OkIP

Tiako P.F. (ed) *Intelligent Computing and Consumer Support Applications*. Chronicle of Computing. OkIP
© 2023 Oklahoma International Publishing
https://doi.org/10.55432/978-1-6692-0003-1_5

Automatic Test Generation for Microservices Based on Consumer-Driven Contracts (CDC)

Shaheen Zahedi[a], Samad Paydar[b]

Shaheen Zahedi [a], Gilan, Rasht, Iran
Samad Paydar [b], Khorasan, Mashhad, Iran

ABSTRACT

Microservices offer a shorter time-to-market by improving productivity with maximizing the automation of the software development lifecycle. However, when dealing with such systems, there are important design principles like their solidness and test coverage, which should be considered to be able to co-operate well in a distributed environment especially from the architecture level. However, for the provider of a service, it might be challenging to maintain the obligations needed by the consumer. Therefore, in this approach a form of agreement is written by the consumer of the service, which is called contract. In this paper, a language agnostic, no-code approach is proposed for conducting the consumer-driven contract concept, through automatically converting the contracts into a set of solid test suites, that runs on each deployment. Consequently, if the service provider alters the returned data in a way that breaks the expectations of the consumer, this breaking change will be detected early by running the auto-generated test suites. The proposed approach also provides an insight for the provider, on how their service is being used and how changes can affect the consumers. Then it gets evaluated with a dataset of 56 open-source projects available on Github which use consumer-driven contract testing. The results demonstrate that for 56 real source projects, the proposed approach has been able to generate 450 tests from the contract files in the sources and 90% of the generated test suites passed.

Keywords: microservices, testing, consumer driven contract testing, test generation, mutation testing

I. INTRODUCTION

According to research on motivations on migration to microservice architecture (Taibi, Lenarduzzi, & Pahl, 2017), Maintainability and scalability were consistently ranked as the most important motivations, therefore, most of the newly-migrated systems need to invest their time and effort on maintainability. In the microservice world, the larger a system becomes the harder it is for service providers to know how their service is being used, and the reason is that it is more complicated with more people making changes to it. In a distributed environment, due to its distributed nature (Like transparency, size of the system etc.) understanding and reaching the actual problem, is difficult. One of the problems we are trying to address when we use integration tests is exactly this. Integration testing requires testing against the actual consumer, but a consumer-driven contract can accomplish such a task without testing against actual consumers. (newman, 2014)

A Contract is commonly defined as a set of different properties between the provider and the consumer of a service. In a broader term, Consumer-Driven Contract (CDC) can be thought of as a guarantee for the communication layer between services. Whenever the provider fails to satisfy the expectations defined by the consumer, the system can accurately exhibit what was the expectation and what is the actual returned value. In a way, this method ensures that any pair of consumers and providers can properly send and receive messages in a well-defined manner.

In comparison with the related works available in this area, the proposed approach has the following main advantages:**1) No obligation to know contract's syntax**: For creating a contract, one needs to get familiar with the syntax of writing a contract. But in the proposed approach this stage is done via interaction with the user interface. This can remove the need to know the basic knowledge. **2) Correct syntactical structure:** Because this method is generated automatically, there is no room for developer mistype and misuse to make any syntactical error. **3) Platform-Independence:** In other approaches, developer needs to execute project tests to verify the contracts. But here the verification is done independently via the stand-alone executable JAR file. **4) Language agnostic:** In current frameworks, because the contract is written and maintained inside the codebase, one should use different technologies for different programming languages for example there is Pact JS, Pact Ruby, etc. But here, there is only one machine generated contract for all available frameworks, as long as it has a REST API there is no boundaries. **5) Other extras:** The proposed approach, offers more features than those available in the market, for example timeout, and there is room for improvement in features as the framework evolves. The rest of the paper is organized as follows: Section II briefly discusses the related work about CDC and test generation techniques. In Section III, the proposed approach is introduced and explained in detail. Evaluation of the proposed approach is presented in Section VI, including result statistics of the evaluation. and finally, Section V concludes the paper by providing some directions for future work.

II. RELATED WORKS

Testing and verifying the integration among components of a software system, is a very vital type of testing. And it is said in (Fowler, TestPyramid, 2012), (Google, 2015) that we should

a sh.zahedi@mail.um.ac.ir
b s.paydar@um.ac.ir

Presented at the 2022 OkIP International Conference on Software Engineering Research & Development (SERD) in Oklahoma City, OK, USA, and Online, on October 3-6, 2022

spend most of the time on the integrations between the components. CDC helps on testing the integration between components. CDC testing has several advantages when considering to deal with complex systems with multiple components involved. A major benefit of this testing method is if the provider service is updated (Kohei Arai, 2021). According to (Lehvä, Mäkitalo, & Mikkonen, 2019) CDC test is also efficient of detecting integration defects. With all the benefits, as it's said in (Muhammad Waseem, 2021) the most rarely used strategy for testing microservices is CDC. In this approach the provider and consumer should consistently be in touch with each other, any breaking changes from consumer and provider is recognized early in the production, therefore with CDC the effort and the maintenance can be greatly reduced. Especially as the interaction between microservices evolves and the system grows bigger.

A. Mutation Testing

Mutation testing creates modified version of the program (in this case the contract) called *mutants*, it is done via *mutation operators* to simulate the fault or lead the tester to edge cases. It is expected for the test suites to fail, then it is said to be killed; Otherwise, the mutant remains alive and it means the generated tests were unable to detect the changes. (Alessandro Viola Pizzoleto, 2019) Which can be quite useful for assessing the productivity of our generated test. We leveraged mutation testing as an approach to assess the quality of the generated tests from the contract. Any mutation testing system represents a set of mutation operators, each of which is a variation of the generated tests that mimics the developer's mistakes. (Jacob Krüger, 2018)

B. End-to-End testing vs Consumer-Driven Contracts

End-to-End (E2E) testing has emerged in the last decade as a reliable and valuable technique the main objective of E2E was to make sure that the system has reliable and consistent behavior. (Cristian Mart´ınez Hern´andez, 2021) but it was facing some drawbacks, most of them are related to its nature. Running a full E2E test requires whole system to be up and running. These drawbacks can be enlisted as 1) being slow 2) easily breakable 3) E2E tests are expensive and hard to maintain 4) For some tests, it requires dedicated testing environment. CDC employs modern methods to reduce these drawbacks.

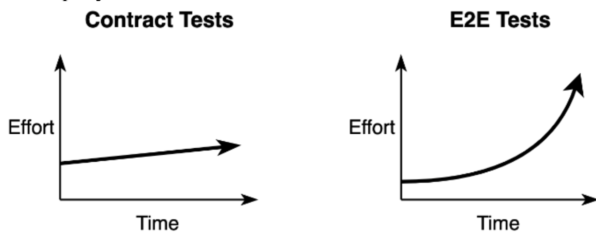

Fig. 1 Comparison between CDC and E2E , from *(Google, 2015)*

The testing pyramid is an idea, in which tests are classified based on their granularity and also the reasonable number of test suites. As it is shown in the testing pyramid (newman, 2014), there are various approaches to testing the microservices. The more it goes to the bottom of the pyramid, the number of tests increases, and as it goes up, the scope of tests increases. CDC tests lie exactly in the middle, which means they have both reasonable granularity and scope. Since microservices communicate over well-defined APIs, the concept of contract here can be applied very well. With the usage of CDC, we can test both sides of the service in isolation. With utilizing design-by-contract paradigm (Meyer, 1992). As Martin Fowler suggested in (Fowler, Consumer-Driven Contracts: A Service Evolution Pattern, 2006) contracts should be 1) **Closed and complete:** It should contain mandatory elements to support consumer's expectations. 2) **Singular and non-authoritative:** From the business functionality point of view, they should be singular and non-authoritative because they come from the union of existing consumer expectations (the provider does not know what consumer it's talking to). 3) **Bounded stability and immutability:** A consumer-driven contract should be stable in the sense that, we can determine the validity of a contract according to a specified set of contracts. And it should be immutable, meaning that the result cannot change with the manipulation of time and space.

Moreover, here we determine a set of characteristics in addition to Martin fowler's statements, 1) **Implementation agnostic:** With the right organizational setup, providers and consumers can talk about expectations without any implementation-specific knowledge. 2) **Quality characteristics:** usually for the teams. It is important to meet some important quality characteristics like latency and throughput. To measure provider's quality. 3) **Stateful:** Over the conversation between provider and consumer, the consumer might require the provider to remember its state. It is important to come up with a solution that can save the state rather than doing the whole process again to perform each relevant test. 4) **No adverse effect:** A test call to an endpoint may produce undesired harmful effects from the test environment. For example, suppose a contract is designed for a bank transaction environment. A test call should not perform any actual call to the core banking system which causes actual money transactions. At present, there is little literature providing a comprehensive view of different aspects of consumer-driven contract testing. However, currently, there is a number of frameworks like Pact and Spring Cloud Contract which allow the concept of CDC tests.

C. Available tools

1) Pact

Currently, Pact is the most widely-used framework for CDC, it is a code-first tool for testing HTTP and message integrations using contract tests. Pact uses its DSL to write contracts. Later it converts the written contract in whatever language (Ruby, JavaScript, Java, etc.) with its engine into test suites. Another solution it provides is PactBroker which solves the problem of maintaining the contract. If the provider and consumer use two separate repositories, PactBroker provides a mutual space for them. (Inc., n.d.) The pact framework does its work simply by following a set of steps. **Step one:** it sets up an HTTP mock server using a fluent API, then it runs all the tests. Once the running is completed, all the interactions are recorded and written to a contract file, called a pact. This pact file defines a

contract that provider and consumer must follow. **Step two:** It runs this pact file or the contract and gets a real response from a real provider. If the provider satisfies the pact file it passes, otherwise, we are facing a failure. (Alex Soto Bueno, How to Test Java Microservices with Pact, 2020) (Alex Soto Bueno, Testing Java Microservices, 2019)

2) Spring cloud contract

The Spring cloud contract is another contract testing framework provided by Sun Microsystems which of course is natively supported by JVM. It leverages Wiremock to apply stubs on the mocked server. It provides test capacities for messaging. Especially Spring related products like Spring AMQP, Spring Cloud Stream, etc. (Spring, 2020) Spring cloud contract (SCC) uses Rest Assured framework to send and receive REST requests. Here the contract is a pre-defined file either in Groovy, YAML, or Pact. In order to run the contract and validate it we need to set up the Spring Cloud Contract plugin and set the contracts in the proper folder on the provider side and when we trigger the build the plugin will read the contracts and generate test classes in the /contracts folder and then it generates stubs and puts in /stubs folder. These stubs will be packaged inside a jar file with the suffix stub.jar and the jar file will run in the package phase of a publish. Both SCC and Pact are essentially solving the same problem. The main difference is with Pact definition and validation of the contract is on the consumer side but SCC defines the contract on the provider side and if the provider validates then the contract gets published on the provider side.

III. PROPOSED APPROACH

As of now, the way a consumer-driven contract works is that after the contract is written, consumer-side developer tries to implement tests in their own language and framework to verify its loyalty to the contract. On the other hand, the provider needs to prove the same thing too. So, they create and maintain two different codes on potentially two different microservices. This paper provides an approach which is considerably less challenging to employ CDC tests. The way it works is that there are two separate executable files that can run on any machine with Java Runtime Environment one for the provider and the other for the consumer. Both take the same contract as input. The provider uses provider component only and the consumer does not need to have the provider component. The only tool it needs to run is the JRE. Then the consumer or provider can be started with a `java -jar consumer.jar` or `java -jar provider.jar` command. After running it asks for the contract and all the information will be loaded from the contract.

A. The contract

In the contract we define how the request and response's body, headers, cookies, request parameters, etc. should look like. In other sense a contract is generated from the expectations defined between the two parties. The most essential part of a contract is the "interaction" which contains a JSON array, each element represents one network call between the provider and the consumer. All the generated tests come from this definition. Each interaction consists of a request and response. TABLE 1 shows the content of a request, the proposed approach uses this structure to form a request to the mocked server and records the response.

TABLE 1 CONTRACT REQUEST STRUCTURE

Name	***Description***
Headers	List of key-values, Defines the request header
Body	Defines the body
Params	Put request params in the request
Cookies	List of key-values, sends request with cookies
Data	Plain-data, it can be anything of value
ParamRules	Set of rules to apply for request params
CookieParams	Set of rules to apply for cookies

TABLE *2* also demonstrates the definition of a response which is used to validate the recorded response.

TABLE 2 CONTRACT RESPONSE STRUCTURE

Name	***Description***
Headers	List of key-values, Defines the response header
Body	Defines the body
BodyRules	Set of rules to apply for the returned body
HeaderRules	Set of rules to apply for the returned headers

Each rule is a way to define exactly how the data should look like. And there are three type of rules that can be defined in a contract. *1) Contains:* means that it should contain the string and the generated tests will pass only if they contain such sequence of characters. *2) Matches:* it takes regular expressions and checks if the value satisfies the expression or not. *3) DoesNotMatch:* the generated test will only pass if the regular expression fails to satisfy.

B. The consumer

The consumer, is an executable JAR file (it only needs JRE to execute) and as it has been pointed out, the consumer takes a contract as an input, and according to the contract, it is going to run a mock server. On the consumer-side, a flexible tool for building mock APIs named Wiremock is leveraged. Wiremock helps with advanced request matching, dynamic response templating, recording responses etc. In the proposed approach there is a module named StubGenerator inside the consumer component and it is responsible to take an interaction and transform it to a Wiremock stub for the mock server. On the consumer-side, it also records all the interactions with the mocked server. When a network call to the mock server is taken place, it will print out all the details related to the call. This can help the developer to decipher, what exactly is happening with the network call. If it can find the called stub according to the contract, the server will return the desired output, otherwise, it will print out the nearest stub match that can be found on the server. Finally, after running the mock server, the consumer

uses the test generator module to generate tests from the contract. These tests need to be executed to demonstrate that the system is doing okay. After running the tests, it shows how the mock server responded to those tests. This feature can come in handy when there was something wrong with the mock server. It can be discovered earlier in the production before executing the tests. Fig. 2 demonstrates with more detail as UML activity diagram.

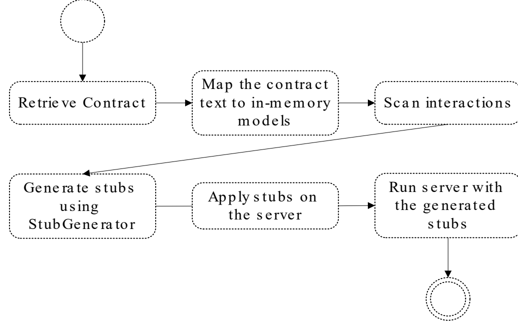

Fig. 2. Consumer-side workflow

Fig. 3 represents the message in the console, for when the mock server is ready. It shows exactly the local and external address of the mock server generated for the user.

```
Mock server is running! Access URLs:
Local:      http://localhost:8080/
External:   http://127.0.1.1:8080/
Press CTRL+C to stop
```

Fig. 3. mock server ready console message

C. The Provider

This component also is an executable JAR file and it takes the contract as an input. It then maps the content of the contract to an object, which is ready for test generation. This module scans each specification of the contract and then generates a number of callbacks. Each of which is a test to be executed. It then executes the network call with the desired specifications only once. And then it creates different callbacks on the response. The result of every callback execution is either true or false which indicates the result of the test also known as Assert. Fig. 4 illustrates a workflow of the provider.

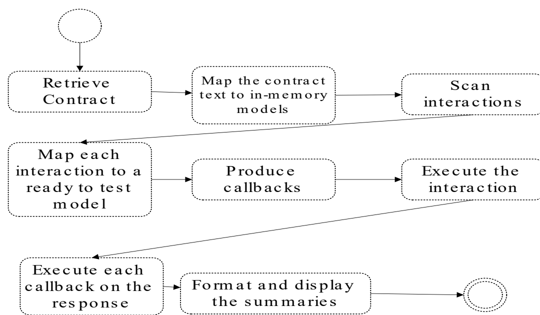

Fig. 4. Provider-side workflow

If the assertion was false the framework shows a detailed message on how the expectations were and narrows down the value of the actual response. In the end, it provides a summary of how many tests are executed and some useful information on the mutations.

IV. The Testing Process

A. Utilizing Continious Integration/Continuous Delivery

An attractive aspect of such approach is the automation. As the software evolves, each time the code base alters, without the CI/CD it needs manual deployment and manual running of each test to make sure that the new change did not break anything. CI/CD facilitates this process down to a single click after making any changes in the project. Most CI/CD platforms contains a series of stages (Sriniketan Mysari, 2020). Each of the stages includes doing part of the deployment and integration job. Thus, it can be said that the most essential use of such approach is, as a CI/CD stage. The idea is that after the application is completely up and running, we can start running contract tests automatically. If any of the tests did not pass, we do not proceed to the next stage of CI/CD.

This way we can ensure that with every deployment, there is healthy code which is capable of satisfying the contract. As it is demonstrated in Fig. 5 it can provide a summary, to show the maintainer, the statistics on the tests. It can be seen in the CDC SUMMARY section how many tests the proposed approach has generated from the contract and how good was the new code, with satisfying the contract. Also, a summary of mutations is provided below, which can help the user to know the quality of the generated tests.

```
----------------------------------------------------------
Application is running! Access URLs:
Local:      http://localhost:8081/
External:   http://127.0.1.1:8081/

---------------- CDC SUMMARY ----------------
Total 3 test cases executed
3 tests passed
0 tests failed
100% of tests were passed


Mutations Executed: 91
Mutations survived: 11
Mutations killed: 80
Mutation score: 87
```

Fig. 5. Jenkins CI/CD platform, contract test execution

B. Mutation testing

For assessment of the generated tests, we utilized mutation testing. Here, mutation testing is a form of testing in which we change specific components of the contract to ensure the generated tests will be able to detect the changes. These changes in the contract are intended to cause errors in the test suites. If they don't, we will know that the test suite is weak. When all generated tests pass, we mutate different parts of the contract, and then we generate tests from the new mutated version. Then we run all mutations. It is important to note that just like the test suites, the web call is made only once and all mutations will run on the recorded request and response.

1) Mutation Engine

There is JSON structure involved in several places in the contract for example body, headers, and cookies. We proposed and implemented an engine that is able to manipulate JSON structure in various ways. Mutation operators perform the better part of their job by leveraging this engine. In TABLE *3* we'll describe each operator involved to manipulate the JSON.

TABLE 3. JSON MANIPULATOR'S ENGINE OPERATIONS

Name	***Before***	***After***
Insert null to array	[1 , 2 , 3]	[1 , 2 , 3, null]
Empty replace	{ "x": "hello"}	{ "x": {}}
Remove an item from array	[1 , 2 , 3]	[2 , 3]
Add empty object	[1 , 2 , 3]	[1 , 2 , {}]
Remove a pair	{ "x" : 0, "y" : 1 }	{"x":0}
Change order	{ "x" : 0, "y" : 1 }	{ "y" : 1 , "x" : 0 }
Change string	{ "x" : "a" }	{ "x" : "b" }
Change numbers	{"y" : [1,2,3]}	{"y":[99,2,3]}

2) *Mutation Operators*

The way mutation testing is performed is with the mutation operators. Each operator acts as a tool to alter the contents of the contract. Some of these operators engage the mutation engine mentioned in Mutation Engine section. some of them are mutated with other methods. For example, the status code mutation is implemented via changing the status code with a list of available HTTP status codes. In TABLE *4* a definition of each operator is listed.

TABLE 4 DEFINITION OF EACH MUTATION OPERATOR

Name	***Definition***	***Involves Engine***
Status code	Replaces response status code with another code.	No
Response body	Applies minor adjustments to the response body, if it's defined	Yes
Response header	Alters the response header in the contract, if there is one	Yes
HTTP Method	Changes the HTTP method	No
Params	If there's query parameters in the contract, it mutates them.	Yes
Cookies	If cookies are defined in the contract.	Yes

3) *Equivalent Operators*

One of the challenging aspects of producing mutation is the equivalent operators. An equivalent mutant always produces the same output as the original program; hence it cannot be told apart from the original program. For example, on the header part of the contract. It will make no difference on any server. If you add an extra header to the request, the response will be the same without any regard to the extra header added as a mutation.

In the proposed approach some of the survived mutations were directly the result of this concept. Causing the false belief in the results of the mutations, that the generated tests are not in a good quality.

V. EVALUATION

For examining the performance of the proposed approach, in various contexts and different projects, we have collected contracts from real-world open-source projects on Github. Here is how the process went: 1) Hundreds of projects implementing the CDC have been accumulated. We did this step with utilizing a section on Github that shoes what other repositories are using the current repository. 2) With doing this a large number of projects gathered. Then, the projects with the most Forks and watches on Github were collected. 3)Then, all the repositories have cloned and with a bit of exploring in the source code, contract files for SCC and Pact were collected. 4) Some contracts were directly supported and some needed to be converted manually, to a contract that can be read by our framework. 5) Then, all contracts are executed, one by one and all results are recorded on separate text files. 6) Then the recorded results counted and summarized with the help of regex.

As it is exhibited in the Fig. *6*, the way it's executed is by utilizing both consumer and provider. First, we give the consumer the contract, and it loads up and generates stubs to make a mock server. Once the mock server is up and running, the same contract is given to the provider. After running is finished, the results will be examined carefully. By using this idea, theoretically, if the system is doing okay, all the tests should pass and all the mutations should be killed. But in the real world, it didn't happen, which will be elaborated in the Test generation section and Fig. 7.

A. *Results*

After the selection phase of the open-source projects completed, the tests get classified and executed. Among the selected projects there were two types of them, one is mostly used for tutorial and practice purposes and the other, which is real world project with real-world use. Most of the projects (about 62%) were real-world projects, and the rest were example demonstration projects.

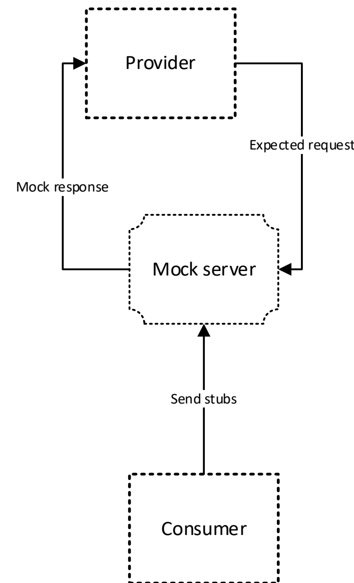

Fig. 6. The idea for evaluating the approach

1) Contracts

We could discover 157 contract files out of 56 projects in the first selection phase. After execution is completed, 104 of the contracts has made their way to the next phase. And 56 contracts were rejected in the execution phase for various reasons. The reasons include unsupported features 14% and invalid selection[1] 61%.

2) Test generation

Among the contracts described in the Evaluation section, 105 different contracts from available projects have been collected. On average, it could generate about 3 tests per contract. And not all of the tests passed, 89% percent of the generated tests passed, and the rest failed for various reasons, which will be highlighted in the **Fig. 7**.

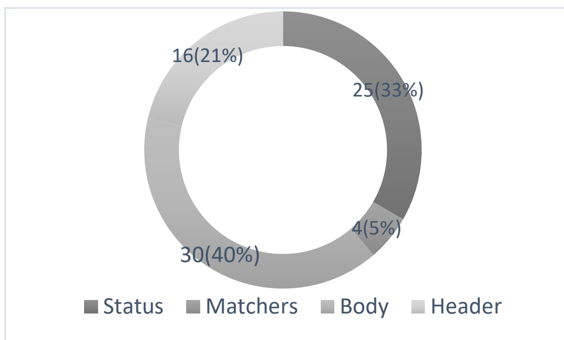

Fig. 7. Statistics on failed tests

3) Mutation Tests

On average, 93 mutants were generated per contract, and while the results were almost similar to generated tests,83%of them have passed. And the reason for 17% failure can partially be the equivalent operators.

4) Mutation Tests

Judging by the number of survived mutants and where the survivors mostly are, it can be said that most of the equivalent operators lie in the header operators, which is well expected. Adding a new key value to the header cannot make any tests fail therefore, it cannot make any mutations killed, as it happened 1927 times for the header and only 77 times for the other parts.

VI. Conclusion and Future Work

Implementing a CDC framework is exhausting work since various aspects need to be taken into account. And since there are very few studies available in this context, it is a bit difficult to find comprehensive information on this matter. In this paper, a novel approach to implementing CDC proposed and its various components have been discussed, later it's evaluated with several open-source contracts available on Github based on their reputation.

In the future, we plan to finalize the proposed framework, since further research is needed to make a comprehensive CDC framework. The priority would be to reduce the number of failed tests mentioned in the Results section. The other priority is to achieve what's mentioned in Related Works section, whichis saving the state and abolishing the side effects of the web call. And the final priority would be to add new features like various 'like' operators or supporting file interactions and also better integration with CI/CD frameworks like Jenkins etc. Finally, another valuable piece of research that can be done here is to add a user interface to the process of writing a contract. With the UI a user can write contracts without the hassle of knowing the syntax of the contract and it can do so without any syntactical errors.

VII. References

Alessandro Viola Pizzoleto, F. C. (2019). A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *The Journal of Systems & Software*.

Alex Soto Bueno, A. G. (2019). *Testing Java Microservices.* New York: Manning.

Alex Soto Bueno, A. G. (2020 , April 06). *How to Test Java Microservices with Pact.* Retrieved from https://blogs.oracle.com/javamagazine/post/how-to-test-java-microservices-with-pact

Cristian Mart´ınez Hern´andez, A. M.-L. (2021). Comparison of End-to-End Testing Tools for Microservices: A Case Study. *Information Technology and Systems*, *1.*

Fowler, M. (2006, June 12). *Consumer-Driven Contracts: A Service Evolution Pattern.* Retrieved from https://martinfowler.com/articles/consumerDrivenContracts.html

Fowler, M. (2012, May 1). *TestPyramid.* Retrieved from https://martinfowler.com/bliki/TestPyramid.html

Google, M. W. (2015, April 22). *Just Say No to More End-to-End Tests.* Retrieved from https://martinfowler.com/bliki/ContractTest.html

Inc., P. (n.d.). *Pact Documentation*. Retrieved from https://docs.pact.io/

Jacob Krüger, M. A.-H. (2018). Mutation operators for feature-oriented software product lines. *Software: Testing, Verification and Reliability, 29*(1-2), 21.

Kohei Arai, S. K. (2021). On Testing Microservice Systems. *Proceedings of the Future Technologies Conference (FTC) 2020, Volume 3.*

Lehvä, J., Mäkitalo, N., & Mikkonen, T. (2019). Consumer-Driven Contract Tests for Microservices: A Case Study. *Product-Focused Software Process Improvement : 20th International Conference.* Barcelona, Spain.

Meyer, B. (1992). Applying "Design by Contract'. *Interactive Software Engineering, 25*(10).

Muhammad Waseem, P. L. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. *The Journal of Systems & Software*.

newman, S. (2014). *Building Microservices desinging find grained systems.* O'Reilly.

Spring. (2020). *Spring Cloud Contract*. (Sun Microsystems) Retrieved from https://spring.io/projects/spring-cloud-contract

Sriniketan Mysari, V. B. (2020). Continuous Integration And Continuous Deployment Pipeline Automation Using Jenkins Ansible. *International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE).* Engineering and Science.

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing, 4*(5), 1083 - 1099.