Behavior-Driven Development of Microservice Applications

Tugkan Tuglular, Deniz Egemen Coşkun, Ömer Gülen, Arman Okluoğlu, Kaan Algan
Izmir Institute of Technology,
Urla Izmir, 35430
Turkey

Received: July 22, 2021. Revised: October 8, 2021. Accepted: November 15, 2021.
Published: December 9, 2021.

Abstract— As the number of microservice applications rises, different development methodologies for them are under consideration. In this manuscript, we propose a behavior-driven development method for microservice applications. The proposed method starts with writing end-to-end tests at the system or application level and then moves down to the microservice level, where component and unit tests are written. Next, code that passes these tests is developed one by one for each level. Once user stories are covered, our method loops again to integrate negative tests to achieve holistic testing for the microservices and the application. Finally, the proposed method is validated with an application with five microservices. Results confirm that the proposed method matches with the generally accepted test pyramid.

Keywords—behavior-driven development, micro-services, test-driven development.

I. Introduction

THE development of microservice applications using Behavior Driven Development (BDD) approach is a yet undiscovered concept. BDD focuses on defining fine-grained specifications of the behavior of the targeted software application [1]. However, as more and more applications are developed in microservices and BDD is gaining popularity, we were interested in combining the two and discovering if BDD is suitable for microservice application development. Towards this end, we first set out a generic microservice internal design and interface design and then defined the type of tests that a microservice application should pass. Afterwards, we defined the relationship between these tests and the generic microservice design. Finally, we developed a BDD method for microservice applications. For evaluation, while developing a microservice application with the proposed BDD method, we measured development times for code and tests and test coverage. The implementation of the proposed BDD method is publicly available at https://github.com/segment17.

This paper defines test types used for microservice applications. They are unit tests component tests, and end-to-end tests resembling a test pyramid with unit tests at the bottom and end-to-end tests at the top, as shown in Fig.1. There are also integration tests and contract tests needed to be written. We consider these tests within unit tests because they dependent on unit tests and they should be written after unit tests.

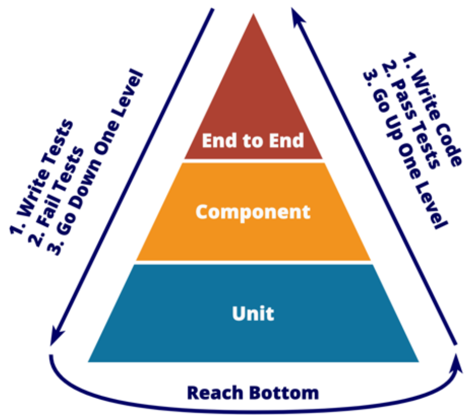

Figure 1. Test pyramid applied to microservices.

The paper also presents a novel BDD implementation of microservice applications using these test types following a test-first approach. The proposed BDD method defines each phase of the software development lifecycle for microservices concerning this test pyramid. It starts with writing the end-to-end tests, component tests, and unit tests in Gherkin. As shown in Fig. 1, the proposed approach first goes from top to bottom of the pyramid through writing tests, which fail since there are no code, level by level. When we reach the bottom, we write the code that will pass these unit tests. Now, the unit tests pass, and we start to move up in the pyramid. Next, we write code that will pass the component tests since the component tests have already been written, and unit code is now available.

Once we clear the component level, we move up to the utmost level and we write the code that will pass end-to-end tests. Finally, at the top of the pyramid, we have all tests and the code ready. With this approach, the developers have always clear vision what to code. Since it is impossible to come up with all tests in one iteration, we expect more iterations until the tests and the code are done. This is what we experienced with our implementation of the proposed BDD method.

The proposed BDD method for microservice applications also defines a generic test writing process utilized at every level of the software development lifecycle for microservices. The generic test writing process starts with developing Gherkin scenarios and their test code, and time for this development is measured. Once the code is completed and all tests at that level are passed, development time for code and test coverage are measured and recorded. If test coverage is below the set value, which was 80% in our case, more tests are written. Measurement of test and code development times continues until the expected test coverage value is achieved. Once we pass the expected test coverage value, then negative tests are written to develop a holistic test suite along with the code that passes these negative tests.

Negative tests describe how the software shouldn't behave, whereas positive tests in user stories define how the software should behave. To pass negative tests, more code should be written until the expected test coverage value is reached. The development times are again measured and recorded. These measurements are valuable in software engineering for two reasons, namely for intra-project and for inter-project comparisons and predictions. We utilized intra-project predictions in the development of our microservice application and here we provide intra-project comparisons in the evaluation section.

The paper is organized as follows. The following section gives the background information about the concepts used in the paper. It describes BDD, Gherkin, Cucumber, and microservices. Section III explains the proposed method followed by its implementation on a case. Section V presents an evaluation of the method approach along with a discussion. Section VI outlines related work, and the last section concludes the paper.

II. Fundamentals

A. Behavior-Driven Development

Historically, behavior-driven development followed test-driven development (TDD). TDD states that tests should be written before code, such that tests become the stopping criteria for writing code [2]. If the code passes all tests, coding ends. Therefore, test scope and coverage have a critical role in software development. TDD is usually applied at the method or class level with unit tests. Since software developers write unit tests and code, TDD excludes customers, analysts, and testers from the implementation process. With the idea of agile development, the inclusion of these parties into the process became very important. BDD became one of the solutions. In BDD, analysts with customers, testers, and even developers collaborate to define what software should do in terms of software behavior. Application of BDD to software development does not impose any order such as test-first or code-first.

B. User Stories

User stories are the starting points of Behavior Driven Development. User stories usually have the "As a role, I want to action, so that value" format. User stories are expected to be written by analysts, customers, and testers together. User stories gave a general idea of what the software should do but the details are left to acceptance criteria, which can be written in natural language or in a language like Gherkin with a preset template. In this work, we used Gherkin to define acceptance criteria. Gherkin uses a set of special keywords to give structure and meaning to executable specifications [3]. Gherkin is a line-oriented language in terms of structure and each line must be divided by the Gherkin keyword except feature and scenario descriptions [3]. These lines are called step.

C. Microservice Applications

Before microservice architecture, monolithic architecture [4] was the primary style for application design. In monolithic architecture, the application runs as a single service; therefore, is tightly coupled and calls for complex development processes as it grows more and more. When a part of it gets a spike, the whole system needs to be scaled, and when a part of it fails, it is more likely that the whole system fails. Furthermore, when the system needs maintenance, it takes more time and effort to apply it due to its dependencies and complexity. In contrast to monolithic architecture, microservice architecture offers a system that solves these issues.

Microservice architecture [5] is an architectural style in which the application is designed as a collection of services. These services are modular, self-contained, loosely coupled, and therefore easier to maintain and test. This approach provides flexibility and agility for the scaling and development of the application. Each service is free to choose any technology for its implementation and can be run independently from each other. One service's failure does not entail another service's failure, making the system much more resilient. With tools like Kubernetes, it becomes easier to orchestrate the services in the cluster so that well known DevOps issues, such as when and which services to scale up or down, which portion of the services should use a new version of a service, or rollback the release on a failure, are easier to resolve.

III. Proposed Method

A. Gherkin Step Definitions

As stated previously, every line specifying a setup or an action in Gherkin is called a step. The functions they are mapped to are called Gherkin step definitions. We used the Cucumber tool to make this mapping. When a Gherkin test is executed using Cucumber, each step's corresponding step definition function is being run in the order that they appear in the Gherkin feature file. The mappings are made by the word content of the step, so it is impossible to map two exact steps in different feature files to different functions. This approach is a plus rather than a drawback because it increases reusability. Multiple steps with exact wording appear across feature files in our work, but they all run the same function.

We used tester classes to implement step definitions. We called these classes ScenarioTester. Gherkin feature files also allow us to use multiple tags to mark features and scenarios. We have utilized these tags to specify the type of test the feature file is, such as unit, component, and end to end, to quickly subclass ScenarioTester and set up the environment and step definitions accordingly.

B. Interface Between Microservices

The proposed architecture depends on Kubernetes and Docker. In the deepest layer, we have the microservice (explained with detail in the next section) which runs inside a Docker container. We have dockerized our microservices to easily upload and run inside Kubernetes using Kubernetes manifest files as shown in Fig. 2. When the dockerized code is pushed to the DockerHub, we do not need to carry code from our repositories to Google Cloud, instead a single line in our Kubernetes manifests downloads the code from DockerHub automatically Our Kubernetes cluster is set up inside Google Kubernetes Engine inside Google Cloud and they use gRPC protocol for in-cluster communication.

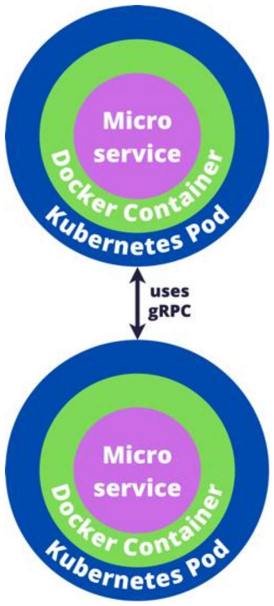

Figure 2. Interface between microservices.

C. Generic Microservice Design

We designed our microservices in four layers, namely Controller, Service, Gateway, Repository. The controller is the facade of the microservice. When a request comes to the microservice via a network, it is directly forwarded to the controller. In our controllers, the first function that a request comes to is called a guard function. We chose this naming because those functions' primary responsibility is to primitively check the validity of the data in the request body. These checks can be about whether all required fields are filled, all data types are correct, and within logical boundaries. If all the simple validations are passed, the controller forwards the request data to its service layer. The other use of the controller happens after it receives the result from the service layer. Depending on the success or failure of the response, the controller creates a network response object from data with the appropriate response code and message. These response object formats are defined in the microservice's contract tests. This concept is explained below.

The service layer is where the coordination of the logic is implemented. Upon receiving the request data from the controller, the service layer creates the necessary domain objects, fetches, or saves data using database via repositories, fetches data, or makes mutating calls using other microservices via gateways. In a simple microservice, the whole service layer can be implemented as one mediator class. A domain object refers to a collection of data and functions concerning one logical entity used inside the microservice. A domain object may be implemented as an object-oriented program.

Repositories are the layers between service layer and the database used by the microservice. The primary use of repositories is to separate database access from the service layer code. The coder of the service layer should not know how to write SQL/NoSQL or even know which type of database is being used. The repository and service layer coders should agree upon an interface required to satisfy the service layer’s needs. Each repository usually concerns one domain object and has functions aiding in its creation, read, update, or delete (CRUD) operations. However, repositories can have functions that concern more than one domain object in situations that require complex cross-table querying. That said, wholly separated functions concerning CRUD operations of two different domain objects should not be in the same repository. Every domain object that requires data persistence should have its repository. Mappers inside the repository handle the formatting of data. For example, a mapper may be designed to read SQL query results and create the appropriate Java objects or vice versa. Mappers can be classes, or in small environments, they can be simple functions.

Gateways are very similar to repositories by their layer separation characteristic. Gateways are the layer between Service Layer and other microservices. Their interfaces are defined to fulfill the needs of the service layer, whether it requires reading data from another microservice or needs to trigger a change in another micro-service. The coders of the service layer do not know the contract between this microservice and the other microservice. They do not need to know how to make requests. They only call the gateway's asynchronous functions, getX(), without considering that X is not even in the same microservice. Every microservice accessed from this microservice needs to have its gateways.

D. Generic Test Design

In the proposed method for BDD of microservice applications we utilized five type tests, namely unit tests, integration tests, component tests, contract tests, and end-to-end tests. Their relationship with microservice layers is shown in Fig. 3.

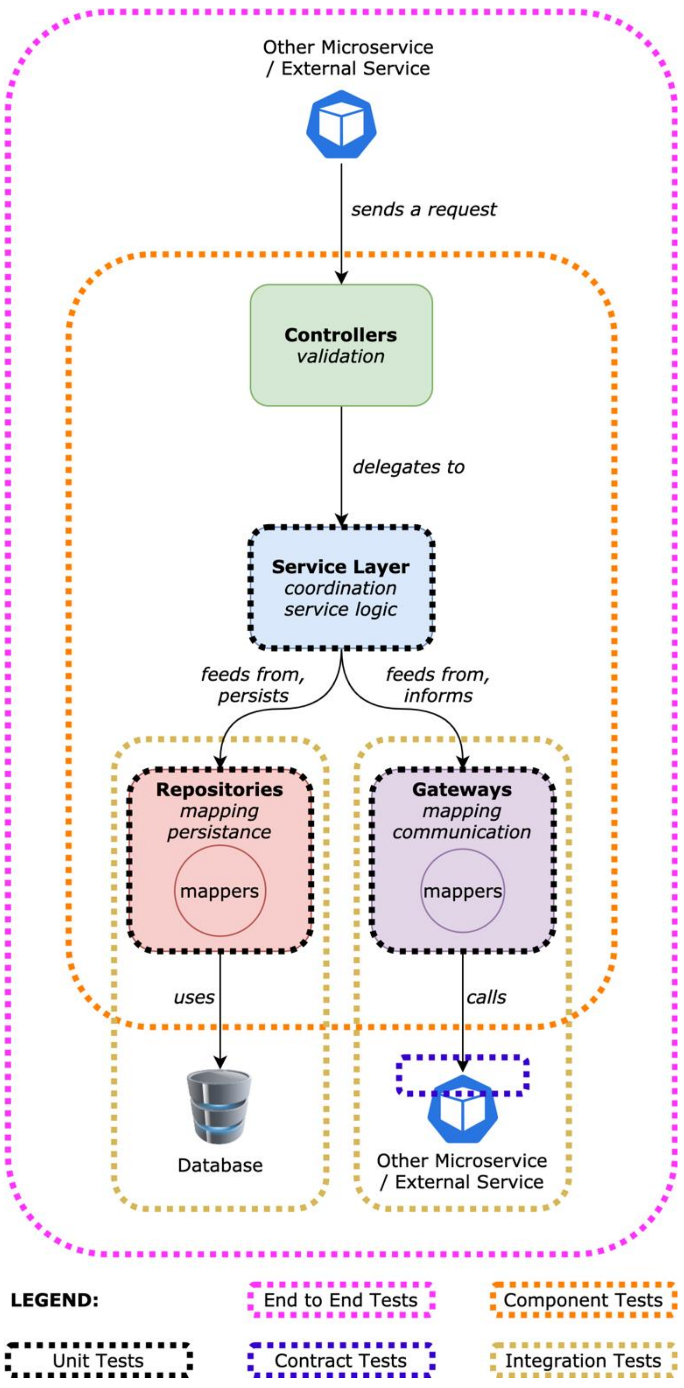

Figure 3. Test types applied to microservice parts.

Unit tests are created for each layer. They can be written for a single method or a single class. The tester/developer decides how fine-grained a unit test should be written. There is one caveat about writing unit tests for the controller. Since the controller is acting as the facade of the microservice, the unit tests for the controller can be skipped as component tests also do the same tests. In our architecture, unit tests use mocks as dependencies of repositories and gateways. For example, if we are writing a unit test for a function of MatchServiceGateway of BoxerService, we should use the MockMatchService Gateway class designed as a subclass of MatchService Gateway. The only change in this subclass is that it replaces the real network call and response with mocked in-code data. This way, while developing BoxerService we can act as if the MatchService exists even though it does not.

Integration tests are derived tests for repositories and gateways. They derive from unit tests. In our design, they share the same test steps. The only difference is that while they are being executed, the repository/gateway is not mocked and makes actual calls outside the microservice.

Component tests are written for the endpoints of the microservice. They make a request and expect a response and check the changes in the repositories if necessary. Component tests do not require the implementation of anything outside the microservice. Instead, they use mock repositories and gateways. That way, even if the database is not set up or the other microservices are not written, we can almost wholly finish implementing and testing our microservice, and as the external dependencies get ready, all we must do is fill in a couple of real-world call functions.

Contract tests are derived tests for the other microservices. The critical point here is that developers and testers of one microservice write the contract tests for other microservices that they access. They specify what kind of data they expect from some endpoints of that microservice. Our architecture has derived these tests from gateway integration tests because a gateway integration test for our microservice includes our expectancies from another microservice.

End-to-end tests are derived tests for the component. They derive from component tests. In our design, they share the same test steps. The only difference is that while they are being executed, the repositories and gateways are not mocked and make actual calls outside the microservice. In cases where an end-to-end scenario covers multiple microservices and needs to check the changes in other microservices' databases or states, they include extra steps that reach outside the microservice.

E. Proposed BDD Method for Microservice Applications

Conventional BDD suggests the following workflow [6]:

1. Identify business features and related user stories
2. Define scenarios and acceptance criteria for the feature
3. Determine steps per scenario
4. Write failing test steps for unimplemented feature
5. Write code to pass the test steps and
6. Refactor the code
7. Produce reports

The above workflow does not address the test types and their implementation. Moreover, it does not consider the application architecture. In this work, we extend this workflow for microservice architectures and for five test types that we consider necessary to test a microservice application.

Our proposed extension creates another workflow, namely microservice BDD workflow. The E2E tests are for system level, and the remaining four test types are for microservice level. Although our system level BDD workflow matches the conventional BDD workflow, the microservice BDD workflow is different and novel. This microservice BDD workflow explained in Section I takes the generic microservice design and its coupled test design explained in Sections III-C and III-D, respectively, into consideration.

For reporting, we take two measurements for every level in every microservice: the duration to complete tests and code and the test coverage rate. We will make measurements in two levels: microservice level relating component tests and system level relating end-to-end tests. The microservice level checks for the validity and completeness of a single microservice, while the system level checks for end-to-end validity and the micro-services working together. We separate each level into two milestones; one when a user story is completed and the second when a holistic test suite is achieved. The holistic test suite, where all lines of the code are covered for positive and negative tests, is expected to result in 80% or more test coverage.

IV. Application of Proposed Method

We applied the proposed BDD method and developed the Unlimited Boxing Championship application with four domain microservices and a front-end microservice, which are shown in Fig.4. The domain microservices are Authentication Service, Boxer Service, Standings Service, and Match Service. The application is a boxing championship information website with two tables filled with matches and boxers' standings on the homepage. There is also a login page for the purposes of getting admin privileges and a boxer details page which shows information about a boxer and that boxer's matches/standing. Admin can create/edit/delete boxers and matches and these are performed with modals on the front-end. The user stories are listed at https://github.com/segment17/ubc/tree/master/stories and their mapping to Cucumber features are given at https://github.com/segment17/code-statistics due to space limitations.

To show the complexity of the implemented application, we present uses relationships of each microservice. The FrontEnd microservice uses other microservices as follows:

- AuthService to log in an admin and store the token to send for requests that re-quire admin access.
- BoxerService to show the list of boxers or the details of a boxer or any mutating operations from admin.
- MatchService to show the list of matches or the details of a match or any mutating operations from admin.
- StandingsService to show the standings (leaderboard) of all boxers.

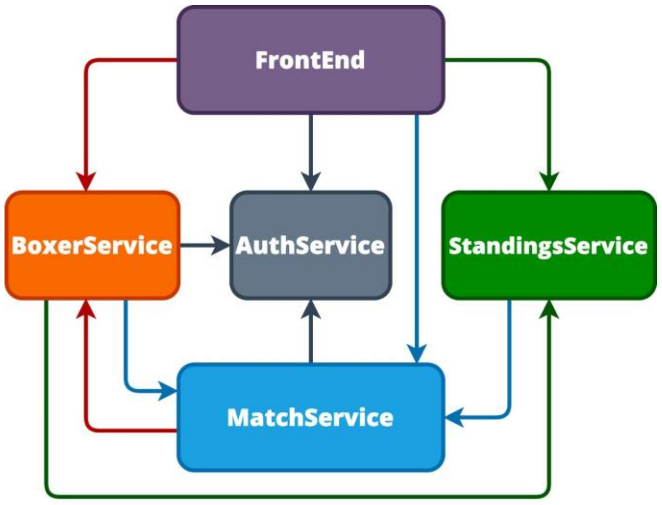

Figure 4. Microservices of Unlimited Boxing Championship application.

The BoxerService microservice uses other microservices as follows:

- AuthService to validate the token in operations such as: add boxer, update boxer, delete boxer.
- StandingsService to get standing and score information as well as match data of a boxer.
- MatchService to delete the matches of a boxer upon deletion of the boxer itself.

MatchService microservice uses other microservices as follows:

- AuthService to validate the token in operations such as: add match, update match, delete match.
- BoxerService to validate boxers exist before adding a match that involves them.

StandingsService microservice uses other microservices as follows:

- MatchService to fetch all matches to create the leaderboard.
- MatchService to fetch the matches of the boxer to decide its score and win/lose count.

To develop microservice-based Unlimited Boxing Championship application, we utilized Docker, Kubernetes, and Ambassador Edge Stack for infrastructure, gRPC for remote procedure calls, Envoy for edge and service proxy, MySQL for storage, and React, Material-UI, and Protoc (grpc-web) for frontend.

V. Evaluation

During the development of the Unlimited Boxing Championship application, we continuously measured test coverage at system level for end-to-end tests and at microservice level for component tests. When we reached above 80% code coverage for holistic test suite, we stopped development. The coverage values for end-to-end tests and component tests are presented in Table I.

Results in Table I confirm the test pyramid in Fig. 1. To pass tests at the highest level, the tests below need to have more code coverage.

Table 1. Test coverage for each microservice

Microservice	*End-to-End Test*	*Component Tests*
AuthService	82.93	95.12
BoxerService	83.15	87.64
MatchService	89.91	93.58
StandingsService	95.56	100.00

We measured user story and holistic test suite completions as explained in Section III-E. The measurements for each user story from M1 to H2 are given in Table II in Appendix. In the table heading, under user story identifier, we also indicated the microservice, where part or whole of the user story is implemented. For instance, M1 user story is wholly implemented in the MatchService whereas B1 user story is partly implemented in BoxerService and partly in StandingsService. From user's point of view, it is not important where the user story is implemented.

Table II in Appendix also shows the consolidated total development time and corresponding number of scenarios, number of steps, and step run counts for each user story. The formula for step run count is given below:

$$step\ run\ count = \sum_{s \in scenarios} s_{number\ of\ steps} \times s_{number\ of\ examples}$$

The total step run count is calculated as follows. For each Gherkin Scenario, the number of steps is multiplied by the number of examples, i.e., test data, since the steps should be executed for each test data once. To observe the relationship between step run count and total development time we draw the trendline between them as shown in Fig. 5. They appear highly correlated.

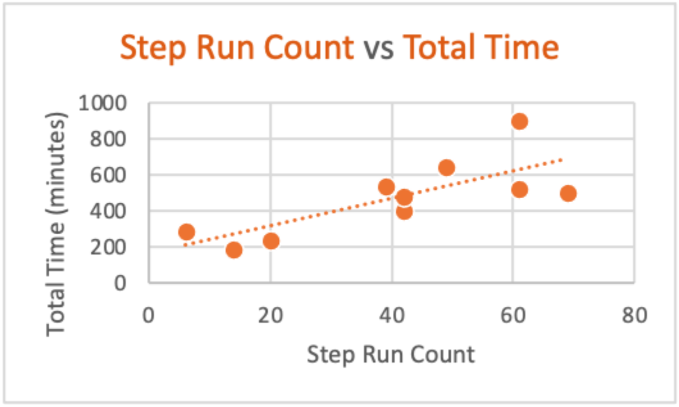

Figure 5. The relationship between step run count and total development time.

The configuration of the computer used for measurements is as follows:

- Model: MacBook Pro 13-inch, 2020
- Processor: 2.3 GHz Quad-Core Intel Core i7
- Memory: 16 GB 3733 MHz LPDDR4X

Since our evaluations are only based on a single microservice-based application, the evaluations we have performed may not be representative and generalized to all microservice-based applications. The proposed BDD method for microservice-based applications may generate different results in different architectures and designs. Moreover, another threat to validity is related to the authors since the authors developed the microservice-based application.

Another threat to validity is the technologies we utilized in software development. We utilized them to the best of our knowledge. There might be better utilization. The same threat to validity applies to the software we used for measurement in evaluation. Moreover, the computer and the operating system, where we took measurements, may be under the effect of some background processes at the time of measurements.

VI. Related Work

According to Wirfs-Brock [7], BDD is set forth to encourage incremental design by writing small behavior-driven specifications, then implementing code that works according to the specifications. The key to success of BDD is the executable acceptance tests that describe the expected behavior of a feature [8]. Behavior-driven specifications have been written in executable forms to be run by Cucumber [9] or Gauge [10]. Existing BDD tools were evaluated in [11]. The tools are categorized with respect to in which type testing they are used as well as with respect to programming languages, such as Java and C#. There are also approaches and tools to automate test generation in BDD, such as [12] and [13]. Bob and Storer developed a tool called developed behave_nicely, which automatically generates step implementation functions from Gherkin [12]. Tuglular and Şensülün extended Gherkin to automate test generation for software product lines [13].

With agility gaining popularity, BDD is applied microservice applications. Rahman and Gao [8] present a reusable automated acceptance testing architecture to address reusability, auditability, and maintainability concerns raised in applying BDD to each microservice. They claim that they propose the first approach addressing these issues.

Zampetti et al. [14] asserted that the availability of frameworks such as Cucumber makes the application of BDD possible in practice. However, they claimed that it is unclear to what extent developers use such frameworks, and whether they use them for performing BDD. Their study showed that approximately 27% of the sampled projects use BDD frameworks. In about 37% of the cases, they found a co-evolution between scenarios/fixtures and production code.

Aghayi et al. [15] proposed a crowdsourced development workflow called CrowdMicroservices for microservices based on BDD. In their proposal, the workflow starting point is the description of the microservice as a set of endpoints with paths, requests, and responses. Then, a crowd implements the endpoints, identifying individual endpoint behaviors that they test, implement, debug, create new functions, and interact with persistence APIs as needed. Our method differs from this workflow in two respects. First, we start with user stories and Gherkin step definitions, whereas CrowdMicroservices start with endpoints with paths, requests, and responses. Second, we write executable tests first and then the code, whereas, in CrowdMicroservices, code is written first and then the tests.

VII. Conclusion

We propose a behavior-driven development method for microservice applications. The proposed method starts with writing tests at the system level and microservice level. Then, code that passes these tests is developed one by one for each level. Once user stories are covered, our method loops again to integrate negative tests to achieve holistic testing for the microservices and the application.

The proposed method is validated with an application with five microservices. We developed this application using the proposed method and reported development time and test coverage measurements.

REFERENCES

[1] M. G. Cavalcante and J. I. Sales, "The Behavior Driven Development Applied to the Software Quality Test," 2019, pp. 1–4.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code Addison-Wesley Professional," *Berkeley, CA, USA*, 1999.

[3] "Gherkin Reference," 2019. https://cucumber.io/docs/gherkin/reference/

[4] P. K. Garg and M. Jazayeri, "Selected, annotated bibliography on process-centered software engineering environments," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 2, pp. 18–21, 1994.

[5] F. Andre and M.-T. Segarra, "A generic approach to satisfy adaptability needs in mobile environments," 2000, pp. 10-pp.

[6] A. Stec, "Understanding BDD," Feb. 23, 2021. https://www.baeldung.com/cs/bdd-guide (accessed Sep. 25, 2021).

[7] R. J. Wirfs-Brock, "Driven to... discovering your design values," *IEEE Software*, vol. 24, no. 1, pp. 9–11, 2007.

[8] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," 2015, pp. 321–325.

[9] "Cucumber Reference," 2019. https://cucumber.io/docs/cucumber/api/

[10] "Gauge: Less Code, Less Maintenance, More Acceptance Testing." https://gauge.org (accessed Sep. 27, 2021).

[11] R. K. Lenka, S. Kumar, and S. Mamgain, "Behavior driven development: Tools and challenges," 2018, pp. 1032–1037.

[12] T. Storer and R. Bob, "Behave Nicely! Automatic Generation of Code for Behaviour Driven Development Test Suites," 2019, pp. 228–237.

[13] T. Tuglular and S. Şensülün, "SPL-AT Gherkin: A Gherkin Extension for Feature Oriented Testing of Software Product Lines," 2019, vol. 2, pp. 344–349.

[14] F. Zampetti, A. Di Sorbo, C. A. Visaggio, G. Canfora, and M. Di Penta, "Demystifying the adoption of behavior-driven development in open source projects," *Information and Software Technology*, vol. 123, p. 106311, 2020.

[15] E. Aghayi, T. D. LaToza, P. Surendra, and S. Abolghasemi, "Crowdsourced behavior-driven development," *Journal of Systems and Software*, vol. 171, p. 110840, 2021.

APPENDIX

Table 2. User Story and Holistic Test Suite Development Durations with respect to Gherkin Steps and Run Counts

GOAL	*M1 (AS + MS)*	*M2 (AS + MS)*	*M3 (AS + MS)*	*B1 (BS + MS + SS)*	*B2 (AS + BS)*	*B3 (AS + BS)*	*B4 (AS + BS + MS)*	*A1 (AS)*	*H1 (MS)*	*H2 (MS + SS)*
User story	5h 25m	4h 38m	6h 23m	9h 4m	4h 20m	5h 40m	4h 42m	2h 20m	3h 10m	1h 42m
Holistic test suite	5h 21m	4h 23m	2h 23m	5h 55m	2h 42m	2h 24m	3h 43m	1h 40m	1h 35m	1h 29m
Total development time	10h 46m	9h 1m	8h 46m	14h 59m	7h 2m	8h 4m	8h 25m	4h 0m	4h 45m	3h 11m
Number of scenarios	10	7	10	15	8	8	13	5	2	4
Number of steps	30	22	32	54	29	29	43	14	6	14
Step run count	49	39	61	61	42	42	69	20	6	14

Tugkan Tuglular received the B.S., M.S., and Ph.D. degrees in Computer Engineering from Ege University, Turkey, in 1993, 1995, and 1999.

He worked as a research associate at Purdue University from 1996 to 1998. He has been with Izmir Institute of Technology since 2000. After becoming an Assistant Professor at Izmir Institute of Technology, he worked as Chief Information Officer in the university from 2003-2007. In addition to his academic duties, he acted as IT advisor to the Rector between 2010-2014. In 2018, he became an Associate Professor in the Department of Computer Engineering of the same university. He has more than 70 publications and an active record of duties with international and national conferences. His current research interests include model-based testing and software quality with machine learning support.

Assoc. Prof. Dr. Tuglular is a member of ACM, IEEE Computer Society, and IEICE.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

Tugkan Tuglular came out with the idea of the paper and the proposed method. The proposed method has been improved by all the authors.

Deniz Egemen Coşkun, Ömer Gülen, Arman Okluoğlu, and Kaan Algan has implemented the proposed BDD method and developed a microservice application. They measured development times for code and tests and test coverage.