test cases to identify the optimal approach. Our results indicate that the Cypress automation code generated by GPT-4o using a few-shot chained prompt approach excels in generating complete code for each test case, with fewer empty methods and improved syntactical accuracy and maintainability. Based on these findings, we developed a novel open-source Visual Studio Code (IDE) extension, "Cypress Copilot" utilizing GPT-4o and a few-shot chain prompt technique, which has shown promising results. Finally, we validate the Cypress Copilot tool by generating automation code for end-to-end web tests, demonstrating its effectiveness in testing various web applications and its ability to streamline development processes. More importantly, we are releasing this tool to the open-source community, as it has the potential to be a promising partner in enhancing productivity in web application automation testing.

INDEX TERMS Agile software development, behavior driven development, large language model, machine learning, prompt engineering, software testing, cypress, selenium, web application, AI assistant tools, GitHub Copilot, code generation, test case generation, test automation, zero-shot, few-shot, OpenAI, GPT-3, GPT3.5, GPT-4, GPT-4o.

I. INTRODUCTION

Agile methodologies drive rapid delivery and continuous improvement in the fast-paced world of modern software development. In order to maintain quality while dealing with faster development cycles, it is crucial to have automated testing in place to ensure that new code integrates seamlessly and remains free of defects [1]. The global test automation market is witnessing a surge in demand, with an expected compound annual growth rate (CAGR) of 18.6% from 2021 to 2031. This report underscores the need for advanced testing solutions to meet this rapidly

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

growing demand[1]. Despite the advantages of adopting test automation, the process of developing and maintaining automation code for web-based applications can be a significant drain on time and resources. Our research proposes an optimal approach that not only identifies the best LLM and prompt techniques for generating automation code but also promises to speed up the automation coding process. This acceleration could lead to a substantial improvement in software quality and a significant reduction in delivery times, while also allowing development teams to focus on innovation and reducing the burden of repetitive tasks. As part of our study, we reviewed various popular test automation frameworks and tools to determine the best options for applying our research approach. We considered the BDD methodology and the Cypress automation tool in our approach to generating the automation code.

We chose the BDD methodology in our study due to its advantages in fostering collaboration between technical and non-technical stakeholders, ensuring that software meets both functional and business requirements [2], [3], [4]. BDD centers on creating executable specifications written in the human-readable Gherkin language, which is easily understood by both developers and domain experts[2,3]. This methodology addresses unclear, missing requirements and bridges the gap between technical implementations and business objectives [5].

We selected the Cypress automation tool in our study because of its extensive adoption and significant advantages. Cypress is widely used, with 5.3 million weekly downloads[4,5]. It is renowned for its ease of use and powerful features for end-to-end web application testing [6]. It runs directly in the browser, offering faster and more reliable tests with minimal setup. Additionally, Cypress is rated as one of the top three automation frameworks by LambdaTest and BrowserStack[6,7], and it has over a million repositories using it[8]. We aim to develop a novel tool, "Cypress Copilot," which generates automation code specifically for Cypress based on the optimal LLM and prompt techniques derived from our research approach. We aim to benefit many teams using this popular framework by enhancing their testing efficiency and productivity with Cypress Copilot. In our research, we have comprehensively evaluated various Open AI model variants such as GPT-4-Turbo, GPT-4o, and GitHub Copilot with zero-shot and few-shot chain prompt techniques and proposed an optimal solution. Results demonstrate that the Cypress automation code generated by our approach has better code completeness, maintainability, and is prone to fewer syntax errors.

[1]https://www.transparencymarketresearch.com/test-automation-market.html
[2]https://cucumber.io/docs/
[3]https://docs.specflow.org/projects/specflow/en/latest/Gherkin/Gherkin-Reference.html
[4]https://www.npmjs.com/package/cypress
[5]https://www.cypress.io
[6]https://www.browserstack.com/guide/best-test-automation-frameworks
[7]https://www.lambdatest.com/blog/best-test-automation-frameworks/
[8]https://github.com/cypress-io/cypress/network/dependents

A. BDD METHODOLOGY

In this section, we discuss BDD methodology at a high level and the Cypress automation code implementation for BDD scenarios to benefit those new to test automation or with diverse professional backgrounds. BDD methodology mainly involves discovery, formulation, automation, and validation phases (Figure 1) in iterative cycles, ensuring continuous collaboration between stakeholders and maintaining alignment between business requirements and system behavior.

FIGURE 1. BDD methodology stages.

1) DISCOVERY

This phase involves understanding and discovering the user story requirements by discussing the application's expected behavior. Stakeholders and team members collaborate to define scenarios, examples, and acceptance criteria.

2) FORMULATION

In this phase, the behaviors discovered during the Discovery phase are formulated into concrete examples and specifications, also known as Gherkin/BDD/cucumber scenarios. These examples are often written in a structured format, such as Given-When-Then, to describe the behavior in an easy-to-understand and implement.

Figure 3 presents a basic BDD scenario using the Gherkin language to demonstrate login functionality. In this example, each line corresponds to a step in the scenario. The "Given" step sets the initial state by confirming the user is on the login page. The "When" step represents the user's action of entering valid credentials and clicking the login button. Finally, the "Then" step specifies the expected result, which is the user being redirected to the dashboard page. This structure improves readability and allows non-technical stakeholders to easily follow and understand the behavior being tested.

3) AUTOMATION

The automation phase involves writing code and implementing automated tests based on the BDD scenarios created in the Formulation phase. These tests are designed to verify that the application behaves as expected according to the specified scenarios.

4) VALIDATION

In this phase, the power of BDD scenarios is unleashed as they are executed to confirm that the application aligns with the specified behaviors. The scenarios are carefully reviewed to verify that all requirements are met and to identify any potential issues or discrepancies.

In the subsequent section, we discuss the steps involved in the test automation phase, as our proposed approach aims to automate the code generation.

5) AUTOMATION STAGE

The automation stage involves translating BDD scenarios into executable test code. The main steps include creating a feature file with the BDD scenarios formulated for a user story and implementing a step definition file, which contains the step methods for the corresponding BDD scenario steps (steps are lines in BDD scenarios that start with keywords such as Given, When, and Then, as well as other Gherkin language keywords). Within each step method, implement code to perform actions on web elements (such as buttons, lists, radio buttons, etc.). Once the automation framework and code are set up and executed, the script launches the web application under test, and the automated tests are executed and validated.

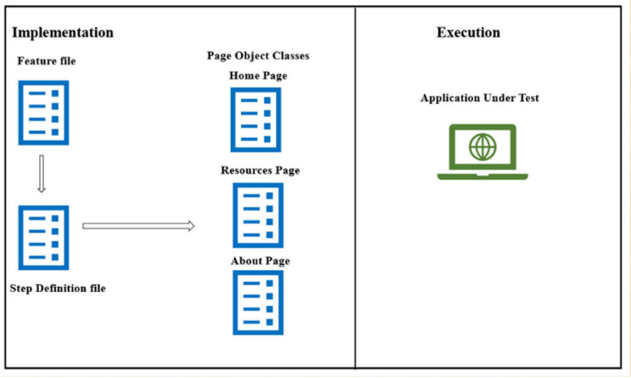

FIGURE 2. BDD automation procedure.

```
Feature: Login Functionality
  As a user
  I want to be able to log in to the application
  So that I can access my account

  Scenario: Successful Login
    Given I am on the login page
    When I enter valid credentials
    And I click the login button
    Then I should be redirected to the dashboard page
```

FIGURE 3. Feature file with login functionality scenario.

One of the important best practices in implementing web application automation tests is to adopt the Page Object Model (POM) design pattern, as shown in Figure 2. POM encapsulates web elements and their interactions into reusable components by representing each page or component of the web application with a corresponding page object class, which declares the web elements and methods for interacting with them[9]. This design enhances maintainability

[9]https://github.com/JoanEsquivel/cypress-cucumber-boilerplate/

```
import { Given, When, Then }
from "@badeball/cypress-cucumber-preprocessor";

// Step definition for navigating to the login page
Given('I am on the login page', () => {
  cy.visit('/login');
});
// Step definition for entering valid credentials
When('I enter valid credentials', () => {
  cy.get("#user-name").type('myusername');
  cy.get("#password").type('mypassword');
});
// Step definition for clicking the login button
When('I click the login button', () => {
  cy.get("#login-button").click();
});
// Step definition for verifying the dashboard page
Then('I should be redirected to the dashboard page', () =>
{
  cy.url().should('include', '/dashboard');
});
```

FIGURE 4. Step definition file that includes implementation of each BDD scenario step.

```
class LoginPage {
  elements = {
  //Locating the username, password, login web elements
    usernameInput: () => cy.get("#user-name"),
    passwordInput: () => cy.get("#password"),
    loginBtn: () => cy.get("#login-button"),
  };

  // Method to type username into the username input field
  typeUsername(username) {
    this.elements.usernameInput().type(username);
  }
  // Method to type password into the password input field
  typePassword(password) {
    this.elements.passwordInput().type(password);
  }
  // Method to click on the login button
  clickLogin() {
    this.elements.loginBtn().click();
  }
}
// Exporting an instance of LoginPage class
export const loginPage = new LoginPage();
```

FIGURE 5. POM class implementation.

by centralizing locators and interactions, making it easier to update web elements in one location. Additionally, POM promotes code reusability and readability and reduces redundancy in test scripts, making it essential for efficient and scalable automation frameworks. Figures 3, 4 and 5 demonstrate a simple Cypress automated test for login functionality, with and without POM.

II. RELATED WORKS

Recent advancements in software testing have increasingly incorporated ML and NLP techniques, particularly

leveraging LLMs. Liu et al. developed GPTDroid, a tool that automates mobile app GUI testing using GPT-3. GPTDroid treats the testing process as a question-and-answer task, generating and refining test scripts based on the app's responses [7]. Similarly, Yoon et al. introduced DroidAgent, an autonomous agent that uses LLMs to interact with mobile apps and accomplish task goals during GUI testing [8]. Dwarakanath et al. proposed a novel method employing computer vision to simulate human behavior in software testing. Their approach interprets textual test scripts and interacts with the application under test (AUT) based on visual input, emulating human testers [9]. Extending this line of work, Hu et al. created AUITestAgent, a natural language-driven GUI testing tool that automates interactions and verifies functionality based on test requirements written in plain language. This tool extracts GUI interactions and employs a multidimensional data extraction strategy to validate app responses [10]. Yu et al. introduced RoboTest to advance GUI testing further. This non-intrusive GUI testing framework uses a visual-based robotic arm and novel screen and widget detection algorithms to simulate human testing across various screen sizes [11]. YazdaniBanafsheDaragh et al. developed a deep learning-based GUI testing approach that generates valid inputs for application testing using captured screenshots, eliminating the need to know the application's internal workings [12]. Lastly, Feng et al. presented CAT, a Retrieval Augmented Generation (RAG) system, to facilitate cost-effective UI automation testing for WeChat. CAT system successfully automated many UI tests and identified numerous bugs [13].

In unit test generation, Pan et al. explored using LLMs and static analysis to generate high-coverage, compliable unit tests for programming languages like Java and Python, focusing on enhancing readability and developer-friendliness [14]. Wang et al. proposed a method that improves unit test coverage and line/branch analysis by decomposing complex methods into smaller slices, allowing LLMs to concentrate on each slice [15]. Lops et al. developed AgoneTest, a system that generates and evaluates unit test suites for Java projects using LLMs, comparing human-written tests with LLM-generated ones to boost efficiency and scalability [16]. Gu et al. introduced TestART, which enhances LLM-generated tests through a co-evolutionary process combining automated test generation and iterative repair [17]. Yang et al. evaluated open-source LLMs and commercial models like GPT-4 for unit test generation in Java projects. They highlighted the impact of prompt design and LLM limitations in test generation [18]. Fakhoury et al. developed TiCoder, an interactive workflow that enhances code generation accuracy by clarifying user intent through tests, reducing cognitive load, and improving AI-generated code evaluation [19]. Alshahwan et al. introduced TestGen-LLM, a tool that refines human-written unit tests using LLMs to ensure measurable improvements and resolve issues such as hallucinations [20].

For API testing, Sri et al. explored automated REST API test generation using LLMs, leveraging OpenA's capabilities to automate the creation of complex test scenarios based on Postman test cases [21]. Pereira et al. introduced APITestGenie, a tool that generates API test scripts from business requirements and API specifications using LLMs. This tool aims to improve productivity by automating test script generation, although human validation is recommended before integration into CI/CD pipelines [22].

As part of our previous research on automating test case creation, we examined LLMs for generating BDD scenarios using zero- and few-shot prompts. We demonstrated that models like GPT-3.5 and GPT-4 produced accurate, error-free BDD tests, with the few-shot prompt technique yielding the best results [23]. In this manuscript, we further extend our research on automating the generation of test automation code using popular tools such as Cypress. Despite significant advancements, we identified an untapped opportunity to leverage LLM-driven automation for generating code for end-to-end test cases in web applications. This presents a promising area for further exploration, with the potential to develop novel approaches that could enhance testing productivity and significantly improve overall quality.

III. METHODOLOGY

Our proposed methodology aims to comprehensively evaluate various OpenAI model variants and prompt design techniques to assess their effectiveness in generating Cypress automation code for the given BDD scenarios. Figure 6 illustrates the evaluation procedure for the generated Cypress automation code across approximately 260 test cases (BDD scenarios covering 56 user stories). The process begins by reading feature files containing BDD scenarios from a CSV file. Each feature file corresponds to the BDD scenarios for a specific user story. The feature file is paired with a prompt, using either zero-shot or few-shot chain techniques, along with model parameters such as temperature, top_p, and max_tokens sent to the OpenAI API endpoint. The generated Cypress automation code is stored in a CSV file by iterating through all feature files. The above mentioned is implemented in Python using Google Colab framework for each OpenAI model variant, specifically GPT-4-Turbo and GPT-4o, and executed the program with zero-shot prompts. Additionally, the few-shot chain prompt technique was evaluated exclusively on GPT-4o, as it is OpenAI's best-performing model and is expected to perform better than GPT-4-Turbo[10]. The code for all models is available in the GitHub repository[11]. After the Cypress code generation step, the generated code is evaluated for various criteria, including syntax issues and the effectiveness of code generation. We comprehensively evaluated various OpenAI model variants and prompt design techniques within

[10]https://github.com/openai/simple-evals?tab=readme-ov-file#benchmark-results

[11]https://github.com/karpurapus/Cypress-Automation-LLM/

this proposed methodology, which integrates automated code generation with rigorous validation.

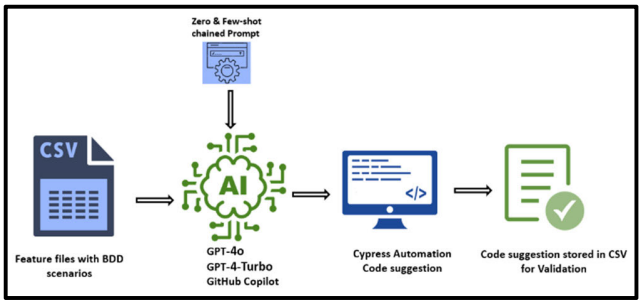

FIGURE 6. Cypress code generation from BDD scenarios and evaluation procedure.

The following sections provide an overview of the key components of our proposed methodology.

A. DATASET

We have not come across a comprehensive, publicly available dataset of user stories from real-time projects. While some datasets include user stories from various research experiments or public repositories, they generally do not represent real-time projects. In order to closely mimic real-world scenarios, for this study, we have collected a diverse set of approximately 55 user stories related to BDD scenarios from our previous research on the automation of BDD scenarios[12] [23]. Our dataset includes user story data from Mendeley[13] and a blog post[14] that highlights the most commonly occurring user stories across different domains. This dataset is hosted in a public GitHub repository[15], where we invite the software development research community to contribute and collaborate in enhancing and expanding this dataset. This compilation ensures a comprehensive representation of use stories across various domains, providing a robust foundation for analysis in our study. The Table 1 shows the list of user stories considered.

We have carefully curated a dataset that spans a broad spectrum of user stories from various application contexts, ensuring coverage across multiple domains. These applications cater to distinct user needs, ranging from seamless real-time interactions and customizations to essential service-driven solutions. This diversity provides a robust foundation for validating our test automation code generation approach, as it reflects a wide array of functional requirements and user interaction patterns. Our dataset also incorporates key features such as data filtering, form handling, user customization, and administrative functionalities, which are critical for creating an adaptable automation solution. By leveraging this dataset, we aim to validate the effectiveness of our approach in mimicking real-world automation demands and handling complex functionalities typical of real-time projects.

TABLE 1. Brief description of applications and various user stories considered for automation code generation.

Application	List of User Stories
Recycling and Waste Management Portal	Users can select recyclable waste types, view facility hours, locations of public bins and special drop-off sites, schedule pick-ups, upload schedules for recommendations, and contact admins. Admins manage facility information, onboard centers, and view usage stats. Facility representatives and superusers update facility info, respond to user questions, and view stats.
Data Analytics and Reporting Platform	Allows users to filter, sort, and aggregate data by various dimensions for detailed views.
Agile Learning and Certification Portal	Trainers list upcoming classes, update courses, and add profile details. Site visitors view and browse certification courses.
Camp Management Portal	Camp administrators set reminders, track facility usage, and manage scheduling. Parents submit forms, enroll children, and camp staff track camper assignments and attendance.
Mobile Applications	Users receive customized content based on interests, enable offline learning in language apps, and get location-specific news recommendations.
Banking Application	Customers receive e-statements, schedule payments, and categorize transactions.
Loan Management Application	Loan officers view customer credit history, borrowers calculate loan eligibility, and loan processors track application status.
County Services Portal	Users search for public information, submit applications, pay fees, check application status, and request inspections. Staff perform reviews, issue permits, and schedule inspections.
Music Discovery and Streaming Platform	Users search and discover music based on profiles and preferences. Admins manage video segmentation.

[12]https://github.com/karpurapus/BDDGPT-Automate-Tests
[13]https://data.mendeley.com/datasets/7zbk8zsd8y/1
[14]https://www.parabol.co/blog/user-story-examples/
[15]https://github.com/karpurapus/BDD-Cucumber-scenario-Dataset

B. PROMPT TECHNIQUES

Prompt engineering involves creating clear and concise instructions or queries to guide LLMs in performing specific tasks. These prompts improve model comprehension and performance by providing explicit guidance, reducing ambiguity, and ensuring accurate responses. Wei et al. demonstrate that instruction tuning is an effective method for refining models through fine-tuning with datasets described using explicit instructions [24]. The zero-shot learning capabilities enable the model to understand and perform tasks effectively without additional training [25], [26]. LLMs have impressive zero-shot capabilities, but they may struggle with complex tasks. A few-shot prompting addresses this issue by including examples directly within the prompt. This technique helps the model learn in context and perform better by conditioning it with minimal instances and examples for subsequent tasks [27], [28], [29], [30], [31], [32].

As part of this research, we developed and introduced a novel few-shot chain prompt technique. This method involves providing prompts in a sequence, with each prompt building on the outputs of the previous ones. This technique is beneficial for handling complex task execution, such as writing Cypress automation code that achieves maximum functionality with minimal required code changes to run effectively on the application under test. The few-shot chain prompt technique is implemented in two steps, aligning with the BDD automation implementation procedure described in the BDD methodology section. As shown in Figure 7, In Step 1, the few-shot technique provides the model with an example feature that includes BDD scenarios and their corresponding step definition code. Based on the example, the model then generates the Cypress step definition code for the requested BDD scenarios. In Step 2, we apply the few-shot technique once again.

The model is presented with an example step definition paired with its corresponding POM class code. It is also given with the step definition generated in Step 1 and asked to produce the corresponding POM class code. The model then uses this information to generate the POM class code corresponding to the step definition from Step 1. The example code used for this prompt technique is sourced from a GitHub repository[9] that incorporates automation for the Sauce Labs web application.[16]

The example code referenced in Figure 4, along with the instructions for generating the step definition JavaScript code included in Figure 8, is utilized as part of Step 1 to demonstrate the process. Similarly, the example code mentioned in Figure 5, along with the instructions shown in Figure 9 to generate the POM class Cypress code, is used as part of Step 2.

C. OPEN AI MODELS

We listed the OpenAI models considered for evaluation, as detailed in Table 2. The selected models include:

[16]https://www.saucedemo.com

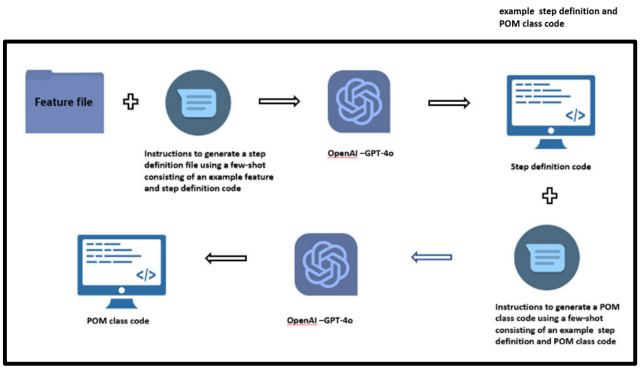

FIGURE 7. Few-shot chain prompt technique.

```
1.Strictly follow the exact syntax for importing
page object classes. Use lower camel case naming
convention. The example syntax is:
import {loginPage} from '@pages/LoginPage';
2.Strictly write multiple imports when importing
more than one page object class.
For example, this is incorrect syntax:
import { calcPage, compPage } from '@pages';
The correct syntax is:
import { calcPage } from '@pages/CalcPage';
import { compPage } from '@pages/CompPage';
3.Write the step definitions for the given BDD scenarios.
```

FIGURE 8. Step 1 instructions in few-shot chain prompt technique.

```
1.Define all web element methods along
with their selectors.
2.Write the complete web element interaction logic
in the UI methods.
3.Provide a complete implementation for every
verification method.
4.In the absence of implementation details, assume
and write the code. Do not leave any method without
full implementation.
5.If application behavior or functionality is
unclear, assume the most common behavior/functionality.
6.Avoid leaving any placeholders; implement the full
logic for each method.
7.Ensure that the Page Object Class is implemented
for all page imports in the step definition file.
8.Ensure the Cypress code does not have any JavaScript
type errors.
9.Review the generated code thoroughly.
Verify each step, and if any step is not followed,
rewrite the code accordingly.
```

FIGURE 9. Step 2 instructions in few-shot chain prompt technique.

D. EVALUATION

We systematically assessed the effectiveness and quality of the code generation process by focusing on two critical aspects: completeness and quality. This comprehensive evaluation aims to identify strengths and areas for improvement in the generated code.

TABLE 2. Open AI models considered for Cypress code generation.

GPT-4o	Our study utilizes the most recent addition – "gpt-4o-2024-08-06" model from OpenAI with zero-shot prompt technique. This model, renowned for its extensive context length of 128,000 tokens and its high intelligence, is well-suited for handling complex, multi-step tasks [33] [34][17].
GPT-4-Turbo	Our study utilizes the latest "gpt-4-turbo-2024-04-09" model with zero-shot prompt technique, known for its advanced vision capabilities and features that include JSON mode and function calling, making it highly effective for vision-related tasks and complex integrations [33] [34][17].
GitHub Copilot	We considered GitHub Copilot version - v1.194.0 in VS Code IDE with a zero-shot prompt technique for our study. GitHub Copilot, powered by OpenAI's Codex, is an AI pair programmer that speeds up coding by suggesting lines or blocks of code based on your context [35-41]. Trained on a vast dataset of public GitHub code, it supports various languages and styles, enhancing productivity and serving as a learning tool with practical coding examples.
GPT-4o with few-shot chain prompt	Our study utilizes the most recent addition - the "gpt-4o-2024-08-06" model from OpenAI with the few-shot chain prompt technique.

Code Generation Completeness Evaluation: Our evaluation of the completeness of code generation is a comprehensive process that involves measuring several vital parameters. This approach provides a robust assessment of the generated code. Specifically, we analyze:

1) STEP METHODS GENERATED

This metric evaluates the total number of step methods generated by each OpenAI model for the corresponding steps for all BDD scenarios in all user stories, offering insights into the completeness of the code generation process using various OpenAI models. Fewer generated step methods indicate a lack of code for the given steps of the BDD scenario.

2) STEP METHODS WITHOUT IMPLEMENTATION

This metric evaluates the total number of step methods generated without any code (i.e., methods that are generated but lack implementation) for all BDD scenarios in all user stories for each model. It highlights areas where the generated code is incomplete and may require further development to be fully functional.

3) POM CLASS METHODS GENERATED

This metric evaluates the total number of POM methods generated by each model for the corresponding steps for all BDD scenarios in all user stories, offering insights into the completeness of the code generation process using various OpenAI models.

4) POM CLASS METHODS WITHOUT IMPLEMENTATION

This metric evaluates the total number of POM class methods generated without any code (i.e., methods that are generated but lack implementation) for all BDD scenarios in all user stories for each model. It highlights areas where the generated POM code is incomplete and may require further development to be fully functional.

5) FRACTION OF STEP METHODS WITH IMPLEMENTATION

As illustrated in Equation (1), we calculated fraction of step methods implementation by dividing the number of step methods with implementation by the total number of step method for each user story, encompassing all BDD scenarios within that story.

$$\begin{aligned} &\textit{Fraction of step`methods with implemenation} \\ &= \frac{\textit{No.of step methods generated with implementation}}{\textit{Total No. of step methods generated}} \end{aligned} \tag{1}$$

6) FRACTION OF STEP METHODS WITHOUT IMPLEMENTATION

Similarly, the fraction of step methods without implementation is computed by dividing the number of step methods lacking implementation by the total number of step methods for each user story, encompassing all BDD scenarios within that story, as shown in Equation (2). This metric is computed for each user story, encompassing all BDD scenarios within that story

$$\begin{aligned} &\textit{Fraction of Step methods with out implemenation} \\ &= \frac{\textit{No. of step methods generated without implementation}}{\textit{Total No. of step methods generated}} \end{aligned} \tag{2}$$

7) FRACTION OF POM METHODS WITH IMPLEMENTATION

As illustrated in Equation (3), as shown at the bottom of the page, we calculated fraction of POM methods implementation by dividing the number of POM methods with

$$\textit{Fraction of POM methods with implementation} = \frac{\textit{No.of POM class methods generated with implementation}}{\textit{Total No. of POM class methods generated}} \tag{3}$$

[17]https://platform.openai.com/docs/models/gpt-4o#4ofootnote

implementation by the total number of POM methods for each user story, encompassing all BDD scenarios within that story. This metric is computed for each user story, encompassing all BDD scenarios within that story.

8) FRACTION OF POM METHODS WITHOUT IMPLEMENTATION

The fraction of POM methods without implementation is computed by dividing the number of POM methods lacking implementation by the total number of POM methods for each user story, encompassing all BDD scenarios within that story, as shown in Equation (4), as shown at the bottom of the page. This metric is computed for each user story, encompassing all BDD scenarios within that story.

Code Generation Quality Evaluation: To assess the quality of the generated code, we focused on identifying syntax and runtime errors. This evaluation includes:

9) SYNTAX AND RUNTIME ERRORS

This metric assesses syntax and runtime errors in the generated code, ensuring it adheres to the correct grammatical rules of the programming language (JavaScript). Identifying syntax errors guarantees code correctness while evaluating runtime errors helps detect issues that arise during execution, ensuring the overall code runs smoothly and error-free.

By thoroughly examining these factors, we aim to ensure the completeness and quality of the generated code, thereby validating the reliability and effectiveness of Cypress code generation using various OpenAI models. This evaluation helps identify the optimal prompt technique and model that performs best across the defined metrics.

IV. RESULTS AND DISCUSSIONS

As outlined in the methodology section, we generated Cypress code for BDD scenarios across all 55 user stories, evaluating various OpenAI models and prompts. The subsequent sections present the findings from the different experiments conducted. Visual representations have been employed to convey critical findings and offer comprehensive insights into the experiments.

Figure 10 highlights the impact of GPT-4o with a few-shot chain prompt on code generation sufficiency for test cases, compared to GPT-4-Turbo, GitHub Copilot, and GPT-4o with a zero-shot prompt. Among these, we observed GPT-4o with a few-shot chain prompt excels in generating a higher number of step methods, leading to more effective implementation of BDD scenario steps and offering superior code coverage for test cases (BDD scenarios). This improved performance is attributed to the few-shot chain prompt technique, which results in more complete code with fewer

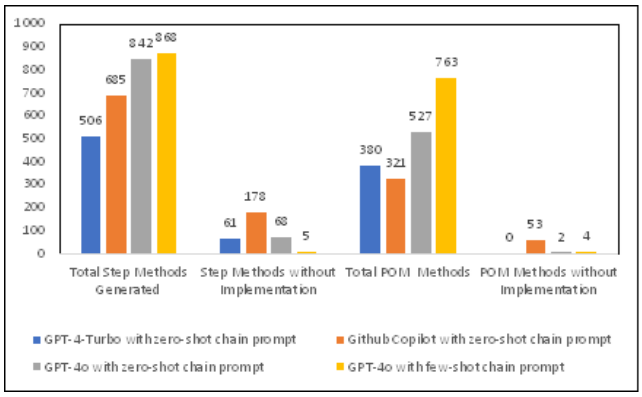

FIGURE 10. Code generation completeness across various models.

unimplemented methods compared to GitHub Copilot, GPT-4o, and GPT-4-Turbo, which use zero-shot prompting. The increased number of step methods and the resulting superior code coverage are key factors that make GPT-4o with a few-shot chain prompt stand out.

The enhanced step method implementation observed in GPT-4o with zero-shot prompt compared to GPT-4-Turbo with zero-shot can be attributed to advancements in GPT-4o. On the other hand, the lower step method implementation by GitHub Copilot with a zero-shot prompt is partly due to errors in generating almost 10% of total BDD scenarios and the fact that GitHub Copilot is based on a GPT model version prior to GPT-4o.

Amon all models, GPT-4o with a few-shot chain prompt demonstrates the highest number of step methods, which can be attributed to the effectiveness of the few-shot chain prompt technique. In terms of POM methods, while there is no direct measure of code coverage extent, the higher number of POM methods in GPT-4o with a few-shot chain prompt can be linked to generating more step methods and adherence to best coding practices. Specifically, this technique declares all web elements as properties within the class and groups actions associated with web elements according to scenario steps. This approach results in more POM methods than GPT-4o and GPT-4-Turbo with a zero-shot prompt and enhances maintainability by organizing code more effectively. Keeping web elements and their associated actions structured and modular makes the resulting code easier to update and maintain and improves overall code quality.

Howeve, these results only demonstrate the overall performance of the models. Therefore, we conducted a detailed analysis to cover the variability aspects across all the feature files (55 feature files with 5 BDD scenarios, one feature file for each user story). The subsequent section discusses the details.

$$\textit{Fraction of POM class method with out implemenation} = \frac{\textit{No. of POM class methods generated without implementation}}{\textit{Total No. of POM class methods generated}} \tag{4}$$

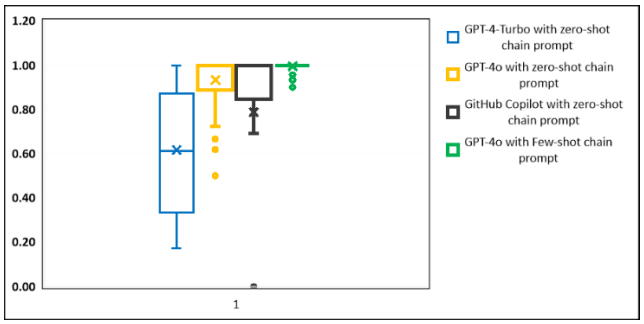

FIGURE 11. Fraction of Step methods with implementation.

In the box plot (Figure 11) we show the fraction of step methods with implementation for all BDD scenarios in user stories for each model. GPT-4-Turbo with a zero-shot prompt shows the lowest median fraction of 0.62 with substantial variability, as indicated by an interquartile range (IQR) from 0.33 to 0.87. In contrast, GPT-4o with a zero-shot prompt demonstrates better performance with a higher median fraction of 0.93, a narrower IQR of 0.89 to 1.0, and fewer outliers. GitHub Copilot with zero-shot prompt has a median fraction of 0.79, with a tight IQR between 0.85 and 1.0 and only minor outliers, reflecting highly reliable performance. GPT-4o with a few-shot chain prompt achieves the highest median fraction of 0.995, with its IQR remaining close to 1.0, indicating near-perfect performance despite occasional outliers below 0.9.

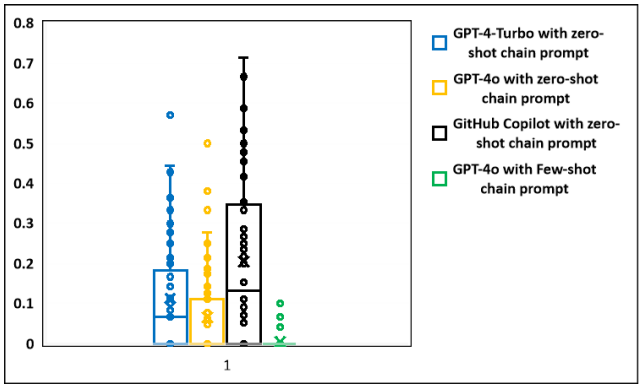

FIGURE 12. Fraction of Step methods without implementation.

The box plot (Figure 12) evaluates the fraction of Step methods without implementation for all BDD scenarios in user stories for each model. GPT-4-Turbo with zero-shot prompt has a median fraction of 0.11, indicating it leaves almost 10% of methods unimplemented, with an IQR spanning from 0.0 to 0.18 and a few outliers above 0.4. GitHub Copilot with zero-shot shows a higher median fraction of around 0.2 and a broader IQR span of 0.0 to 0.34, with noticeable outliers, suggesting it leaves a more significant percentage of methods unimplemented and exhibits less consistent performance. GPT-4o with zero-shot prompt has a median fraction of 0.07, with an IQR spanning from 0.0 to 0.11. GPT-4o with a few-shot chain prompt achieves the best performance with a median fraction close to 0.0 and a narrow IQR, indicating it leaves very few methods unimplemented, though occasional outliers above 0.1 are observed.

Our analysis shows that GPT-4o with a few-shot chain prompt delivers the highest and most consistent implementation in step method generation. GPT-4o with a zero-shot prompt follows with high and reliable implementation but slightly lower performance. GitHub Copilot with a zero-shot prompt ranks third, demonstrating reliable performance but with more variability compared to GPT-4o with the zero-shot prompt. GPT-4-Turbo with zero-shot prompt shows the lowest implementation and more variability, suggesting inconsistent performance across BDD scenarios.

In the box plot (Figure 13) we show the fraction of POM methods with implementation for all BDD scenarios in user stories for each model. GPT-4-Turbo, GPT-4o with a zero-shot prompt, and GPT-4o with few-shot chain prompts demonstrate higher POM methods implementation (Figure 13), with their results clustering near 1.0, indicating that they consistently generate a higher number of fully implemented methods with minimal variance. In contrast, GitHub Copilot with zero-shot prompt exhibits a wide range of variability, with many methods falling below 0.5, reflecting less consistency and less no of fully implemented methods generation. Additionally, we observed that in almost seven instances, GitHub Copilot with zero-shot prompt threw errors related to being unable to generate the required code.

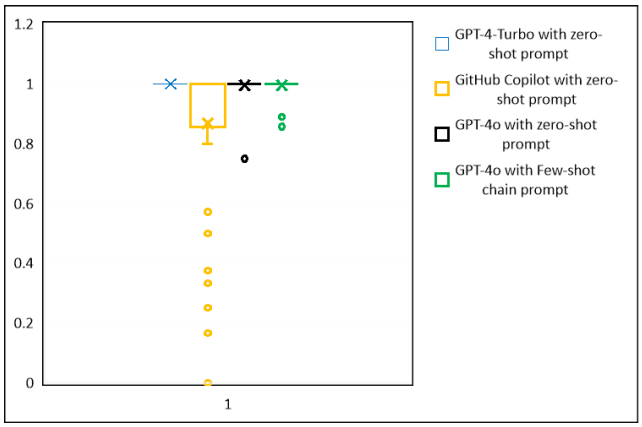

FIGURE 13. Fraction of POM methods with implementation.

The box plot (Figure 14) shows the fraction of unimplemented POM class methods across GPT-4-Turbo, GPT-4o, GitHub Copilot with zero-shot prompt, and GPT-4o with a few-shot chain prompt. GPT-4-Turbo, GPT-4o, and GPT-4o with a few-shot chain prompt show low and consistent fractions of unimplemented methods, clustering near 0. In contrast, GitHub Copilot exhibits high variability, with some results exceeding 0.8, indicating more frequent and inconsistent unimplemented methods. Although the POM methods implementation is good for most models - GPT-4-Turbo with zero-shot prompt, GPT-4o with zero-shot prompt,

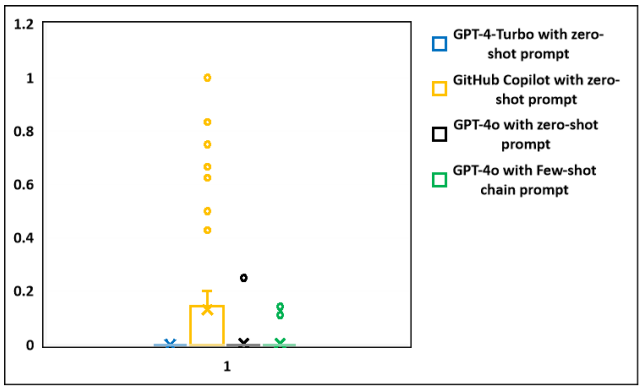

FIGURE 14. Fraction of POM methods without implementation.

and GPT-4o with few-shot chain prompt, we observed that GPT-4o with a few-shot chain prompt model outperforms all others in terms of variance and the high number of POM methods generated.

Overall, the results highlight the superior performance of GPT-4o models, particularly when enhanced with a few-shot chain prompt, in generating fully implemented POM methods. The minimal variance we observed in the GPT-4o few-shot chain prompt model indicates that providing additional context and examples helps optimize the quality and consistency of the generated code. On the other hand, GitHub Copilot with zero-shot prompt demonstrates lower implementation, suggesting it may need further refinement to handle complex automation tasks, such as POM method generation within the Cypress framework using BDD methodology. It is important to note that GitHub Copilot observations are specific to the context of Cypress code generation with BDD, and we may not be generalizable to other coding tasks or methodologies.

Code Generation Quality: To set up the automation repository, one can follow the general structure provided by the Cypress-Cucumber Boilerplate[9] or set up their own. We used this repository to set up the code generated from OpenAI models. While using this specific boilerplate is not mandatory, the foundational setup may remain similar to ensure compatibility with the generated code. The prerequisites include installing Node.js, which comes with Node Package Manager (npm), and installing essential npm packages like Cypress and Cypress-Cucumber Preprocessor.

The models generate code for two main types of files: step definition.js files, which define the test steps, and POM class files, which handle web application automation (web element interactions). These files are integrated into the automation repository, and no modifications are made initially. After the addition of the code, we executed test cases (BDD scenarios) using Cypress commands such as ''npx cypress open'' for interactive testing or ''npx cypress run'' for headless execution. After running the tests, we captured and analyzed various syntax and runtime errors reported in the log file during the Cypress test execution. The log files were further analyzed to categorize the major error types and how the errors occurred across the models. Table 3 describes the error type details.

TABLE 3. Cypress error descriptions.

Import POM statement missing	This error occurs when the step definition file is missing the import statement for the POM class. The POM class file contains web element identifiers and methods for interacting with those elements, which are essential for the proper functioning of the step definitions.
POM class file path not found	This error occurs when the step definition file references a POM class, but the specified file path is incorrect or cannot be located. As a result, the necessary POM class file containing web element identifiers and action methods cannot be accessed, leading to a failure in the test execution.
Import "cypress-cucumber-preprocessor" statement missing	This error occurs when the step definition file is missing the import statement for the "`badeball/cypress-cucumber-preprocessor`" package, which is necessary for integrating Cucumber with Cypress.
Cypress related Type Errors	This error occurs when the test script attempts to use a custom Cypress command, such as "cy.loginAsAdmin", but Cypress does not recognize it as a valid function. This typically happens if the custom command has not been properly defined or imported into the test file.

The aggregation of syntax errors across all models, as shown in Figure 15, reveals that GPT-4-Turbo with zero-shot prompt had the highest incidence of errors (36%), followed by GPT-4o with zero-shot prompt (32%), followed by GitHub Copilot with zero-shot (24%), and then GPT-4o with a few-shot chain prompt technique (only 8%). We showed the distribution of syntax error types in Figure 16, GPT-4o a with few-shot chain prompt encountered only one type of syntax error compared to other models. GitHub Copilot and GPT-4-Turbo, GPT-4o with zero-shot prompt exhibited a variety of syntax error types, requiring more effort to fix compared to GPT-4o with a few-shot chain prompt. The implementation of all models is available in GitHub

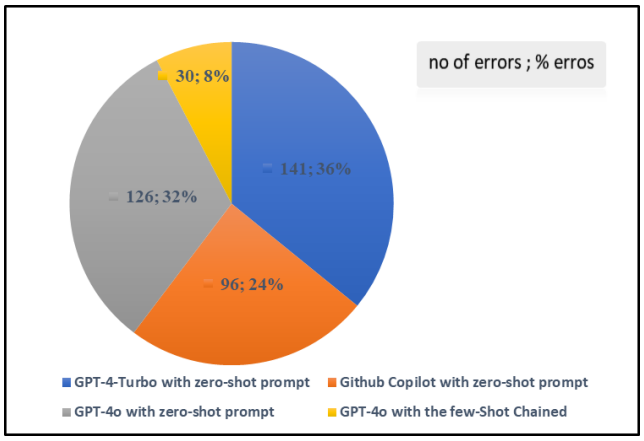

FIGURE 15. Syntax error distribution.

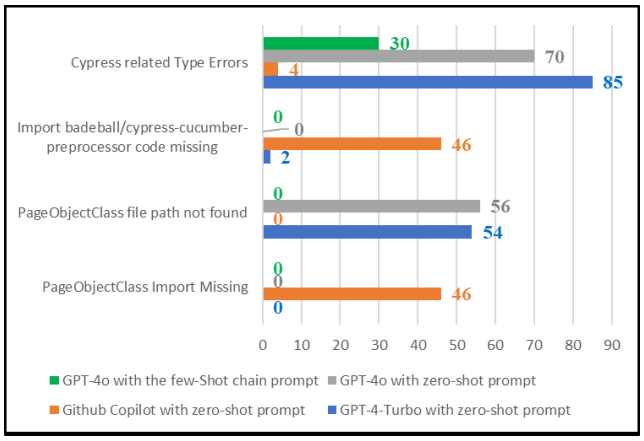

FIGURE 16. Syntax error type distribution.

repository[18]. Overall, these findings highlight the pivotal role of advanced prompt engineering techniques in enhancing the performance of LLMs for Cypress test automation code, particularly in handling BDD scenarios. This is especially significant despite improvements in GPT-4o's ability to manage complex tasks.

V. VISUAL STUDIO EXTENSION DEVELOPMENT

Our results demonstrate that the GPT-4o with a few-shot chain prompt proposed in this manuscript offers superior code generation effectiveness with fewer errors. As a result, we have developed the Cypress-Copilot tool, a Visual Studio Code extension leveraging this technique. Designed to support the Cypress developer community, this extension aims to enhance user productivity through its usability-focused features. After installing the extension, users can follow the steps outlined in Figure 17 below to integrate and utilize the tool effectively. The extension will be made available following acceptance of the paper.

Figure 17 illustrates our recommendation to streamline workflow for developers using the Cypress Copilot extension

[18]https://github.com/karpurapus/OpenAIModels_BDD

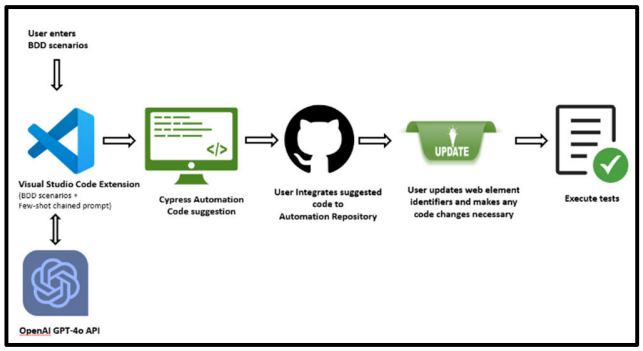

FIGURE 17. Cypress Copilot Visual studio extension workflow.

within the Visual Studio Code IDE. The process begins when the user inputs a BDD scenario into Visual Studio Code, facilitated by the extension, which connects to the OpenAI GPT-4o API. This integration provides automated suggestions for Cypress test code based on the BDD scenarios, significantly enhancing productivity by minimizing manual coding efforts. The user can then integrate the suggested code into their automation repository, typically hosted on a platform such as GitHub. The final step involves updating web element identifiers as needed and executing the tests after making any necessary code adjustments.

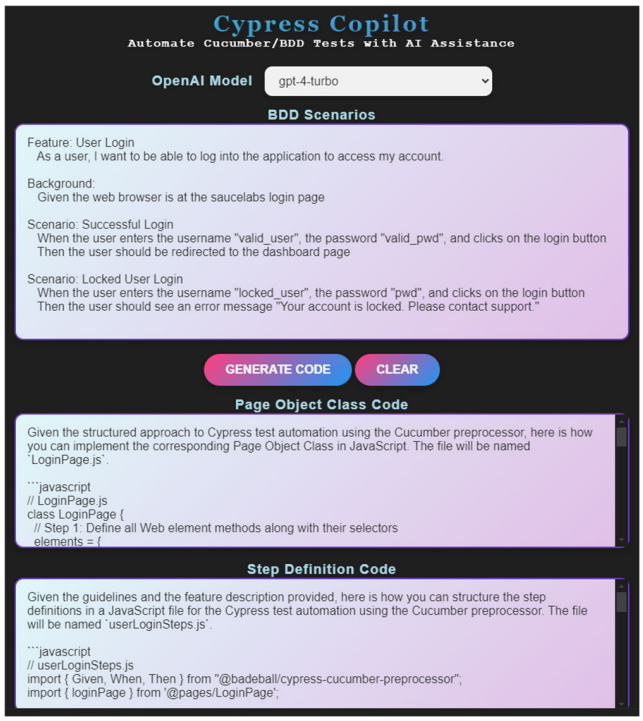

FIGURE 18. Cypress copilot user interface.

The user interface of the Cypress Copilot extension is shown in Figure 18. In this interface, the user needs to enter the BDD scenarios and click on "Generate Code." The extension then suggests code primarily for step definitions and POM implementations, displaying the results

in separate output boxes. We validated the Cypress Copilot extension by conducting web end-to-end (E2E) tests in production environments, utilizing demo websites such as OrangeHRM,SauceLab,and ComputerAssetManagement (Table 4). These demo websites provide environments where automation code can be safely tested and evaluated without disrupting live web services. The automation code generated by the Cypress Copilot extension for these demo websites is publicly available, allowing others to try the tool and execute the code in similar environments and observe the results firsthand. We refrained from using publicly available websites (production environments) for generating the automation code due to permission constraints. Also, running automation scripts on live websites without explicit consent could result in unintended interactions, such as multiple hits on the website by external users, especially as we are making the automation code publicly available. This could negatively affect the website's user experience or operational integrity. Our goal is to share the generated automation code, enabling others to run it in controlled, non-production environments that closely mimic real-time scenarios, while avoiding any potential disruptions to public web services. The objective was to assess the effectiveness and productivity of the code generated by our Cypress Copilot extension. We evaluated the accuracy and efficacy of the generated Cypress code by measuring the ratio of updated lines of code to the total number of lines generated. The code generated for automating these websites, along with the nature of the changes made to ensure successful execution, is stored in GitHub repositories[19,20,21].

The results from Figure 19 and 20 indicate a generally high level of Cypress automation code accuracy across different web applications. The trends reveal that bot POM class files and step definition files require few corrections, suggesting that the initial implementations are largely reliable. Also, we didn't observe any syntax errors in the code. Overall, the data reflects consistent performance, with only minor code adjustments necessary, highlighting the effectiveness of Cypress Copilot extension.

VI. LIMITATIONS AND FUTURE IMPROVEMENTS

Interestingly, we observed that increasing the level of detail in BDD scenarios leads to generating code that closely aligns with the expected behavior. For example, in an e-commerce website, if the product search filter's sorting option requires selecting and applying a filter, the BDD scenario should encompass both steps to ensure that the code generated by Cypress Copilot accurately represents the application's behavior. In addition, we observed that the maximum number of BDD scenarios that can be provided to Cypress Copilot for code generation depends on the model output token limit. For example, GPT-4o (gpt-4o-2024-08-06) has a 16,384-token limit. Therefore, we recommend inputting a number of tests (BDD scenario) based on whether the output can accommodate the required code generation.

The cypress copilot results we presented in this study are based on a limited number of instances, primarily due to the restricted availability of diverse demo websites for automation testing. Although the number of websites tested is limited, the quality and effectiveness of code generation have been rigorously evaluated across a diverse dataset, encompassing nearly 260 test cases, as detailed in the Results and Discussion section. The approach taken for evaluation indicates that the method for generating automated tests and validation is comprehensive and is anticipated to demonstrate significant robustness, suggesting that major discrepancies in outcomes are unlikely when applied to a more diverse set of web applications. As Cypress Copilot is validated in

TABLE 4. Demo web site details.

Website	Description
OrangeHRM[22]	An open-source human resource management platform offering HR services like employee records management, leave tracking, and performance management. Used for testing scenarios related to managing employee details, attendance systems, and performance tracking workflows.
SauceLabs[23]	A cloud-based platform for testing web and mobile applications across multiple browsers and devices. Used for testing scenarios like login, data submissions and cross-browser compatibility.
ComputerAssetManagement[24]	A platform for managing IT assets such as hardware and software. Used for testing scenarios related to asset tracking, maintenance schedules, and resource management workflows.

[19]https://github.com/karpurapus/Cypress-Validation-On-Websites-OrangeHRM

[20]https://github.com/karpurapus/Cypress-Validation-On-Websites-ComputerAssetManagement

[21]https://github.com/karpurapus/Cypress-Validation-On-Websites-Saucelab

[22]https://opensource-demo.orangehrmlive.com/web/index.php/auth/login

[23]https://www.saucedemo.com/

[24]https://computer-database.gatling.io/computers/

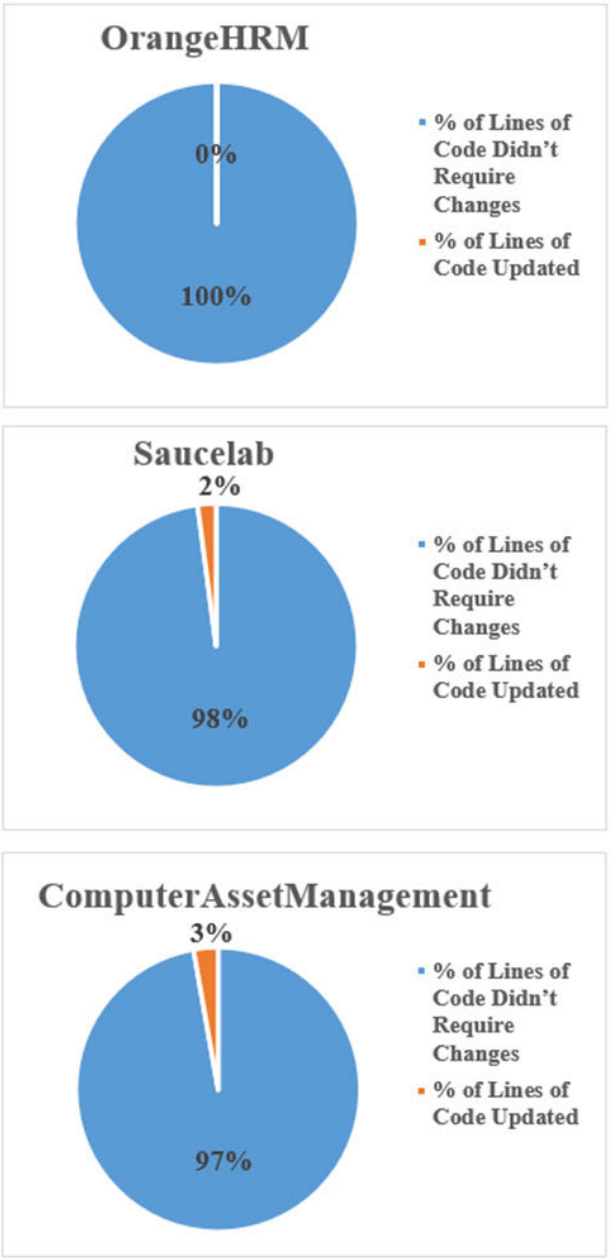

FIGURE 19. Step definition code generation.

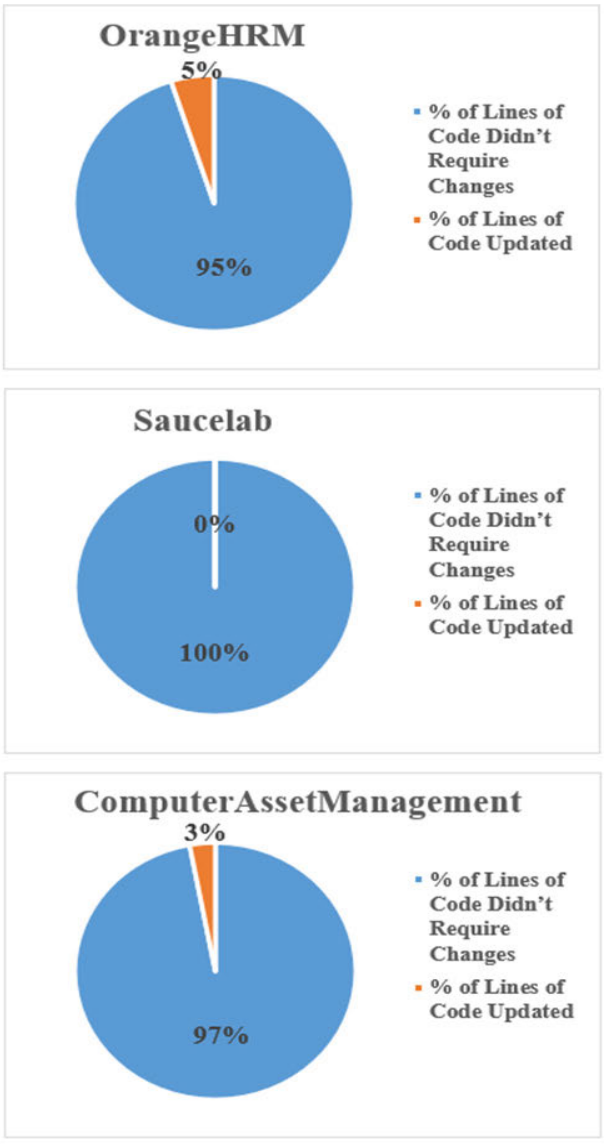

FIGURE 20. POM code generation.

the future across a broader range of web applications, potentially with contributions and feedback from the open-source community, we anticipate the findings will receive additional support in terms of overall code effectiveness and quality.

While test case coverage in terms of meeting requirements is not the focus of our current study, as we are primarily evaluating code generation quality and effectiveness, we see it as an important area for future work. In the next phase of this research, we plan to explore and obtain comprehensive test case suites from real-time projects. We intend to consider a wide range of complexities, from simple to highly intricate test scenarios, to evaluate the proposed approach.

The key benefit of using Cypress Copilot lies in its ability to streamline test automation code generation, significantly reducing the manual effort required for test development and validation. By automatically generating code that aligns with application behavior, the tool accelerates the automation process and enhances development efficiency. Its seamless integration with BDD workflows ensures that the generated tests accurately represent user scenarios and business requirements. Cypress Copilot also aids in the onboarding of new team members by providing clear, generated code examples that accelerate learning. This reduces the learning curve for testers and developers, enabling faster contributions to the project. Furthermore, it promotes the standardization of coding practices across teams by generating consistent, well-structured code, which helps maintain high-quality codebases

and simplifies code reviews. Cypress Copilot's ability to generate quick and efficient code can facilitate seamless integration into CI/CD pipelines, resulting in faster turnaround times for feature implementation and ultimately reducing time to market.

VII. CONCLUSION

In this study, we demonstrated superior performance in generating automation code for web application testing when we applied LLMs in combination with our novel few-shot chain prompt technique. We selected the BDD methodology and Cypress automation tool for web application automation testing. Among various OpenAI models, namely GPT-4o, GPT-4-Turbo, and GitHub Copilot with zero-shot prompt, GPT-4o with a few-shot chain prompt demonstrates superior performance in generating sufficient code for test cases, enhancing code maintainability, and reducing syntax errors. Based on this optimal technique, we developed the novel ''Cypress Copilot'' open-source tool with aim to significantly improve testing efficiency and boost productivity by minimizing manual coding efforts. Validation of the Cypress Copilot Visual Studio Code extension confirms its practical applicability and effectiveness in testing various web applications. We are releasing this tool to the open-source community, as it has the potential to be a promising partner in enhancing productivity in web application automation testing. In future, we plan to extend the validation of Cypress Copilot-generated code to a broader range of web applications and integrate community feedback to enhance the tool's performance and reliability. We believe the foundational findings from our research may serve as part of the key building blocks for a comprehensive end-to-end AI testing solution. By this, we mean a solution that spans the entire testing lifecycle, including developing test scenarios for requirements, test execution, and defect identification. Furthermore, our findings have the potential to encourage further research and inspire new ideas in this field.

REFERENCES

[1] S. Tyagi, R. Sibal, and B. Suri, ''Adopting test automation on agile development projects: A grounded theory study of Indian software organizations,'' in *Agile Processes in Software Engineering and Extreme Programming*, vol. 283, H. Baumeister, H. Lichter, and M. Riebisch, Eds., Cham, Switzerland: Springer, 2017, doi: 10.1007/978-3-319-57633-6_12.

[2] G. Downs, ''Lean-agile acceptance test-driven development: Better software through collaboration by ken pugh,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 4, p. 34, Aug. 2011, doi: 10.1145/1988997.1989006.

[3] L. P. Binamungu, S. M. Embury, and N. Konstantinou, ''Characterising the quality of behaviour driven development specifications,'' in *Proc. 21st Int. Conf. Agile Processes Softw. Eng. Extreme Program.* Copenhagen, Denmark: Springer, Jun. 2020, pp. 87–102.

[4] H. M. Abushama, H. A. Alassam, and F. A. Elhaj, ''The effect of test-driven development and behavior-driven development on project success factors: A systematic literature review based study,'' in *Proc. Int. Conf. Comput., Control, Electr., Electron. Eng. (ICCCEEE)*, Feb. 2021, pp. 1–9.

[5] C. Solis and X. Wang, ''A study of the characteristics of behaviour driven development,'' in *Proc. 37th Euromicro Conf. Softw. Eng. Adv. Appl.*, Oulu, Finland, Aug. 2011, pp. 383–387, doi: 10.1109/SEAA.2011.76.

[6] F. Mobaraya and S. Ali, ''Technical analysis of selenium and cypress as functional automation framework for modern web application testing,'' in *Proc. 9th Int. Conf. Comput. Sci.*, 2019, pp. 27–46, doi: 10.5121/csit.2019.91803.

[7] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, ''Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions,'' in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13, doi: 10.1145/3597503.3639180.

[8] J. Yoon, R. Feldt, and S. Yoo, ''Intent-driven mobile GUI testing with autonomous large language model agents,'' in *Proc. IEEE Conf. Softw. Test., Verification Validation (ICST)*, Toronto, ON, Canada, May 2024, pp. 129–139, doi: 10.1109/icst60714.2024.00020.

[9] A. Dwarakanath, N. Dubash, and S. Podder, ''Machines that test software like humans,'' 2018, *arXiv:1809.09455*.

[10] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, ''AUITestAgent: Automatic requirements oriented GUI function testing,'' 2024, *arXiv:2407.09018*.

[11] S. Yu, C. Fang, M. Du, Y. Ling, Z. Chen, and Z. Su, ''Practical non-intrusive GUI exploration testing with visual-based robotic arms,'' in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Apr. 2024, pp. 1–13.

[12] F. YazdaniBanafsheDaragh and S. Malek, ''Deep GUI: Black-box GUI input generation with deep learning,'' in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Melbourne, VIC, Australia, Nov. 2021, pp. 905–916, doi: 10.1109/ASE51524.2021.9678778.

[13] S. Feng, H. Lu, J. Jiang, T. Xiong, L. Huang, Y. Liang, X. Li, Y. Deng, and A. Aleti, ''Enabling cost-effective UI automation testing with retrieval-based LLMs: A case study in WeChat,'' in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2024, pp. 1973–1978.

[14] R. Pan, M. Kim, R. Krishna, R. Pavuluri, and S. Sinha, ''Multi-language unit test generation using LLMs,'' 2024, *arXiv:2409.03093*.

[15] Z. Wang, K. Liu, G. Li, and Z. Jin, ''HITS: High-coverage LLM-based unit test generation via method slicing,'' in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2024, pp. 1258–1268.

[16] A. Lops, F. Narducci, A. Ragone, and M. Trizio, ''AgoneTest: Automated creation and assessment of unit tests leveraging large language models,'' in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 2440–2441, doi: 10.1145/3691620.3695318.

[17] S. Gu, C. Fang, Q. Zhang, F. Tian, and Z. Chen, ''TestART: Improving LLM-based unit test via co-evolution of automated generation and repair iteration,'' 2024, *arXiv:2408.03095*.

[18] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, ''On the evaluation of large language models in unit test generation,'' in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Oct. 2024, pp. 1607–1619.

[19] S. Fakhoury, A. Naik, G. K. Sakkas, S. Chakraborty, and S. K. Lahiri, ''LLM-based test-driven interactive code generation: User study and empirical evaluation,'' *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, pp. 2254–2268, Sep. 2024, doi: 10.1109/TSE.2024.3428972.

[20] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, ''Automated unit test improvement using large language models at meta,'' in *Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, Jul. 2024, pp. 185–196.

[21] S. D. Sri, R. C. S. P. Raman, G. Rajagopal, and S. T. Chan, ''Automating REST API postman test cases using LLM,'' 2024, *arXiv:2404.10678*.

[22] A. Pereira, B. Lima, and J. P. Faria, ''APITestGenie: Automated API test generation through generative AI,'' 2024, *arXiv:2409.03838*.

[23] S. Karpurapu, S. Myneni, U. Nettur, L. S. Gajja, D. Burke, T. Stiehm, and J. Payne, ''Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation,'' *IEEE Access*, vol. 12, pp. 58715–58721, 2024.

[24] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, ''Finetuned language models are zero-shot learners,'' in *Proc. 10th Int. Conf. Learn. Represent.*, 2022.

[25] W. Wang, V. W. Zheng, H. Yu, and C. Miao, ''A survey of zero-shot learning: Settings, methods, and applications,'' *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 1–37, Mar. 2019, doi: 10.1145/3293318.

[26] Y. Xian, C. H. Lampert, B. Schiele, and Z. Akata, "Zero-shot learning—A comprehensive evaluation of the good, the bad and the ugly," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 9, pp. 2251–2265, Sep. 2019.

[27] Y. Song, T. Wang, P. Cai, S. K. Mondal, and J. P. Sahoo, "A comprehensive survey of few-shot learning: Evolution, applications, challenges, and opportunities," *ACM Comput. Surv.*, vol. 55, no. 13, pp. 1–40, Dec. 2023.

[28] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020, *arXiv:2001.08361*.

[29] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, and S. Agarwal, "Language models are few-shot learners," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2020, pp. 1877–1901.

[30] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" in *Proc. EMNLP*, 2022, pp. 1–14, doi: 10.18653/v1/2022.emnlp-main.759. [Online]. Available: https://par.nsf.gov/biblio/10462310

[31] 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," 2023, *arXiv:2302.13971*.

[32] T. Sun, Z. He, Q. Zhu, X. Qiu, and X. Huang, "Multitask pre-training of modular prompt for Chinese few-shot learning," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics*. Toronto, ONT, Canada: Association for Computational Linguistics, 2023, pp. 11156–11172.

[33] J. Huang and J. Zhang, "A survey on evaluation of multimodal large language models," 2024, *arXiv:2408.15769*.

[34] T. Ball, S. Chen, and C. Herley, "Can we count on LLMs? The fixed-effect fallacy and claims of GPT-4 capabilities," 2024, *arXiv:2409.07638*.

[35] I. Siroš, D. Singelée, and B. Preneel, "GitHub copilot: The perfect code compLeeter?" 2024, *arXiv:2406.11326*.

[36] W. Cheng, K. Sun, X. Zhang, and W. Wang, "While GitHub copilot excels at coding, does it ensure responsible output?" 2024, *arXiv:2408.11006*.

[37] M. Borg, D. Hewett, D. Graham, N. Couderc, E. Söderberg, L. Church, and D. Farley, "Does co-development with AI assistants lead to more maintainable code? A registered report," 2024, *arXiv:2408.10758*.

[38] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024, *arXiv:2406.00515*.

[39] J. Res, I. Homoliak, M. Perečíni, A. Smrč ka, K. Malinka, and P. Hanacek, "Enhancing security of AI-based code synthesis with GitHub copilot via cheap and efficient prompt-engineering," 2024, *arXiv:2403.12671*.

[40] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the security of GitHub Copilot's generated code—A targeted replication study," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2024, pp. 435–444.

[41] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, "Demystifying practices, challenges and expected features of using GitHub copilot," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 33, pp. 1653–1672, Dec. 2023.

SURESH BABU NETTUR received the B.Tech. degree in computer science engineering from Acharya Nagarjuna University, Guntur, India, and the M.Sc. degree from Birla Institute of Technology and Science (BITS), Pilani, Rajasthan, India.

He has over two decades of experience from conceptualization and design to developing software automation solutions, with a strong focus on scalable solutions for various platforms across banking, healthcare, telecom, and manufacturing industries using Agile and Waterfall methodologies. He has worked extensively in various industries, particularly financial services, developing complex applications using technologies, such as Spring Boot, RESTful Web Services, and DevOps tools. In addition, he is proficient in leveraging AI tools, such as GitHub Copilot and OpenAI models to enhance productivity and code quality, incorporating these innovations into both development and testing processes. His experience includes leading cross-functional teams, managing onsite-offshore development models, and leveraging cloud platforms like AWS. He is also skilled in Agile methodologies, Test-Driven Development (TDD), and the design and implementation of service-oriented architectures (SOA). His work is driven by a passion for delivering innovative solutions through AI and machine learning, integrating these technologies into software engineering, healthcare, and finance for impactful and sustainable outcomes.

SHANTHI KARPURAPU received the B.Tech. degree in chemical engineering from Osmania University, Hyderabad, India, and the Master of Technology degree in chemical engineering from the Institute of Chemical Technology, Mumbai, India.

She has over a decade of experience leading, designing, and developing test automation solutions for various platforms across healthcare, banking, and manufacturing industries using Agile and Waterfall methodologies. She is experienced in building reusable and extendable automation frameworks for web applications, REST, SOAP, and microservices. She is a strong follower of the shift-left testing approach, a certified AWS cloud practitioner, and a machine learning specialist. She is passionate about utilizing AI-related technologies in software testing and the healthcare industry.

UNNATI NETTUR is currently pursuing the bachelor's degree in computer science with Virginia Tech, Blacksburg, VA, USA. She possesses an avid curiosity about the constantly evolving field of technology and software development, with a particular interest in artificial intelligence. She is passionate about gaining experience in building innovative and creative solutions for current issues in the field of software engineering.

LIKHIT SAGAR GAJJA is currently pursuing the bachelor's degree in computer science with BML Munjal University, Haryana, India. He is evident in showing his passion for the dynamic field of technology and software development. His research interests include artificial intelligence, prompt engineering, and game-designing technologies, highlighting his dedication to obtaining hands-on experience and developing innovative solutions for real-time issues in software engineering.