26 Mar 2024

Large Language Model Driven Automated Software Application Testing

Fei Wang

Kamakshi Kodur

Michael Micheletti

Shu-Wei Cheng

Yogalakshmi Sadasivam

See next page for additional authors

Inventor(s)
Fei Wang, Kamakshi Kodur, Michael Micheletti, Shu-Wei Cheng, Yogalakshmi Sadasivam, Yue Hu, and Zening Li

Large Language Model Driven Automated Software Application Testing

ABSTRACT

Testing, including user interface testing and A/B testing is an important part of application development. It can be difficult for human testers to write appropriate test assertions or checks for user interface tests. Human testing is costly and not scalable. This disclosure describes the use of a large language model (LLM) to automate testing for software applications, including UI tests, A/B tests, and simulating manual QA tests. A dataset of prior known errors in the UI rendering is constructed and labels are associated with different types of errors. Prompt engineering is carried out to develop a library of prompts that, when provided to an LLM along with UI screenshots, cause the LLM to return responses that indicate whether the UI has errors and details about the error. UI screenshots are obtained by executing test cases for a codebase under test, e.g., using a mobile device simulator, and are analyzed using the LLM with appropriate prompts. The LLM responses include assertions that are integrated into a testing framework. A/B testing is carried out by providing the LLM UI screenshots or results from both control and experiment arms along with appropriate prompts. Automated LLM-driven testing is scalable, can reduce costs, expand test coverage, and reduce errors in production.

KEYWORDS

- LLM-based software testing
- User interface testing
- UI testing
- Test assertion
- Test coverage
- A/B testing
- QA testing
- Screenshot comparison
- Mobile device simulator
- Bug report

BACKGROUND

Testing, including user interface testing and A/B testing (comparing different versions) is an important part of application development. When the user interface of an application is not rendered correctly, the user experience suffers. The types of user interface rendering errors can vary and can be different in different applications.

For example, consider digital map applications that users rely on for a variety of purposes such as navigating to a location, orienting themselves in the physical world, seeking information on routes between locations, identifying points of interest near them, etc. Digital maps are typically split into regions that are rendered on the screen of the user's device. In some cases, one or more of the regions in the map region of the user's interest can fail to render accurately. The inaccurate display of a digital map can include problems such as a blank region, a blurry region, a partially rendered region, a region with inaccurate or missing labels, etc. Such cases can result from updates and refinements to the functionality of the digital map application and/or changes to the underlying map data.

In a digital map application, the large number of possible states of the visual rendered output makes creating exhaustive assertions via traditional techniques difficult-to-impossible, especially factoring in that map data evolves over time. For a given image of a rendered map, it may be feasible to assert the presence of all expected text. However, it is possible that a point of interest, e.g., a business such as a restaurant associated with a portion of the text is closed and no longer shown on the map, while the text remains. In certain cases, the text may have temporal characteristics, e.g., text indicating that a business closes soon. In another example, if the text size is increased, it may lead to certain text being rendered larger and getting truncated with "...". Other unpredictable effects may also be seen, e.g., a map region rendering in an inaccurate color.

There are some traditional techniques to at least partially address these. For example, it may be possible to generate comparison screenshots of a UI in test for comparison with a previously vetted version, both being based on the same static data. However, implementing such techniques requires manual maintenance and can be fragile over time. To detect rendering problems, application development teams rely on human testers in the quality assurance (QA) phase of product development. However, during pre-production testing, it can be difficult for human testers to write appropriate test assertions or checks to ensure correct rendering of regions and labels in the case of a digital map, and different types of errors for different applications. Moreover, human testing is not scalable. However, automatically determining whether a map has been rendered correctly is challenging.

Manual QA testing on mobile devices is also a time-consuming and repetitive task. Testers need to manually execute test cases, capture screenshots, and compare the captured screenshots to expected results to identify any discrepancies. This process can be tedious and error-prone, especially for complex applications (e.g., digital maps) with a large number of test cases.

DESCRIPTION

This disclosure describes techniques to automate the testing of the user interface of a software application to determine if the user interface is rendered correctly. The techniques are explained below with reference to the user interface of a digital map application.

User interface (UI) testing

Testing the user interface (UI) of a digital map application that utilizes map regions to render the map involves checking whether the map user interface, including map regions, are rendered correctly and without known defects that have been observed previously. An LLM can

be deployed for automated testing by providing the user interface under test (e.g., image of the UI) along with appropriate prompts. The appropriate prompts can be determined by compiling a database of problematic historical map UI renderings and consulting domain experts to identify and label commonly occurring rendering issues. Effective prompts are developed via prompt engineering. The prompts can be used to query the LLM to identify similar issues in the application under test, e.g., a production or pre-release version of a digital map application.

The LLM can be provided the prompt and a render of the user interface of the digital map, e.g., that includes a map region composed of regions as well as other UI elements such as navigation guidance, icons, buttons, text boxes, etc. The response from the LLM can accurately identify any rendering issues in the user interface. For instance, the prompt “Are map regions rendered accurately?” may produce an LLM response of “The map appears to be from a website zoomed into a specific location with no labels with a faint grid-like structure in the background that does not correspond to individual map regions.”

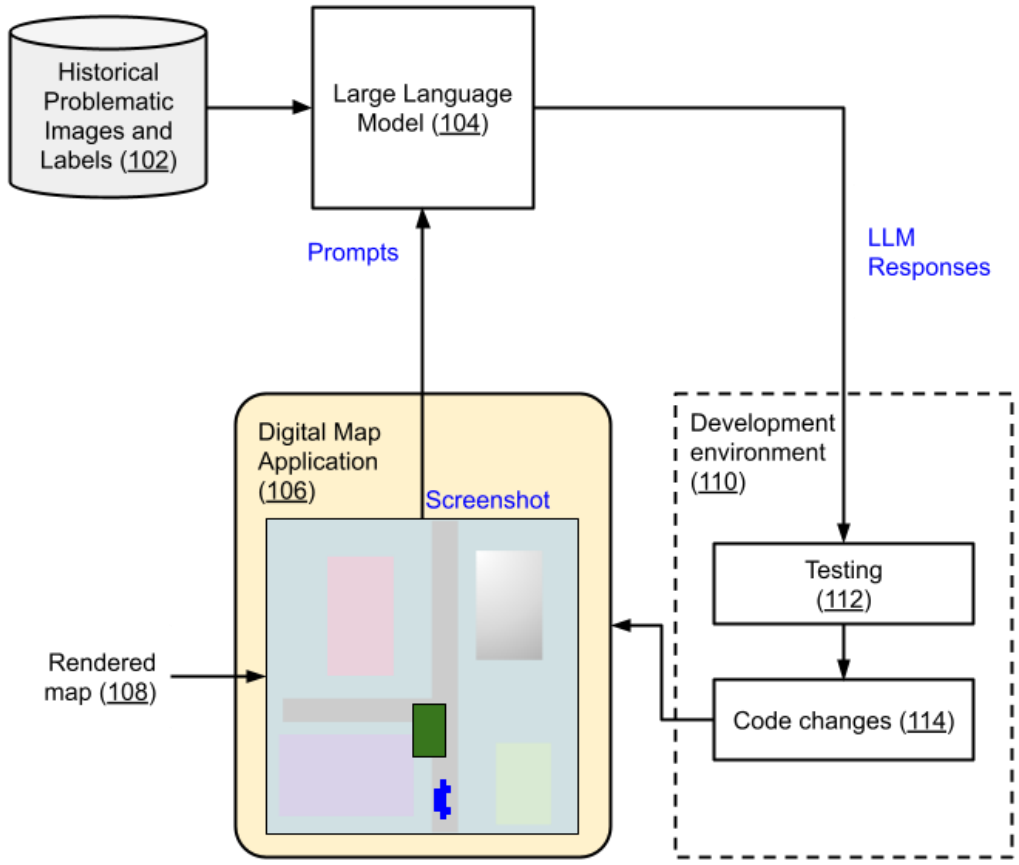

Fig. 1: Prompting an LLM to detect problems in digital map rendering

Fig. 1 shows an example operational implementation of the techniques described in this disclosure. A dataset (102) is gathered of historical images of the UI with rendering issues and corresponding labels. The images in the dataset are preprocessed to ensure consistency and compatibility with a suitable LLM (104). Prompt engineering includes prompting the LLM with known images and evaluating the responses to tweak the prompts such that the LLM responses reliably identify (and optionally, label) the known past problems captured in a dataset. The prompts can then be utilized to perform UI testing for the application.

The user interface (108) of a digital map application (106) under test is rendered. The UI can be rendered by executing the application in a test environment (110). Images of the user

interface (e.g., screenshots) in various states that are being tested are provided to the LLM along with prompts based on the prompt engineering. The prompts include a request to the LLM to produce responses that indicate if the UI has a rendering error.

The LLM generates responses that indicate whether the input images exhibit a problem, e.g., corresponding to previously labeled UI errors. The LLM generated responses can additionally include possible reasons for the detected issues. The LLM responses are incorporated in the testing processes (112) for the application. The testing framework includes mechanisms to generate test assertions or checks based on the LLM responses. If the LLM responses indicate UI errors, corresponding code sections (including UI assets) can be modified (114) to address the problems until problems are resolved.

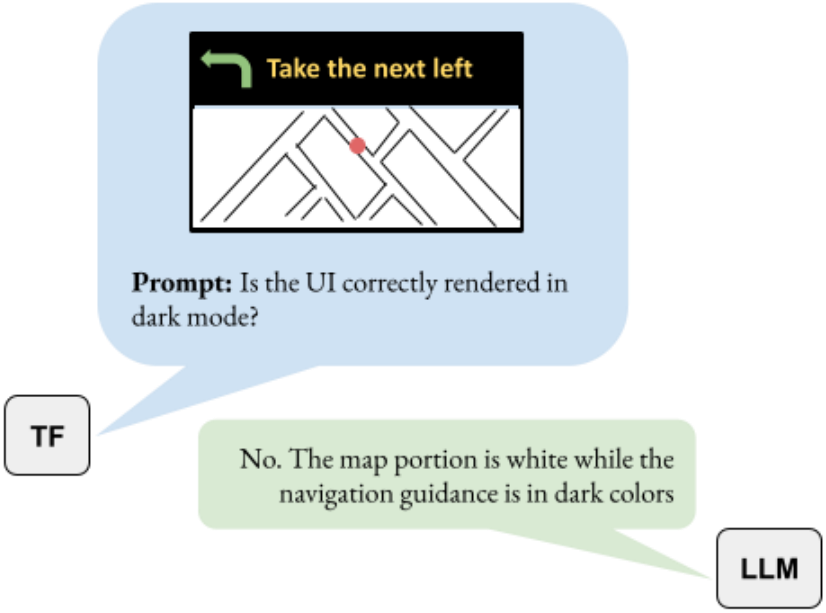

Fig. 2: LLM detects dark mode violation

Fig. 2 illustrates another example evaluation of a digital map UI using an LLM. In the prompt, a screenshot of the UI of the map in navigation mode is shown. As seen, the navigation guidance (“Take the next left” and the arrow) are rendered in dark colors, while the map itself is

rendered in white. The prompt to the LLM requires the LLM to return whether the app UI conforms to “dark mode” - a mode in which light colors are eliminated to provide a UI suitable for use at night. The LLM correctly returns that a portion of the UI - the map portion - is rendered in white, violating requirements of dark mode.

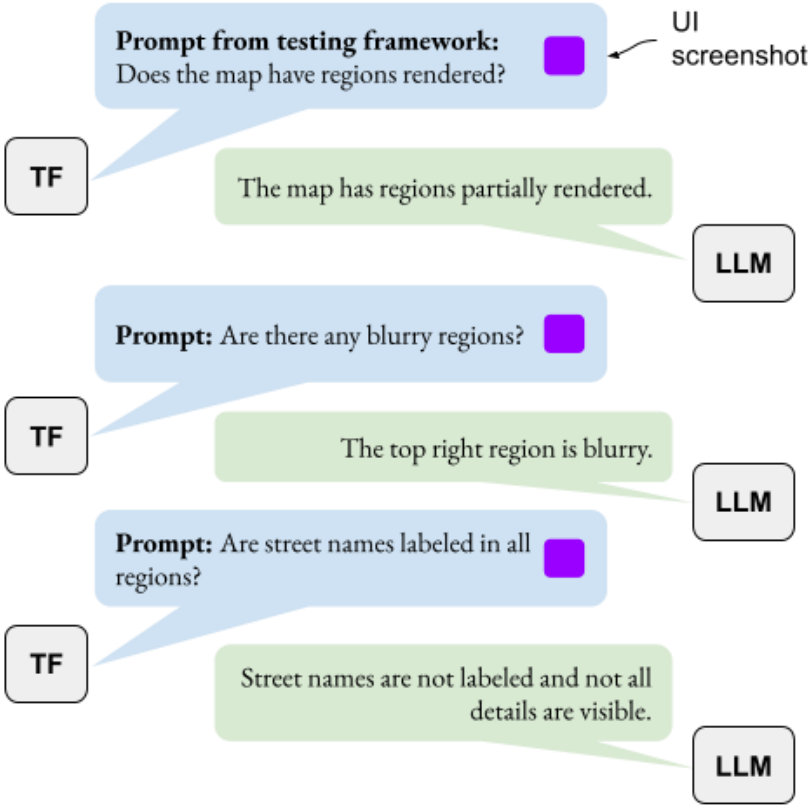

Fig. 3: Example prompts and LLM responses

Fig. 3 illustrates a set of prompts generated by the testing framework and provided to the LLM along with the rendered user interface of a digital map application. In each case, the LLM-generated response indicates whether the rendered UI has a problem, e.g., whether all the map regions have been rendered, whether any regions are blurry (and if so, which ones), whether street names are labeled in the regions.

The techniques described above can be employed to automate testing and QA processes for user interfaces. The LLM can be integrated into pre-production and/or production testing frameworks. The automated tests can then detect known types of rendering problems caused by updates to the application codebase and/or other assets (e.g., regions in a digital map database). The LLM performance is monitored, and necessary adjustments are made to the LLM (e.g., use an updated version of the LLM) and/or to the prompts provided to the LLM.

A/B testing

In some implementations, the LLM can be used in an A/B test framework to generate assertions to detect issues. For example, such a framework can detect unknown UI issues. By providing the LLM application UI screenshots or results from both A(control) and B(experiment) and proper prompt engineering, the LLM can detect issues more thoroughly and intelligently with lower maintenance costs than human crafted and maintained assertions. By combining A/B testing with LLM-powered assertions, the number of prompts can be significantly reduced by applying “Chain of Thought” in the interactions with the LLM. The chain of thought at a high level can be described as the following steps:

1. Providing the LLM the baseline screenshot from tests A and telling the LLM that this is the baseline
2. Providing the LLM with the experimental screenshot from tests B
3. Prompting the LLM to provide an answer that indicates whether there is significant difference between the screenshots
4. Parsing the response from the LLM in the test framework and marking the test as pass or fail

In the example of Fig. 3, this technique can be applied by supplying a different screenshot of an error-free “dark mode” UI to the LLM as the baseline and providing the screenshot in Fig. 3 as the experimental screenshot.

Simulating manual quality assurance (QA) testing

A large language model (LLM) can also be used to simulate manual QA testing on mobile simulators. Such an approach leverages the capabilities of LLMs to automate the process of executing test cases and performing visual assertions on screenshots, and can reduce the time and effort required for manual testing.

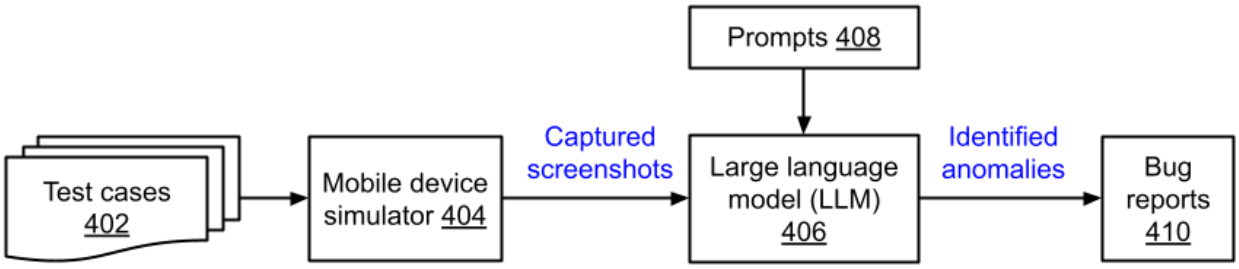

Fig. 4: Automating QA testing using an LLM

Fig. 4 illustrates an example process flow to use an LLM to automate executing test cases, performing visual assertions on screenshots, and reporting bugs. The technique leverages the ability of an LLM to understand natural language instructions, generate text, perform image recognition tasks, and assert on the visual questions posed to the LLM, e.g., via a prompt.

To deploy an LLM to simulate testing, input test cases (402) are provided in the form of QA testing documents or testing steps. Human written natural language test notes can be turned into test cases. A mobile device simulator (404) simulates user actions such as login, user interface actions such as tapping buttons or other user interface elements and navigating through

the app being tested. Screenshots of the app are captured at different steps of the test case execution. These can be used as input to an LLM (406)

The LLM executes the test cases and performs visual assertions on screenshots (e.g., as described above with reference to Fig. 1) based on prompts (408) and identifies anomalies. Bug reports (410) are generated at the end of testing, similar to those generated during manual QA testing.

The described techniques for user interface testing, A/B testing, and simulating manual QA testing can be implemented using any suitable LLM. The dataset to identify appropriate prompts to detect UI rendering problems can be obtained and pre-processed as necessary to ensure consistency and compatibility with the LLM. With proper prompt engineering, implementation of the techniques described above can accurately detect UI rendering issues that human testers find challenging to catch. LLM-based testing is a cost-effective solution, especially for production environments where the volume of tests can be substantial.

With A/B testing, differences in the behavior of different app versions (e.g., current production version and a version under test) can be intelligently detected without writing specific test cases. LLMs can generate a wide range of assertions that cover a variety of scenarios and edge cases, enabling more comprehensive testing and fewer issues in production. Since the LLM generates assertions automatically, the cost to create and maintain human-crafted assertions is reduced, saving time and resources.

By automating the process of test execution and visual assertions using an LLM, the testing time can be significantly reduced. This can enable the QA test phase for an application under test to be reduced significantly, thus improving feature end to end time between development to launch. With good prompts, LLMs can perform visual assertions with high

accuracy, reducing the risk of false positives and false negatives during the test process. The use of LLMs makes testing scalable such that a large number of test cases and different types of mobile devices can be tested. The techniques are flexible and can easily be adapted to different testing scenarios and requirements.

Implementation of the techniques can significantly reduce the need for the manual effort of writing assertions or checks to detect problems in software applications (including the user interface), thus substantially improving the speed, scale, efficiency, and effectiveness of testing and QA, and resulting in substantial cost savings.

CONCLUSION

This disclosure describes the use of a large language model (LLM) to automate testing for software applications, including UI tests, A/B tests, and simulating manual QA tests. A dataset of prior known errors in the UI rendering is constructed and labels are associated with different types of errors. Prompt engineering is carried out to develop a library of prompts that, when provided to an LLM along with UI screenshots, cause the LLM to return responses that indicate whether the UI has errors and details about the error. UI screenshots are obtained by executing test cases for a codebase under test, e.g., using a mobile device simulator, and are analyzed using the LLM with appropriate prompts. The LLM responses include assertions that are integrated into a testing framework. A/B testing is carried out by providing the LLM UI screenshots or results from both control and experiment arms along with appropriate prompts. Automated LLM-driven testing is scalable, can reduce costs, expand test coverage, and reduce errors in production.

REFERENCES

1. Pichai, Sundar and Dennis Hassabis “Our next-generation model: Gemini 1.5” available online at https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/, published Feb 15, 2024; accessed Mar 18, 2024.
2. Liu, Zhe, 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." *arXiv preprint arXiv:2310.15780* (2023).
3. Wang, Junjie, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. "Software testing with large language models: Survey, landscape, and vision." *IEEE Transactions on Software Engineering* (2024).