UDC: 004.93 https://doi.org/10.15407/jai2024.04.195

O. Harasymchuk
Lviv Polytechnic National University, Ukraine
12, Stepan Bandera str., 79013, Lviv, Ukraine
oleh.ih.harasymchuk@lpnu.ua
https://orcid.org/0009-0008-6794-8599

ARTIFICIAL INTELLIGENCE IN CONSUMER-DRIVEN CONTRACT TESTING OF DISTRIBUTED SYSTEMS

Abstract. This article explores the case of the usage of artificial intelligence (AI) for optimizing the process of covering distributed systems with consumer-driven contract test, analyzing the pros and cons of this approach. Considering the complexity of development of modern distributed systems, like microservices, and the need to ensure the system components interactions keep reliable as long as the system keeps evolving this study is focused on finding the most effective way to introduce the contact testing into such systems to maximize the contracts tests coverage while minimizing development costs.

The contract testing has its challenges: steep learning curve, impact on the delivery lifecycle, spreading the approach consistently across the organization. These challenges often lead to teams sacrificing the benefits of the approach and using more traditional ways of testing, like end-to-end (E2E) testing, which however does not fit well into distrusted system.

The described methodology includes generating (by AI platform) the contract between the parties (consumer and provider), generating the consumer test to verify the provider is compatible with the expectations the consumer has of it. It is proposed to use following inputs for AI as the source for generation: request-response pairs, OpenApi specification, consumer codebase.

The research employs Pact as a tool that allows to define a contract between a consumer and a provider as well as verify that both sides adhere to this contract. NodeJS is used as a framework for consumer and provider development. PactFlow platform with its HaloAI executes contracts and tests generation. The proposed approach simplifies the road to introduce the contact testing into the distributed systems, increases the development team effectiveness in system implementation and a confidence in its stability.

Keywords: contract testing, artificial intelligence, pact, distributed systems, microservices, consumer, provider.

Intoduction

In today's software development landscape, the use of distributed architectures is prevalent. Each component owns and handles specific functionality and communicates with others through the variety of network protocols, like http, messaging or grpc. In this context, the need arises to ensure compatibility between services, ensuring they interact as expected and that changes in one do not negatively affect others [1]. Miscommunication between services can lead to broken functionalities, delayed releases, and ultimately, a poor user experience.

This is where consumer-drivent contract testing [2] plays a crucial role. Contracts serve as a formal agreement between the two teams, ensuring that both sides adhere to predefined expectations [3]. Contract tests act as a safety net, catching potential issues before they reach production. This approach ensure that services (or components) interact with each other as expected and changes in one service do not break others. Unlike traditional end-to-end tests, which test the entire system, contract tests focus on the interactions between specific components. This makes them faster and more reliable. However contract testing is a technique, and there are many ways to achieve it. Various tools offer contract testing capabilities, each with its own learning curve, which can sometimes detract from the true benefits of the approach. The objective is to optimize the way organizations and teams have to pass to start using it, decrease the entrance threshold into the approach. With the rise of Generative AI, organizations and practitioners have been wondering how best to harness the emerging technologies to alleviate common problems across many workflows, with contract testing not an exception. This article examines current tools on the market that offer AI capabilities, the benefits and caveats of it, tactics and methodologies of GenAI usage to optimize contract testing implementation.

Analysis of recent research

In recent years, the use of consumer-driven contract testing in distributed systems has become more considered but still it's not as popular as the traditional approaches. There are few materials describing why the traditional ways of testing are not the best choice and what are the pitfalls. Integration tests are a good way of verifying our system as they use real (not mocked out) components but quite a lot can go wrong [1,5]. Integration tests are [6]:

– **Unstable.** A lot of efforts need to be taken to keep it up to date. Unstable tests are worse than no tests [7] as they train your team to ignore test results.

– **Broadly scoped & unspecific.** When integration test fail it could be a whole host of possible reasons. But a tight scope should apply to your tests. When your tests fail you should be able to quickly pinpoint where the failure happened quickly and start to understand why.

– **Slow**. It is quite expensive (resource vise) to spin up the subset of your system under test. It often requires additional servers and computing power to have the test suite up and running.

– **Expensive.** To keep your integration tests up to date and fast comes at a cost. Developers or QAs will need to spend quite a lot of time keeping the tests green.

On the other hand, the contract tests are:

– **Fast.**

– **Focused** on the single integration at a time.

– **Cheaper**. No need for dedicated test environments.

– **Reliable**. They have fast and reliable feedback, that is easier to debug.

– **Scalable**. They scale linearly [4] with the number of integrations, instead of exponentially (Fig. 1).

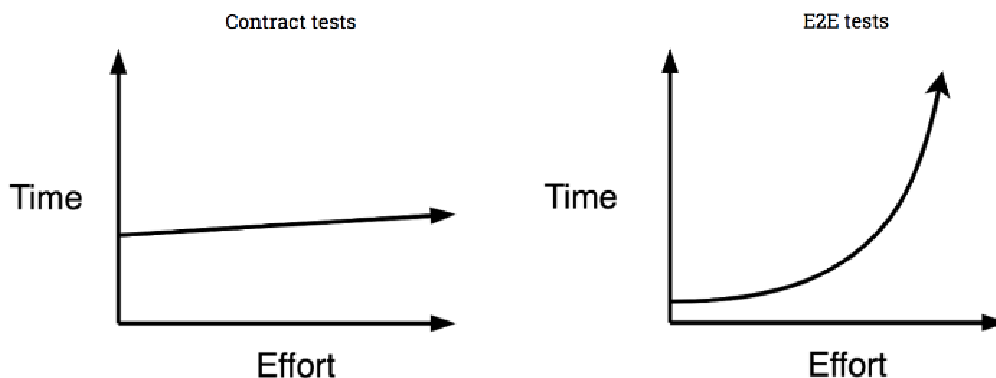

Fig. 1. Contract vs E2E tests comparison

Regarding the use of GenAI for contract test generation, this specific topic is quite new in the industry. We can find some thoughts about it in a few articles over the network, describing that AI integration into contract testing can overcome hurdles, enhancing efficiency, accuracy, and overall software quality [8]. GenAI tools like ChatGPT and CoPilot have demonstrated the potential to dramatically enhance efficiency, accelerate learning, and offer valuable insights that were previously difficult to uncover. There is even ready to use solution, that we are going to use in this research [9], however this solution is the single one on the market. It supports only test generation for services communicating over *http* protocol. Moreover, it can't be used for free and supports only 2 technologies that can integrate with it: JS and Java. So, a plenty of other popular technologies, like. Net or Ruby are still uncovered.

The aim of the study

The main purpose of this work is to demonstrate the GenAI usage to generate contracts and consumer tests for the distributed systems based on different type of inputs, analyze which inputs are better to use to calibrate the end result, so that human involvement into the process is minimized.

Usual approach of manual test implementation often results into several problems: human errors, long learning curve, ineffective tests that can provide a false sense of confidence by either testing too little or testing the wrong parts of your code. Introducing GenAI possibilities into the process we get the next benefits:

- Reducing human error.
- Predictive insight.
- Quickly scaffolding.
- Improved time to market.
- Scale across organization.

We are going to exercise the following modes of consumer test generation:

- Traffic capture,
- OpenAPI descriptions.
- Existing client code.
- OpenAPI descriptions improved by client code.

```
1.  // Consumer Code
2.  const ProductRepository = require("./product.repository");
3.  const repository = new ProductRepository();
4.  exports.getAll = async (req, res) => {
5.      res.send(await repository.fetchAll())
6.  };
7.  exports.getById = async (req, res) => {
8.      const product = await repository.getById(req.params.id);
9.      product ? res.send(product) : res.status(404).send({message: "Product not found"})
10. };
11. exports.repository = repository;
12. class Product {
13.     constructor(id, type, name, version, price) {
14.         this.id = id;
15.         this.type = type;
16.         this.name = name;
17.         this.version = version;
18.         this.price = price;
19.     }
20. }
21.
22.  // Provider Code
23. const axios = require('axios').default;
24. const adapter = require('axios/lib/adapters/http');
25. axios.defaults.adapter = adapter;
26. class Product {
27.     constructor(id, type, name) {
28.         this.id = id;
29.         this.type = type;
30.         this.name = name;
31.     }
32. }
33. export class API {
34.   constructor(url) {
35.    //... some code here
36.   }
37.   async getAllProducts() {
38.     return axios
39.       .get(this.withPath('/products'), {
40.         headers: {
41.           Authorization: this.generateAuthToken()
42.         }
43.       })
44.       .then((r) => r.data.map((p) => new Product(p)));
45.   }
46.   async getProduct(id) {
47.     return axios
48.       .get(this.withPath('/product/' + id), {
49.         headers: {
50.           Authorization: this.generateAuthToken()
51.         }
52.       })
53.       .then((r) => new Product(r.data));
54.   }
55. }
```

Fig. 2. Consumer & provider codebase

Methods and materials

To setup our framework and working environment we would need to use several existing technologies and libraries.

– **NodeJS** is used to implement out distributed system parties between which we are going to setup contract and tests: consumer and provider.

– **Pact** is a code-first tool that allows to define a contract between a consumer and a provider as well as verify that both sides adhere to this contract.

– **Pactflow** is AI-powered developer platform that allows us to enable test generation feature and run it through different data inputs.

– **Killercoda** is an online platform that – allows us spin up the working environment and run our experiments on it.

Testing results

The following (Fig. 2) provider and consumer codebase was created using NodeJS (only important parts left just to illustrate the idea, rest of the codebase was omitted).

Traffic capture

Let's consider the case when we have an existing provider that we wish to generate contracts for. Pactflow-ai allows to generate Pact tests from capture files. These capture files, are request/response pairs. Specifically, the capture files should conform to an HTTP Messages, using the HTTP/1.1 format (Fig. 3).

```
1.  //Request
2.  GET /product/10 HTTP/1.1
3.  Host: api.example.com
4.  User-Agent: curl/8.1.2
5.  Accept: application/json;
    charset=UTF-8
6.
7.  //Response
8.  HTTP/1.1 200 OK
9.  Content-Type: application/json;
    charset=utf-8
10. Content-Length: 96
11. {
12.   "id": "10",
13.   "name": "Aussie",
14.   "type": "Pizza",
15.   "version": "v1",
16.   "price": 9.99,
17. }
```

Fig. 3. Request & response example

Let's use this pair as the input for the pactflow-ai to generate the consumer test.

```
pactflow-ai generate request-response \
--request ./capture/get.request.http \
--response ./capture/get.response.http \
  --language typescript \
  --output ./src/api.pact.spec.ts
```

The generated consumer test is shown in Fig. 4 below:

```
1. import { PactV3, MatchersV3 } from "@pact-foundation/pact";
2. describe('Consumer Pact with Product Service', () => {
3.   const provider = new PactV3({
4.     consumer: 'ProductConsumer',
5.     provider: 'ProductService',
6.   });
7.   describe('Pact for retrieving a product', () => {
8.     it('returns the product details', () => {
9.       provider.given('Product with ID 10 exists')
10.        .uponReceiving('a request for product with ID 10')
11.        .withRequest({
12.          method: 'GET', path: '/product/10',
13.          headers: {'Accept': 'application/json; charset=UTF-8'},
14.        })
15.        .willRespondWith({
16.          status: 200,
17.          headers: {'Content-Type': 'application/json; charset=utf-8'},
18.          body: {
19.            id: MatchersV3.string('10'), name: MatchersV3.string('Aussie'),
20.            type: MatchersV3.string('Pizza'), version: MatchersV3.string('v1'),
21.            price: MatchersV3.number(9.99)},
22.        });
23.      return provider.executeTest(async (mockserver) => {
24.        const client = new ProductClient(mockserver.url);
25.        const response = await client.getProduct('10');
26.        expect(response).to.deep.equal({
27.          id: '10', name: 'Aussie',
28.          type: 'Pizza', version: 'v1',
29.          price: 9.99,
30.        });});});});
31. });
```

Fig. 4. Consumer test

The are several problems with this consumer test. The ProductClient is generated by AI but it’s not a real http client The test expectation is to get response from provider with 5 fields inside, but our consumer uses only 3 of them: id, *type,* name. These problems would cause the test failure if we ran it, so still it needs some manual work to be done to fix it.

OpenAPI descriptions

We may have documented services using API descriptions that adhere to the OpenAPI specification. These OpenAPI descriptions can aid shared understanding of a providing services expectations, for both providers and consumers. For consumers, they can easily leverage tools like swagger-codegen to bootstrap clients from OpenAPI specifications in multiple languages. So having OpenAPI descriptions its now possible to execute test generation passing these files as an input.

```
pactflow-ai generate openapi \
--spec ./products.yml \
--endpoint "/product/{id}" \
--output ./src/api.pact.spec.ts \
--language typescript
```

We received almost identical consumer test, but this time the "dummy" http client is the real working one, so the the only reason for test to fails would be still 2 extra fields that consumer does not use: price, version. We can try to improve the test generation. We will use an OpenAPI description as an input but will also provide the context of our client code, which will become our system under test.

```
1.  openapi: 3.0.1
2.  info:
3.    title: Product API
4.    description: Pactflow Product API demo
5.    version: 1.0.0
6.  paths:
7.    /product/{id}:
8.      get:
9.        summary: Find product by ID
10.       description: Returns a single product
11.       operationId: getProductByID
12.       parameters:
13.       - name: id
14.         in: path
15.         description: ID of product to get
16.         schema:
17.           type: string
18.         required: true
19.         example: 10
20.       responses:
21.         "200":
22.           description: successful operation
23.           content:
24.             "application/json; charset=utf-8":
25.               schema:
26.                 $ref: '#/components/schemas/Product'
27.               examples:
28.                 application/json:
29.                   value:
30.                     id: "1234"
31.                     type: "food"
32.                     price: 42
33.         "400":
34.           description: Invalid ID supplied
35.           content: {}
36. components:
37.   schemas:
38.     Product:
39.       type: object
40.       required:
41.         - id
42.         - name
43.         - price
44.       properties:
45.         id:
46.           type: string
```

Fig. 5. Provider OpenApi specification

```
1.  import { PactV3, MatchersV3 } from "@pact-foundation/pact";
2.  import { expect } from "chai";
3.
4.  describe('Consumer Pact with Product Service', () => {
5.    const provider = new PactV3({
6.      consumer: 'ProductConsumer',
7.      provider: 'ProductService',
8.    });
9.
10.   describe('when a request to get a product by ID is made', () => {
11.     it('returns the product details', () => {
12.       provider
13.         .given('a product with ID 1234 exists')
14.         .uponReceiving('a request to get a product by ID')
15.         .withRequest({
16.           method: 'GET',
17.           path: MatchersV3.regex('/product/\\d+', '/product/1234'),
18.           headers: {
19.             'Accept': 'application/json; charset=utf-8'
20.           }
21.         })
22.         .willRespondWith({
23.           status: 200,
24.           headers: {
25.             'Content-Type': 'application/json; charset=utf-8'
26.           },
27.           body: MatchersV3.like({
28.             id: MatchersV3.string('1234'),
29.             name: MatchersV3.string('Sample Product'),
30.             price: MatchersV3.number(42),
31.             type: MatchersV3.string('food'),
32.             version: MatchersV3.string('1.0')
33.           })
34.         });
35.
36.       return provider.executeTest(async (mockserver) => {
37.         // Dummy client to simulate the request
38.         const client = {
39.           getProductById: async (id: string) => {
40.             const response = await fetch(`${mockserver.url}/product/${id}`, {
41.               method: 'GET',
42.               headers: {
43.                 'Accept': 'application/json; charset=utf-8'
44.               }
45.             });
46.             return response.json();
47.           }
48.         };
49.
50.         const response = await client.getProductById('1234');
51.         expect(response).to.deep.equal({
52.           id: '1234',
53.           name: 'Sample Product',
54.           price: 42,
55.           type: 'food',
56.           version: '1.0'
57.         });
58.       });
59.     });
60.   });
61. });
```

Fig. 6. Consumer test (2)

By providing our codebase as a context, pactflow-ai can replace the dummy client generated in the previous step, with our real client. If the code provided also contains the object model used, then pactflow-ai should ensure that only fields used by a consumer are added to the contract. This avoids a common pitfall, whereby more fields are added to a consumer test, than the consumer client code uses.

This puts tension between provider teams, as they are unduly bound to honour those fields for consumers that do not use them.

```
pactflow-ai generate openapi \
  --spec ./products.yml \
  --endpoint "/product/{id}" \
  --code ./src/api.js \
  --code ./src/product.js \
  --output ./src/api.pact.spec.ts \
  --language typescript
```

```
1. import { PactV3, MatchersV3 } from "@pact-foundation/pact";
2. import { API } from "./src/api";
3. import { expect } from "chai";
4.
5. describe('Consumer Pact with Product Service', () => {
6.   const provider = new PactV3({
7.     consumer: 'ProductConsumer',
8.     provider: 'ProductService',
9.   });
10.  describe('when a request to get a product by ID is made', () => {
11.    it('returns the product details', async () => {
12.      provider
13.        .given('Product with ID 1234 exists')
14.        .uponReceiving('a request to get a product by ID')
15.        .withRequest({
16.          method: 'GET',
17.          path: MatchersV3.regex('/product/\\d+', '/product/1234'),
18.          headers: {'Accept': 'application/json; charset=utf-8'},
19.        })
20.        .willRespondWith({
21.          status: 200,
22.          headers: {'Content-Type': 'application/json; charset=utf-8'},
23.          body: MatchersV3.like({
24.            id: '1234', name: MatchersV3.string('Sample Product'),
25.            price: MatchersV3.number(42),
26.            type: MatchersV3.string('food'),
27.            version: MatchersV3.string('1.0'),
28.          }),
29.        });
30.      await provider.executeTest(async (mockServer) => {
31.        const api = new API(mockServer.url);
32.        const product = await api.getProduct('1234');
33.        expect(product).to.deep.equal({
34.          id: '1234', name: 'Sample Product',
35.          price: 42, type: 'food',
36.          version: '1.0'
37.        }); }); }); });
38. });
```

Fig. 7. Consumer test (3)

We can see the improved test version, where read API clients is used as well as response is expected to contain only 3 fields used by consumer.

Existing client code

When we do not have OpenAPI descriptions for the provider we can generate tests from the client code, without the need for an OpenAPI description.

```
pactflow-ai generate code
   ./src/api.js ./src/product.js \
  --output ./src/api.pact.spec.ts \
  --language typescript
```

```
1. import { PactV3, MatchersV3 } from "@pact-foundation/pact";
2. import { API } from "./src/api";
3. import { Product } from "./src/product";
4.
5. describe('Consumer Pact with Product Service', () => {
6.   const provider = new PactV3({
7.     consumer: 'ProductConsumer',
8.     provider: 'ProductService',
9.   });
10.
11.   describe('Fetch product details', () => {
12.     it('returns the details of a product', () => {
13.       const productId = 1;
14.
15.       provider
16.         .given(`product with ID ${productId} exists`)
17.         .uponReceiving('a request to fetch product details')
18.         .withRequest({
19.           method: 'GET',
20.           path: `/product/${productId}`,
21.           headers: {
22.             'Authorization': MatchersV3.string('Bearer some-auth-token'),
23.           },
24.         })
25.         .willRespondWith({
26.           status: 200,
27.           body: {
28.             id: MatchersV3.integer(productId),
29.             name: MatchersV3.string('Product Name'),
30.             type: MatchersV3.string('Product Type'),
31.           },
32.         });
33.
34.       return provider.executeTest(async (mockserver) => {
35.         const api = new API(mockserver.url);
36.         const product = await api.getProduct(productId);
37.         expect(product).to.be.an.instanceOf(Product);
38.         expect(product.id).to.equal(productId);
39.         expect(product.name).to.be.a('string');
40.         expect(product.type).to.be.a('string');
41.       });
42.     });
43.   });
44. });
```

Fig. 8. Consumer test (4)

This consumer test version looks the best and perfectly matching the consumer needs. However, GenAI does not have any information about the provider this time, so it can't make any expectations on values that could be returned by the real provider during the runtime. Finally, if we run our test we should get the Pact contract

"npm run test:pact"

You can check the test result below. The product details test passed succesfully.

```
1.  ubuntu $ npm run test:pact
2.   PASS  src/api.pact.spec.ts
3.    Consumer Pact with Product Service
4.      Fetch product details
5.        √ returns the details of a product (11 ms)
6.  Test Suites: 1 passed, 1 total
7.  Tests:       1 passed, 1 total
8.  Snapshots:   0 total
9.  Time:        2.513 s
10. Ran all test suites.
```

Fig. 9. Test results

During this research we exercised different options that GenAI can provide, and how it can help us to quickly generate consumer Pact tests and pacts between parties as well. The generation process is not ideal and different inputs can produce slightly different results.

Whilst GenAI may need a small amount of tweaking to run, it quickly and accurately generate Pact tests, using the latest client library DSL's, following recommended Pact best practices, including the usage of Provider States and Matchers.

By refining the inputs we provide to Gen AI, we can find ways to tailor it the particular use case.

Conclusions

In this article we've investigated the GenAI consumer test generation capabilities, the different options provided, and how they can help to quickly generate consumer Pact tests.

While this approach may need a small amount of tweaking to run, it quickly and accurately uses the latest client library DSLs, and follows the recommended Pact best practices, including the usage of provider states and matchers. By refining the inputs you provide to GenAI paltform, we can tailor it to the particular use case.

There is still a room for improvements and future invetigations: improving the inputs for the AI, extending the framework to support more techologies, such as Python, .NET & Go and more communication protocols, like grpc.

However, even the existing mechanism is powerfull enough to help development teams and organizations to apply consumer-driven contract testing faster in their projects and optimize the testing strategies with contract testing benefits.

```
1.  {
2.    "consumer": {
3.      "name": "ProductConsumer"
4.    },
5.    "interactions": [
6.      {
7.        "description": "a request to get a product by ID",
8.        "providerStates": [
9.          {
10.           "name": "a product with ID 1234 exists"
11.         }
12.       ],
13.       "request": {
14.         "headers": {
15.           "Accept": "application/json; charset=utf-8"
16.         },
17.         "matchingRules": {
18.           "header": {},
19.           "path": {
20.             "combine": "AND",
21.             "matchers": [
22.               {
23.                 "match": "regex",
24.                 "regex": "/product/\\d+"
25.               }
26.             ]
27.           }
28.         },
29.         "method": "GET",
30.         "path": "/product/1234"
31.       },
32.       "response": {
33.         "body": {
34.           "id": "1234",
35.           "name": "Sample Product",
36.           "type": "food"
37.         },
38.         "headers": {
39.           "Content-Type": "application/json; charset=utf-8"
40.         },
41.         "matchingRules": {
42.           "body": {
43.             "$.id": {
44.               "combine": "AND",
45.               "matchers": [
46.                 {
47.                   "match": "type"
48.                 }
49.               ]
50.             },
51.             //...other rules
52.           },
53.           "header": {}
54.         },
55.         "status": 200
```

Fig. 10. Pact (contract)

References

1. Lehvä, J., Mäkitalo, N. & Mikkonen (2019). Consumer-Driven Contract Tests for Microservices: A Case Study. https://doi.org/10.1007/978-3-030-35333-9_35
2. Marie Cruz & Lewis Prescott. Contract Testing in Action
3. Robinson. I.: Consumer-Driven Contracts: A Service Evolution Pattern. Available:https://martinfowler.com/articles/consumerDrivenContracts.html
4. Matt Fellows. What is contract testing and why should I try. Available: https://pactflow.io/blog/what-is-contract-testing/
5. Elliott Murray. Proving E2E tests are a scam. [Online] Available: https://elliottmurray.medium.com/proving-e2e-tests-are-a-scam-21d39e88913
6. Thomas Shipley. Contract Testing with Pact in Net Core. Available:https://tomdriven.dev/.net%20core/c%23/contract%20testing/pact/test/2018/03/13/contract-testing-with-pact-in-net-core.html
7. Just Say No More to End-to-End Test. [Online] Available: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html
8. Yousaf Nabi. Unpacking GenAI's Role in Contract Testing. Available: https://pactflow.io/blog/ai-automation-part-2
9. Pactflow [Online]. Available: https://docs.pactflow.io/
10. Manuel Simosa; Frank Siqueira, Contract Testing in Microservices-Based Systems: A Survey. doi:10.1109/ICSESS58500.2023.10293058

The article has been sent to the editors 11.10.24.
After processing 20.10.24.
Submitted for printing 30.12.24.

Copyright under license CCBY-NC-ND