CS 5130 – EpicAutomator Report

Tucker Cook Jacob Greenberg Guillermo Rached

Abstract & Problem Description

As compute resources become cheaper and faster, software testing has become more accessible than ever. Despite its accessibility, software testing requires a huge time commitment to set up and maintain. Not only this, but test writing is often boring and seen more as a chore by developers. As such, there is a demand for systems which make test writing and maintenance easier.

Our project, *EpicAutomator* seeks to take the chore out of user interface-based testing. UI tests are often some of the most frustrating tests to maintain because they can be affected by seemingly innocuous changes and tend to be more fragile than other tests. One of the main reasons behind this frustration is because UIs are intuitive for humans but are extremely complex from the computer's perspective. We hope to bridge the gap by allowing developers to write tests in natural language and have them executed by an LLM. LLMs provide a powerful tool allowing developers to spend less time fiddling with tests and more time writing code.

Implementation Details

EpicAutomator is a Python program designed to be run as part of an automated testing pipeline. We went through great lengths to ensure the program is modular and easy to iterate upon. There are two main systems working in harmony to allow the program to work.

The first component is the LLM. Our system is intended to work with Ollama, a highly modular framework for running LLMs locally on a capable machine. We picked Ollama because it made it very easy to test different models and is one of the most commonly used programs for locally hosting LLMs. We felt that hosting the LLM locally, was critical since it allows for testing of private or in-development apps without concern of leaks. We also made sure that it was easy to modify the program to allow the use of API-based LLMs like ChatGPT or Gemini.

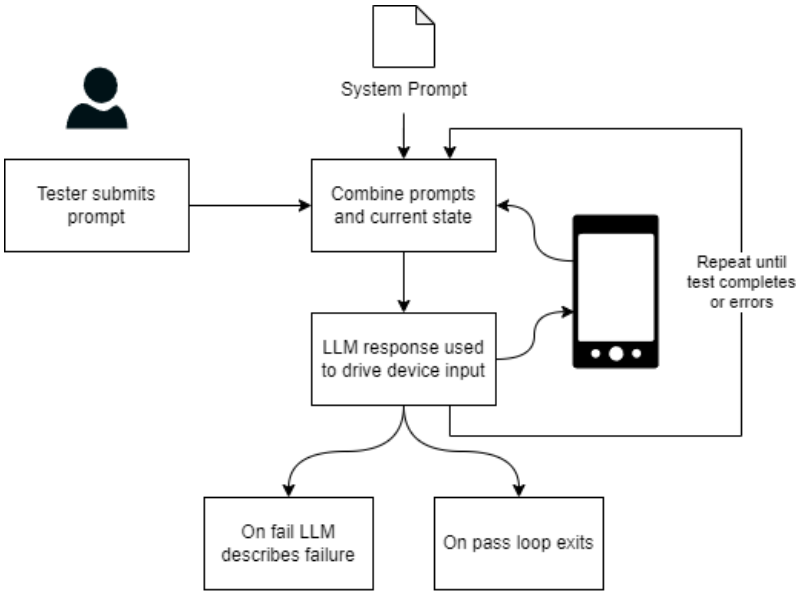
Regardless of the model, our program provides the LLM with three pieces of information. First, a system prompt, this prompt tells the LLM what commands it should use and how it should interact with the other systems. Second, the tester prompt, this prompt contains the LLM's objective. Objectives can be something simple like "open the settings app" or can be more complex like "navigate to Google, search for cute cat pictures, and share it with someone from your Contacts". Lastly, the LLM is provided with an XML based view hierarchy of the phone. Unlike a screenshot, the hierarchy provides information about the pixel location of buttons along with text

names. This allows the LLM to spend more time analyzing the view instead of trying to decipher a potentially incomplete transcription of the screen.

The second system is the Automator itself. The Python program works to connect the user's input, the LLM, and the device under test together. The Automator prompts the LLM with the initial screen state and uses the response to enter an input on the device via the Android DeBugger (ADB). This process is repeated ad nauseam until the LLM is satisfied and completes or believes it has encountered an error and reports it.

```
//sdcard//window_dump.xml: 1 file pulled, 0 skipped. 4.7 MB/s (39693 bytes in 0.008s)
==========
{"command":"text settings"}
==========
```

In this command the LLM types 'settings' via the phone's keyboard



Flow chart of the Automator system

There are two distinct advantages to using an LLM over traditional UI testing. First, LLMs are able to apply some logic, when they see an unexpected scenario, they are able to navigate around it instead of panicking like a traditional system. This means the developers don't need to update tests every time they make a change to the UI. When developers do need to update tests, they only need to update the natural language prompt instead of having to dig through the testing code.

The second, more subtle advantage deals with handling errors when they occur. The LLM is capable of detecting errors and reporting them, but unlike traditional UI testing systems which effectively say 'there's an error' the LLM is capable of describing the error it encountered in more detail. Additionally, since the LLM has to enter specific, discrete inputs there is a clear log of the inputs used allowing for much reproducibility in errors.

Results

In its current implementation, EpicAutomator is limited by the LLM used to power it. Many of the freely accessible Ollama LLMs do not have the capacity to interpret the large XML dump and often get confused when interacting with the device. More powerful, remotely accessed LLMs, like Gemini and ChatGPT are more capable of understanding the prompts and XML dump. However, due to the nature of UI testing it is difficult to gather empirical data about the system's performance.

Even though it is difficult to gather empirical data, we did make an effort to gather some control data which may be useful as we continue to build on this project. One of the comparisons we conducted was racing a human tester and EpicAutomator using different models.

Prompt	Human Time (s)	AI Time (s)
Open the settings app from the home screen (Ollama)	1.36	DNF
Open the settings app from the home screen (Ollama)	1.9	DNF
Open the settings app from the home screen (Ollama)	0.8	DNF
Search for 'cute cat pics' (Ollama)	6.29	6.69
Search for 'cute cat pics' (Ollama)	6.46	8.5
Search for 'cute cat pics' (Ollama)	5.99	9.16
Open YouTube application and search "Cat videos" (Gemini)	11.53	14.00
Open Messages application and click 'Start Chat' and type message to 'EpicAutomator' (Gemini)	9.64	21.31

This data set, while far from objective, hints at comparable abilities of the LLM to human tester. Even with large prompts and complicated system the LLM is able to remain in the realm of human testing in terms of speed. This allows the human tester to focus on more complicated test cases while EpicAutomator takes care of the simpler ones.

We also compared a number of models. Again, the data collected isn't complete or objective, but it does point towards more powerful, modern LLMs preforming better in the sorts of reasoning tasks we assigned to them.

Model	Failure Rate
Llama2-uncensored	100%
Llama3.2 (3 billion parameters)	98%
Gemini	32%

Closing Thoughts

We are happy with the progress we have made on this project and feel that this is an area with a lot of potential. We ultimately weren't expecting the sticking point of this project to be the LLMs, but we routinely had issues with consistency and over 'intelligence'.

For future work in this area, we'd like to see tests with more LLMs, especially paid ones. A reinforcement learning approach to try and get better outputs out of the LLM. And combining image recognition LLMs with existing solutions in a hybrid approach.