International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 12 Issue: 2
***Article Received:** 25 January 2024 **Revised:** 12 February 2024 **Accepted:** 30 March 2024*

BDD for Testing Microservices and Distributed Systems

Sagar Aghera
Independent Researcher, Sr Staff Engineer in Test, Netskope Inc, USA
Email: saghera@netskope.com

Nikhil Yogesh Joshi
Independent Researcher, Manager II, Fiserv, USA
Email: nikhilyogesh.joshi@fiserv.com

Abstract: This research uses Behaviour-Driven Development (BDD) to test microservices and distributed systems for scalability, fault tolerance, and concurrency. By using natural language specifications, BDD helps stakeholders collaborate and record and validate system behaviours. Unit testing, integration testing, and end-to-end (E2E) testing are evaluated inside the BDD framework. Integration testing balances coverage, maintainability, and complexity best. Compared to TDD and ATDD, BDD excels in behaviour specification and stakeholder alignment, complementing TDD's unit test coverage and ATDD's acceptance criteria validation.

***Keywords**: BDD, microservices, distributed systems, integration testing, E2E, TDD, ATDD.*

I.INTRODUCTION

Microservices and distributed systems are key architectures for current software applications. Software systems are becoming more sophisticated and require scalable, flexible, and maintainable paradigms. Microservices are popular and useful and used by over 61% of organizations, according to O'Reilly [1]. Complex systems make testing challenging.

BDD is a popular microservices and distributed system testing solution. BDD natural language test requirements promote developer-tester-business stakeholder collaboration [2]. Distributed system implementation requires better communication and software behaviour that fits business needs.

Traditional testing approaches struggle with dynamic, decentralized microservices and distributed systems. Multiple services communicating over network protocols complicate integration, coordination, and failure tolerance in these systems. In 2018, poor software quality cost the US $2.84 trillion, mostly due to testing and debugging [3]. Effective testing reduces expenses and ensures software quality.

BDD connects technical and non-technical stakeholders with plain-language executable specs and Cucumber, SpecFlow, and JBehave [4]. This method improves system behaviour comprehension by creating understandable and maintainable test scenarios. BDD in microservices can automate and test agile development methods in CI/CD workflows.

Distributed systems assess system behaviour across nodes and services using BDD for distributed transactions, synchronization, and coordination. Probabilistic distributed systems with network splits and node failures need rigorous testing to show resilience and dependability. BDD's behaviour specification provides realistic test scenarios to cover edge cases and failure types.

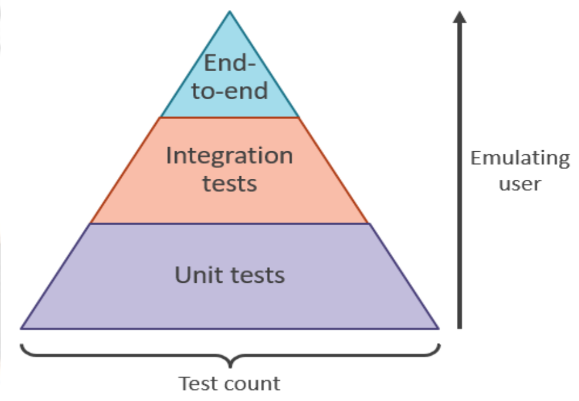

Fig 1.1:Different Testing methods used in microservices and distributed systems (“https://khorikov.org/images/2020/2020-03-04-test-pyramid.png”)

II.LITERATURE REVIEW

Microservices and distributed systems architectures are becoming widely used as a result of the quick growth of software development methodologies. A microservices design breaks programs into loosely connected and deployable services, improving scalability, flexibility, and maintainability. Distributed systems have multiple nodes that interact and collaborate to achieve goals. Resilience and fault tolerance improve with this architecture. However, their complexity, diversity, and ever-changing qualities make these architectural ideas difficult to test.

Behaviour-driven development (BDD) has helped stakeholders bridge these gaps by using common language and executable specifications. BDD encourages developers, testers, and domain experts to collaborate on TDD. This cooperation guarantees that the software matches business requirements [1].

Effective microservice and distributed system testing is essential. According to a Cloud Native Computing Foundation survey, 63% of companies had problems testing microservices. These problems generally involved inter-service dependencies, state management, and data consistency [2]. Gartner estimated that 70% of significant enterprises delay release cycles due to insufficient distributed system testing [3]. Data shows the necessity for comprehensive testing methodologies that can handle modern software designs.

In BDD, user stories and business requirements are used to create systematic test scenarios. Natural language scenarios created by Cucumber, SpecFlow, and JBehave can be automated in CI/CD workflows. This improves test coverage, tracking, early problem detection, and error resolution costs [4].

Although BDD has benefits, microservices along with distributed systems make it challenging to deploy. These designs require advanced solutions for test orchestration, inter-service communication, and data synchronization due to their decentralization. Due to network oscillations and partial failures, resilient and tolerant test cases are needed [5].

RESEARCH GAP

The decentralized service communication of microservices and distributed systems makes software design scalable and resilient [1]. BDD was implemented as these architectures are challenging to test [2]. BDD enhances testing collaboration and clarity using natural language constructs, however quantitative analysis and developing technology integration are absent

Gaps in research are :

- A restricted quantitative study on BDD's impact on performance, mistake detection, and testing efficiency.
- Challenges in managing BDD test suites in large microservices environments.
- Inadequate investigation of distributed component synchronization behaviour requirements.
- The underutilized combination of BDD, orchestration, containerization, and service mesh topologies (e.g., Kubernetes, Istio)
- Insufficient frameworks for BDD concerns in distributed systems and microservices.

III.BDD FOR MICROSERVICES AND DISTRIBUTED SYSTEM

3.1. Definition and Characteristics of Microservices Architecture

Microservices architecture breaks apart a huge, integrated software into independent services. Each standalone service uses HTTP/REST or messaging protocols. This architecture supports modularity, autonomous service generation, deployment, and scalability. Every microservice is a business function with one responsibility and can be programmed in many languages and technologies [1]. Decentralized microservices enable polyglot programming and fault isolation because one service failure may not affect the complete system.

The primary attributes of microservices are as follows:

- **Decentralization:** Independent teams managing and operating independently.
- **Scalability:** Each service can be scaled autonomously.
- **Resilience:** Isolating faults improves the overall stability of the system.
- **Polyglot Persistence:** Various services can be provided by distinct databases.
- **Continuous Delivery:** Promotes quick development and implementation [1, 5].

Difficulties in Testing Microservices

Testing microservices entails intricacies such as:

- **Inter-service Communication:** Maintaining accuracy and asynchronous communication.
- **Service Dependencies:** Mocking and service virtualization are needed for isolated testing.
- **Deployment Environments:** Setting up tests is more difficult in dynamic containerized environments.
- **Data Consistency:** Ensuring uniformity among dispersed services.
- **Integration Testing:** Testing service interactions thoroughly might be difficult [6].

According to a survey conducted by O'Reilly, 86% of firms encounter substantial difficulties when it comes to testing microservices [7].

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 12 Issue: 2
***Article Received:** 25 January 2024 **Revised:** 12 February 2024 **Accepted:** 30 March 2024*

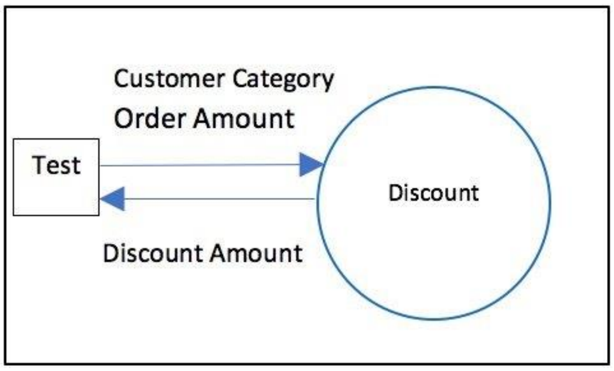

Fig 3.1: Microservice using BDD Framework (“https://imgopt.infoq.com/fit-in/3000x4000/filters:quality(85)/filters:no_upscale()/articles/microservices-bdd-interface-oriented/en/resources/1microservices-bdd-interface-oriented-2-1548446601454.jpg”)

Application of (BDD) in Microservices

Behaviour-Driven Development (BDD) improves testing by encouraging open communication and collaboration. BDD scenarios are written in a simple language, making test cases easy to access and update.

3.2. Definition and Characteristics of Distributed Systems

Distributed systems have many separate computing nodes that network and synchronize. These nodes work together to achieve a goal, usually appearing as a single system to end-users.

The essential attributes of distributed systems encompass:

- **Scalability:** Effectively manage larger workloads by including more nodes.
- **Fault Tolerance:** Maintain operations despite node failure via redundancy.
- **Concurrency:** Enable simultaneous execution across several nodes.
- **Transparency:** Conceal the intricacy of the distributed infrastructure from users.
- **Heterogeneity:** Combine several software and hardware elements [8].

Distributed systems enable excellent availability and performance in cloud computing, big data processing, and huge enterprise applications.

Difficulties in Testing Distributed Systems

Testing distributed systems presents numerous substantial challenges:

- **Network Partitioning and Latency:** Network partitions and variability complicate behaviour prediction.[11]
- **Concurrency Issues:** The simultaneous execution of many tasks generates race situations and deadlocks.
- **State Management:** Ensuring a uniform state across nodes is a complicated task.
- **Fault Injection and Recovery:** Simulating faults and testing recovery requires complex methods.
- **Scalability Testing:** Effective scaling with more nodes requires extensive load testing [9].

According to an IEEE survey, 72% of distributed system developers struggle to verify state management and fault tolerance [10].

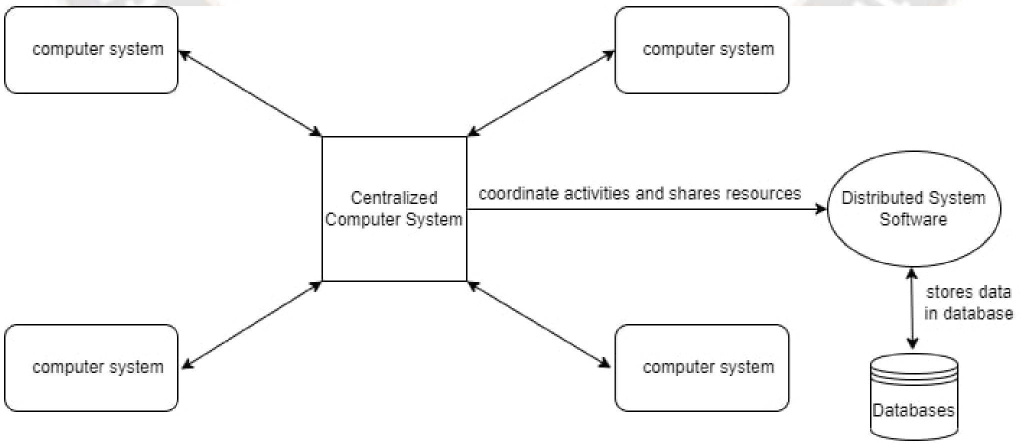
computer system
computer system
Centralized Computer System
coordinate activities and shares resources
Distributed System Software
stores data in database
computer system
computer system
Databases

Fig 3.2: Distributed Systems using BDD Framework (“https://media.geeksforgeeks.org/wp-content/uploads/20220525155747/distributedsystem1.jpg”)

Application of (BDD) in Distributed Systems

Behaviour-Driven Development (BDD) standardises distributed system testing and encourages developer, tester, and stakeholder communication. Behaviour-driven development (BDD) uses language to create test scenarios to ensure that everyone understands the system's expected behaviour.

IV.DIFFERENT TECHNIQUES AND ALGORITHMS USED IN MICROSERVICES AND DISTIBUTED SYSTEMS

Unit testing, integration testing, and end-to-end testing are types of testing procedures used in distributed systems and microservices. Each technique focuses on distinct facets of system functionality, guaranteeing the system's overall dependability and performance.

4.1. Unit Testing

Unit testing examines individual pieces or actions for functionality. In microservices, this means testing functions and methods separately from other services and dependencies.

Algorithm:

- **Identify Unit:** Choose the unit (function/method) that has to be tested.
- **Establish Test Environment:** Arrange the essential components for the test environment, such as mocks and stubs.
- **Define Test Cases:** Create test cases that encompass a range of input circumstances.
- **Execute Tests:** Execute the test cases.
- **Verify Results:** Verify the results by comparing them to the anticipated outcomes.
- **Report:** Document the outcomes of the exam.

Mathematical Model:

Consider the function $f : X \rightarrow Y$ that is being tested, where X represents the set of all possible inputs and Y represents the set of all possible outputs. The unit test can be defined as:

$\forall x \in X, \exists y \in Y$ such that $\mathrm{f}(x) = y$

where y is the expected output for input x.

Applications:

- Microservices require unit testing before integration for consistency. It helps identify flaws quickly and minimizes development costs [12].

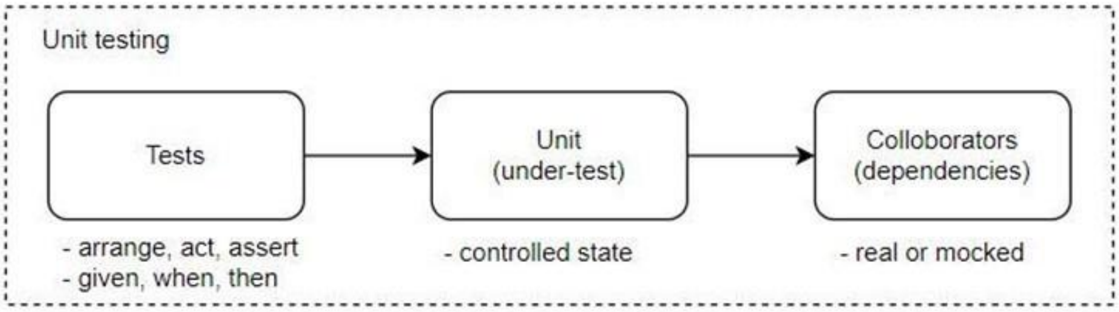

Fig 4.1: Unit Testing Architecture (“https://media.licdn.com/dms/image/D5612AQGiGg2RnS9pKg/article-inline_image-shrink_1000_1488/0/1693913625883?e=1724889600&v=beta&t=rStHibLzc7fZ70N7yByqU_MEF0i_vy3L96ATLKFgpDw”)

4.2. Integration Testing

Integration testing examines the connection between system components and services to guarantee appropriate operation. Microservices ensure expected service communication and collaboration.

Algorithm:

- **Interface Identification:** Identify service interfaces and interactions.
- **Setup Environment:** Set up the test environment by configuring all essential services.
- **Define Test Cases:** Create test cases that encompass interaction possibilities.
- **Execute Tests:** Carry out the integration tests.
- **Verify Results:** Verify the proper interaction of the services and ensure that they generate the anticipated results.
- **Report:** Record the examination outcomes.

Mathematical Model:

Let $S_1, S_2, , \ldots, S_n$ be the services and I_{ij} be the interaction between service S_i and service S_j. The integration test can be represented by the following mathematical expression:

$\forall i, j \in \{1,2, \ldots, n\}, I_{ij}(S_i, S_j)$ produces expected results

Applications:

- Microservices and distributed systems need integration testing to ensure service integration. It identifies service communication, data exchange, and interdependencies issues [13].

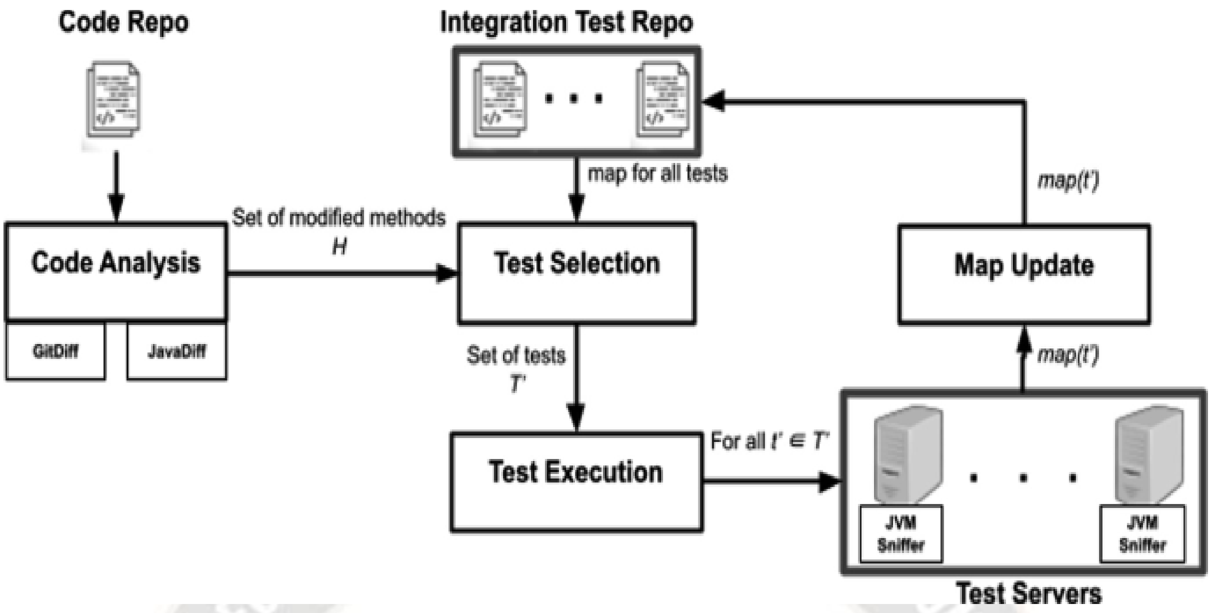

Fig 4.2: Integration Testing Framework (“https://media.springernature.com/lw685/springer-static/image/chp%3A10.1007%2F978-981-99-3091-3_14/MediaObjects/539848_1_En_14_Fig5_HTML.png”)

4.3. End-to-End Testing

End-to-end (E2E) testing evaluates the entire system workflow to ensure that all components and services work together to offer the desired functionality.

Algorithm:

- **Identify User Scenarios:** Specify user scenarios that encompass whole workflows.
- **Establish Environment:** Create a test environment that matches production.
- **Define Test Cases:** Generate test cases for every user scenario.
- **Execute Tests:** Execute the end-to-end tests.
- **Verify Results:** Verify the system's functionality in all scenarios.
- **Report:** Document the outcomes and any challenges encountered.

Mathematical Model:

Let W be the collection of workflows, denoted as $\{w_1, w_2, \dots, w_m\}$, where each workflow w_i involves a sequence of interactionsI_{ij}. The E2E test can be represented by the universal quantifier $\forall$, where w_i belongs to W.

$\forall w_i \in W$, w_i executes successfully and produces expected outcomes

Applications:

- End-to-end testing ensures that all microservices and distributed components work together to provide the desired functionality. It helps discover system-wide issues [5].

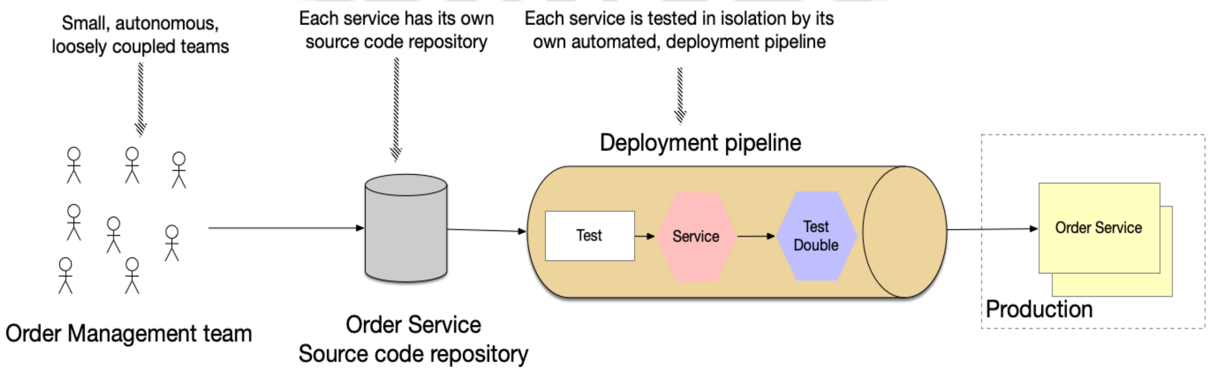
Small, autonomous, loosely coupled teams
Each service has its own source code repository
Each service is tested in isolation by its own automated, deployment pipeline
Deployment pipeline
Test
Service
Test Double
Order Service
Production
Order Management team
Order Service Source code repository

Fig 4.3: End-to-end (E2E) Testing Framework (“https://microservices.io/i/posts/testable-in-isolation.png”)

V.COMPARATIVE ANALYSIS OF VARIOUS TECHNIQUES AND ALGORITHMS FOR TESTING MICROSERVICES AND DISTRIBUTED SYSTEMS

Key Performance Metrics

Based on the following critical performance indicators, unit testing, integration testing, and end-to-end testing are compared:

- **Test Coverage:** Indicates how much testing has been done on the source code.
- **Execution Time:** The amount of time needed to run the test cases.
- **Fault Detection Rate:** The ability to find flaws.
- **Maintenance Effort:** The test cases need to be maintained.
- **Scalability:** The capacity to manage a growing system size and test count.
- **Complexity:** The difficulty of creating and carrying out test cases.

Unit Testing, Integration Testing, and End-to-End Testing techniques are compared in table 5.1 based on performance criteria.

Metric	Unit Testing	Integration Testing	End-to-End Testing
Test Coverage	High (90-100%)	Medium (60-80%)	Low (30-50%)
Execution Time	Low (milliseconds)	Medium (seconds)	High (minutes)
Fault Detection Rate	Medium (component-level bugs)	High (interaction-level bugs)	Very High (system-level bugs)
Maintenance Effort	Low (isolated tests)	Medium (dependent on interactions)	High (entire workflow tests)
Scalability	High (easily scales with components)	Medium (scales with interaction complexity)	Low (scales with entire system)
Complexity	Low (simple, isolated tests)	Medium (involves multiple components)	High (involves full workflows)

Table 5.1: Comparison of Different Techniques and Algorithms for Testing Microservices and Distributed Systems

Integration testing works better for microservices and distributed systems. Test coverage, execution time, error detection, maintenance, and complexity are balanced. For component evaluation, unit testing is essential, whereas end-to-end testing verifies the workflow. Integration testing ensures reliable and effective service interactions and communication. The BDD framework and three methodologies create a complete testing plan.

Comparison of BDD with Other Testing Methods

Behaviour-Driven Development (BDD) testing emphasizes stakeholder engagement utilizing natural language requirements. It is comparable to TDD and ATDD but has a different focus and approach.[15]

Test-Driven Development (TDD):

- TDD is a development strategy where developers write tests before producing code. It uses a "red-green-refactor" cycle to write failing tests (red), implement code (green), and improve code quality.

Acceptance Test-Driven Development (ATDD):

- ATDD expands TDD by helping stakeholders specify acceptance criteria early in development. Based on these criteria, acceptance tests influence development. The table 5.2 below compares different Testing Methodology with BDD.

Approach	Coverage	Maintainability	Readability	Integration Complexity
BDD	Focus on behaviour	Aligns tests with business	Natural language	Moderate, focuses on interactions
TDD	Unit level coverage	Modular code	Concise, specific	Low, isolates units of code
ATDD	Acceptance criteria	Business-driven tests	Business domain language	Higher, ensures user requirements

Table 5.2: Comparison of Different Testing Methodology

TDD, ATDD, and BDD have various benefits due to their focus and technique. BDD aligns corporate goals and improves maintainability with natural language standards. ATDD enhances integration with higher-level acceptance criteria validation, whereas TDD improves unit test coverage and modular code. The best strategy relies on project needs, stakeholder input, and testing granularity. These strategies increase software quality and satisfy customers.

VI.DISSCUSSION

This paper emphasizes BDD testing methods and methodologies for microservices and distributed applications. Microservices and distributed systems need testing for stability and performance due to their scalability, fault tolerance, and concurrency. These systems require considerable testing for network latency, state management, and fault recovery. BDD organizes system testing using natural language components and stakeholder communication.

In microservices, BDD leverages Cucumber and SpecFlow to construct executable tests from Given-When-Then user stories and scenarios. This systematic approach enables unit, integration, and E2E testing. Unit testing evaluates parts before integration. Microservices must communicate seamlessly, requiring integration testing. E2E testing ensures workflows offer intended functionality.

Comparisons of these testing approaches utilizing key performance measures provide insights. Unit testing finds component issues early due to its great test coverage and low execution time. Scalable and easy to maintain, it's crucial for microservices testing. Unfortunately, it cannot detect interaction- and system-level faults. Integration testing balances test coverage, speed, and error detection. Correctly connecting with services solves a major microservices and distributed system challenge. Interface complexity limits scalability, although complexity and maintenance are medium. E2E testing finds system-level errors to verify workflows. Its size makes it less scalable and takes the longest to execute, maintain, and understand.[14]

The focus and technique of TDD, ATDD, and BDD offer several benefits. BDD excels at plain language requirements for business alignment and maintainability. TDD improves unit test coverage and modular code, while ATDD improves integration with higher-level acceptance criteria validation. Best technique depends on project needs, stakeholder involvement, and testing granularity. These methods improve software quality and customer satisfaction.

BDD works well with CI/CD workflows. By providing ongoing feedback and early issue discovery, CI/CD pipelines with automated test execution increase software quality. Studies show that BDD improves microservice and distributed system test coverage and reduces failures.

VII.CONCLUSION AND FUTURE SCOPE

This research examined Behaviour-Driven Development (BDD) for testing microservices and distributed systems, including testing methods, algorithms, and comparisons. The introduction stressed the necessity of testing in assuring complex systems' reliability, scalability, and fault tolerance. BDD addresses microservices and distributed system testing difficulties by using natural language specifications to improve stakeholder participation.

Through BDD testing, unit, integration, and end-to-end (E2E) testing were examined to determine their roles and efficacy. Unit testing ensures great coverage and early bug detection by validating individual components. Integration testing checks service interactions, essential for microservices system stability. Complete workflows are tested in E2E to ensure end-user satisfaction. Comparing these methods showed that integrated testing balanced coverage, maintainability, and complexity well.

BDD's strengths stood out when compared to TDD and ATDD. BDD's behaviour specification and stakeholder participation improve maintainability and business goals. TDD thrives in unit test coverage and modular code architecture, while ATDD rigorously validates acceptance criteria to satisfy user expectations.

Future scope

The outcomes of this study provide numerous opportunities for further investigation:

- **Advanced Fault Injection:** Explore advanced fault injection methods to simulate failures and strengthen systems.
- **AI and Machine Learning:** Create automated test scenarios and forecast defects with AI/ML.
- **Scalability Improvements:** Parallel execution and resource management scale E2E testing.
- **BDD Framework Enhancements:** Develop microservice and distributed system BDD tools that focus on service interactions.
- **Performance Testing:** Add latency and throughput to BDD.
- **Security Testing:** Use BDD security testing to find and fix system vulnerabilities.

By addressing these issues, testing approaches for distributed systems and microservices will improve, utilizing BDD's advantages to satisfy contemporary software requirements.

REFERENCES

[1] Newman, S., 2015. Building Microservices: designing fine-grained system. Oâ€™ Reilly Media, Inc., California, p.2.

[2] North, D., 2018. Introducing BDD, 2006. Verfügbar unter: http://dannorth. net/introducingbdd.

[3] Krasner, H., 2021. The cost of poor software quality in the US: A 2020 report. Proc. Consortium Inf. Softw. QualityTM (CISQTM), 2.

[4] Bruschi, S., Xiao, L. and Kavatkar, M., 2019. Behavior driven development (BDD): a case study in healthtech. In Pacific NW Software Quality Conference.

[5] Lewis, J. and Fowler, M., 2014. Microservices: a definition of this new architectural term. MartinFowler. com, 25(14-26), p.12.

[6] Zimmermann, O., 2017. Microservices tenets: Agile approach to service development and deployment. Computer Science-Research and Development, 32, pp.301-310.

[7] O'Reilly Media. (2018). Microservices Adoption in 2018. O'Reilly Media.

[8] Coulouris, G.F., Dollimore, J. and Kindberg, T., 2005. Distributed systems: concepts and design. pearson education.

[9] Van Steen, M., 2002. Distributed systems principles and paradigms. Network, 2(28), p.1.

[10] Lahami, M. and Krichen, M., 2021. A survey on runtime testing of dynamically adaptable and distributed systems. Software Quality Journal, 29(2), pp.555-593.

[11] Binamungu, L.P., Embury, S.M. and Konstantinou, N., 2018, March. Maintaining behaviour driven development specifications: Challenges and opportunities. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 175-184). IEEE.

[12] Meszaros, G., 2007. xUnit test patterns: Refactoring test code. Pearson Education.

[13] Beizer, B. (1990) Software Testing Techniques. 2nd Edition, Van Nostrand Reinhold, New York.

[14] Luo, L., 2001. Software testing techniques. Institute for software research international Carnegie mellon university Pittsburgh, PA, 15232(1-19), p.19.

[15] Dehghanpour, K., Wang, Z., Wang, J., Yuan, Y. and Bu, F., 2018. A survey on state estimation techniques and challenges in smart distribution systems. IEEE Transactions on Smart Grid, 10(2), pp.2312-2322.