CS453 Automated Software Testing

Spring 2023

Team Project

Debugger-CLI: Automatic Test Code Generation CLI

Jihun Ko
Kristina Shelemba
Akhdan Dzaky Maulana
Minseong Hwang

1. Introduction

When it comes to writing E2E test codes, there are lots of problems as follows.

- **Time and resources**: E2E tests can be resource-intensive, often requiring significant time to write, run, and maintain. As a result, they can slow down the development cycle if not managed effectively.
- **Reducing human error**: E2E testing is complex and requires meticulous attention to detail. Automating these tests reduces the risk of human error that can occur when testers become fatigued.
- **Maintainability**: As the application evolves, E2E tests must be updated to reflect changes in the system. This ongoing maintenance can be challenging.
- **Test flakiness**: E2E tests often interact with many layers of an application and its environment, leading to greater chances of test flakiness, or non-deterministic behavior, making the test results unreliable and the debugging process difficult.
- **Complex scenarios**: E2E tests cover the entire software application from start to finish. This means designing tests that capture complex, real-world scenarios, which requires a deep understanding of the application and the business logic.

Based on these problems, automating E2E test code generation can be beneficial for the industry. Recently, Powerful-enough LLM has emerged and is open to the public through OpenAI. By utilizing this GPT's brilliant power, we can automate complex and repetitive, and faulty processes with a test code generation agent. Using LLM iteratively to reduce the human workload is feasible and useful. We can imagine GPT automatically builds test codes and runs iteratively as developers create more codes.

However, there are some problems that exist such as token limitation for the input of LLM and using GPT solely without guiding heuristics yields poor test codes. So we've mainly focused on squeezing the codebase down to a more GPT-usable format(such as YAML) and how to guide the GPT model so that it can create decent and reliable test codes.

2. Debugger-CLI

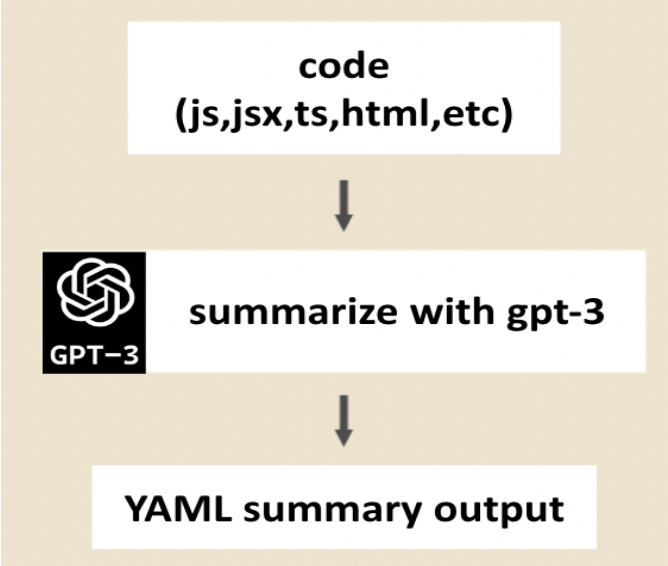
2-1. Yaml Generation

First, the entire codebase is loaded with langchain's GitLoader. The target parameter is stored in GIT_REPOSITORY. Then, we run the below prompt for each file using "gpt-3.5-turbo", while preserving the codebase's original structure:

```
query = """
    You are a professional Software Engineer who has scarce knowledge about E2E testing and file
summarization. Create a definition file that summarizes and explains about file in yaml format. In the
generated yaml format text, it should contain important information about the file.

    [File : {file_path}]
    {page_content}

    File Summary in yaml Format:
    """
```

(Prompt 1: Summarize files for better capturing the context)

In the snippet above, {file_path} is the location of file within the codebase, and {page_content} is the content of the file.



(Figure 1: Summarization Logic)

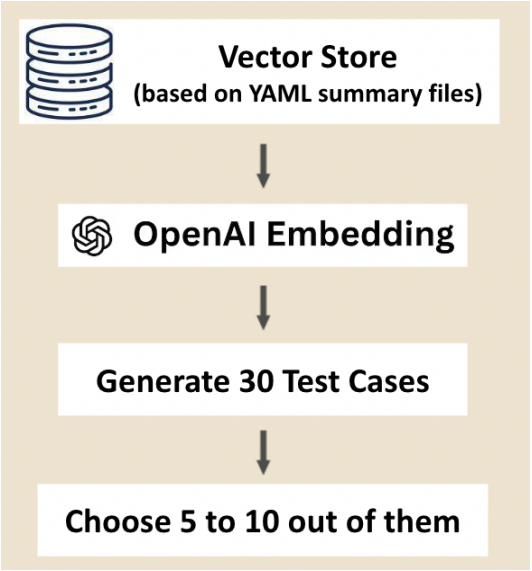
2-2. Test Scenario Generation

After we have the yaml summary of the codebase, we can use it to generate the test scenarios. These yaml files will serve as a context which is stored as a Vector Store object. Next, we send below prompts to "GPT-4":

```
prompt_template = """You are a Software Engineer who writes E2E test scenarios. Use the following
    pieces of context to do actions at the end.
    {context}

    Action: {question}
    """

query = f"""
        Create 30 E2E business logic test scenarios based on document, and choose only
        {TEST_SCENARIO_COUNT} important test scenarios related to users in Project Manager's
        perspective.

        Ignore configuration files such as webpack, package.json, etc. Embed business-logic-related
        files only.

        {TEST_SCENARIO_COUNT} E2E detail test cases(from 30 generated E2E tests) in BULLET POINTS:
        """
```

(Prompt 2: Creating Test Scenarios)

In the snippet above, {context} is the yaml files stored as a Vector Store, {question} is the query string, and {TEST_SCENARIO_COUNT} is the number of test scenarios to generate.



(Figure 2: Test Scenario Creation Logic)

2-3-1. Test Code Generation

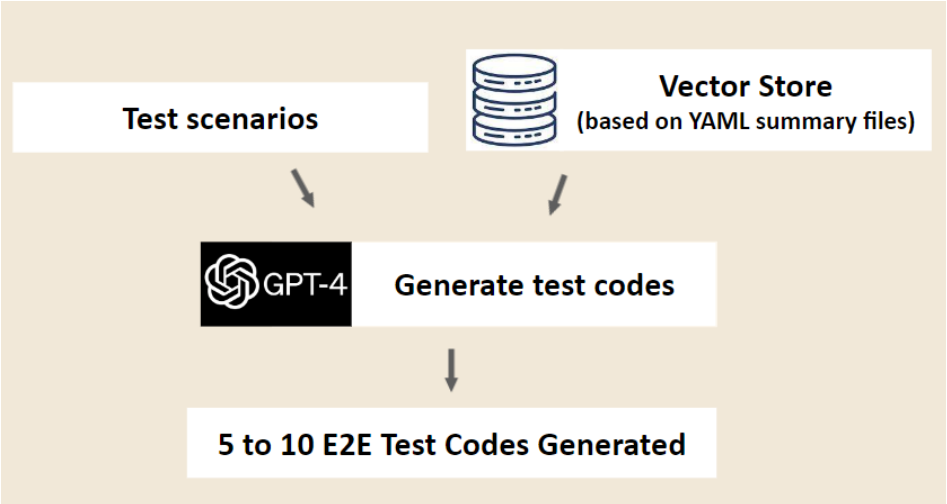
Finally, we can use the yaml summary files and the scenario list file to generate the actual test codes. The yaml summary files are used again as a Vector Store object to provide context. The scenario list was generated as a numbered list so that we can generate separate test code files for each test scenario. Since we want to generate the test codes for all scenarios, we can

create a loop to iterate on each scenario number. For each iteration, we send below prompt to "GPT-4" to generate the test code for the i-th scenario:

```
prompt_template = """You are a Software Engineer who writes test codes.
            Your language/framework preference is Javascript(Node.js, Jest). Use the following pieces
            of context to do actions at the end.
            {context}

            Action: {question}
            """

query = f"""Create E2E test code for {i + 1}th business logic of below test scenario document. E2E test
            code should be in Javscript language which works in Node.js environment. At the beggining
            of the code, test scenario must be embedded in comment section.

            [test-scenario.txt]
            {test_scenarios}

            Professional & Detail E2E test code in Javascript:
            """
```

(Prompt 3: Test Code Generation Prompt)

In the snippet above, {context} is the yaml files stored as a Vector Store, {question} is the query string, {i+1} is the scenario number, and {test_scenario} is the list of scenarios.



(Figure 3: Test Code Generation Logic)

2-3-2. Performance Improving Techniques

Based on what we've done for 2-3-1, we was able to create general test codes which implement the chromium based test that locates some elements and fill out the required data and test the functionality. Eventhough, the general idea for this generated test codes were valid enough but it requires some additional setup(set up required frameworks, and other environment settings). So, for codebases with human-written test codes in specific frameworks(NestJS HTTP Mocking test, etc..), we've tried using this codes as a few shot learning

samples. And we were able to create some fuzzed test input by conditionally adding a prompt as following, this was only available when the example inputs of endpoint was provided.

```
"""
Also, make sure to create lots of test cases(test inputs) using fuzzing technique from provided test
input.(Create multiple blocks of 'it' for each test case)
When fuzzing the inputs, change the input words to random words.
"""
```

(Prompt 4: Conditional prompt to apply some fuzzing techniques)

3. Evaluation

3-0. Assumption

We assumed that we have access to test scenarios and test codes. If this requirement is not met, then the test scenarios and test codes have to be created manually.

3-1. Coverage

As a coverage measure, we generated test scenarios by hand, with our Debugger-CLI for login-flow software in https://github.com/mxstbr/login-flow and using ChatGPT without pre-processing steps. Using human-written test scenarios as the upper bound, we observed how many generated test scenarios are actually present in the human-written scenarios and whether our pipelined approach works better than ChatGPT out of the box.

The human-written test scenarios are as below.

Test Scenarios:

1. Verify if a user will be able to login with a valid username and valid password
2. Verify if a user cannot login with a valid username and an invalid password.
3. Verify if a user cannot login with an invalid username
4. Verify the login page when both fields are blank and the Login button is clicked.
5. Verify if a user will be able to register with a new username and password
6. Verify if a user cannot register an existing username
7. Verify the register page when both fields are blank and the Register button is clicked.
8. Verify the time taken to log in with a valid username and password.

9. Verify if the user is able to log out after logging in.
10. Verify the Login page against SQL injection attack.

Generated scenarios that exist in human-written scenarios:

- [1] Verify if a user will be able to login with a valid username and valid password
- [2] Verify if a user cannot login with a valid username and an invalid password.
- [3] Verify if a user cannot login with an invalid username
- [5] Verify if a user will be able to register with a username and password
- [6] Verify if a user cannot register an existing username

Generated scenarios that don't exist in human-written scenarios:

1. User registration:

- Test user registration with invalid username formats and ensure the error message is displayed.

3. User profile: Entire test suite

4. Project creation: Entire test suite

5. Project editing: Entire test suite

6. Task assignment: Entire test suite

7. Task status updates: Entire test suite

8. Notifications: Entire test suite

9. Document management: Entire test suite

10. User onboarding: Entire test suite

Thus, the generated scenarios reached a decent 50% coverage, since it reached 5 out of 10 scenarios. However, it seems that the generated scenarios tried to cover a broader topic than just login, register, or logout. This might be caused by the large number of test scenario demands compared to the small handful of features of the target software (login, register, dashboard, logout), so that GPT starts imagining features that don't exist, such as project creation, task assignment, notifications, etc.

ChatGPT-generated scenarios that exist in human-written scenarios: [1], [2], [3], [5], [6], [8]

ChatGPT-generated scenarios that do not exist in human-written scenarios: N/A

Human-written scenarios that do not exist in ChatGPT-generated scenarios: [4], [7], [9], [10]

Thus, the scenarios generated by ChatGPT achieved 60% coverage since it reached 6 out of 10 scenarios. However, ChatGPT did not manage to come up with any test scenarios that had not already been written by a human. This indicates that while ChatGPT did do a decent job coming up with test scenarios similar to a human, there were no unique scenarios that did not already exist in the human-written ones. When provided with the actual project source code, ChatGPT operates strictly within the bounds of the source code, without considering any possible extensions to the project. While this may be good in some cases, the lack of capacity to produce entirely new test scenarios hinders ChatGPT's ability to offer fresh insights and innovative solutions that were not explicitly present in the existing source code.

3-2. Accuracy

Compare the results of generated codes and hand-written codes

First, Let's compare the human-written testing codes with the ones generated by ChatGPT without any preprocessing.

```
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { INestApplication } from '@nestjs/common';
import { AppActions } from './app.actions';
import { AppConstants } from './app.constants';
import { MessageConstants } from './message.constants';
import { auth } from './auth';
import { genSalt } from './salt';
import { BrowserHistory } from 'react-router';

describe('AppActions', () => {
  let app: INestApplication;

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [AppActions],
    }).compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });
```

(NestJS setup code generated by ChatGPT w/o preprocessing)

That setup code is mostly correct, however, there are too many unnecessary imports, which negatively affect the code's readability and maintainability.

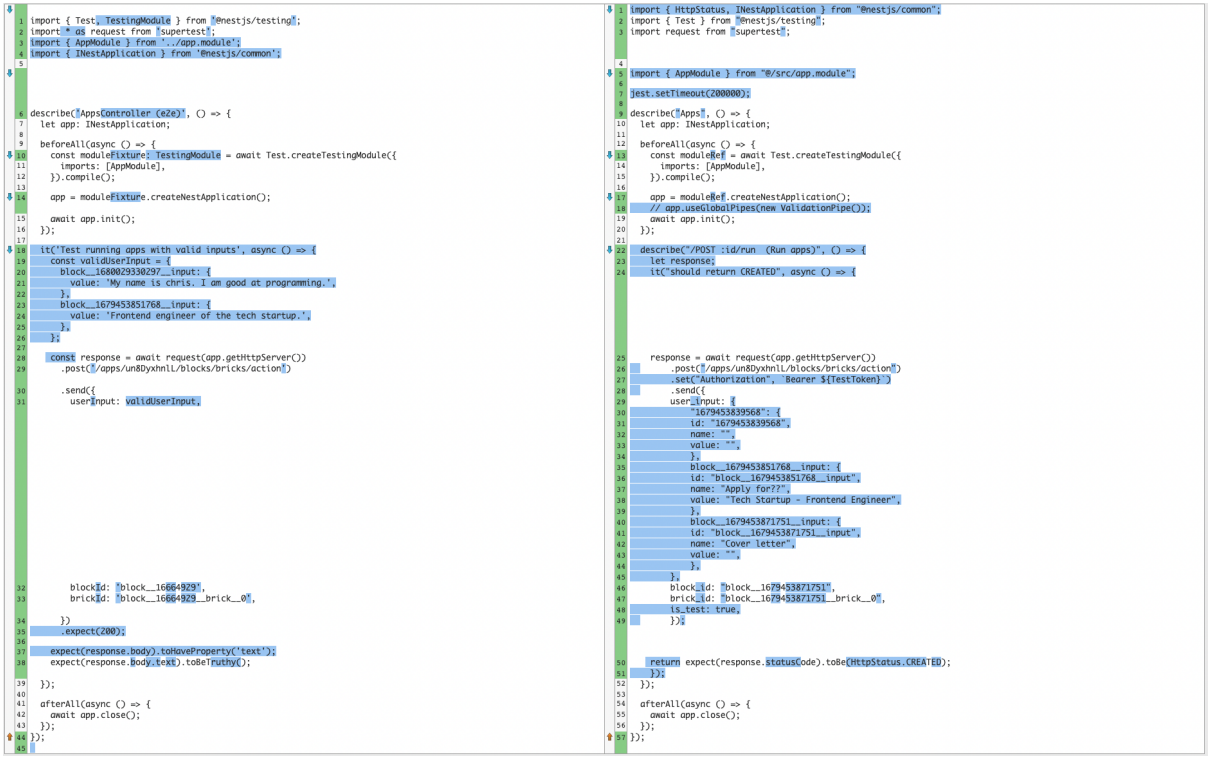
```
  describe('POST /login', () => {
    it('should log in a user', async () => {
      const response = await request(app.getHttpServer())
        .post('/login')
        .send({ username: 'testUser', password: 'password123' });

      expect(response.status).toBe(200);
      expect(response.body.authenticated).toBe(true);
      expect(response.body.token).toBeDefined();
    });
```

(Example of a NestJS testing code generated by ChatGPT w/o preprocessing)

It is important to note here that one of the main advantages of our approach is the fact that we can provide an entire embedded codebase to generate scenarios and testing codes, which is not something you can do with GPT/ChatGPT out of the box due to the limit on the number of characters/tokens in the request. Thus, while GPT/ChatGPT may work just fine for generating test codes for small toy projects, using them without any pre-processing for more complex systems is not feasible.

Now, here is some result based on our methodology.

An evaluation Case with testing framework specification(NestJS Testing) Test Scenario and example input/output definition from the codebase is provided.



```
import { Test, TestingModule } from '@nestjs/testing';
import * as request from 'supertest';
import { AppModule } from '../app.module';
import { INestApplication } from '@nestjs/common';

describe('AppsController (e2e)', () => {
  let app: INestApplication;

  beforeAll(async () => {
    const moduleFixture: TestingModule = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleFixture.createNestApplication();

    await app.init();
  });

  it('Test running apps with valid inputs', async () => {
    const validUserInput = {
      block__1680029330297__input: {
        value: 'My name is chris. I am good at programming.',
      },
      block__1679453851768__input: {
        value: 'Frontend engineer of the tech startup.',
      },
    };

    const response = await request(app.getHttpServer())
      .post('/apps/un8Dyxhnll/blocks/bricks/action')

      .send({
        userInput: validUserInput,

        blockId: 'block__16664929',
        brickId: 'block__16664929__brick__0',

      })
      .expect(200);

    expect(response.body).toHaveProperty('text');
    expect(response.body.text).toBeTruthy();

  });

  afterAll(async () => {
    await app.close();
  });
});
```

```
import { HttpStatus, INestApplication } from "@nestjs/common";
import { Test } from "@nestjs/testing";
import request from "supertest";

import { AppModule } from "@/src/app.module";

jest.setTimeout(200000);

describe("Apps", () => {
  let app: INestApplication;

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleRef.createNestApplication();
    // app.useGlobalPipes(new ValidationPipe());
    await app.init();
  });

  describe("/POST :id/run  (Run apps)", () => {
    let response;
    it("should return CREATED", async () => {

      response = await request(app.getHttpServer())
        .post("/apps/un8Dyxhnll/blocks/bricks/action")
        .set("Authorization", `Bearer ${TestToken}`)
        .send({
          user_input: {
            "1679453839568": {
              id: "1679453839568",
              name: "",
              value: "",
            },
            block__1679453851768__input: {
              id: "block__1679453851768__input",
              name: "Apply for??",
              value: "Tech Startup - Frontend Engineer",
            },
            block__1679453871751__input: {
              id: "block__1679453871751__input",
              name: "Cover letter",
              value: "",
            },
          },
          block_id: "block__1679453871751",
          brick_id: "block__1679453871751__brick__0",
          is_test: true,
        });

      return expect(response.statusCode).toBe(HttpStatus.CREATED);
    });
  });

  afterAll(async () => {
    await app.close();
  });
});
```

(Diff N: GPT Generated (Left) vs Human Generated (Right) test code for app running endpoint)

For this specific case, we can see that overall structure for writing test code is same, but few implementation detail(validUserInput variable is defined before running request.send method and human did not, etc..). Furthermore, I've tested this same case with fuzzing prompt enabled. And I was able to get more various test inputs as following.

```
{
  userInput: {
    block__1680029330297__input: {
      value: 'My name is Alice. I am good at designing.',
    },
    block__16794538517 68__input: {
      value: 'Graphic designer of design team of tech startup',
    },
  },
  blockId: 'block__1679453871751',
  brickId: 'block__1679453871751__brick__0',
},
```

(Fuzz Result 1: Name and specialty changed chris->alice, program->design)

By requesting minimum number of fuzzing input, GPT was able to automatically create list of inputs and changed the test code to iterate over those test cases properly.

3-3. Flexibility

To test the flexibility of our software, we can test it with many different kinds of websites, and measure the coverage and accuracy for each website. As expected, this will take a lot of time. So, unfortunately, we will not be doing the flexibility testing due to time constraints.

4. Conclusion

LLM is powerful enough to substitute demanding human works. However, there exists still lots of considerations because of explicit limitations of GPT(context size limit, needs a lot of prompt engineering, etc..). For tackling these problems, we've created a pipeline from summarizing code files and feed them the model to define a test scenarios and lastly, based on those scenario it creates a actual test codes. We evaluated our GPT pipeline to automate test code generations by comparing those of human creations. And for the general results, we were able to conclude that this test code automation technique can reduce lots of burdens for developers writing test code themselves. However, there were some limitations this technique have.

First, GPT models basically result the output probabilistically which means that no matter how we change the configurations to lower the randomness (temperature parameter), the result of generation was non-deterministic. So this could lead to inconsistent performance for some cases.

Second, actually putting this automated test code in real use(integrating tests into CI/CD pipelines and further upgrading test code to cover more specific cases) requires human works. So this entire test code generation to fine-tuning and continuously integrating in a development pipeline it would be further developed based on this project.

5. Github Repository

https://github.com/xMHW/debuggers-cli

6. Reference

https://platform.openai.com/docs/guides/embeddings
https://python.langchain.com/en/latest/index.html
https://www.softwaretestinghelp.com/login-page-test-cases/