

Local LLMs for End-To-End Testing in Practice: Lessons from a Smart City Web Application

Fabio Moretti  

ENEA, ENEA Smart Cities and Communities Laboratory, Rome, Italy

Simone Ronzoni 

University of Bergamo, Italy

Patrizia Scandurra  

University of Bergamo, Italy

Vincenzo Scotti  

KASTEL, Karlsruhe Institute of Technology, Germany

Abstract

End-to-End (E2E) testing of modern web applications simulates real user interactions to verify the complete application flow, ensuring seamless integration of UI, functionality, and data. It is, in general, costly to create and maintain, especially when the backend is a smart city platform with a data lake and complex user flows that introduce additional complexity. Recent advances in Large Language Models (LLMs) offer new opportunities for automating parts of this process, but their practical adoption in real development environments remains challenging, particularly when privacy, monetary cost, deployment, and control requirements call for local rather than cloud-based models.

This paper investigates the feasibility of introducing local LLMs into the E2E testing process of a real-world smart city web application, the ENEA PELL-IP portal for monitoring public lighting infrastructures. We developed and evaluated GenE2E, a two-stage pipeline that generates test cases from use case specifications and transforms them into executable Playwright tests. The approach was integrated with an existing testing environment and evaluated against manually written baseline tests.

Our findings show that local LLMs can effectively support E2E test generation, but not yet as fully autonomous tools. In our setting, Llama 3.3 outperformed CodeLlama mainly due to its larger context window; single-test generation improved focus and coverage, whereas batch processing reduced model invocations and showed lower generation time in our setup, but the timing results are hardware-dependent and based on a single run; broader efficiency claims require repeated measurements on representative hardware. We also observed that specification quality, completeness of Page Object Model (application-specific API wrapping HTML pages), and use case dependency management had a major impact on generated test quality. Based on this feasibility study, we derive preliminary lessons for practitioners considering local LLMs for E2E test generation.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Computing methodologies → Artificial intelligence

Keywords and phrases End-to-End testing, Use Case Scenarios, Web applications, Large Language Models

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.88

Category Software Engineering in Practice Track Paper

Supplementary Material *Software*: <https://zenodo.org/records/21629002> [19]

Funding *Patrizia Scandurra*: Italian MASE-ENEA Programme Agreement on Electrical System Research, PTR 2025–2027, Project 1.7, Activity Line 2.6, “Extension of the automation testing framework for PELL School Buildings.”

Vincenzo Scotti: Pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF)



© Fabio Moretti, Simone Ronzoni, Patrizia Scandurra, and Vincenzo Scotti;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 88; pp. 88:1–88:21



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

End-to-end (E2E) testing [13] is a critical phase of software quality assurance that validates the overall system behaviour from the user's perspective. Traditional E2E testing relies heavily on manual procedures or semi-automated frameworks requiring significant human effort for creating, maintaining, and executing test cases. These conventional methods face challenges in scalability and maintenance due to the dynamic nature and evolvability of modern web applications with complex interfaces and business logic. Modern web applications adopt a multi-tier architecture where the implementation is scattered across different layers and run on different machines. When the backend is a smart city platform with a data lake, deeper interaction flows introduce additional complexity, requiring validation across distributed services, heterogeneous data sources, and dynamic system behaviors.

Large Language Models (LLMs) have emerged as transformative AI tools offering new opportunities for automating software lifecycle tasks [6, 18], including test generation. LLMs can transform natural language specifications into structured test scenarios and executable scripts, potentially reducing manual effort and improving testing productivity. However, most existing studies focus on cloud-based models and controlled experimental settings, while practical adoption in industrial environments raises additional challenges related to privacy, deployment constraints, monetary cost, and the need for operational control. These factors often motivate the use of *self-hosted* (local) LLMs, whose effectiveness in real-world E2E testing pipelines remains underexplored. Moreover, current research has largely focused on unit or specialized testing (e.g., accessibility or security), while E2E testing – due to its broader scope and dependency on system-level context – remains comparatively less investigated [3, 12].

Conventional E2E testing involves multi-stage manual analysis of use case scenarios to produce comprehensive test suites covering typical and edge-case user flows. Automation frameworks (like Playwright¹) simulate user interactions programmatically, but developing such scripts remains resource-intensive and prone to maintenance issues. LLMs' capabilities in contextual reasoning and code generation promise substantial benefits, potentially automating the E2E testing process and improving efficiency, coverage, and adaptability.

This paper reports our experience introducing local LLMs into the E2E testing process of a real-world smart city application, namely the ENEA PELL-IP web portal for monitoring public street lighting infrastructures [1]. To this end, we developed GenE2E, a two-stage pipeline that first generates test cases from use case specifications and, then, transforms them into executable Playwright scripts. The approach follows a *pipe-and-filter* architecture and has been integrated into an existing industrial testing workflow.

Rather than presenting a field deployment or a mature experience report, this study provides an initial feasibility assessment of using self-hosted LLMs to generate executable E2E tests within an existing industrial testing environment. Its core contribution is demonstrating that a complete requirements-to-test pipeline can be instantiated for a real-world web application under practical constraints such as local deployment, integration with an existing Page Object Model, and use-case dependencies. The evaluation is conducted offline by comparing generated tests with pre-existing manually written tests; therefore, our findings characterize the current capabilities, limitations, and trade-offs of the approach rather than its impact on an operational development process.

¹ Popular open-source automation tool for testing modern web applications <https://playwright.dev/>

The results show that local LLMs can provide a useful starting point for E2E test development, but the generated artifacts still require human review and correction. Based on the observed outcomes, we derive preliminary guidance for practitioners.

This paper is organized as follows. Section 2 reports on related work. Section 3 presents the web application under test, clarifying the overall domain-related scope. Section 4 describes the GenE2E software architecture based on LLMs. Sections 5 and 6 present the design and results of an empirical evaluation of GenE2E. Sect. 7 discusses lessons learned, limitations, validity threats, and actionable lessons for practitioners, while Sect. 8 concludes the paper.

2 Related Work

This section reviews LLM-based E2E test generation, focusing on preliminary empirical studies and two approach families: *feature inference* and *multi-stage generation* from requirements.

Preliminary studies

Several empirical studies have evaluated the effectiveness of LLM-supported test automation. The study in [15] compares manual coding, capture-and-replay techniques (CRT), and natural language translation (NLT) using Gherkin specifications². It reports that manual implementation requires the highest effort, CRT achieves the fastest development with minimal human intervention, and NLT provides an intermediate solution but suffers from scalability limitations. However, CRT sacrifices maintainability due to poor script reuse. Similarly, [24] explores the use of LLMs to translate Gherkin specifications into Selenium scripts, showing that LLM-based approaches can reduce repetitive test engineering tasks and provide a promising baseline for automation.

In [14], four approaches are compared: manual coding, ChatGPT Lite, ChatGPT Max, and GitHub Copilot. All approaches rely on Gherkin descriptions, Selenium WebDriver³, and JUnit. Results show that manual coding is the most time-consuming (54 sec/LOC), followed by ChatGPT Lite (47 sec/LOC) and Copilot (45 sec/LOC), while ChatGPT Max – leveraging iterative prompt refinement – achieves the best performance (38 sec/LOC). These studies emphasize that existing LLM-based solutions mainly act as development assistants requiring human interaction and refinement, whereas GenE2E aims to substantially automate the lifecycle from requirements to executable tests, while retaining human oversight for feasibility filtering and validation, and to support their integration into a CI/CD pipeline for continuous testing.

GenE2E differs by addressing requirement-based E2E testing of web applications, starting from use case scenarios, handling use-case dependencies, and generating Playwright tests through a two-stage local-LLM pipeline.

Feature inference approaches

Feature inference approaches generate E2E tests by directly exploring application behavior rather than relying on predefined requirements. AutoE2E [3] autonomously navigates web applications, observes UI states, and uses probabilistic scoring to identify features, which are then transformed into action sequences for testing. However, its one-to-one mapping between features and test cases may limit coverage in complex scenarios.

² The semi-formal Gherkin language is an intermediary specification to address the gap between test requirements and executable test scripts.

³ <https://www.selenium.dev/>

Scriptless testing [29] follows a similar philosophy by dynamically generating test sequences through GUI exploration, selecting actions using strategies such as random exploration, model-based inference, or reinforcement learning. GPTDroid [16] further refines this concept by iteratively extracting GUI context, encoding it into prompts, and translating LLM responses into executable actions.

XUAT-Copilot [30] extends this paradigm with a multi-agent architecture, where LLM-based agents collaborate on action planning, state checking, and parameter selection, supported by modules for state sensing and test case rewriting. This system has been validated in a production environment (WeChat Pay).

While these approaches enable high levels of automation, they do not rely on requirements specifications. Instead, they infer features purely from observed behavior, which may conflict with established testing principles and introduce a circular dependency: the correctness of generated tests depends on the correctness of the application under test.

Multi-stage generation from requirements

Multi-stage approaches instead use requirements as input to progressively derive test scenarios and executable scripts. The work in [8] adopts a two-step pipeline: deriving Gherkin acceptance tests from user stories and then translating them into executable Cypress scripts using knowledge of the HTML structure. While aligned with requirement-driven testing, it does not consider dependencies among test cases. GenIA-E2ETest [12] proposes a multi-level prompting strategy to guide the LLM across three tasks: restructuring and modularizing test scenarios, identifying and refining UI elements, and generating executable scripts. Although it supports natural language scenarios, it does not start from use case requirements.

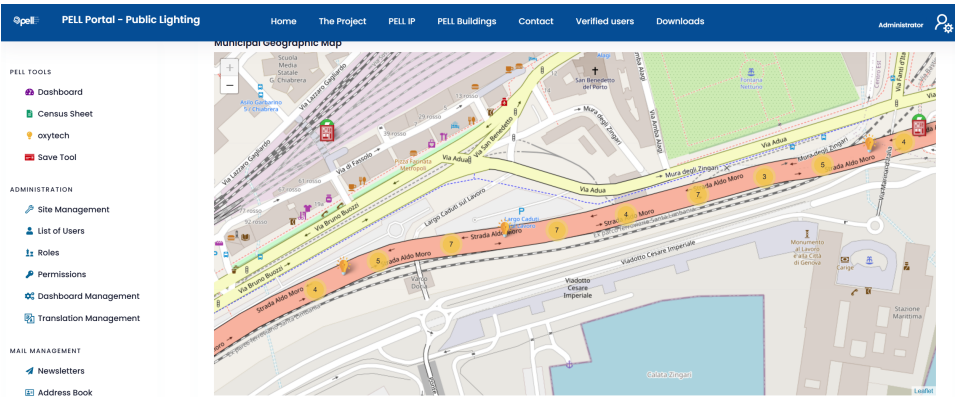
Overall, multi-stage approaches better align with traditional testing practices by grounding test generation in requirements. In this context, GenE2E advances the state of the art by aiming at full automation of the entire process, from requirements analysis to executable E2E test generation, while addressing limitations such as test completeness and dependency handling.

3 Running example: the ENEA PELL-IP web portal

The Italian National Agency for New Technologies, Energy and Sustainable Economic Development (ENEA) focuses on improving the management of energy-intensive urban infrastructures. To support this objective, ENEA developed the smart city platform PELL (Public Energy Living Lab) [1], which collects, organizes, processes, and monitors infrastructure data at a national scale, enabling more effective and data-driven management.

Public street lighting has been selected as a primary application domain, as it accounts for nearly 40% of municipal energy consumption in Europe [21,23]. Despite the availability of advanced solutions such as LED technologies, solar-powered systems, and data-driven tools, many municipalities still operate inefficient legacy systems [7]. The PELL-IP platform [2] addresses this issue by collecting and analyzing both static and dynamic data to continuously assess infrastructure performance in terms of functionality, energy use, and costs. The platform relies on public administrations conducting infrastructure censuses and adopting smart meters to enable accurate energy consumption monitoring.

Figure 1 shows a portion of the PELL-IP platform dashboard section, which provides users with a tailored interface composed by a set of widgets describing the data acquired. Each user has its own dashboard according to specific scopes and interests (e.g. citizen, municipality, utility, regional administrators, national administrators). In this specific case, a geo-referenced map shows the lighting spots located nearby the Genoa port. Each lighting spot can be navigated and its features visualized through a popup.



■ **Figure 1** PELL IP web portal dashboard.

The PELL-IP portal implements 41 use cases, all of which are available for reference in the ENEA GitLab repository ⁴. Of the existing use cases and their test cases, a subset (related to the PELL tools Census Sheet and Dashboard) has been considered for representation of the portal’s operational scenarios, ensuring a reference suite of test cases.

4 The GenE2E framework

The goal is the automated generation of comprehensive test cases and their corresponding Playwright test code from high-level use case descriptions. To achieve this, we first present the key requirements that GenE2E must fulfill, along with an overview of how these requirements are addressed by the proposed solution. Then we describe GenE2E’s architecture, illustrating how its components interact to support the automated generation process.

4.1 GenE2E requirements and implementation overview

GenE2E was designed to meet the following key requirements:

- *Automated test generation from use cases:* The system shall generate both test cases and executable JavaScript code compatible with the Playwright framework, directly from textual descriptions of use case scenarios. *Limited human intervention:* The system shall reduce manual effort by automating the main test-generation steps, while retaining human oversight for feasibility checks, validation, and correction of generated artifacts when needed.
- *Modular architecture:* The system shall adopt a modular architecture that enables independent operation of components for different use cases and scenarios.
- *Open-source and on-premises solutions:* The system shall support open-source and on-premises solutions for the main building blocks of the framework, deployed on local server infrastructure. This requirement addresses privacy and security concerns, provides fine-grained control over the generation process, ensures reliable data persistence, and reduces the recurring monetary costs associated with cloud-based LLMs, especially since continuous test generation is a token-intensive activity.

⁴ <https://gitlab.com/fmoretti/pell2.0-automation-testing>

- *Integration with a pre-established testing toolchain:* The generated test scripts are to be structured to operate within a pre-established testing environment. This environment leverages the Playwright tool for test execution and Python-based utility modules for common functionalities such as test reporting and data persistence. The test scripts are designed to invoke functions from these utility modules, ensuring seamless interaction with the framework’s reporting and storage mechanisms.

To meet all requirements listed above, we built GenE2E on a preexisting testing framework, wherein test cases were manually generated from requirements expressed as use case scenarios, and subsequently implemented in Playwright. This previous framework was built upon the conventional structure of a Playwright project, enhanced with environment variable support for parameter configuration and a utility component (called *TestResultReporter*) for common functions (e.g., database persistence of all test execution steps). We further enhanced the previous testing framework by introducing, at the initial stage of the pipeline, an additional module (the GenE2E framework) powered by LLMs to automate the test generation and coding process. To this purpose, the GenE2E architecture was designed as a modular pipeline comprising two independent LLM-augmented sub-systems that can operate both sequentially and individually, providing flexibility in deployment and usage scenarios.

We also considered and evaluated different design alternatives according to the following dimensions: *execution mode*, single or batch processing (explained later in the paper); *type of LLM*; and *prompting technique* (zero-shot, one-shot, and few-shot). The primary development phases encompassed comprehensive system design and prompt engineering optimization to maximize model performance.

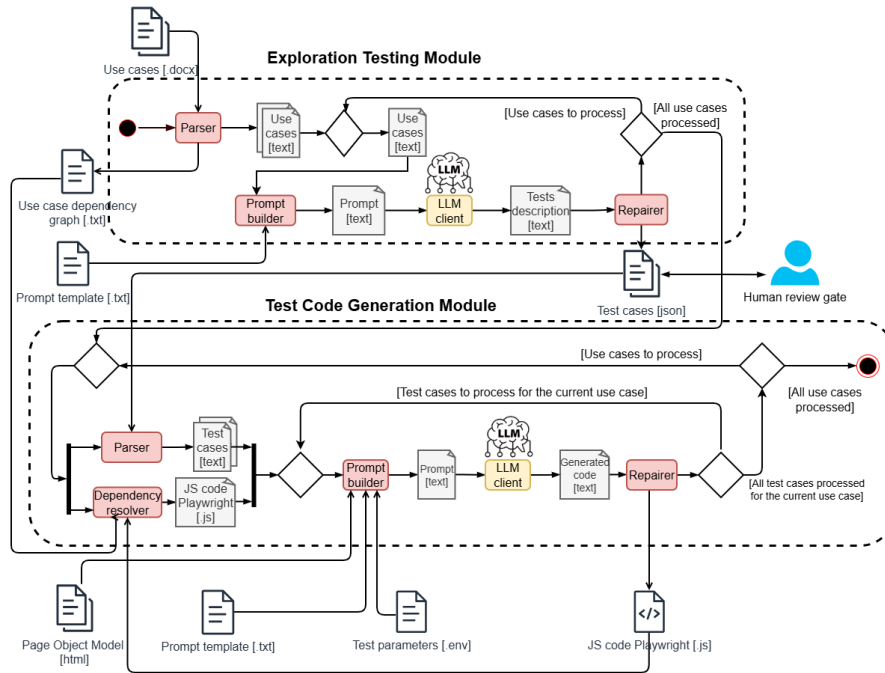
Note that the primary objective of this work was on “building the system that integrates LLMs” rather than to “benchmark or compare the performance of a broader range of LLMs available on the market” for the scope. For this reason, only two models, Meta Llama 3.3 70B [9] and CodeLlama [26] 70B (a foundational model for code generation tasks, built on top of Llama 2 [28]), were employed during the first development of GenE2E, while a comparison with other LLMs was postponed as future work. The selection of the LLM was guided by the need for open-source foundation models that could be executed locally, ensuring full accessibility without reliance on proprietary cloud services. While this constraint limits the use of other models available remotely, it offers key benefits such as reduced operational costs, improved data privacy, and independence from network latency.

Moreover, these two specific models were selected for conducting a comparative performance analysis between a general-purpose language model (Llama 3.3) and a fine-tuned one from the same family for software development tasks (CodeLlama). The goal of such a comparison was that of obtaining insights about the effectiveness of specialized versus generalized foundational models for our scope of automating test code generation from use case scenarios. The most significant difference between these two models is the context window dimension (131072 of Llama 3.3 against 2048 of CodeLlama). This difference, as will be shown in the evaluation results, considerably influence on the quality and comprehensiveness of generated test code, particularly when dealing with complex test scenarios that require extensive contextual information with test case dependencies.

GenE2E was implemented as a CLI tool in Python, designed for easy integration into development workflows and CI pipelines. It was developed upon the Ollama framework⁵ to manage and run local language models efficiently on local hardware.

⁵ ollama.com/

4.2 GenE2E framework



■ **Figure 2** GenE2E workflow (single processing mode).

The framework follows a pipeline architecture composed of two main stages: the *Exploration Testing Module*, which generates test cases from requirements, and the *Test Code Generation Module*, which converts these test cases into executable Playwright scripts.

To balance quality and efficiency, the framework supports two execution modes: *single* and *batch*. In single processing mode (Fig. 2), each use case and test case is processed individually, allowing the LLM to focus on a limited context and typically producing higher-quality outputs, at the cost of longer execution time. In contrast, batch processing reduces the number of LLM invocations by generating multiple test cases or scripts in a single call, improving throughput but potentially degrading output quality due to increased prompt complexity and context dilution.

4.2.1 Exploration Testing Module

The first module automatically derives test cases from textual use case descriptions stored in *.docx* documents. A *Parser* extracts the use case information and converts it into a structured JSON representation, while also constructing a *dependency graph* (a DAG capturing use case relationships such as *include*, *extend*, and *generalization*). This graph is later used to manage dependencies across test cases.

Each use case is then processed by a *Prompt Builder*, which embeds its description into a predefined prompt template and sends it to the *LLM Client*. The latter invokes Llama 3.3 (temperature 0.3), producing a structured JSON output containing multiple test cases. Each test case includes identifiers, descriptions, execution steps with expected outcomes, pre/postconditions, classification (positive/negative), and priority. Table 1 reports a universally known scenario for the *Login* use case, while Listing 1 shows a natural-language test case generated from its base flow by the *Exploration Testing Module*.

■ **Table 1** Use Case Scenario: User Login.

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

To ensure robustness, a *Repairer* component automatically validates and cleans the generated JSON file by enforcing JSON well-formedness. The resulting test cases are then persisted and passed through a lightweight human review gate, allowing practitioners to filter infeasible, underspecified, or non-reproducible outputs before code generation.

4.2.2 Test Code Generation Module

The second module transforms test cases into executable test scripts. A *Parser* reformats the JSON test cases into a textual representation suitable for prompting. A *Dependency Resolver* uses the previously built dependency graph to retrieve relevant code fragments from dependent use cases, focusing on “happy path” scenarios, which are then reused to improve consistency and reduce test code duplication.

The *Prompt Builder* plays a central role in this stage by assembling prompts enriched with multiple sources of contextual information. In particular, it injects knowledge of the web application structure through the Page Object Model (POM) API, ensuring that only

■ **Listing 1** Example of generated natural-language 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"
}
```

relevant page objects are included to limit token usage and avoid context dilution. Additional information, such as input constraints embedded in POM methods and environment variables from configuration files, is also incorporated to guide the model toward generating valid and executable code.

Dependency handling is explicitly supported by incorporating previously generated code into the prompt when needed. This enables reuse of existing functionality, although its effectiveness depends on proper code modularization; poorly structured code may lead to duplication or inconsistencies, especially in the presence of long dependency chains.

Prompt design follows the TELeR taxonomy [27], using a fixed configuration (*Turn=Single*, *Expression=Instruction*, *Role=defined*, *Level=5*) to ensure consistency. The framework supports zero-, one-, and few-shot prompting; in the latter case, the number of examples is limited to two, based on evidence of diminishing returns beyond this threshold [5] and to control token consumption. Prompt templates and prompt examples are available in the online replication package⁶, together with data and code artifacts.

The *LLM Client* (temperature 0.0) generates deterministic JavaScript code for Playwright tests, along with execution statistics such as runtime and token usage. Finally, a *Repair* component extracts the generated code from the model output, fixes import statements, and adapts it to the target testing project structure, ensuring compliance with the testing team’s Playwright best practices before saving the final artifacts.

5 Evaluation design

This section presents the experimental design we followed to evaluate the GenE2E framework (Section 5.1) and the approach we followed to analyze the experimental results (Section 5.2).

We considered the following research questions. RQ1 (Test Generation Feasibility): Can GenE2E generate correct executable test cases? We examine whether (Stage 2) of GenE2E can produce test cases that are both syntactically well-formed and semantically valid with respect to the intended testing scenarios. RQ2 (Alignment): To what extent do the test cases generated by GenE2E align with the reference tests manually defined by engineers? We investigate whether GenE2E can generate test cases that comply with the coding style, structural conventions, and implementation constraints observed in human-written tests (considered the *gold standard*), thereby approximating the practices of human testers. RQ3 (Effectiveness) What is the effectiveness of the tests generated by GenE2E compared to the baseline test suite? RQ4 (Efficiency): What is the cost of executing the GenE2E?

5.1 Methodology

Following the implementation of the GenE2E framework, we designed our experiments to systematically assess the system’s performance in automatically generating test cases and producing Playwright-compatible test code that accurately reflects the specified test scenarios. Due to space constraints, we focus the main evaluation on the *Test Code Generation Module*, while reporting only summary data for the *Exploration Testing Module*. In the evaluation, we refer to existing E2E tests – manually scripted Playwright tests for test cases derived from UML use case scenarios – as our *baseline*. Table 2 provides a comprehensive overview of a suite of 18 test cases that constitute this baseline.

⁶ Software artifacts and supplementary material are available in the replication package, which also includes an appendix with key examples and details: <https://zenodo.org/records/21629002>

■ **Table 2** List of all test cases that are part of the baseline (translated from Italian).

Use case	Test case	Description
1	1	User login with correct credentials
1	2	User login with incorrect credentials
2.1	1	Download PDF
2.2	1	Search lighting systems
2.3	1	View general data table
2.4	1	KPI benchmarking by municipality
3.1	1	View census records
3.2.1	1	Search census record by name
3.3	1	Upload census record
3.4	1	Action on census record
3.4.1	1	Download census record
3.4.2	1	Delete census record
3.4.2	2	Delete census record with operation cancel
3.4.3	1	Edit census record
3.4.4	1	Freeze census record
3.4.5	1	Census record details
5	1	Select language
6	1	User logout

■ **Table 3** Experiment configurations.

Configuration	Processing	Prompt technique	Model
1	Single	Zero-shot	Llama3.3
2	Single	Zero-shot	CodeLlama
3	Single	One-shot	Llama3.3
4	Single	One-shot	CodeLlama
5	Single	Few-shot	Llama3.3
6	Single	Few-shot	CodeLlama
7	Batch	Zero-shot	Llama3.3
8	Batch	Zero-shot	CodeLlama
9	Batch	One-shot	Llama3.3
10	Batch	One-shot	CodeLlama
11	Batch	Few-shot	Llama3.3
12	Batch	Few-shot	CodeLlama

In our experiments, we compare 12 distinct configurations of the Test Code Generation Module that are detailed in Table 3. We compared two different processing approaches: *single* (where the test specifications and the code alternate in the LLM context) and *batched* (where first all test specifications are provided in the context and then all the code follows), to understand the impact of input format. We compared three different prompting schemes (depending on the number of test specifications and examples in the context): *zero-shot*, *one-shot* and *few-shot* (with two examples), to assess the impact of *in-context learning* [5]. We compare two different LLMs, namely Llama 3.3 [9] and CodeLlama [26], to understand the difference between a generic model and one specialized in code generation.

To evaluate the considered configurations, we resorted to the test cases extracted using the *Exploration Testing Module*. For the Exploration Testing Module, we executed three independent generation rounds for each of the 41 use cases and using three prompting strategies (Zero-Shot, One-Shot, and Few-Shot), resulting in 369 generated test-suite outputs.

These generated test specifications were manually post-processed and filtered to ensure that only test cases that are feasible and really reproducible are forwarded to the *Test Code Generation Module* LLM⁷. After this data curation step, we obtained a dataset of 95 natural-language test cases; each of the 12 configurations of the second LLM was used to generate the code for all test cases in this dataset. We resorted to greedy decoding to ensure a deterministic output generation. We do not exclude the evaluation of other decoding schemes, which is now out of the scope of this work.

We run our experiments on a workstation running Ubuntu server 24.04.2 and equipped with AMD Ryzen Threadripper PRO 7985WX 64-Cores CPU, 256 GB RAM and a GeForce RTX 4090 GPU. Concerning LLMs, we experimented with the pre-trained Llama 3.3 70B (Q4_K_M quantization, CPU offloading and 131072 tokens context) and Instruction tuned CodelLlama 70B (Q4_0 quantization and 2048 tokens context) as baseline.

5.2 Metrics

For the evaluation, we resorted to a quantitative approach. For evaluating the test generation feasibility (RQ1), we checked the syntactic validity of the generated test cases and ran them to assess their successful or unsuccessful execution (compared to the 18 baseline test cases). For code alignment (RQ2), we also analyzed the generated test cases statically, comparing their code syntactically and semantically to that of the 18 references. Additionally, we analyzed the effectiveness (RQ3) and the efficiency (RQ4) of the generated tests for all 95 test cases produced by the Exploration Test Module.

Test Generation Feasibility. For the test execution evaluation, we considered the following aspects: *compilation success*, *execution with no errors*, and *assertion validity*. Compilation success indicates whether the code compiles successfully without errors. Although JavaScript is an interpreted language, this refers to errors identifiable through static code analysis. Execution with no errors indicates whether the code executes successfully to completion without raising any runtime exceptions. Assertion validity determines whether the test case passes all embedded assertions, indicating functional correctness of the generated code against the specified requirements and expected behaviors.

Test Code Alignment. Given the test code generated for a test case and the corresponding baseline test code, we measured their similarity both syntactically and semantically. In both cases, we resorted to metrics and techniques coming from the NLP area.

We measured syntactic similarity using the BiLingual Evaluation Understanding (BLEU) [22] and CodeBLEU [25] metrics. Both metrics compute the similarity measuring the overlapping of tokens in the compared string sequences. BLEU was originally designed to evaluate the quality of machine translation output [11, Chapter 25], it provides a quantitative assessment of surface-level textual correspondence. We used the implementation available with the *NLTK* library [4]. CodeBLEU extends traditional BLEU specifically for source code evaluation by incorporating code-specific characteristics. We resorted to the CodeXGLUE implementation⁸, setting the target language to JavaScript. BLEU and CodeBLEU should be interpreted only as approximate indicators of syntactic alignment with the reference

⁷ Some test scenarios can not be implemented in Playwright as it does not support certain operations, especially those involving backend processes or server-side events, like triggering errors during a PDF download.

⁸ <https://github.com/k4black/codebleu>

implementation. Because multiple Playwright implementations may correctly operationalize the same test scenario, low overlap with a single reference does not necessarily indicate an incorrect test. We therefore complement these measures with compilation success, execution without runtime errors, and assertion validity.

Semantic similarity was computed resorting to more recent deep learning based approaches. We used an encoder language model (namely *jina-embeddings-v2-base-code* [10]) to generate embeddings of the compared pieces of code. These embeddings represent a piece of code in a semantically meaningful space. Computing the *cosine similarity* between a pair of embeddings gives us a measure of the semantic similarity between the two pieces of code. Before embedding the code we pre-processed it to remove extraneous elements including comments, import statements, multiple consecutive empty lines, and leading/trailing whitespace characters, ensuring that the similarity assessment focuses on the core functional content.

Effectiveness metrics. To understand the effectiveness of the GenE2E we consider two metrics: *test case coverage* and *test execution rate*. The former is the proportion of test scripts successfully generated and executed relative to the total number of feasible test cases created from the Exploration Testing Module, and the latter is defined as the number of test scripts that are successfully executed.

Efficiency measures. The analysis of efficiency is designed to inform us on the costs of the GenE2E framework. We considered two key factors for this analysis: *generation time* (i.e., the time required to generate the test code) and *number of processed tokens* (divided into number of prompt tokens and generated response tokens). For these two metrics, we report the average values and the total across all generated tests.

6 Evaluation results

Across the Stage 1 evaluation, the *Exploration Testing Module* produced 369 test-suite outputs, corresponding to 41 use cases, three prompting strategies, and three independent generation rounds. The average number of test cases per generated suite was 5.44 for Zero-Shot, 7.84 for One-Shot, and 5.09 for Few-Shot. On the first generation attempt, Zero-Shot and One-Shot produced syntactically valid JSON in all cases (123/123 each), whereas Few-Shot produced 92/123 syntactically valid outputs (74.8%). When both syntax and structure were considered, the fully valid outputs were 108/123 for Zero-Shot (87.8%), 119/123 for One-Shot (96.7%), and 86/123 for Few-Shot (69.9%). JSON repair was automatic and implemented by the **Repairer** compiler; feasibility filtering, instead, was performed through human review before forwarding test cases to Stage 2.

In the following subsections, we report and comment on the results coming from the quantitative analysis of the Stage 2. In general, we noticed that the limited context window of the CodeLlama model made the generation of code almost impossible, subsequently impacting all metrics. Moreover, the presence of highly restrictive safeguards implemented in CodeLlama required to alter slightly the prompt as mitigation to the inability of the model to generate test cases connected to the login for safety reasons.

6.1 Test Generation Feasibility – RQ1

Figure 3 reports test execution metrics, including the number of successfully generated tests (out of 18 baseline test cases).



■ **Figure 3** Analysis of execution metrics histograms.

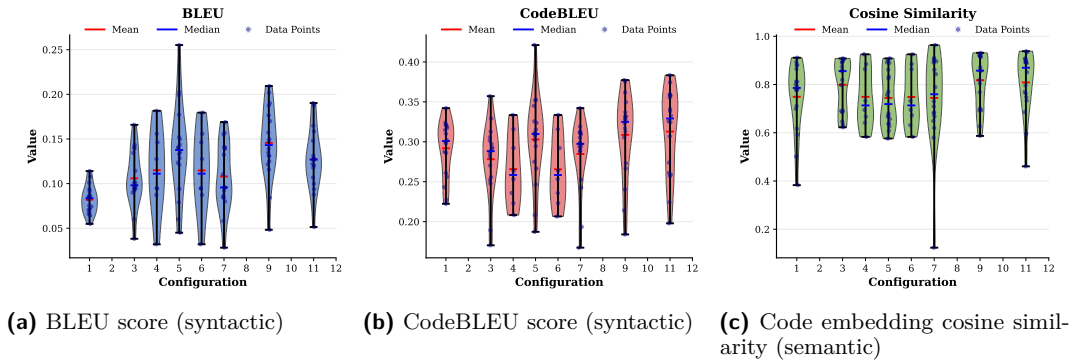
As anticipated, CodeLlama frequently fails to generate even a single test case. We obtained executable code for evaluation only in configurations 4 and 6. While all CodeLlama-generated tests compile successfully, none execute without runtime errors.

Llama 3.3 performs better, successfully generating code across all its configurations. In single processing mode, zero-shot and few-shot prompting each produce five tests that pass all assertions. The best overall result is obtained by batch processing with few-shot prompting, with seven tests passing all assertions.

When interpreting assertion failures, it is important to account for differences in assertion granularity. Although the generated and manually written baseline tests originate from the same use cases, the baseline tests generally validate fewer, higher-level outcomes, whereas GenE2E may generate an assertion for the expected result of each test step. Consequently, some generated tests correctly execute the intended interaction flow but fail on an additional requirement-derived assertion that is not present in the corresponding baseline test and is not satisfied by the current portal implementation. For example, the login use case specifies the display of a successful-operation message, whereas the corresponding baseline test checks successful authentication without asserting the presence of that message. We retained these outcomes as assertion failures in the reported metrics; however, they do not necessarily indicate errors in the generated interaction logic. In these cases, the generated tests are therefore more closely aligned with the documented specification than the corresponding baseline tests.

Further manual analysis identified inadequate details in the initial use case specifications as the primary cause of runtime errors in generated tests. This ambiguity propagates to the test cases created by the Exploration Testing Module, preventing the LLM from inferring the correct implementation. A secondary error source involves input parameters: LLMs rely exclusively on parameters provided through environment variables, whereas some required variables were hard-coded in baseline source code rather than being exposed through the environment.

Concerning compilation errors, we identified significant challenges in dependency management, particularly for tests with extensive dependency chains spanning multiple sources. The *Repair* component of the GenE2E framework is limited to correcting dependency paths for folder structure consistency but cannot insert missing import statements. While static analysis could detect missing dependencies, automatically determining the appropriate source files for importing specific methods remains challenging. Another prevalent issue in batch processing is the invocation of non-existent POM methods. We attribute this to context dilution that occurs when processing large amounts of information simultaneously. As



■ **Figure 4** Analysis of code syntactic and semantic similarity.

documented in the literature, model parameter count significantly influences in-context recall performance [17]. Consequently, employing larger models could potentially mitigate these issues.

Finally, errors coming from the use of deprecated Playwright methods are likely attributable to the model’s training data cutoff, which presumably references outdated library versions.

6.2 Test code alignment – RQ2

Figure 4 reports the code similarity scores. Overall, we observe that similarity scores increase with the number of shots, meaning that using few-shot learning is not only helping the LLM to solve the task, but also capture the expected style presented in the examples.

From a syntactic perspective, both BLEU and CodeBLEU scores exhibit consistent monotonic improvement across configurations, with configuration 11 being the exception. In that case, CodeBLEU more accurately captures the actual overlap between generated and reference code compared to BLEU. The progressively higher scores suggest that providing multiple examples enables the LLM to better reproduce patterns found in the reference samples. These scores indicate closer alignment with one particular implementation, however, and should not be interpreted as direct evidence of greater functional correctness. This effect is particularly pronounced in batch processing configurations.

From a semantic perspective, we observe analogous trends when applying batch processing. In batch processing, semantic similarity plateaus after incorporating a single example, mirroring the behavior observed for syntactic similarity. Conversely, in single processing mode, increasing the number of examples degrades semantic similarity performance.

6.3 Effectiveness – RQ3

The first part of Table 4 presents test effectiveness metrics. CodeLlama fails completely on this task, while Llama 3.3 demonstrates consistent capability across all prompting techniques.

Single processing achieves better test case coverage compared to batch processing, suggesting that focusing on the most recent information helps the LLM maintain attention on the current test case. Conversely, batch processing may introduce additional complexity through the alignment step between inputs and outputs.

One-shot learning procedures the worst results in terms of both test case coverage (39%) and test execution rate (11%). Zero-shot and few-shot learning yield comparable results, with zero-shot showing marginally better test case coverage. This suggests that excessive

■ **Table 4** Effectiveness and efficiency results (efficiency values of batch processing configurations are scaled on the batch size).

Conf.	No. test cases	Test effect. [%]		Time [hh:mm:ss]		No. of tokens			
		Coverage	Exec. rate	Avg.	Tot.	Prompt		Response	
						Avg.	Tot.	Avg.	Tot.
1	95	61	28	03:26	05:26:10	2240	212759	1858	176469
2	-	0	0	-	-	-	-	-	-
3	95	39	11	05:03	07:59:45	4016	381512	3766	357778
4	40	0	0	04:09	02:46:00	3059	122364	2897	115872
5	95	56	28	05:01	07:56:35	4063	385985	3811	362070
6	40	0	0	03:47	02:31:20	2980	119200	2818	112728
7	95	0	0	02:30	03:58:30	583	55396	644	61154
8	-	0	0	-	-	-	-	-	-
9	95	33	33	02:33	04:02:30	673	63894	633	60101
10	-	0	0	-	-	-	-	-	-
11	95	44	39	01:34	02:28:40	663	63025	649	61659
12	-	0	0	-	-	-	-	-	-
Basel.			55						

examples may constrain the model’s generative flexibility or introduce bias toward specific testing patterns that fail to generalize across diverse scenarios. In batch processing, the absence of examples in zero-shot led to critical errors that made the tests not runnable. When compared to the 55% baseline of manual tests, most configurations fall short in coverage, but configuration 11’s (batch processing, few shot learning and Llama 3.3) execution rate approaches a reasonable threshold.

6.4 Efficiency – RQ4

Results on efficiency (LLM processing time and processed tokens) are reported in the second part of Table 4. We reported the scores on the generated test cases out of the available 95 specifications. Single scores refer to the average over all the generated tests. The reported generation times should be interpreted as measurements of the evaluated configurations in our specific execution environment, rather than as general efficiency estimates for GenE2E or the considered model families. In particular, the 70B models did not fit entirely within the 24 GB GPU memory available on the experimental machine, requiring CPU offloading and making the observed runtimes dependent on the hardware configuration. Each configuration was executed once on a shared workstation. Therefore, although greedy decoding and a temperature of 0 make the generated outputs reproducible under the same setup, the timing measurements may be affected by transient system load and were not collected with the repetitions required for a statistically robust performance analysis. Thus, we use generation time only to describe the relative behavior observed in this experimental run, not to establish statistically supported performance differences among configurations.

The most relevant performance difference emerges between single and batch processing modes. In fact, batch processing turns out to have a better throughput as its prompt are shorter and so are the generated responses (if scaled on number of generated tests), thus requiring less inference time. The prompting technique itself does not substantially impact total execution time, with configuration 1 being an exception. Regarding model performance, CodeLlama exhibits reduced execution times compared to Llama 3.3. We attributed this to the slight architectural differences between the models and the weights quantization approaches.

Token count is primarily determined by the prompting technique. Specifically, transitioning from zero-shot to few-shot learning increases prompt size proportionally with the number of examples provided. This represents a critical constraint for models like CodeLlama with its 2048-token context limit.

Beyond the execution and effectiveness challenges, which we acknowledge as significant limitations for current LLMs, an important consideration for the proposed GenE2E framework concerns the time required for automated test code generation compared to manual implementation by human developers. The mean execution time of 2 minutes and 30 seconds observed in the optimal configuration represents a substantial efficiency gain over manual implementation. Human developers typically require variable amounts of time depending on test complexity, ranging from minutes for simple cases to several hours per test for complex scenarios [14, 15].

Regarding RQ4, batch processing reduced the number of model invocations and the associated token-processing overhead, indicating a potential throughput advantage. However, the wall-clock measurements are specific to a single run on our shared, hardware-constrained workstation. They should therefore be regarded as descriptive observations rather than general evidence of the efficiency of the configurations or models.

Summary. The evaluation showed that there is no optimal configuration. In our setup, Llama 3.3 was the only model that consistently generated outputs for all input requests. A plausible explanation is its much larger context window, which better accommodates the contextual information required by E2E test generation.

While not effective as a fully autonomous solution, GenE2E proved highly valuable as an assistive tool, reducing developer workload and generating more diverse functional tests, thereby achieving broader test coverage than the manual baseline.

7 Discussion

7.1 What configuration should a team use?

Our results do not identify a universally optimal configuration. Instead, they reveal a set of practical trade-offs that teams should consider when adopting local LLMs for end-to-end (E2E) test generation. The most appropriate configuration depends on whether the team prioritizes test coverage, execution reliability, generation throughput, or reduced post-processing effort. Table 5 summarizes the main practical recommendations.

First, we do not recommend using CodeLlama 70B in the current setup. Although CodeLlama is specialized for code generation, its limited context window made it unsuitable for complex E2E testing scenarios involving use case descriptions, POMs, dependencies, examples, and execution constraints. In our experiments, this limitation had a stronger negative impact than the potential benefits of code specialization.

Second, Llama 3.3 should be preferred as the default model. Despite being a general-purpose model, it consistently produced usable outputs across configurations. Its larger context window made it better suited to the amount of contextual information required for realistic E2E test generation.

Third, single processing with zero-shot or few-shot prompting is preferable when coverage is the main objective. Processing one test case at a time helps the model focus on the current scenario and reduces context dilution. This configuration is therefore more suitable when the goal is to generate a broader set of candidate tests, even at the cost of longer generation time.

■ **Table 5** Recommended GenE2E configurations for different practical goals.

Goal	Recommended configuration	Rationale
Use CodeLlama 70B in the current setup	Not recommended	Context window limitations outweigh code-specialization benefits.
Minimize manual repair	Llama 3.3, single processing	Lower risk of dependency and alignment errors.
Maximize generated test coverage	Llama 3.3, single processing, zero-shot or few-shot	Better focus on individual scenarios and less context dilution.
Improve throughput	Llama 3.3, batch processing, few-shot	Faster generation, but requires stronger post-processing.

Fourth, batch processing with few-shot prompting is preferable when throughput is the main objective and the team can tolerate additional post-processing. Batch processing reduces generation time and token consumption per test, but it also increases the risk of misalignment between inputs and outputs, missing dependencies, or calls to non-existing POM methods. For this reason, batch few-shot prompting is promising when teams need faster generation and have a review or repair step in place.

Thus, rather than selecting a single “best” configuration, practitioners should treat GenE2E as a configurable assistant. For teams prioritizing correctness and coverage, single processing with Llama 3.3 is the safest starting point. For teams prioritizing scalability and generation speed, batch few-shot prompting with Llama 3.3 can be effective, provided that a subsequent validation and repair phase is included.

7.2 Limitations and threats to validity

We discuss here the main limitations and threats to the validity of the proposed GenE2E approach, along with the mitigation strategies adopted. *Internal validity* may be affected by hallucinations introduced by LLMs. Specifically, LLMs may infer functional steps that are incorrect or do not exist or also add hard-code for client/environmental conditions creating assumptions purposefully. These potentially result in not executable test codes. To mitigate this threat, we configure the temperature to 0 to reduce randomness in generation. Moreover, instead of feeding all use case specifications to the LLM, we split and process each use case individually. This decomposition helps the LLM focus more precisely on the semantics of a specific use case scenario and related test cases, improving inference accuracy. In addition, we feed the LLM with a context model, including client/environmental conditions. In general, we enriched the prompt template trying to reflect the structure and content of a standard test code script from what we consider our golden test set⁹. Most of the cases where we obtained not runnable hard-code/template-code by the LLM are due to the fact that some conditions were missing in the context model.

External validity is primarily threatened by the quality of use case specifications. Use cases with incomplete or low quality scenario descriptions pose significant challenges for test case inference. To address this and make our approach more robust, the LLM-augmented generation pipeline includes a human-powered manual inspection task that examines each generated test case description (on average, 5/6 test cases per use case) to filter out (before feeding the LLM for test code generation) test case descriptions that look like under-specified or untestable or do not cover the expected behavior.

⁹ A predetermined collection of expert-curated test case scripts.

Another issue is that our study is restricted to test only one web application, the ENEA PELL-IP web portal. However, this web portal encompasses most mainstream features of common web applications and complex and deep interaction workflows typical of smart city platforms. So we are confident that the proposed E2E testing approach is generalizable and applicable to other web applications.

Construct validity is also affected by the use of BLEU and CodeBLEU against a single reference implementation: these metrics capture alignment with that implementation but cannot account for the full set of functionally equivalent Playwright tests. We mitigate this limitation by jointly considering compilation, runtime execution, and assertion validity.

7.3 Actionable Lessons and Hypotheses for Practitioners

Based on our experience with GenE2E, we derive the following practical lessons for teams aiming to introduce local LLMs into E2E test generation.

Lesson 1: Specification quality dominates test quality. *Context.* GenE2E relies on use case scenarios and contextual artifacts as primary inputs. *Evidence.* Many runtime failures stemmed from incomplete or ambiguous specifications, propagating errors to generated test cases. *Recommendation.* Teams should make specifications “LLM-ready” by clearly defining preconditions, expected outcomes, input constraints, alternative/exception flows, and test data. Priority should be given to high-quality use cases and well-documented POMs, as LLMs cannot compensate for missing context or unclear requirements.

Lesson 2: Context capacity may be critical for E2E test generation. *Context.* E2E test generation requires handling heterogeneous contextual inputs, including use cases, POMs, dependencies, constraints, examples, and generated code. *Evidence.* Llama 3.3 outperformed CodeLlama in our experiments; however, this comparison is confounded because the models differ not only in code specialization, but also in model generation, training data, quantization, and especially context-window size. Our results do not isolate whether the observed gap is due to context capacity or to other model differences. *Implication.* We therefore report this as a tentative hypothesis rather than a definitive recommendation. Future work should compare recent code-specialized local models with larger context windows to assess the relative impact of context capacity and code specialization.

Lesson 3: Batch generation improves throughput but increases fragility. *Context.* Batch mode reduces LLM calls and improves efficiency. *Evidence.* It also increased issues such as context dilution, misalignment, missing dependencies, and invalid POM calls. *Recommendation.* Use batch generation only when throughput is critical and downstream validation or repair mechanisms are available.

Lesson 4: A small human gate is still necessary. *Context.* Some generated test scenarios may be infeasible or not executable via the UI. *Evidence.* Filtering test cases before code generation prevented invalid scenarios from propagating. *Recommendation.* Introduce a lightweight human review step focused on feasibility, testability, and coverage.

Lesson 5: Manual baselines are not always perfect ground truth. *Context.* Manual tests are typically used as evaluation baselines. *Evidence.* Generated tests sometimes captured requirements missing from manual ones. *Recommendation.* Treat manual tests as reference points rather than absolute oracles, leveraging LLM outputs to identify gaps in existing test suites.

8 Conclusion and future directions

This paper presents GenE2E, a LLM-enhanced framework to support the generation of end-to-end test suites and executable test scripts from use case specifications for web applications.

The main goal of this work was to assess the feasibility, rather than the operational maturity, of introducing self-hosted LLMs into an existing E2E testing process and to evaluate their effectiveness in a realistic industrial setting. In particular, we compared two models from the same family, Llama 3.3 and CodeLlama, as an initial feasibility baseline. However, this comparison cannot be interpreted as isolating the effect of general-purpose versus code-specialized capabilities, since the two models also differ substantially in context-window size, generation, training data, and quantization. The experimental evaluation on the ENEA PELL-IP web portal showed that local LLMs can effectively support test generation, although their role remains primarily assistive rather than fully autonomous. The evaluation explored 12 configurations varying processing strategies, prompting techniques, and model choice. Overall, the evidence supports the feasibility of GenE2E as an assistive test-development tool, but not yet its use as an autonomous replacement for human test engineers.

As future work, we plan to extend this study in several directions. First, we aim to replicate the evaluation on additional web applications to strengthen the validity and generalizability of the findings. Second, we intend to deepen the investigation of local LLM variants, including recent code-specialized models with larger context windows, such as Qwen2.5-Coder and similar alternatives, to disentangle the effects of context capacity and code specialization on E2E test generation performance. Third, we plan to complement the current quantitative evaluation with a qualitative assessment of the generated test scripts, adopting both an expert assessment and an LLM-based evaluation approach (LLM-as-a-judge, as proxy of expert evaluation). Finally, we aim to further automate the framework by integrating mechanisms for the automatic retrieval of contextual information (e.g., POMs and UI structure), potentially leveraging LLM-based agents to reduce manual configuration effort and improve scalability. This is a promising research avenue, as proposed in [20].

References

- 1 Portale pell. <https://www.pell.enea.it/>.
- 2 Mubashir Ali, Patrizia Scandurra, Fabio Moretti, and Laura Blaso. Architecting a big data-driven software architecture for smart street lighting. In *20th International Conference on Software Architecture, ICSA 2023 - Companion, L'Aquila, Italy, March 13-17, 2023*, pages 1–10. IEEE, 2023. doi:10.1109/ICSA-C57050.2023.00019.
- 3 Parsa Alian, Noor Nashid, Mobina Shahbandeh, Taha Shabani, and Ali Mesbah. Feature-driven end-to-end test generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering, ICSE '25*, pages 450–462. IEEE Press, 2025. doi:10.1109/ICSE55347.2025.00141.
- 4 Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly, 2009. URL: <http://www.oreilly.de/catalog/9780596516499/index.html>.
- 5 Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfc4967418bf b8ac142f64a-Abstract.html>.
- 6 Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. *ACM Trans. Intell. Syst. Technol.*, 15(3), March 2024. doi:10.1145/3641289.

- 7 Pathomthat Chiradeja and Suntiti Yoomak. Development of public lighting system with smart lighting control systems and internet of thing (IoT) technologies for smart city. *Energy Reports*, 10:3355–3372, 2023.
- 8 Margarida Ferreira, Luis Viegas, Joao Pascoal Faria, and Bruno Lima. Acceptance test generation with large language models: An industrial case study, 2025. doi:10.48550/arXiv.2504.07244.
- 9 A. Grattafiori et al. The Llama 3 Herd of Models. *CoRR*, abs/2407.21783, 2024. doi:10.48550/arXiv.2407.21783.
- 10 Michael Günther et al. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents, 2024. arXiv:2310.19923.
- 11 Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.
- 12 Elvis Júnior, Alan Valejo, Jorge Valverde-Rebaza, and Vânia Neves. Genia-e2etest: A generative ai-based approach for end-to-end test automation. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 282–292, Porto Alegre, RS, Brasil, 2025. SBC. doi:10.5753/SBES.2025.9927.
- 13 M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Chapter five - approaches and tools for automated end-to-end web testing. In Atif Memon, editor, *Advances in Computers*, volume 101, pages 193–237. Elsevier, 2016. doi:10.1016/bs.adcom.2015.11.007.
- 14 M. Leotta, H. Zeeshan Yousaf, F. Ricca, and B. Garcia. AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study. In *EASE '24: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 339–344. ACM, 2024. doi:10.1145/3661167.3661192.
- 15 Maurizio Leotta, Filippo Ricca, Alessandro Marchetto, and Dario Olianias. An empirical study to compare three web test automation approaches: Nlp-based, programmable, and capture&replay. *Journal of Software: Evolution and Process*, 36, August 2023. doi:10.1002/SMR.2606.
- 16 Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Make LLM a Testing Expert: Bringing Human-Like Interaction to Mobile GUI Testing via Functionality-Aware Decisions. In *2024 IEEE/ACM 46th Int. Conf. on Software Engineering (ICSE)*, pages 1222–1234, Los Alamitos, CA, USA, April 2024. IEEE.
- 17 Daniel Machlab and Rick Battle. LLM In-Context Recall is Prompt Dependent, 2024. doi:10.48550/arXiv.2404.08865.
- 18 Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2025. doi:10.48550/arXiv.2402.06196.
- 19 Fabio Moretti, Simone Ronzoni, Patrizia Scandurra, and Vincenzo Scotti. Replication package for the paper “Local LLMs for End-to-End Testing in Practice: Lessons from a Smart City Web Application”, 2026. Computer Software. doi:10.5281/zenodo.21629002.
- 20 Takuro Morimoto and Harumi Haraguchi. A study on the improvement of code generation quality using large language models leveraging product documentation, 2025. doi:10.48550/arXiv.2503.17837.
- 21 Andrzej Ożadowicz and Jakub Grela. Energy saving in the street lighting control system – A new approach based on the en-15232 standard. *Energy Efficiency*, 10(3):563–576, 2017.
- 22 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL, 2002. doi:10.3115/1073083.1073135.
- 23 Francesc Pardo-Bosch, Ana Blanco, Eukén Sesé, Félix Ezcurra, and Pablo Pujadas. Sustainable strategy for the implementation of energy efficient smart public lighting in urban areas: case study in san sebastian. *Sustainable Cities and Society*, 76:103454, 2022.

- 24 Alexander Poth, Olsi Rrjolli, Huiyu Wang, and Klaus Schmid. Baseline evaluation of llm-facilitated ui test-case generation from gherkin specifications. In Murat Yilmaz, Paul Clarke, Andreas Riel, Richard Messnarz, Mikus Zelmenis, and Ivi Anna Buce, editors, *Systems, Software and Services Process Improvement*, pages 39–53, Cham, 2026. Springer Nature Switzerland.
- 25 Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. [arXiv:2009.10297](#).
- 26 Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, and other. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. [doi:10.48550/arXiv.2308.12950](#).
- 27 Shubhra Kanti Karmaker Santu and Dongji Feng. Teler: A general taxonomy of LLM prompts for benchmarking complex tasks. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 14197–14203. Association for Computational Linguistics, 2023. [doi:10.18653/V1/2023.FINDINGS-EMNLP.946](#).
- 28 Hugo Touvron, Louis Martin, Kevin Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. [doi:10.48550/arXiv.2307.09288](#).
- 29 Colin Van Hooren, Fernando Pastor Ricós, Stefano Bromuri, Tanja E. J. Vos, and Beatriz Marín. LLM-Empowered Scriptless Functional Testing. In *2025 25th International Conference on Software Quality, Reliability and Security (QRS)*, pages 1–12, 2025. [doi:10.1109/QRS65678.2025.00012](#).
- 30 Zhitao Wang, Wei Wang, Zirao Li, Long Wang, Can Yi, Xinjie Xu, Luyang Cao, Hanjing Su, Shouzhi Chen, and Jun Zhou. Xuat-copilot: Multi-agent collaborative system for automated user acceptance testing with large language model, 2024. [doi:10.48550/arXiv.2401.02705](#).