End-to-End Testing of Modern Web Applications Using Large Language Models: Appendix

Anonymous Authors

Appendix A: Examples of Use case scenario and test case

An intuitive, universally recognized example of use case scenario is reported in Table I. A test case generated from it by the *Exploration Testing Module* is shown in Listing 1. This use case is presented here in English for the benefit of international readers; however, the original version was written in Italian and used as it is by GenE2E.

Listing 1: Example of test case for the *Login* use case

```
{
  "test_case_id": "UC1_TC1",
  "title": "Login with valid credentials",
  "preconditions": "User not authenticated",
  "postconditions": "User authenticated
       with awareness that the operation was successful",
  "test_steps": [
    {
      "step
          ": "Enter correct credentials in the login form",
      "expected": "The system accepts the credentials"
    },
    {
      "step": "Click the Login button",
      "expected": "The user is successfully authenticated"
    },
    {
      "step": "View
           the message confirming successful operation",
      "expected":
            "The message confirms successful authentication"
    }
  ],
  "test_type": "Positive",
  "priority": "High",
  "use_case_id": "UC1"
}
```

Appendix B: Prompt structure and components

The structure of all prompts (see the prompt template shown in Table II) was defined according to the TELeR (Turn, Expression, Level of Details, Role) taxonomy [1]. This taxonomic approach provides a structured methodology for prompt engineering optimization in complex natural language processing tasks. We adopted the configuration *(*$Turn = Single$*,* $Expression = Instruction$*,* $Role = defined$*,* $Level = 5$*)* for the four dimensions of such a taxonomy. A comprehensive description of these configuration parameters is as follows:

- *Turn*: Prompts can be either single or multi-turn, depending on the number of turns used while prompting LLMs in order to perform a complex task.
  While prompts for complex computational tasks often include detailed descriptions and sub-task specifications, which could benefit from multi-turn interactions with a language model, empirical evidence showed that a single-turn approach was more effective for this application domain.
- *Expression style*: Depending on how the overall directive and its sub-tasks are expressed, prompts can take the form of either questions or instructions. The instruction style was selected in our work since it provides explicit directive-based communication that optimizes model comprehension and task execution accuracy.
- *Role*: Prompts can be categorized as either system-role defined or undefined. A system role was defined to enhance response quality through explicit expertise attribution, leveraging the model's capacity for role-based behavioural adaptation.
- *Level of Details*: Prompts are divided into seven distinct levels (levels 0-6). The degree of detail is determined by the presence or absence of different aspects like clear goals, sub-task division, explanation seeking, few-shot examples, etc. By definition, Level 0 means minimal details (i.e., no aspects/no directive), while Level 6 means the directives include clear goals, distinct sub-tasks/steps, an explicit requirement of explanation/justification, well-defined criteria for evaluation, additional information fetched via information retrieval techniques and/or few shot examples.
  Given the inherent complexity of the target task, which necessitates numerous specific directives and comprehensive guidance, a high level of detail was deemed essential for optimal performance. According to the established TELeR's reference scale we adopted the penultimate detail level 5, ensuring comprehensive task specification via information retrieved and assembled by the dependency resolver and prompt builder components of GenE2E, but without further justification/explanations.

The prompt architecture consists of the following sections:

1) Role definition: This section assigns a specific professional role to the language model, explicitly designating it as a domain expert to leverage improved response quality through specialized contextual framing and enhanced task-specific knowledge application.
2) Task definition: Provides a clear, concise, and unambiguous description of the primary objective, establishing the foundational understanding required for successful task execution and ensuring alignment between system expectations and model output.
3) Existing code: This represents an optional component that

Use Case	User Login
Summary	Authentication of a user registered in the system
Actors	Region, Mayor, Supplier, Designer, Citizen
Precondition	The user is not authenticated
Postcondition	The user is authenticated and aware that the operation was successful
Base Sequence	1. The user enters credentials in the form 2. The user starts the authentication process by clicking the "Login" button 3. The user sees a message indicating the operation was successfully completed
Branch Sequence	2.1 The user is asked to change their initial access password 2.2 The user enters the new password and confirms
Exception Sequence	3. The user sees a message indicating the operation was completed with an error related to the exception (e.g., incorrect credentials, password mismatch, etc.) 4. The user can restart from step 1 of the base sequence

TABLE I: Use Case Scenario: User Login

is dynamically included only when the use case for which test code is being generated exhibits direct dependencies on previously processed use cases. The module incorporates previously generated code from test cases representing nominal execution paths (happy path scenarios), enabling contextual continuity and code reuse optimization.

4) Parameters: Contains essential environmental variables and configuration data to be utilized within the generated code, including comprehensive test data specifications, system configuration parameters, and runtime environment variables necessary for proper test execution.
5) Context Integration: Incorporates relevant Page Object Model components identified during the Context Injection phase, providing the model with detailed knowledge of the portal's structural architecture, available user interface interactions, and system behavioural patterns necessary for accurate test case implementation.
6) Instructions Specification: Delivers refined, specific guidance regarding code generation requirements, encompassing code organization principles, file structure specifications, integration methodologies with existing system components, and adherence to established software engineering best practices.
7) Test case input: contains the detailed description of the test case specifications in Italian language, formatted using Markdown syntax to enhance readability and comprehension, while preserving the semantic integrity of the original requirements documentation.
8) Output format: Establishes comprehensive specifications for structuring the generated output, including detailed file organization schemas, naming convention standards, formatting requirements, and integration protocols with existing codebase architecture.

During the prompt construction phase, empirical analysis revealed that the sequential ordering of these modular sections exhibits critical importance for optimal performance outcomes. Research demonstrates that in-context recall performance is highly prompt-dependent [2], particularly when utilizing substantial portions of the model's context window capacity. Through systematic experimentation, positioning the context section at the prompt beginning and the test case input section at the conclusion yielded superior results, with generated code demonstrating enhanced alignment with requirements specified in the test case descriptions. This performance differential was particularly pronounced when utilizing CodeLlama due to its more constrained context window limitations compared to other model architectures.

The prompt is written in English to optimize instruction comprehension, leveraging the fact that English represents the primary language of the extensive documentation corpus utilized during the model's pre-training phase, despite Llama3.3's demonstrated proficiency in Italian language processing. However, the input test cases are deliberately maintained in their original Italian format to preserve semantic integrity and avoid potential meaning degradation that could occur during translation processes, ensuring that domain-specific terminology and contextual nuances remain intact throughout the generation workflow.

Table III reports the principal differences between the prompts specifically engineered for Llama3.3 and CodeLlama models. Despite these targeted modifications, the fundamental structural framework remains consistent across both implementations, as these models belong to the same transformer architecture family and consequently exhibit analogous behavioural patterns and processing mechanisms.

An example of prompt, as instance of the template prompt presented above, is reported in Listing 2.

TABLE II: Prompt template (configuration 1: single processing, zero-shot, Llama3.3)

Section	**Prompt**
Role	*You are a testing engineer, expert in writing Javascript end-to-end tests with Playwright.*
Task	*Task: Analyze the test case provided below and generate a complete and working Playwright test script in Javascript, following the steps described in the test case. When creating the test script, use only the page object model provided for interacting with the DOM and the parameters provided.*
Context	*Use only this page object models for interacting with the DOM:* *“‘javascript* *...* *“‘*
Parameters	*Use only the following parameters from environment variables in the test scripts:* *...*
Existing code	*When in the test case there is a step, a precondition or a postcondition with a name of use case UC*, reuse this functions passing null as reporter parameter:* *“‘javascript* *...* *“‘*
Instructions	*Instructions:* *1. Analyze the test case provided* *2. For the test case, generate a test script in Javascript with Playwright* *3. Use only Page object model provided to interact with the DOM.* *4. Call the TestResultReporter object methods as described below.* *5. Put the code in functions for reusability, with the reporter object as parameter not mandatory. If not null, call reporter methods.* *6. Put functions in a separate file called ”....functions.js” and the test in a file called ”....spec.js”* *7. Do not rewrite functions provided in the prompt, just refer to them.* *8. Add the required import for:* ** test, expect, ... from ’@playwright/test’ in all files* ** TestResultReporter from ”../../models/test-result-reporter.js”* ** page object models from ”../../models/page_object_models/pom_name .js”* ** existing code from the path specified in the code snippet. Pay attention to add imports used also by the existing code.* ** make the generated ”....spec.js” file to reference functions in ”....functions.js” generated file*
Input (test case)	*Here the test case to be converted in Playwright script:* *...*
Output format	*Output format:* ** Output only valid and runnable JavaScript code* ** Do not write code implementation provided in the prompt, only reference it* ** Structure the code in files using markdown code blocks specifying the file name*

Element	**Llama3.3**	**CodeLlama**
Code delimitators	Markdown	XML tags
Task section	Concise	Concise with sentences to pass safeguards

TABLE III: Prompt differences between models

Listing 2: Prompt example for configuration 1

```
ROLE: system
You are a testing engineer, expert in writing Javascript end-to-end tests with Playwright.

Task: Analyze the test case provided below
     and generate a complete and working Playwright test script in Javascript, following the steps described in the test case.
When creating
     the test script, use only the page object model provided for interacting with the DOM and the parameters provided.

ROLE: user
Use only this page object models for interacting with the DOM:
'''javascript
File: login_page.js
export class LoginPage {
  constructor(page) {
    this.page = page;

    // Locators
    this.loginLink = page.getByRole('link', { name: 'Login' });
    this.emailInput = page.getByLabel('E-mail');
    this.passwordInput = page.getByLabel('Password', { exact: true });
    this.loginButton = page.getByRole('button', { name: 'Login' });

    // Selectors
    this.emailFieldSelector = 'input[name="email"]';
    }

  // Login flow methods
  async clickLoginLink() {
    await this.loginLink.click();
  }

  async isEmailFieldVisible() {
    return await this.page.isVisible(this.emailFieldSelector);
  }

  async fillEmail(email) {
    await this.emailInput.fill(email);
  }

  async fillPassword(password) {
    await this.passwordInput.fill(password);
  }

  async clickLoginButton() {
    await this.loginButton.click();
  }
}

'''

Use only the following parameters from environment variables in the test scripts:
E2E_BASE_URL=""
E2E_LOGIN_URL=""
E2E_HOME_URL=""
E2E_DASHBOARD_URL=""
E2E_CTS_URL=""
EMAIL=""
PASSWORD=""



Instructions:
1. Analyze the test case provided
2. For the test case, generate a test script in Javascript with Playwright
3. Use only Page object model provided to interact with the DOM.
4. Call the TestResultReporter object methods as described below.
5. Put the code
     in functions for reusability, with the reporter object as parameter not mandatory. If not null, call reporter methods.
6. Put functions in a separate file called "UC1_TC1.functions.js" and the test in a file called "UC1_TC1.spec.js"
7. Do not rewrite functions provided in the prompt, just refer to them.
8. Add the required import for:
   * test, expect, ... from '@playwright/test' in all files
   * TestResultReporter from "../../models/test-result-reporter.js"
   * page object models from "../../models/page_object_models/<page_object_model_name>.js"
   * existing code from the path specified in the code snippet. Pay attention to add imports used also by the existing code.
   * make the generated "UC1_TC1.spec.js" file to reference functions in "UC1_TC1.functions.js" generated file


<example>
export const functionNameDescribingStep = async function(page, reporter) {
   const startTime = DateTime.now();
```

```
   // Put here test case step implementation

   const endTime = DateTime.now();
   const executionTime = endTime - startTime;
   if (reporter) {
      reporter.addStep('UC1_TC1_ID1',
           'Step 1 description', expectedResults, actualResults, passFail, parametersUsed, executionTime);
   }

   // Include Playwright assertions
   expect(passFail).toBeTruthy();
}

test("UC1_TC1 - Login test with success", async ({page, browserName}) => {
   reporter.setBrowserName(browserName);
   reporter.setTestCase("UC1_TC1", "Login test with success");

   // Call step functions in sequence
   await step1(page, reporter);
   await step2(page, reporter);
   // Additional steps...

   reporter.onTestEnd(test, { status: "passed" }); // status can be "passed" or "failed"
});
</example>

Here the test case to be converted in Playwright script:
Test Case ID: UC1_TC1
Title: Login con credenziali valide
Use Case ID: UC1
Priority: Alta
Type: Positivo
Preconditions: L'utente non è autenticato
Postconditions: Lutente è autenticato ed è a conoscenza che loperazione ha avuto successo
Test Steps:
  1. Step: Inserisci le credenziali corrette nel form di login
Expected: Il sistema accetta le credenziali
  2. Step: Clicca il tasto ``Login''
Expected: Lutente viene autenticato con successo
  3. Step: Visualizza il messaggio di operazione completata con successo
Expected: Il messaggio conferma lavvenuta autenticazione


Output format:
* Output only valid and runnable JavaScript code
* Do not write code implementation provided in the prompt, only reference it
* Structure the code in files using markdown code blocks specifying the file name
```

References

[1] S. K. K. Santu and D. Feng, "Teler: A general taxonomy of LLM prompts for benchmarking complex tasks," in *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, 2023, pp. 14 197–14 203. [Online]. Available: https://doi.org/10.18653/v1/2023.findings-emnlp.946

[2] D. Machlab and R. Battle, "Llm in-context recall is prompt dependent," 2024. [Online]. Available: https://arxiv.org/abs/2404.08865