Automated Test Generation for Microservices from UML Sequence Diagrams

Tomas Vainoris [1] and Lina Čeponienė [1]

[1] *Department of Information Systems, Faculty of Informatics, Kaunas University of Technology, Studentų St. 50, Kaunas, 51368, Lithuania*

Abstract

Microservice architecture increases system flexibility but also introduces challenges in automated testing due to complex inter-service HTTP communication. The paper presents an approach for generating automated tests from UML 2.5 sequence diagrams in microservice-based systems. A rule-based transformation converts diagram message interactions into structured Gherkin scenarios, which become executable through a reusable step definitions library integrated into a behavior-driven development workflow. Prototype implementation was evaluated using a representative microservice project. Evaluation results, including code coverage and mutation testing, indicate that the generated tests can detect faults and validate the behavior of microservice-based systems, demonstrating the feasibility of deriving executable tests directly from design models.

Keywords

Microservices, UML, sequence diagrams, testing, BDD

1. Introduction

In recent years, most companies have been migrating from monolithic architecture to microservice architecture to increase flexibility and improve system maintainability and scalability [1]. However, when testing microservices, it is important to note that they communicate with each other via HTTP requests, which makes testing complex and confusing. When creating automated tests, testers must manually analyze how and in what order these systems communicate. As a result, writing automated tests for microservices is difficult and time-consuming [2].

On the other hand, UML [3] diagrams can serve as a valuable tool for visualizing system behavior, facilitating comprehension of the overall architecture and inter-service communication, such that even those unfamiliar with the system can understand its operation. UML class diagrams are used when specifying API documentation, indicating the possible fields of HTTP requests and responses. Class diagrams are also used to describe the system components through which microservices communicate with one another [4]. UML sequence diagrams are often used to show which microservices interact with each other and the order in which requests are executed. They can also be used to represent error scenarios, illustrating how the system reacts when an error response is returned from another part of the system. Thus, preparing UML diagrams that describe the structure and behavior of a microservice-based system constitutes a valuable step in the overall development process. Moreover, the utility of such diagrams can be extended beyond visualization by using them as a basis for test generation.

While methods for automatically generating tests from UML diagrams already exist for testing monolithic systems by testing individual classes and integrations [5], this area is not yet well developed for microservices. The aim of this paper is to improve microservice testing and facilitate automated test creation by proposing a methodology that reduces manual effort in analyzing inter-service communication and improves alignment between system design artifacts and automated test implementation.

*IVUS 2026: Information Society and University Studies 2026, May 12, Kaunas, Lithuania

tomas.vainoris@ktu.edu (T. Vainoris); lina.ceponiene@ktu.lt (L. Ceponiene)

0000-0002-6812-0215 (L. Ceponiene)

Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-06

CEUR-WS.org/Vol-4317/paper15.pdf

Therefore, the microservice test generation methodology was proposed as a solution to this problem and implemented as a tool that enables the automatic generation of tests from UML sequence diagrams describing how microservices communicate with one another. The test generation tool was implemented as an MCP server and integrated into the Visual Studio Code IDE. Preliminary evaluation results demonstrate that the tool successfully generates Gherkin scenarios from UML diagrams. The generated tests can be executed against a real microservice implementation and can detect faults in the code.

The rest of the paper is organized as follows. The related work section reviews existing methodologies for test generation from sequence diagrams, as well as other approaches applied to microservice testing. The proposed approach section presents the suggested methodology for generating test scenarios from UML sequence diagrams. The prototype section describes the implementation of the test generation tool and its validation.

2. Related Work

Microservice architecture has been widely adopted as an architectural style for building autonomous services with clearly defined responsibilities [6]. To address recurring challenges in distributed systems, microservice design patterns have been proposed as reusable solutions to common architectural problems [7]. Prior studies highlight the importance of isolated testing in microservice-based systems, emphasizing component and contract testing as key practices for maintaining service independence, especially when compared to monolithic architectures [8]. Behavior-Driven Development (BDD) helps teams to express automated tests in terms of expected behavior using structured *Given–When–Then* statements [9]. These specifications are commonly written in the Gherkin language and can be executed using frameworks such as Cucumber [10]. Automated testing is commonly integrated into CI/CD pipelines to support continuous quality assurance and rapid feedback [11]. For modeling microservice architectures, UML class diagrams are commonly used to represent service structure, while UML sequence diagrams are used to describe inter-service communication [12].

One of the existing solutions for generating tests from UML diagrams is Skyfire [13]. It is a tool that generates Cucumber test scenarios from UML state diagrams. The approach transforms a state diagram into a generic graph representation, from which test scenarios are derived and subsequently converted into Cucumber-based tests. States and transitions defined in the UML model are directly mapped to Cucumber steps, where transitions correspond to Given or When steps, and system states are expressed as Then steps [13]. While this method improves functional coverage by systematically deriving tests from behavioral models, the generated scenarios lack executable step implementations and therefore require additional manual effort to implement state checks and transitions.

Several approaches generate automated tests from microservice API specifications such as OpenAPI [14]. These specifications define request attributes and input constraints, enabling the generation of tests that verify specification compliance and input validation, as demonstrated by the UTEST approach [15]. However, since the OpenAPI specification does not capture business logic, such tests cannot detect failures related to deviations from intended API behavior or inconsistencies between internal execution and returned HTTP status codes, for example, when a success response is returned despite an internal error [15].

Another approach to generating test scenarios uses an extended finite state machine (EFSM) as an intermediate representation between UML sequence diagrams and executable test code [16]. Sequence diagrams, which capture detailed interactions between system objects and actors, are transformed into EFSMs where actions, loops, and conditional statements are represented as states, cycles, and branches, respectively [16]. Test scenarios are then generated from the EFSM using the ModelJUnit library and executed with JUnit, allowing testers to focus on creating sequence diagrams rather than writing code manually. A limitation of this approach is that it is tied to Java-based systems; testing applications implemented in other programming languages would require additional components to translate generated tests into the target language or framework.

In general, the reviewed approaches demonstrate that automated test generation from specification models is feasible. However, analyzed approaches focus on individual class interaction. In addition, the transformation from sequence diagrams is often tightly coupled with specific

programming languages, limiting applicability in a heterogeneous microservice environment. OpenAPI-based solutions focus on request and response structures and input validation, but they do not capture inter-service communication flows or business logic captured in behavior models. Therefore, there remains a need for a methodology that generates executable, behavior-oriented tests directly from a UML sequence diagrams, while remaining independent of a particular implementation language and suitable for modern microservice architectures. The approach proposed in this paper addresses this gap by transforming UML sequence diagrams into Gherkin test scenarios.

3. The Proposed Approach for Automated Microservices Test Generation from UML Sequence Diagrams

The proposed approach presents an automated test generation methodology that facilitates the creation of executable test scenarios from UML sequence diagram models, compliant with the UML 2.5 standard. The methodology operates on formal diagram models rather than graphical representations. UML sequence diagrams are used to model interactions between microservices, where each message is interpreted as an HTTP request. Based on these models, test scenarios in the Gherkin format are generated following a predefined, rule-based process that ensures the correctness and consistency of the modeled communication flows. In Figure 1, a full overview of the methodology is presented; yellow boxes represent the test generator and the shared library, which are used to generate executable tests from UML diagrams created during the requirement-gathering and design process.

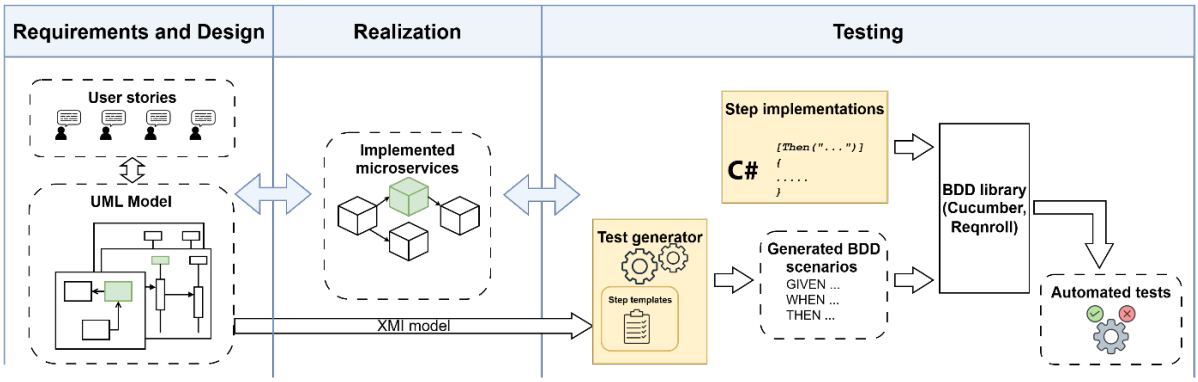


Figure 1: Overview of the process of microservices test generation from UML sequence diagrams

3.1. Modeling Microservice Interactions in Sequence Diagrams

UML sequence diagrams are modeled according to a set of predefined rules established within the proposed methodology. In these diagrams, lifelines represent individual microservices. An exception is made for the first lifeline, which may represent an external actor invoking a microservice. To ensure traceability between sequence diagrams and structural design, a corresponding class diagram is created for each microservice depicted in the sequence diagram. Figure 2 presents an example UML class diagram illustrating the microservices and their corresponding request and response models used in HTTP communication. Each microservice class defines a set of operations representing the exposed REST endpoints. Operation names follow the convention "{MethodName}{EndpointPath}" (e.g., PostLicense). This naming convention was selected to simplify identification of specific endpoints in the diagram and to eliminate the need for additional UML stereotypes.

If an endpoint accepts a request body or produces a response body, each body is modeled as a separate contract class. Classes such as *CreateLicenseRequest* from Figure 2 represent the JSON payload structure, where each attribute corresponds to a JSON field and retains the same name and data type as defined in the JSON schema. Explicit specification of primitive data types (e.g., Boolean, String, and Integer) is required to ensure correct serialization and consistency between the model and

implementation. Contract classes are referenced as parameter types within the corresponding operations. Operations may include multiple URL parameters, which are specified in the same order as they appear in the endpoint path. The parameter names “body” and “response” are reserved: the “body” parameter denotes the HTTP request body, and the “response” parameter denotes the HTTP response body. Both parameters are optional, depending on the endpoint specification.

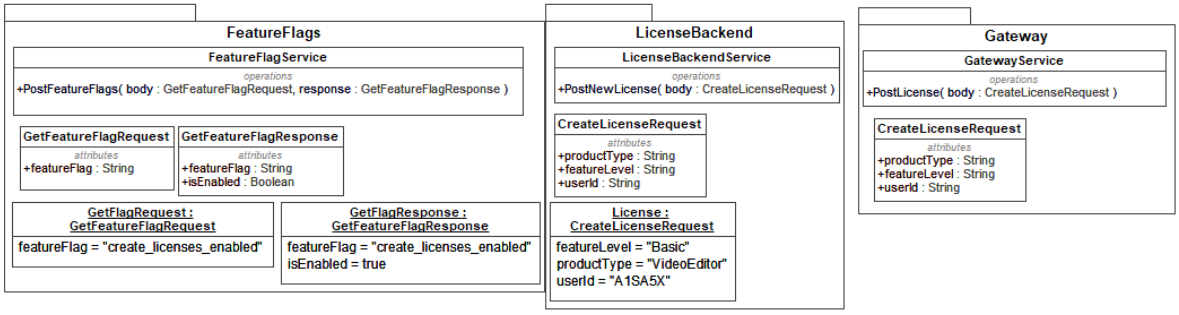


Figure 2: Example class diagram demonstrating microservices

Sequence diagrams are constructed using the defined microservice classes and their operations. Each sequence diagram represents a specific interaction scenario. These scenarios correspond to a concrete use case that will be transformed into automated tests. In Figure 3, an example sequence diagram is presented with *GatewayService*, which calls two external services and uses alternative fragments to visualize different scenarios. The presented sequence diagram uses classes from Figure 2 as lifeline classifiers. *GatweayService* is the test subject, *FeatureFlagService* and *LicenseBackendService* are external dependencies that are called during the depicted scenario. This distinction is important for automated test generation, as only the *GatewayService* behavior is verified, while the remaining services are treated as mocked dependencies. The first message from the sequence diagram uses the *PostLicense* operation from the *GatewayService*, which represents the initial HTTP call to the service. Messages exchanged between lifelines correspond to individual HTTP requests, while reply messages indicate the HTTP status code returned by the service. Status codes are modeled as integer values, and any valid HTTP status code may be specified.

To provide test data, object instance specifications are created with example values and used as arguments within the sequence diagrams. For example, the *PostLicense* operation has a body parameter of type *CreateLicenseRequest*, and in the sequence diagram, it has assigned a value to the *License* instance specification from the class diagram in Figure 2. These instance specifications represent concrete test inputs that will later be used during automated test generation.

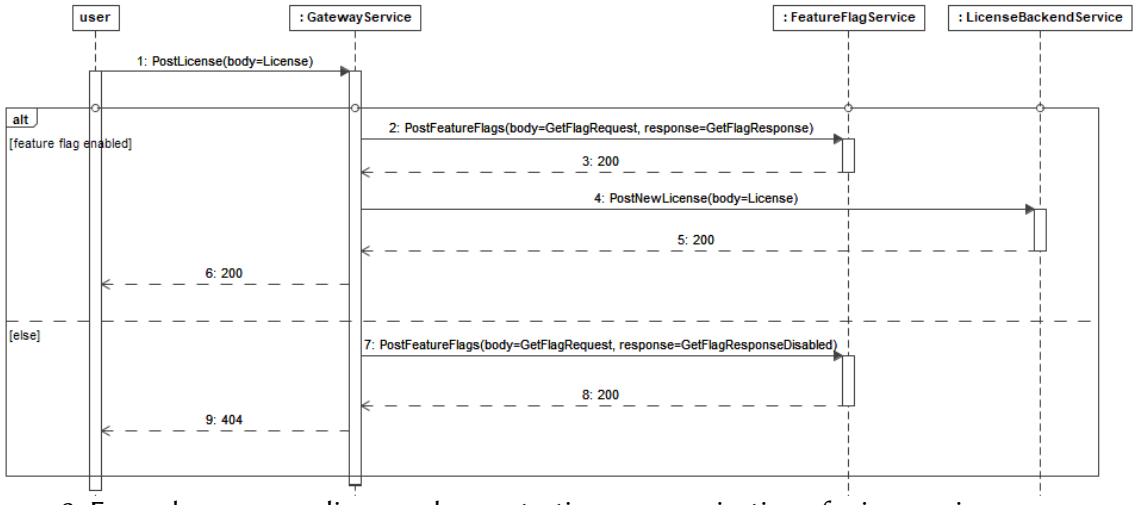


Figure 3: Example sequence diagram demonstrating communication of microservices

Alternative execution paths are modeled using the UML alternative (alt) and optional (opt) fragments. This mechanism enables representation of non-primary flows, such as error scenarios involving responses like 400 Bad Request, 404 Not Found, or other relevant HTTP error codes. Each

alternative branch represents a separate execution path that can be transformed into an individual generated test case. In general, modeling microservice communication using UML class and sequence diagrams provides concrete representations of inter-service interactions and formally establishes service contracts.

3.2. Sequence Diagram Transformation into Gherkin Scenarios

To ensure consistency and verify that implementations conform to the agreed design, each sequence diagram is transformed into executable scenarios written in the Gherkin language. In this transformation process, each message exchanged between lifelines is mapped to a predefined step definition, which serves as the basis for automated scenario generation. The selection of step definitions depends on the role of the microservice under test within the interaction. In Table 1, all step definitions and descriptions are listed. Messages directed to the tested microservice are translated into request-forming steps. Specifically, SD1 and SD2 are used to construct an HTTP request to the tested service, depending on whether the interaction includes a request body. If a message contains a body argument, the corresponding step with a body parameter is applied; otherwise, the simpler request step is used. Interactions involving external microservices invoked by the tested service are handled differently. SD3 and SD4 are used to define the expected responses of dependent microservices, mocking their behavior within the scenario. SD5 represents the execution phase, during which the previously defined request is sent under the configured mock conditions. SD6 verifies that the tested microservice returned the correct status code. SD7 verifies the response body returned by the microservice. SD8 is used to verify that the tested microservice invoked the appropriate external service endpoint with the specified HTTP method and path.

Table 1
Predefined step definitions used in scenarios

Step	Step keyword	Step definition	Description
SD1	Given	I have an HTTP "{method}" "{path}" request	Forms an HTTP request with a path
SD2	Given	I have an HTTP "{method}" "{path}" request with body “{json}”	Forms an HTTP request with path and request body
SD3	Given	service "{serviceName}" returns response with status code {statusCode}	Sets up that the service with the specified name will respond with the status code
SD4	Given	service "{serviceName}" returns {statusCode} for "{method}" "{path}" with body “{json}”	Sets up that the service with the specified name will respond with the body status code
SD5	When	I send the request	Executes constructed requests
SD6	Then	the response status code should be {statusCode}	Asserts that the microservice under test returns a response code
SD7	Then	the response body should match “{json}”	Asserts that the microservice under test returned a response body.
SD8	Then	service "{serviceName}" was called with HTTP "{method}" "{path}"	Asserts that the external service endpoint was called

The proposed test scenario generator imports the UML model and parses its UML elements. The parsed model serves as the input for scenario generation. As an initial step, all distinct execution scenarios are identified. To handle alternative and optional paths, messages and message occurrence specifications are represented as a directed graph. A depth-first search (DFS) traversal is then applied to identify all unique paths within this graph. Each identified path is treated as an independent sequence diagram and processed separately, ensuring that the generator of one scenario does not influence or interfere with others. As demonstrated in Figure 4, the test generator first splits the diagram into separate diagrams, thus removing alternative and option fragments.

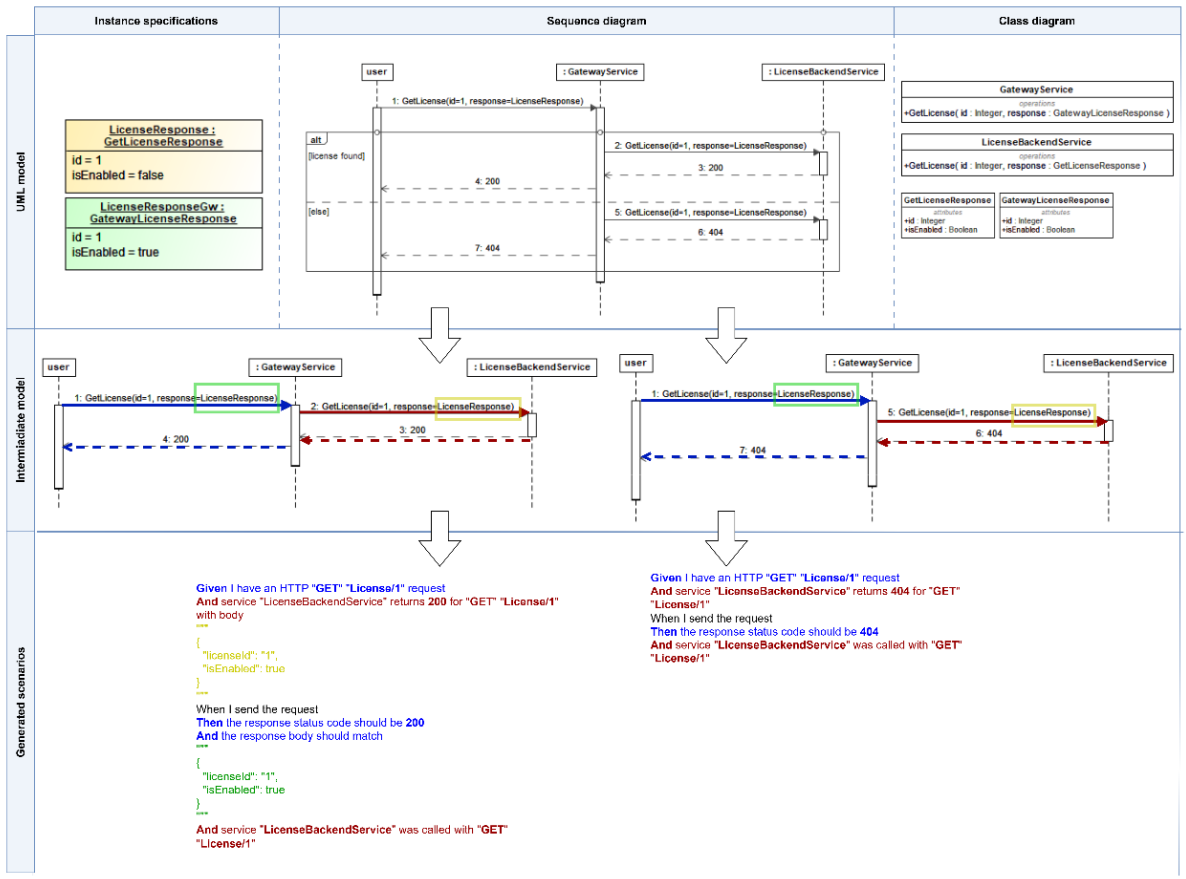


Figure 4: Test scenario generation example

At the beginning of the scenario generation process, all messages are classified into two categories: messages sent to the microservice under test and messages sent by the microservice under test. In Figure 4, the blue messages represent messages sent to the microservice; the red messages represent messages sent by the microservice under test. For request construction, SD1 is applied when the corresponding operation does not include a request body parameter, and SD2 is used when the operation requires a request body parameter. In the first generated scenarios, SD4 is used to mock that *LicenseBackendService* will return a response with a body. In the second scenario, SD3 is used to mock an external service because an error response code is returned. For response status verification, SD6 is used in both scenarios. SD7 was used to verify the returned response body, and from the instance specification, as expected, the response JSON body was constructed. SD8 is used to verify that external services were called by the tested microservice.

4. Prototype Implementation of the Test Generation Tool

The test generation tool was implemented as an MCP server using a .NET 10 application. The MCP server enables the integrated test generation tool to be invoked directly within an IDE that supports Copilot plugins, such as Visual Studio Code. The tool exposes two primary commands: one for listing available sequence diagrams associated with a microservice, and another for generating test scenarios from a selected diagram. This integration allows developers to generate test scenarios directly from UML model files without leaving the IDE, thereby reducing context-switching and simplifying the overall workflow.

In addition to the generator, a shared step implementation library was implemented as a class library using .NET 10, Reqnroll [17], and xUnit [18]. The library contains reusable step definitions matching the predefined Gherkin steps described in Table 1. Reqnroll provides the BDD infrastructure for binding Gherkin steps to the executable code, while xUnit serves as the underlying test execution

framework. Once the shared library is referenced in a project, the generated Gherkin scenarios become fully executable without requiring manual step implementation from the developers.

In Figure 5, test generation tool usage is demonstrated within the Visual Studio Code environment with an integrated Copilot extension. The right-hand panel shows the interaction with the MCP server, where diagrams are listed, and the test generation command is executed. The generated scenarios are automatically created in a feature file, visible in the main editor window. Tests were generated for the sample diagram from Figure 2. On the left side, the test explorer displays the execution results, confirming that the generated scenarios (e.g., “Scenario 1” and “Scenario 2”) were successfully executed.

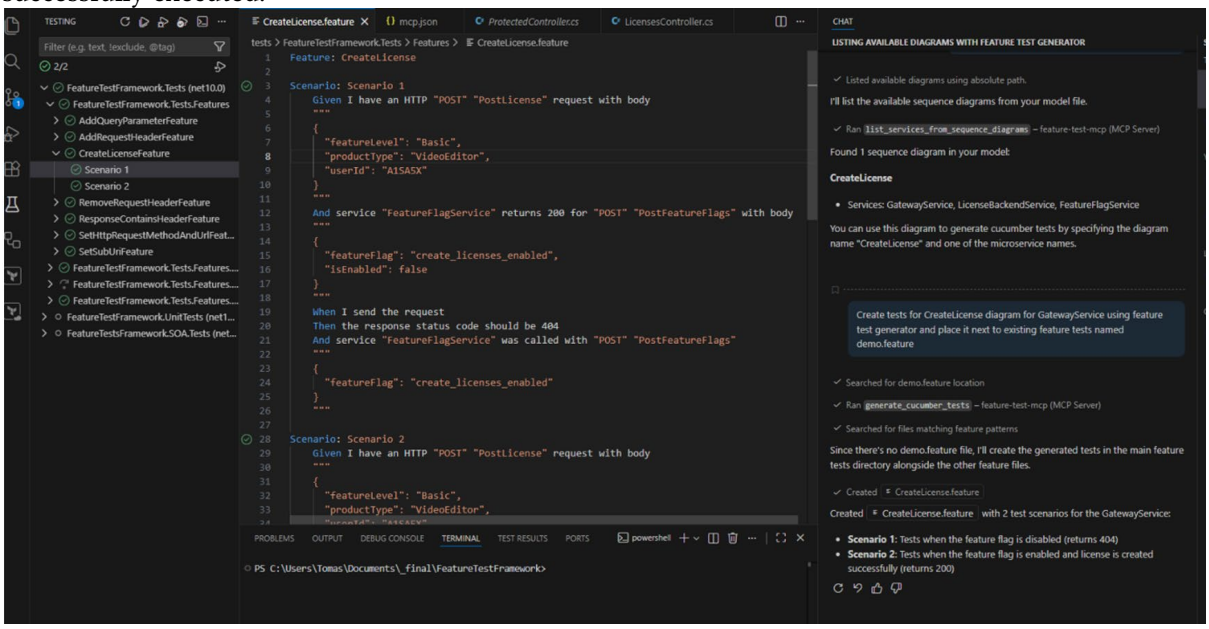


Figure 5: Example test generation tool usage in Visual Studio Code

Prototype evaluation was performed in two phases. First, the correctness assessment of the transformation process from the UML sequence diagram to the Gherkin scenarios was performed. Second, the effectiveness of the generated Gherkin scenarios was validated by running them against an actual microservice implementation to assess the ability to detect defects.

To verify the correctness of the transformation logic, a set of initial validation tests was performed. These tests ensured that sequence diagram elements were accurately translated into the expected Gherkin steps. The results of this validation are summarized in Table 2. The table presents core characteristics of sequence diagrams used in the experiments, including the number of message calls to external systems, the number of lifelines representing a microservice, the number of instance specifications used in body and response parameters, and the presence of alternative and optional fragments. In total, 14 test scenarios were generated from the 10 evaluated sequence diagrams. For all diagrams, request-forming steps SD1 and SD2 were used once per individual-generated scenario. Sequence diagrams that did not have opt or alt fragments, produced a single test scenario (e.g., D1, D4, D6, D7, D9, D10). In contrast, diagrams with either alt or opt fragments generated 2 scenarios, representing the alternative execution paths defined in the sequence diagram (e.g., D2, D3, D5, D8).

Also notable in Table 2 is the variation in the number of instance specifications used in test diagrams. For example, diagrams D7 and D8 contain 4 instance specifications, which are used to define concrete requests or response values; as a result, steps SD7 and SD8 were used to validate response bodies. In the D8 diagram, the operation from the first message call had a body, so SD2 was used to specify the JSON request body. Overall, all predefined step definitions (SD1-SD8) were used across the evaluated cases, confirming that the transformation logic covers the set of supported interaction patterns.

Table 2
Summary of Evaluated Sequence Diagrams and Generated Test Scenarios

Evaluated Sequence diagram						Test scenarios generation results	
No.	External Message Calls	Lifelines	Instance specifica-tions	Alt fragments	Opt fragments	Total generated scenarios	Used step definitions: Step(count)
D1	0	1	0	0	0	1	SD1(1), SD5(1), SD6(1),
D2	2	3	0	0	1	2	SD1(2), SD3(3), SD5(2), SD6(2), SD8(3)
D3	2	3	0	1	0	2	SD1(2), SD3(3), SD5(2), SD6(2), SD8(3)
D4	0	1	1	0	0	1	SD2(1), SD5(1), SD6(1)
D5	0	1	0	1	0	2	SD1(2), SD5(2), SD6(2),
D6	0	1	1	0	0	1	SD1(1), SD5(1), SD6(1), SD7(1)
D7	2	3	4	0	0	1	SD2(1), SD3(1), SD4(1), SD5(1), SD6(1), SD7(2)
D8	2	3	4	1	0	2	SD2(2), SD3(1), SD4(2), SD5(2), SD6(2), SD8(3)
D9	1	2	1	0	0	1	SD1(1), SD3(1), SD5(1), SD6(1), SD8(1)
D10	1	2	1	0	0	1	SD1(1), SD4(1), SD5(1), SD6(1), SD8(1)

The generated scenarios were compared against manually defined expected outputs and were found to match the anticipated results. These validated scenarios were used as a foundation for end-to-end testing of the test generator. In these automated tests, UML models are provided in XMI format as input, and the generated feature files were automatically compared against predefined expected feature files to ensure correct scenario generation.

To assess the validity of the generated tests, a single microservice project was developed as an experimental subject. The microservice exposes a single endpoint that implements the scenario shown in Figure 2. The solution includes a separate test project that leverages a shared step library. Code coverage was measured using the built-in .NET tooling, and mutation testing was done using Stryker.NET tool [19]. Table 3 contains the results of code coverage and mutation testing of tests generated from UML diagrams using a test generation tool.

Table 3
Results of code coverage and mutation testing for single endpoint components

Class name	Lines of code	Code coverage, %	Detected defects	Undetected defects
LicensesController	15	100	4	0
FeatureFlagHttpClient	8	100	3	0
LicensesHttpClient	7	100	2	1
Total	**30**	**100**	**9**	**1**

Although 100% code coverage may appear impressive, it is important to note that the evaluated endpoint is relatively small, comprising only 30 lines of code. Therefore, the coverage metric alone does not necessarily indicate test effectiveness. Nevertheless, mutation testing results demonstrate that the generated tests can detect most of the defects created. The applied mutations included modifications of request body values, the removal of method calls to external services, and other mutations. In the case of *LicensesHttpClient*, one mutation remained undetected. This mutation removed the validation logic that verifies whether the Licenses service returned the correct status code. The reason for this gap can be attributed to the UML sequence diagram, which did not include an alternative fragment specifying the system's behavior when the *LicenseBackend* service returns an invalid response. Overall, the combination of code coverage and mutation testing results indicates that the generated tests are valid and can detect defects in the implementation.

5. Conclusion

The paper presented an approach for automated microservices test generation based on UML sequence diagrams. The proposed methodology transforms message interactions specified in a sequence diagram into predefined Gherkin steps, generating structured behavioral test scenarios. These scenarios become executable through reusable step definitions implemented in a shared step implementation library. The test generation results were evaluated using a representative microservice implementation. The evaluation results, including code coverage and mutation testing, indicate that the generated tests can detect faults and validate the implemented behavior. The findings suggest that the proposed approach can produce meaningful and practically applicable tests derived directly from design artifacts such as sequence diagrams. Future work will involve conducting experiments with a broader range of sample projects to further evaluate the applicability of the methodology. Additional improvements may include support for event-driven architecture and extended handling of HTTP-specific aspects such as headers.

6. Declaration on Generative AI

The authors have not employed any Generative AI tools.

7. References

[1] G. Blinowski, A. Ojdowska and A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access, vol. 10, pp. 20357-20374, 2022.

[2] M. Waseem, P. Liang, M. Shahin, A. D. Salle and G. Márquez, Design, monitoring, and testing of microservices systems: The practitioners' perspective, Journal of Systems and Software, vol. 182, 2021.

[3] Object Management Group, Unified Modeling Language, 2015. URL: https://www.omg.org/spec/UML/2.5/About-UML.

[4] F. Carranza-García, C. Rodríguez-Domínguez and J. L. Garrido, Addressing Expressiveness for a UML Microservices-Based Modeling within the Life Cycle of the Ubiquitous System Development, 2021 17th International Conference on Intelligent Environments (IE), pp. 1-8, 2021.

[5] S. K. Swain, D. P. Mohapatra and R. Mall, Test Case Generation Based on Use case and Sequence Diagram, International Journal of Software Engineering, vol. 3, pp. 21-52, 2010.

[6] P. D. Francesco, P. Lago and I. Malavolta, Architecting with microservices: A systematic mapping study, Journal of Systems and Software, vol. 150, pp. 77-97, 2019.

[7] O. C. Oyeniran, A. O. Adewusi, A. G. Adeleke, L. A. Akwawa and C. F. Azubuko, Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability, International Journal of Advanced Research and Interdisciplinary Scientific Endeavours, vol. 1, no. 2, p. 92–106, 2024.

[8] I. Ghani, W. M. Wan-Kadir, A. Mustafa and M. I. Babir, Microservice Testing Approaches: A Systematic Literature Review, International Journal of Integrated Engineering, vol. 11, p. 65–80, 2019.

[9] M. M. Diepenbeck, U. Kühne, M. Soeken ir R. Drechsler, Behaviour driven development for tests and verification, International Conference on Tests and Proofs, pp. 61-77, 2014.

[10] A. Hellesoy, M. Wynne and S. Tooke, The cucumber book: behaviour-driven development for testers and developers, 2017.

[11] Y. Jani, Implementing continuous integration and continuous deployment (ci/cd) in modern software development,"International Journal of Science and Research, vol. 12, pp. 2984-2987, 2023.

[12] N. Santos, H. Rodrigues, N. Ferreira and R. J. Machado, Inputs from a Model-based Approach towards the Specification of Microservices Logical Architectures: An Experience Report, International Conference on Product-Focused Software Process Improvement, 2019.

[13] N. Li, A. Escalona and T. Kamal, Skyfire: Model-Based Testing with Cucumber, 2016 IEEE International Conference on Software Testing, Verification and Validation, pp. 393-400, 2016.

[14] A. Arcuri, Automated black- and white-box testing of RESTful APIs with EvoMaster, IEEE Software, vol. 38, pp. 72-78, 2021.

[15] L. Giamattei, A. Guerriero, R. Pietrantuono and S. Russo, Assessing Black-box Test Case Generation Techniques for Microservices, International Conference on the Quality of Information and Communications Technology, vol. 1621, p. 46–60, 2022.

[16] M. Rocha, A. Simão and T. Sousa, Model-based test case generation from UML sequence diagrams using extended finite state machines, Software Quality Journal, vol. 29, p. 597–627, 2021.

[17] Reqnroll, Reqnroll, URL: https://reqnroll.net.

[18] xUnit.net, xUnit.net, URL: https://xunit.net.

[19] Stryker.NET, Stryker.NET, URL: https://stryker-mutator.io/docs/stryker-net/introduction/.