Testing GenAI Applications: Addressing the Challenges of Non-deterministic Language Models

JULY 31, 2023

Fabien Zucchet, **aleios**

The recent emergence of Large Language Models (LLMs) has given rise to a new breed of applications powered by Generative AI (GenAI). These applications leverage third-party LLM services to process inputs in a more creative way than traditional applications. With GenAI applications, developers can easily harness the power of AI to generate content, write code, or solve problems in innovative ways.

However, despite their different characteristics, GenAI applications should still be subject to best practice software development principles. Developers still value version control, code review, and rigorous testing for these applications. In this article, we will explore the challenges involved in testing GenAI applications, given the underlying nature of LLMs, and highlight our approach to address these complexities.

The Emergence of Prompt-Centric Applications and the Need for Testing

Unlike traditional applications, where the core value lies in the code, the main value of GenAI applications comes from interacting with third-party LLMs to process user inputs. This paradigm shift has several implications, the most prominent being the non-deterministic nature of LLMs. Running the same model twice with the same prompt can return slightly different responses. Developers have no way to predict precisely how a model will respond to a given prompt (instructions given to an LLM). In addition, the response from the model is returned as loosely structured or unstructured plain text.

Efficient prompt engineering relies on a quick feedback loop to evaluate the quality of LLM responses. Additionally, updating prompts in GenAI applications can introduce unintended consequences and may lead to regressions in the application's behaviour. There has been a huge rise in the number of GenAI based proof of concepts, but little in the way of production applications. In order to bridge the gap, having a robust testing mechanism is key.

Developers need ways to effectively measure and evaluate the impact of changes to the internal prompts on responses, leading to the critical question: How can we effectively test the output of a non-deterministic application?

Our team faced these exact challenges while developing an automatic code review tool using GPT to comment on pull requests (PRs). After a successful proof-of-concept, we had a lot of features and ideas to enhance the tool. As the project gained traction and was applied to real-world cases, ensuring a consistent experiences when implementing new features became the top priority.

Defining Test Cases and Generating Test Prompts

Before testing an application, the first step is to identify what aspects require the most quality assurance. The answer to this question depends on the specific application, and there is no one-size-fits-all approach. In the case of Code Review GPT, our primary concern was the consistency of the tool's reviews. To measure this consistency, we focused on a list of common coding flaws that the tool should identify, such as unintentionally exposing secrets, excessive code nesting, complex functions, and missing "await" keywords for promises, among others. Each item on this list became a test case, and we aimed to ensure that the tool consistently identified and commented on these patterns in the future.

The next step is to run these cases through the application. Since Code Review GPT takes code as input, manually creating a code snippet for each test case would be cumbersome. LLMs excel at this task. We added a step in the test pipeline to generate example code snippets based on plain text descriptions of the test cases. This process translated each business requirement into a format ready to test our application. Below is an example of a simple test case for a code review tool. Once generated, we cached the code snippets for cost optimisation and to prevent introducing unnecessary entropy in the threshold test pipeline.

```
{
  "name": "Unawaited Promise",
  "description": "A code snippet with an unawaited promise"
}
```

Test Case

The test definition is translated through the test pipeline into a code snippet implementing the described bug: in this example of an unawaited promise.

```
function asyncFunction() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Promise resolved');
        }, 1000);
    });
}

function unawaitedPromise() {
    asyncFunction();
}

unawaitedPromise();
```

Generated Code Snippet for Test Case

The next step involves running all the translated test cases through the LLM powered application. For Code Review GPT, each code snippet undergoes a review and a markdown report containing the feedback is created.

Evaluating the Quality of AI Responses

The last step involves evaluating the quality of the review produced by the LLM. While such evaluation is straightforward in traditional applications, it becomes challenging when dealing with non-deterministic LLMs returning loosely structured plain text. Using snapshot testing approaches (like those found in the JS testing library jest) is often not an effective option to assess the non-deterministic output quality.

Why not rely on AI to overcome this limitation? Asking an LLM if a given review matches the test case and if the test should be passing would probably work. But doing so implies using a prompt to test prompts, which lead to spiralling uncertainties.

We opted for a different approach, using snapshots (examples of results we consider satisfying and that we expect) as reference points for evaluation. The snapshots we used are plain text files containing an ideal response we would expect from the LLM to each test case. In the case of Code Review GPT, the snapshots are the markdown review reports (including a 3-emojis summary). You can find an example catching unawaited promises below.

The function `unawaitedPromise` calls `asyncFunction` but does not await its result. This could lead to unexpected behavior if `asyncFunction` is performing any asynchronous operations that `unawaitedPromise` depends on. Consider using the `await` keyword to wait for the promise to resolve before continuing execution. For example:

```
async function unawaitedPromise() {
    // This is an awaited promise
    await asyncFunction();
}

unawaitedPromise();
```

Also, it's good practice to handle promise rejections to avoid unhandled promise rejections. You can do this using `.catch()` or a try/catch block.

⚙️🔄❗

Example Code Review GPT Report

By comparing the test results to these snapshots, we can gauge the quality of the reviews accurately. Using snapshots reduced the complexity of evaluating the quality of AI responses down to comparing two plain text documents. AI happens to be very good at comparing text documents. We can use an AI embeddings model to generate vector embeddings of the two documents.

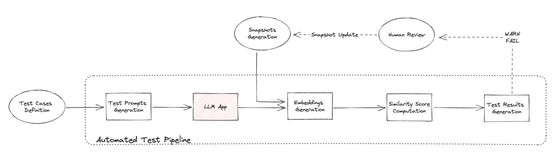
An embedding of a document is a translation of the text into multiple vectors which encapsulate the semantic meaning of the underlying text. Two similar documents will have embeddings which have are closely distanced, whereas very different documents will have embeddings spaced far apart. An approach we refer to as “Embedded Snapshots”.

A very small distance between the embeddings of the test result document and the snapshot indicated a very high semantic similarity, a consistent response from the LLM and therefore a successful test. Conversely, a low similarity indicated that the result was no longer consistent with the snapshot and the test failed. But how should less extreme cases be handled?

We opted for two different thresholds, leading to three possible results: PASS, WARN, and FAIL. The use of three states instead of the traditional PASS/FAIL enables human reviewers to add value to the tests. In cases where the response quality improves but falls just under the similarity threshold, a human can easily detect the enhancement and update the snapshot accordingly. However, manual intervention should be limited. Therefore, human review is only triggered when the similarity score results are uncertain and marked as the WARN status.

Generalising the Test Pipeline

This test pipeline has been presented through the example of Code Review GPT. But the testing strategy can be applied to different projects. The automated threshold testing pipeline including; generating the test prompts, feeding them to the model and computing a similarity score with the embedded snapshots is a generic concept that can be applied to many GenAI application.



GenAI Testing Pipeline

With this test setup run on a CI/CD pipeline, developers can not only confidently update prompts without introducing regressions, but also add regression testing for the changes in the underlying LLM (fine tuning, changing the underlying LLM, etc.) and therefore maintain the stability of their GenAI applications.

Conclusion

As GenAI applications continue to evolve and appear in production, having confidence in their behaviour becomes increasingly crucial. Threshold testing as described in this article serves as a simple yet effective approach to tackle this challenge. As these applications grow in complexity, more advanced testing practices will undoubtedly be required. Adopting proper testing methodologies for GenAI applications can aid in conducting model benchmarks and updates. With LLMs acting as black boxes, prompt testing empowers developers to switch models and ensure consistent behaviour with model updates. Embracing a robust testing approach will be fundamental to the successful development and deployment of GenAI applications in the future.

Fabien Zucchet

Fabien Zucchet is a Serverless Cloud software engineer at Aleios, with experience building GenAI applications and contributing to the development of testing methodologies for non-deterministic LLMs.

aleios

With help from Aleios, Theodo & SICARA