A COMPARATIVE ANALYSIS OF API TESTING APPROACHES IN CONTINUOUS INTEGRATION/CONTINUOUS DEPLOYMENT (CI/CD) PIPELINES IN CLOUD BASED SYSTEMS

Lappeenranta–Lahti University of Technology LUT

Master's in digital systems and service development, Master's thesis

2025

Author: Hafiz Muhammad Umar

Examiner: Kari Smolander

Abstract

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Sciences

Software Engineering

Hafiz Muhammad Umar

A comparative analysis of API testing approaches in continuous integration/continuous deployment (CI/CD) pipelines in cloud based systems.

Master's thesis

2025

54 pages, 9 figures and 12 tables

Examiners: Professor Kari Smolander, Katja Karhu

Keywords: API Testing, API automation testing, Contract testing, Integration testing, E2E testing, Continuous integration, Continuous deployment.

This study examines how a multi-layer testing framework, which is made up of Contract Testing, Integration Testing, and End-to-End (E2E) Testing, can be used to guarantee the reliability and robustness of RESTful APIs. The main aim was to test the contribution of each of the testing layers toward the identification of structural, logical, and communication-level flaws, in the API-based system. The case study is a Flask-based booking API, in which seven control defects were introduced systematically to test the detection efficiency and execution performance at each of the testing phases. The experimental results showed that Contract Testing had the lowest execution time (55 seconds) but had the lowest Defect Detection rate (10 percent), with schema and structural mismatches in the API contracts being the most frequent. Integration Testing took more time (155 seconds) and had the best

DDR (80 percent), which is successful in the detection of most interface, data-flow, and cross-endpoint behavioral defects. End-to-End (E2E) Testing was the slowest (230 seconds), and the DDR of 56.7%; however, it was realistic in validating entire user processes and business rules and revealed some logic and process-level anomalies that had not been exposed by Contract Testing. Overall, the aggregated values of the execution time and the DDR allow stating that a layered testing approach, which involves contract, integration, and E2E testing, is a decent trade-off between speed, coverage depth, and reliability of API-based architectures.

Table of Contents

Figures

Tables

1. Introduction

In the modern world, software is evolving rapidly. Cloud based tools and Continuous Integration/Continuous Deployment (CI/CD) are utilized by many companies to develop, test and release apps (Jani, 2023). They are leaving large and single apps behind and migrating to small and distinct services, which interact with APIs. The primary mechanism through which these services interact is through RESTful APIs in order to ensure that data flows seamlessly (Master Thesis of Lasse Kajavalta & Tempere Univeristy, 2022) . With the increased use of APIs, there is increased importance in the need to test them well so that the entire system remains stable, expandable, and reliable in the cloud(Adeleke et al., 2024).

CI/CD pipelines and testing APIs are an issue. Failure of an API may cause the entire deployment to fail. In contemporary DevOps, the testing is not only a final step, but it continuously occurs to ensure that the bugs are identified early. There are three testing types: contract testing, integration testing, and End-to-End (E2E) testing (Shahin et al., 2017). Checking contract testing ensures that the inputs and outputs of the API are as expected by other services. Integration testing examines the capacity of multiple services to interact, and it can identify misconfigurations. E2E tests execute complete user flows via public interfaces, provide confidence that real-world use is functional, but may be slow and expensive to maintain (Golmohammadi et al., 2024).

It is significant to select the appropriate API testing strategy. It influences the number of bugs that are discovered, the speed of deployment, and its reliability (Izzat & Nada, 2023). The overdependence on end- to-end tests may slow down releases and render them weak. Failure to do integration tests will allow errors into the system. Checking schemas Contract tests are fast but can be lost to problems that can only manifest when services interoperate. These trade-offs should be considered by the teams; cases, when speed, reliability, and maintainability are paramount.

Despite the fact that API testing is identified as being significant, the number of studies that compare the three primary forms of testing in actual CI/CD configurations is minimal (Arcuri, 2019). Available literature presents theoretical basis and individual case studies but

does not include controlled experiments that would compare each methodology (Atlidakis et al., 2020). This becomes particularly the case with cloud-native development, where systems have to scale, be deployed in numerous locations, and operate automatically.

This research will serve this gap by providing a comparison between contract, integration, and E2E tests in cloud-based CI/CD pipelines. In our experiment, we used containerized API along with automated pipeline and injected bugs. The metrics that were analyzed during the evaluation consist of the length of tests, number of defects that tests identify, and their portability. The results will be able to show which tests are most successful under which conditions and inform the teams about the tradeoff between speed and stability.

The research will attempt to make contributions to knowledge in two ways. First, it will present factual experimental evidence of testing trade-offs in the inventory of cloud CI/CD setting, which is not present in the current knowledge. Second, it will provide useful recommendations to software engineers and DevOps individuals that need to develop effective testing processes that may be utilized in actual cloud deployments. Such lessons have applications in theory and practice in an era where software dependability is paramount to the company and the client.

1.1. Significance of Study

The rise of the need of cloud-based CI/CD pipelines in the contemporary world of software development informs the need to undertake a severe assessment of testing strategies that would avoid endangering the consistency of the system and ingenuity in the implementation of the latter (Ullah et al., 2017). This is the case in such environments, where the microservices must communicate with each other predominantly by using RESTful within an API, where the reliability of the APIs is an essential aspect of operational excellence (Mohammed & Kiran, 2015). The evidence-based information that will be presented in the proposed study will refer to the relative effectiveness of particular API testing techniques, contract testing, integration testing, and E2E testing, within the context of realistic cloud-native environments.

The study is critical in bridging a gap in available literature that is academic. Despite various experiments that have been conducted on testing, individually, few of them have conducted a systematic comparison of those methods under identical experimental conditions, and

quantifiable performance measure. The research presented empirical data that might be used in the ongoing discussions regarding the testing pyramid and its adaptation to the cloud-based environment by assessing the time required to implement the experiment, the rate of defect detection, and the maintainability in the experimentally controlled environment (Lomotey & Deters, 2013). It also advances theoretical understanding as it brings to the fore trade-offs where practitioners have to choose between the depth of testing, speed, and reliability which contribute to the existing body of research on software testing and cloud computing and DevOps and also serves as a point of reference of further studies (Eisty et al., 2025).

In reality, those outcomes can be applied directly to software engineers, DevOps specialists, and those businesses that adopt cloud-native pipelines. Software development teams are faced with constant pressure to produce software quickly, and with low rates of production defects (Lu et al., 2013). The absence of comparative standards of API testing modalities require that numerous practitioners rely on either anecdotal experience, or direction by vendors. This study removes that barrier by presenting clear-cut and empirical facts that can be utilized in making decisions during pipeline construction (Petrillo et al., 2016). An example of this is that the results will help the teams to decide when the lightweight contract tests are adequate or when the integration tests are worth the risk profile of the system or when the E2E testing cost is reasonable as compared to the risk profile of the system.

The study has other economic and organizational implications as well (Nguyen, 2021). Unsuitable test strategies could initiate high deployment failures, customer dissatisfaction and costly post release fix . Conversely, the optimized test approaches enhance productivity, time-to-market, and enhance the confidence of the users in the cloud-based applications (Hong et al., 2018). The study will contribute to the resilience of organizations and their ability to survive in an ever-digitized economy through provision of a systematic roadmap to the selection of the testing strategy.

In general, the study is significant since it assists in filling the gap between theory and practice. It has two benefits: one, it can assist the reader to enhance their knowledge of API testing in CI/CD pipelines, and, two, it can offer the professionals in the industry with the practical suggestions. The results obtained will be highly significant to the academic and professional communities because cloud-native architectures are still on the rise to a large extent.

1.2. Problem Statement

The widespread uptake of cloud-based continuous integration / continuous delivery (CI/CD) pipelines has radically transformed the delivery of software and enabled the delivery of applications fast, automatically and continuously. RESTful Application Programming Interfaces (APIs) are very critical within these infrastructures to mediate communication between microservices that are distributed (Jani, 2023). However, the reliability of these pipelines depends on the effectiveness of API testing plans; a non-suitable testing method can lead to the existence of defects, unreliable releases, long-term deployments, and high-maintenance costs(Adeleke et al., 2024).

Although contract testing, integration testing and end to end testing are widely used, all methods have their trade off. Contract testing is efficient but has a drawback of poor workflow-level error detection (Golmohammadi et al., 2024). Integration testing does not present significant demands in terms of set-up or execution but does capture inter-service interactions. E2E testing provides the best confidence with real-life processes but is slow, fragile and costly to maintain (Shahin et al., 2017). Nevertheless, despite these clearly documented differences, comparative evidence collected in controlled experimental situations is scarce. The literature on this subject often discusses these strategies separately or in limited case studies, thus leaving practitioners with no interactive advice on the best strategy to use cloud-native pipeline situations. This lack of systematic, empirical comparison is therefore a big gap in knowledge. Without well-organized evidence, the development teams struggle to streamline the testing processes, thus fostering inefficiency and increasing exposure to the deployment risks. This research gap is unavoidable in improving the stability, efficiency, and reliability of modern cloud-based CI/CD systems.

1.3. Research Questions

This study will seek to perform a comparison of API testing procedures, namely contract, integration, and E2E testing, in cloud-based continuous integration and continuous deployment (CI/CD) pipelines. The study critically reviews these strategies by their implementation time, effectiveness with regards to the detection of defects and maintainability of the strategies and what their strengths and their weaknesses are. Through these analyses, the study will be geared towards providing evidence-based recommendations

to the software teams regarding the selection of the most appropriate API testing methods that can be useful in bringing efficiency in the deployment, system reliability and maintainability over time in cloud-native systems.

1.3.1. Main Research Question (MRQ)

Which REST API testing implementation contract testing (Pact), integration testing (REST Assured), or E2E testing (Postman) is the most suitable to include in cloud-based CI/CD pipelines (e.g., GitHub Actions, GitLab CI) as it balances performance because of under case coverage and defect and test maintenance, respectively?

1.3.2. Sub-Research Questions (SRQs)

RQ1. How does each testing approach (Pact, REST Assured, Postman) compare in terms of execution time when integrated into automated CI/CD pipelines?

RQ2. Which REST API testing approach demonstrates the highest effectiveness in detecting both seeded faults (e.g., schema mismatches, logic errors, workflow disruptions) and real API defects during automated executions in a CI/CD pipeline?

RQ3. What is the relative maintenance effort associated with contract, integration, and E2E testing approaches when deployed in dynamic cloud-based environments, particularly in terms of test flakiness, required updates after API changes, and long-term sustainability

RQ4. Under what system conditions or pipeline setups (e.g. microservices-based vs. monolithic APIs) does each testing approach deliver the most reliable, stable, and resource-efficient deployments?

1.4. Objectives

The study focusses on the following objectives:

- To conduct contract, integration and E2E API testing techniques experimentally to a cloud-based CI/CD pipeline based on containerized APIs and, therefore, providing a controlled environment to perform systematic testing.
- These methods will be measured and compared with each other basing on the assessment of the key measures such as the time taken to execute, the frequency of

detecting defects (including seeded and real ones) and the maintainability of the execution of the methods in different test executions.

- The study will make evidence-based recommendations and best practice in the selection and application of the most appropriate API testing strategy in the cloud-native software development and trade-offs between speed, reliability, and resource cost.

1.5. Contribution of the Study

The contribution of the study is in research and practice in cloud-based software engineering. It has an academic value since it makes a comparatively systematic, empirically grounded comparison of three most popular API testing modalities, including contract testing, integration testing and E2E testing, applied to cloud-native CI/CD pipelines. In a hypothetical approach of these methodologies, or in discrete case studies, the existing literature tends to discuss these methodologies, though this study presents controlled experimental data based on conventionalized measures such as the amount of time required to implement the method, the efficiency of defect detection, and the maintainability. It thus adds to the academic cognition of the inherent trade-offs between different testing methods and introduces academic logic on the applicability of the testing pyramid on the current cloud-based settings.

Ideally, the study has feasible implications to the software practitioner, DevOps engineer, and the organization intending to optimize their testing processes. The work defines the advantages and the weaknesses of each of the modalities, and in that respect, it proposes a decision-making model based on which the teams are expected to be oriented on the application of the suitable testing strategies, according to the requirements of the system and availability of the resources. This helps organizations to reduce the deployment risk, advance the system reliability and operational efficiencies in cloud-native ecosystems, including creating healthier and stronger software delivery pipelines.

1.6. Outline

The study is outlined in 5 chapters. Chapter 1 outlines the study by describing the background of the problem, purpose of the study, particular objectives, meaning of the study,

and contributions. Chapter 2 is a comprehensive survey of current literature related to API testing and cloud-based CI/CD pipelines, presupposes the presence of the agreed-upon methods of foregrounding, such as contract, integration, and E2E testing methods, and identifies gaps in the literature that justify the current study. Chapter 3 identifies the research methodology that includes experimental design, used tools and technologies, and assessment metrics, specifically the ability to reproduce cloud-based practices using Docker and CI/CD. Chapter 4 elaborates on experimental implementation and results which provides a comparative evaluation of the three testing methods based on the execution time, defect detection performance and maintainability, which is supported by statistical analysis. Lastly, Chapter 5 summarizes the findings, making conclusions about the contributions and practical implications and suggests the future research directions related to API testing strategies in the cloud-native setting.

2. Theoretical Background

API testing has become a critical factor of reliability, scalability, and performance in modern applications(Adeleke et al., 2024). The increase in the use of CI/CD pipelines has resulted in a paradigm shift, shifting testing activities out of discrete and isolated quality-assurance phases and on to continuous verification, which is effectively part of deployment processes (Jani, 2023). The most important element of inter-service communication in microservice architectures is RESTful APIs that require strict validation processes to prevent the occurrence of cascading failures in distributed systems (Golmohammadi et al., 2024). The existing body of literature outlines three main types of testing methodologies, namely contract testing, integration testing, and end-to-end (E2E testing, each of which offers different benefits and presents a set of challenges. Contract testing Contract testing is testing which ensures that services are compatible; testing which examines the interactions amongst components is called integration testing, and testing which examines workflows is called end to end testing. Though these researchers examined the effectiveness of these techniques separately, there is little comparative analyses, which would compare the trade-offs of these methods with each other through the pipelines of the operations of the CI/CD. The recent chapter thus takes into consideration the recent research, theoretical framework, and methodological developments to place the current enquiry in the larger scholarly context .

2.1. Contract Testing

Contract testing is a method of software testing that is applied to ensure that the communication between two services, usually a consumer (client) and a provider (API), is in accordance with a pre-established agreement referred to as a contract (Ren et al., 2021). The agreement outlines the anticipated request and response format, data format, endpoints, headers and status codes. As opposed to integration or end-to-end testing, contract testing does not involve the deployment or connection to the entire system. Rather the consumer develops expectations (contracts), which the provider must fulfill (Barboni et al., 2022). It particularly makes contract testing useful in microservices architectures, where there are many independent services and using APIs to communicate with each other. Contract testing dramatically minimizes integration failures, delays in a built pipeline and test costs by

identifying mismatches at an early stage, e.g. schema changes or missing fields. Pact is an example of a tool that is used in CI/CD pipelines to automate the creation and validation of the contracts so that the two parties remain compatible whenever modifications are made. Contract testing, in general, makes the process of integration more reliable and much faster by eliminating API-breaking changes prior to integration.

2.1.1. Contract Testing in Cloud-Native Systems

Contract testing has emerged as an important tool of ensuring the stability and reliability of distributed applications, particularly in cloud-native ecosystems that are loosely coupled with microservices (Simosa & Siqueira, 2023). Unlike monolithic applications, microservice-oriented architectures pursue virtually no other means of inter-service communication than APIs, necessitating stricter verification that the data that is shared between providers and consumers adheres to the preset agreements of the contract (Panahande & Miller, 2023). Contract testing therefore meets this need by focusing on the formalized contracts that define the expected structure, content and behavior of the API requests and responses.

According to Sangannagari (2024) The most prominent advantage of contract testing is that it helps to detect breaking changes at an early point in the continuous integration/continuous delivery (CI/CD) pipeline e.g., a change like a renamed field name, a new data type or a different HTTP status code in the implementation of a service provider will be detected immediately by contract tests before further spreading to integrations or E2E process. This early warning makes contract testing especially beneficial to cloud-based CI/CD systems, where regular deployments and updates are the norm. Contract testing prevents inter-service failures, thereby reducing the risk of failures downstream and hence lowering the cost of remediation defects.

Several frameworks have been developed to help with contract testing with Pact and Spring Cloud Contract being the most used. These structures allow consumer teams to express expectations in the machine-readable contract documents that are then checked against implementation by the provider (Indrasiri & Suhothayan, 2021). Besides, the introduction of Pact Brokers enables the teams to share and handle contracts within distributed environments, thus strengthening the collaborative spirit of cloud-native development.

However, despite its effectiveness, there are natural constraints of contract testing. The fact that it focuses on providing consumer boundaries can lead to the exclusion of problems that arise due to the multi-service workflows or business logic between or among components (Viggiato et al., 2018). As an example, although a contract test can be used to confirm that a booking service will respond with the right schema, it can fail to identify logical errors that occur during the service using a subsystem of payment or authentication (Izzat & Nada, 2023). Moreover, the maintenance of a repository of contracts within the fast-changing working environments will require constant alignment of teams, which can in turn create an administrative burden without the appropriate management (Eisty et al., 2025).

In the framework of the current study, the contract testing proves to be the fastest and least bulky of the three strategies under comparison (Simosa & Siqueira, 2023). It is useful at the levels of guaranteeing compatibility and preventing schema-level defects, but its overall usefulness should be judged in terms of more general performance metrics, such as defect detection coverage, execution time, and maintainability, when used as part of cloud-based CI/CD pipelines.

2.2. Integration Testing

Integration testing is a method used in testing software that is aimed at testing the interaction of individual parts or modules of a system. Integration testing is unlike that of unit testing which verifies isolated functions or methods; it also ensures that the flow of data and communication between modules is correct when put together. Integration testing is often used in API-based systems to ensure that the API is connected with external dependencies in either databases or authentication services or third-party APIs (W.T. Tsai et al., 2001). It aims to detect interface failures, misplaced data mapping, configuration errors or module incompatibility failures. Integration tests are implemented once unit tests have been succeeded before full system testing or end-to-end testing. Applications such as REST Assured can be used to automate the process of API integration testing using real HTTP requests and the verification of responses, API schemas and business logic. Integration testing is essential in CI/CD pipelines as it works out the defects at an early stage, before the deployment so that the related components are reliable. In general, the role of integration testing is to enhance the trust in the stability of the system, minimize defect leakage and allow the seamless interaction between modules in current distributed architecture.

2.2.1. Integration Testing Approaches

Integration testing plays a central role in the validation of interfaces between two or more components or services of cloud applications. Whereas contract testing will ensure that agreements between consumers and providers are isolated, integration testing will determine the behavior of services when integrated in a real or simulated environment (Bai et al., 2011). In the context of cloud-based CI/CD pipelines, where applications consist of distributed microservices, integration testing can make sure that, in addition to the API working well at the boundaries, it works when integrated with other services, databases, or other dependencies.

The key advantage of the integration testing is its ability to reveal misconfigurations, schema differences, and logical inconsistencies that could be hidden in the process of contract testing . As an example, a contract test is only able to check that an API returns data in the correct format, but only integration testing is able to reveal problems like database errors, authentication failure, or relational database conflicts (Sotiriadis et al., 2017). Integration testing has the benefit of revealing these issues, which provides more confidence in the reliability of the system before proceeding to E2E testing or even production deployment.

The following tools have gained popularity in assisting the testing of integration of RESTful APIs (Arcuri, 2019; Golmohammadi et al., 2024). REST Assured is a library written in Java often used to test endpoints according to expected output especially when using CI/CD where automation is of the most importance (Iyer, 2016). Similarly, the pytest in Python will offer a simple but robust testing framework to write an integration test, which will be flexible to include setup and tear down of fixtures like databases and outside services. Such tools can easily be combined with such platforms as GitHub Actions, GitLab CI, or Jenkins, thus facilitating the process of running the tests whenever the code has been changed.

Integration testing has its merits; however, it has a right to certain trade-offs. The big problem is that it is not as easy to establish as the contract testing. Setting up test environments that may simulate or emulate dependent services can thus be a very resource-heavy task, especially in cloud-native ones where the number of microservices is very large (Gao et al., 2011). It also has longer execution times, and the test script functioning might cost a great deal of money as the architecture of the system is modified. Also, integration tests tend to be flaky with some tests succeeding or failing occasionally due to a dependency

failure, or environmental variations, which can cause loss of confidence in the reliability of a pipeline by developers (Trad, 2023).

Within the present research, integration testing introduces a middle ground approach that reconciles between the efficacy of contract testing and complete nature of E2E testing. It gives more substantial information about service interactions, is more realistic in validation, but does not involve the overhead of running complete user workflows. The comparative analysis will put integration testing into perspective based on its functionality in terms of speed of execution, ability to detect defects and maintainability in comparison with the other methods used in cloud-based CI/CD pipelines.

2.1. E2E Testing

End-to-End (E2E) testing is a software testing method which validates the complete execution of an application, starting with the original request and ending with the final output and making sure all the integration components and subsystems works correctly as a unit. In contrast to a unit or integration test, which tests individual modules or interactions between given components, E2E testing approximates actual user conditions at the entire application stack. In the case of REST APIs, E2E testing would ensure that the API calls communicate well with databases, authentication services, external third-party systems, and user interfaces. It tests business logic and data flow, security rules, error handling and general system behavior when used in realistic operational conditions (Leotta et al., 2016). Applications like Postman have automated E2E API testing that can be used to test a sequence of dependent requests as a representation of a real user workflow. E2E testing in CI/CD pipelines is very important in making sure deployments are not regressive or functional. E2E testing is more time-intensive and more demanding in terms of maintenance but gives the greatest confidence in the reliability of the system and user experience.

2.1.1. E2E Testing Approaches

E2E testing has generally been considered the most comprehensive API testing approach because it evaluates workflows in different services in relation to the subject of real-world user interactions (Gu et al., 2023). Unlike other types of testing, such as contract and integration testing, which look at individual agreements or service-service interactions , E2E

testing ensures the entire system is functioning as desired when APIs, databases, authentication systems, and external systems are coordinated with each other. In cloud-native CI/CD pipelines, E2E testing offers an assurance that business critical scenarios would remain stable after being deployed to production (W. T. Tsai et al., 2001).

The main benefit of the E2E testing is its realism. It can be used to reveal problems that might not be detected during previous phases of testing, including data-consistency flaws, workflow breakages or security breaches in authentication and authorization processes (Zendeli et al., 2025). An example of this is in an online booking system where an E2E test would be performed to ensure that the entire sequence is correct which includes user authenticity, creating a booking, confirming payment and then accessing records. This E2E coverage makes E2E testing especially useful to systems of critical importance to the business where failure has a direct impact on user experience and business continuity.

E2E testing of RESTful APIs has an extensive range of tools, including Postman, which is one of the most popular tools used to write and run collections of tests. It has a command-line counterpart, Newman, which allows the automated running of these collections in CI/CD pipelines to generate detailed reports that allow teams to monitor execution time, pass or fail, and payload validation (Pogudin, 2023). Other tools, like Cypress and Katalon, expand E2E testing to more of a user-interface and API testing. Postman/Newman can be used in the current study because it has the optimal ratio of ease of use, automation, and the ability to integrate with cloud-based CI/CD systems.

E2E testing is limited to significant drawbacks even though it has its advantages. It can be generally considered the most resource-consuming and slowest method, because of the necessity to execute entire workflows through a variety of components. Maintaining is also expensive because a simple change in API response or workflow may require many test scripts to be changed. In addition, E2E tests are more prone to flakiness, which can be explained by the network latency, rate limiting, or dependency instability, which can compromise the trust in the test results. In response to these difficulties, practitioners frequently advise such recommendations as limiting E2E tests to important user journeys but use contract and integration tests to cover a wider area (Hu et al., 2014).

In the present study, E2E testing is most realistic and general testing layer. It will be compared and contrasted on the basis of such factors as the execution time, the suitability of

the methods to identify the defects, and maintainability, therefore, it will become evident that it will simplify the cloud-based CI/CD pipelines.

2.2. Recent Studies

Recent research continues to emphasize the significance of robust API testing in the cloud-native context, additional to the fact that firms are employing CI/CD pipelines in their continuous delivery.

Behailu Getachew Wolde & Boltana (2021) propose a REST API composition methodology to address the challenges of testing systems within the clouds. They are grounded on input domain modeling (IDM) to produce testable functions and demonstrate the behaviors through the assistance of the composition of services through REST. With a model-based black-box testing strategy, the study may be used to test E2E functionality without having access to internal code. The other behavior-driven development (BDD) and domain specific languages (DSL) are also integrated by the authors to transform REST APIs to executable test cases. The greatest contribution is the fact that the composition of REST API was proven to be effective in order to make the tests more effective, make the validation of SLA easier, and conduct tests that can be scaled up on the cloud.

Ehsan et al. (2022) synthesized a literature review (SLR) to evaluate methodologies in testing RESTful API and unit test generation. Their search went by the review protocol of Kitchenham, which considered databases like IEEE, Scopus, and ScienceDirect and found 16 high quality research. The study classifies testing methods into tools, methods, and frameworks, such as property-based testing, model-driven, fuzzing, AI-driven testing, and specification-based methods. Major contributions consist in the first complete SLR in the field, mapping the issues (like security, code coverage, and inconsistent API documentation), and pointing out the solution directions. This study forms a research foundation to assess the practices of testing RESTful APIs.

Yatini et al. (2023) conducted a study to determine the performance of PHP micro frames based on REST API development through load testing techniques. The research used four widely used frameworks, including FatFree, Lumen, Phalcon-micro, and Slim, to develop REST APIs and systematically examined them on varying workloads. In the analysis of performance, the metrics used included requests per second (RPS), latency, CPU usage, and

RAM consumption and wrk2 tool was used to run. The most valuable part of this study is the comparative microframework performance analysis in which Phalcon-micro had the best throughput, and FatFree was the most stable in terms of latency, and therefore, this study will inform the developer in choosing the microstructure when developing scalable applications on REST API.

Kim et al. (2023) proposed a novel method in testing a REST API, which combines Natural Language Processing (NLP) algorithms to automatically generate API-documentation-based test cases. They work by semantically analyzing API specifications and user stories to generate executable tests, which consume less manual effort and has better coverage. The method uses machine learning models to recognize intent and map it to API endpoints, such that meaningful test generation will be achieved. The main value of the research is the ability to prove the efficiency of NLP-based automation in improving the work of REST API tests, their relevance to the test, and their ability to undergo scalable validation in agile and cloud-native development frameworks.

Venkatraj et al. (2019) suggested a test automation model in generalized Excel format of REST API testing, which is implemented based on the Groovy scripting language. The framework automates the data reading of the input in Excel files, sending REST API requests, checking output validation, and storing outcomes; hence, less manual testing is required. The framework allows reusable and flexible automation by incorporating Java compatible libraries with Groovy. The main contribution of this work is the creation of cost-efficient, reusable, and scalable automation system, which reduces testing time and effort, and increases accuracy and especially in the situation where the functionality is often modified in the software development cycles.

Nurhayati & Agussalim (2023) developed and deployed a back-end REST API architecture of the AyamHub mobile application that was built on the Node.js Express framework and the waterfall development model. The deployment was done on the services of Google Cloud Platform (GCP) such as App Engine, Cloud SQL, and Cloud Storage, to make it scalable and reliable. The black box approach to testing was also done to confirm the correctness of endpoints and communication with the mobile client. The main value of the present work is that it shows that the use of Node.js and cloud integration can provide efficient, scalable and maintainable back-end APIs that can be adapted to real-life mobile apps.

The system of Suthendra & Pakereng (2020) was written in the Go programming language and Go-Kit in Domain-Driven Design (DDD) approach, breaking down business processes into bounded contexts and deploying the system using Docker with a database-per-service pattern. There was an API Gateway that handled endpoints and system analysis used white-box testing and resilience testing through simulated service failures. The major impact of this study is the ability to show the enhancement of microservice services in system resilience, scalability, and adaptability, which provides the loosely coupled architecture that minimizes dependencies and guarantees the efficient maintenance of large-scale web services.

Felício et al. (2023) came up with a tool RapiTest, which tests RESTful web APIs continuously, without the program being visible to it, using CI/CD pipelines. It produces and executes API requests automatically; therefore, you do not need the code, which is useful when testing the services of other vendors. RapiTest is compatible with continuous delivery pipelines and is able to identify failures and regressions in a short period of time. It was tested on a number of open-source APIs and obtained good results at finding reliability issues. The primary argument of the study is that persistent and automated black-box testing contributes to the reliability of RESTful APIs and manual labor saving agile and cloud-native environments.

Table 2. 1: Summary of the recent techniques for the API testing

Paper ID	Key Work (Main Contribution)	Method Used	Target API
Wolde & Boltana (2021)	Proposed REST API composition technique with IDM, BDD, and DSL for scalable cloud testing	Model-based black-box testing using Input Domain Modeling (IDM) and REST service composition	REST APIs in cloud-based systems (E2E functionality)
Ehsan et al. (2022)	First complete SLR on RESTful API testing and unit test generation, mapping issues and solutions	Systematic Literature Review (Kitchenham protocol)	RESTful APIs (generalized review, no single API)

Yatini et al. (2023)	Comparative performance analysis of PHP microframeworks for REST APIs under load	Load testing with wrk2 (RPS, latency, CPU, RAM)	REST APIs built with FatFree, Lumen, Phalcon-micro, Slim
Kim et al. (2023)	Introduced NLP + ML approach to auto-generate API test cases from documentation	NLP and ML-based semantic analysis for test case generation	REST APIs in agile and cloud-native environments
Venkatraj et al. (2019)	Developed Excel-based Groovy test automation framework for REST APIs	Automation framework with Groovy scripting and Excel integration	REST APIs in evolving software applications
Nurhayati & Agussalim (2023)	Built Node.js Express + GCP back-end REST API for AyamHub app with cloud deployment	Waterfall model, Node.js Express, black-box testing	AyamHub mobile app REST API
Suthendra & Pakereng (2020)	Designed microservices architecture with DDD and resilience testing in Go	Domain-Driven Design (DDD), Go-Kit, Docker, white-box + resilience testing	REST APIs in e-commerce web services
Felício et al. (2023)	Created RapiTest tool for continuous black-box REST API testing in CI/CD pipelines	Continuous black-box testing framework integrated with CI/CD	Open-source RESTful APIs (pipeline testing)

2.3. Research Gap and Proposed Research

While much work has been done on individual API testing methods such as: contract testing (Simosa & Siqueira, 2023), integration testing (Sotiriadis et al., 2017), E2E testing (Gu et al., 2023), there is a noticeable lack of empirical and comparative analysis of their use in the context of cloud-based CI/CD environments. Most previous works are theoretical discussions or isolated implementations without analyzing how different testing layers will

perform under the same pipeline conditions in terms of execution time, defect detection efficiency and maintainability.

Furthermore, available frameworks (e.g., Pact, REST Assured, Postman/Newman) are often evaluated in isolation and do not consider the interdependencies and trade-offs between them when used together in automated pipelines. Existing literature also lacks quantitative evidence about the contribution of these testing strategies to deployment reliability and operational stability in containerized microservice architectures.

This gap illustrates the need for a controlled experimental comparison of the three main API testing approaches in a standardized CI/CD setup. Addressing this will create valuable practice for DevOps teams and add new empirical evidence for academic debates about testing pyramid optimization, cloud native validation frameworks, and software quality assurance automation.

3. Research Methodology

This chapter discusses the research approach which was adopted to compare various methods of testing APIs in cloud-based CI/CD pipelines. It is experimental research that meticulously carries out and verifies three testing techniques namely: contract testing, integration testing and E2E testing (Figure 3.1). The approach describes the primary research plan, the tools and technology to be used, and the procedures to be undertaken in establishing the testing environment. It also lists the actions that have been undertaken including time taken in passing through the tests, the number of defects found and its convenience in terms of maintaining the tests to continue in business to keep the comparison updated. This open system of doing things gives a solid base that can be replicated by others and makes the outcomes dependable.

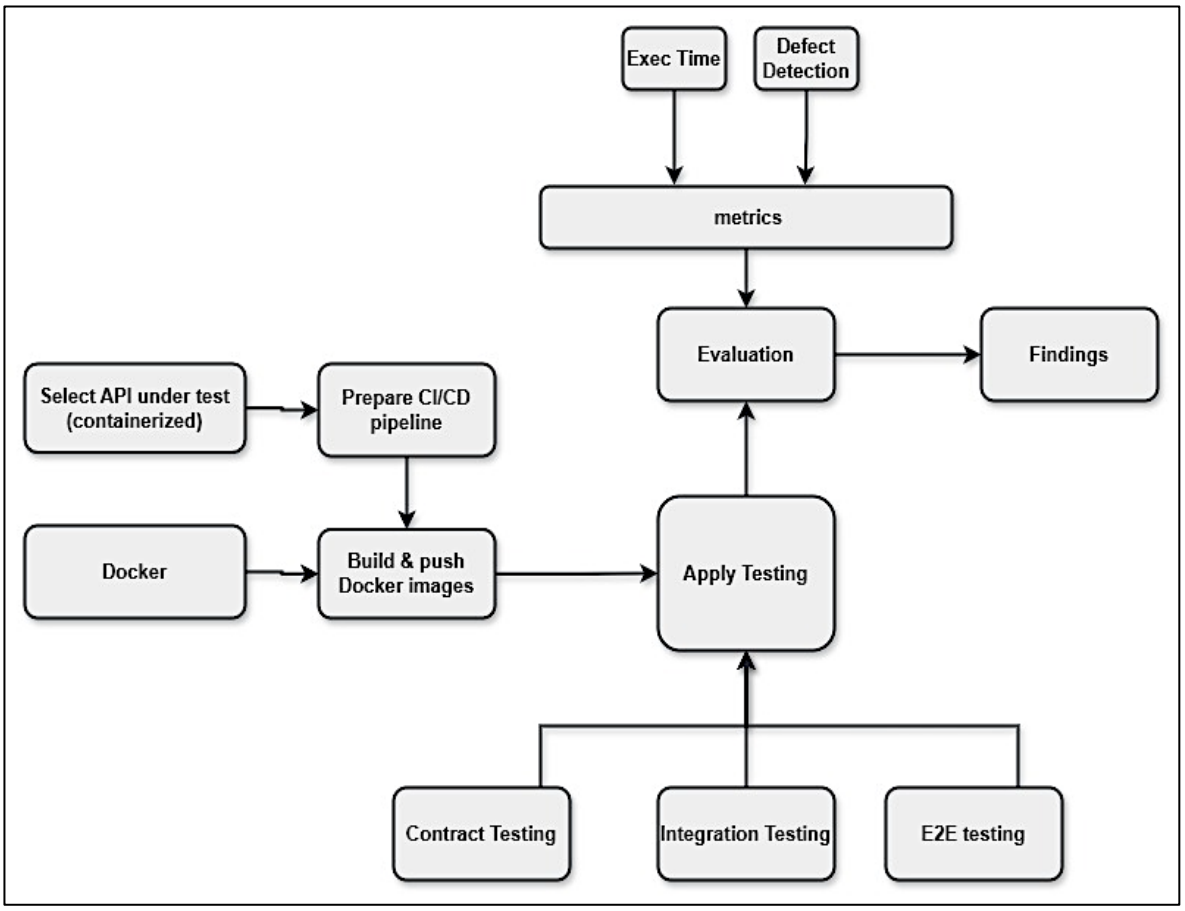

Figure 3. 1: Proposed Methodology Diagram

3.1. Research Design

This section offers an experimental design where all the approaches to testing are controlled. Each test is performed on the same API system and CI/CD pipeline (Shahin et al., 2017). The containerized API is the subject of testing, and this makes the tests consistent and repeatable. CI/CD tools such as GitHub Actions or GitLab CI are automated to build, test, and deploy.

The flow of testing begins by selecting the API to be tested and packing the API to simulate test scenarios. The ready CI/CD pipeline is then created, Docker images are created and deployed, the three testing methods are used, and the results are collected and analyzed in the end.

3.1.1. Selection and Containerization of API

First, we pick an API to test. It may either be an open-source API or a bespoke RESTful API which provides endpoints to incorporate integration and E2E testing (Ehsan et al., 2022). The API is containerized with Docker to ensure that the same environment is used every time to avoid issues that arise due to system variance.

3.1.2. CI/CD Pipeline Preparation

The CI/CD pipeline was installed after the process of containerization. This pipeline is used to automate the construction of the API, testing and evaluation. The CI/CD platform is connected to a version control system like GitHub, which means that the code can be pushed and then builds and tests are automatically carried out as a result. The pipeline stages are:

- Build - Build code and Docker images and push them to a registry.
- Test - Run contract, integration and E2E test suites.
- Export Metrics - Logs, test, and run time.
- Assess and Archive Findings- store information to analyze in future.

Such an order maintains the API constantly validated as well as adhering to DevOps best practices.

3.1.3. Developing and Implementing Docker Images

To the procedure, Docker is indispensable. The API is packaged in a Docker image, which enables it to work across all platforms and scale. An API is started with whatever required services, such as databases or message queues, by Docker Compose or similar tools.

The CI/CD pipeline is configured in a manner that these images are constructed and pushed every time a new run is being launched. This makes all experiments reproducible as each of the tests is being performed in the same container.

3.1.4. Implementation of Testing Approaches

Three particular API testing methods are used in the course of the experiment:

Contract Testing

Contract tests are used to verify that there are no differences between the API and its consumers in the rules of communication. Contracts in such tools as Pact or Spring Cloud Contract are made and verified. This makes data formats; URLs and the expected responses remain consistent and schema errors are identified early.

Integration Testing

Integration tests examine the interface of the API with other components. Instructional tools, such as REST Assured with Java or pytest with Python, verify that the API and the services, such as databases and authentication, interact. This is able to detect logic errors or configuration errors that would otherwise be overlooked by contract tests.

End-to-End Testing

E2E tests confirm the entire process of the work of the user. Postman test suites that are constructed and executed using Newman are simulated. These checks are ensured so that the API should be operating properly at all stages of its life, such as login, data storage, and responses. They consume a lot of resources but offer the best guarantee of reliability.

3.1.5. Evaluation Process

Three measures of comparison of the testing methods are as follows:

- Execution Time - the duration of each test.
- Defect Detection - This is the fraction of seeded bugs such as schema problems, logic errors, or workflow issues that each test detects.
- Maintenance - Ease of updating, maintaining and executing the tests in dynamic cloud environments.

The metrics are provided as a result of pipeline logs, reports, and a test complexity check that is performed manually.

The metrics are compared following the testing. CI/CD logs provide execution times. The judgment of defect detection is based on the comparison of anticipated and actual failures in the seeded bug cases. Maintainability is also considered based on how hard it is to adjust test cases in case there is a change in the API.

These findings are determined statistically by the number of descriptive criterion and non-parametric test to understand which approach works the best.

3.1.6. Recommendations

Lastly, we derive the analysis outcomes so that we can learn the advantages and disadvantages of the contract, integration, and E2E testing with reference to speed, accuracy, and long-term usefulness. These lessons can be used to provide evidence-based advice on the selection of an appropriate testing strategy in cloud-based CI/CD pipelines.

3.2. Pipeline Execution per Commit

The pipeline run of each commit is a bundle of automated workflow that is executed each time a developer commits the changes to a repository connected to a continuous integration/continuous delivery (CI/CD) system. The main aim behind it is to ensure that all the code changes are automatically created, tested, and verified in terms of contract, integration, and E2E testing. The methodology makes RESTful APIs to be constantly checked regarding their functionality, reliability, and maintainability, since it is implemented in the development lifecycle. The following paragraphs will outline the process step by step based on the order as shown in the sequence diagram (Figure 3.2).

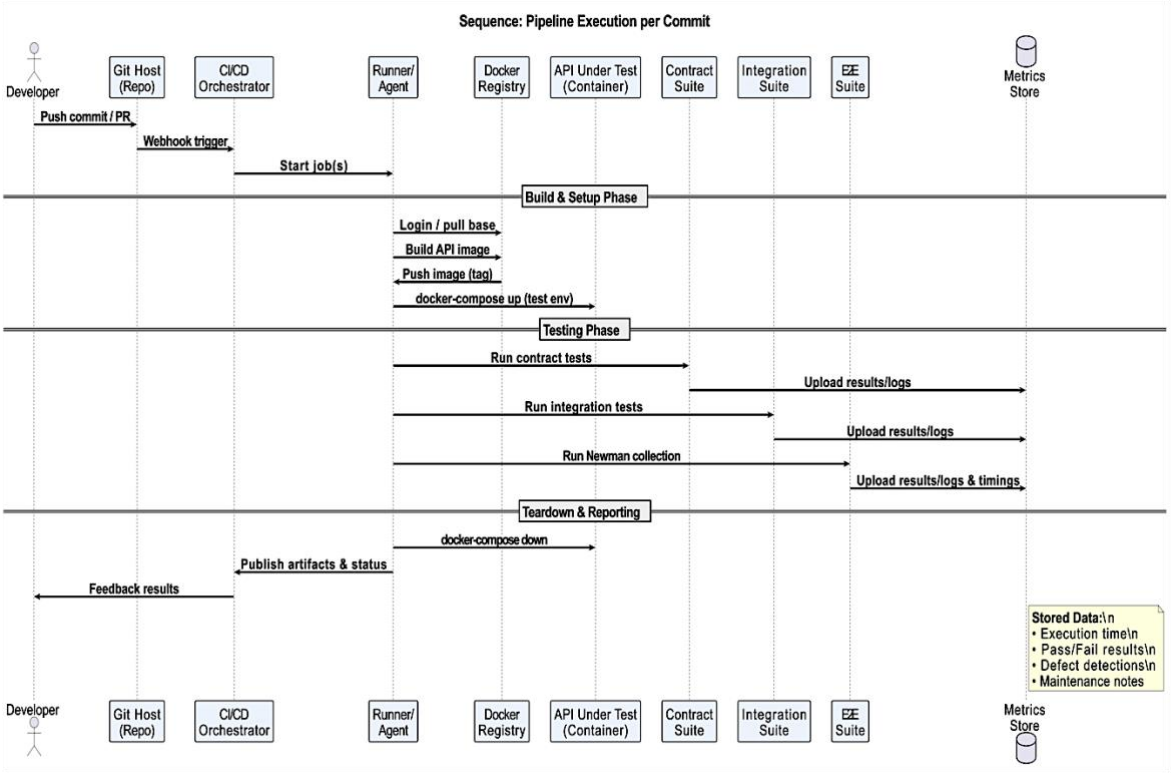

Figure 3. 2: Sequence Diagram for per execution Commit

3.2.1. Commit and Webhook Trigger by the Developer

It is triggered by a developer making a commit or pull request (PR) to the version control system (e.g., GitHub or GitLab) (Figure 3.3). This move will activate a webhook indicating to the CI/CD orchestrator to start the pipeline. The webhook is used as an automated notification system, which eliminates the necessity of a person acting to initiate testing whenever a code change is made. The practice given is core to the continuous integration where each commit is tested to avoid introducing regressions into the system.

3.2.2. CI/CD Runner Job and Orchestrator Job Initialization

On webhook activation, the CI/CD orchestrator (e.g., GitHub Actions, GitLab CI or Jenkins) allocates the task to a runner or agent. The runner is the execution environment, which controls all the tasks that have been defined in the pipeline configuration such as building the system under test, setting up the environment, executing tests and aggregating results.

The job init stage involves preparing the resources needed like the containers and scripts, and dependencies. This arrangement ensures that the environment is clean and isolated at any one time and thus removes any interference with a past execution.

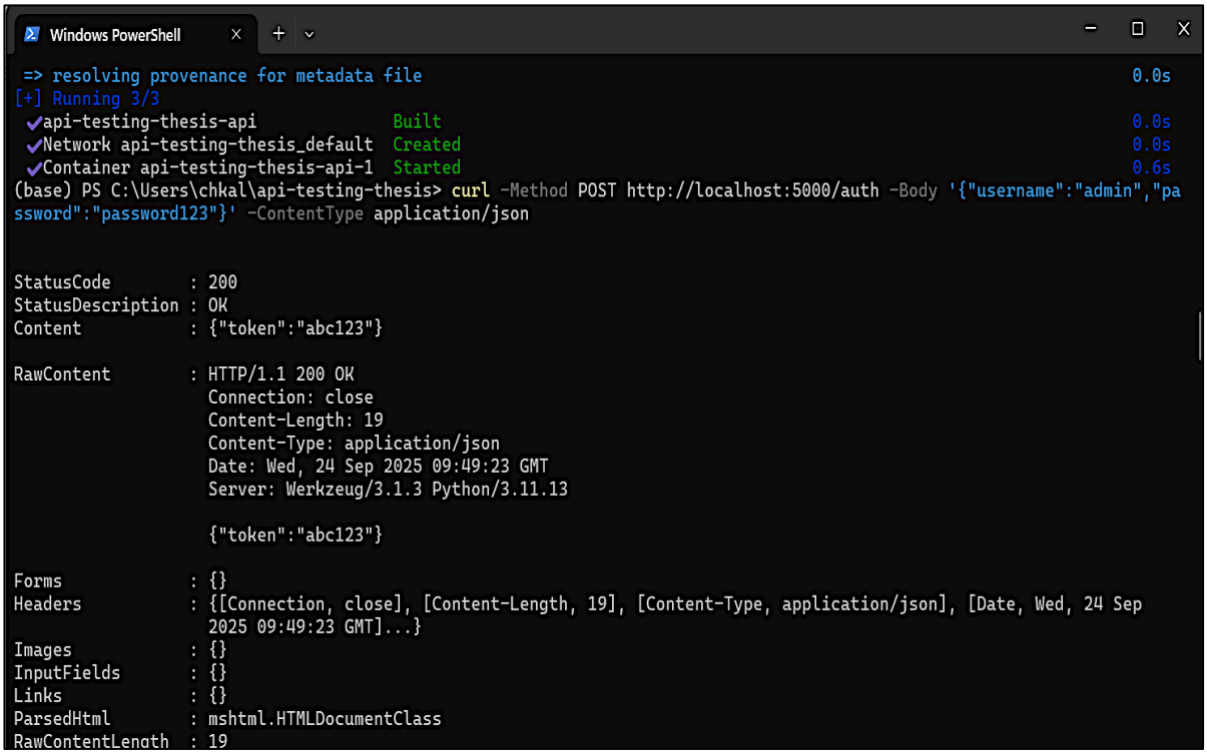

Figure 3. 3: Building a RESTful API Environment on System

3.2.3. Image Build and Deployment Docker

The runner communicates with the Docker Registry and pulls base image and builds the new API container image (Figure 3.4). Figure 3.5 is a representation of the JSON response that was initiated by the local Flask based Booking API endpoint (/booking/{id}), when accessed on localhost:5000. The result proves that the data is retrieved, and the booking information is available: the check-in and check-out dates, the state of the deposit, the name of the guest, and the final price. The response shows that API works properly and has a correct schema structure, which proves the accuracy and consistency of the endpoint in the way it processes data.

The tested API is then packed into Docker image after which it is tagged and re-pushed into the registry to enable versioning and reproducibility. The runner uses docker-compose to start the testing environment that consists of the API container as well as other services that

the former may need (e.g., databases or message queues). Through Docker, the process promotes environmental consistency, whereby the API will have the same behavior regardless of the place of execution.

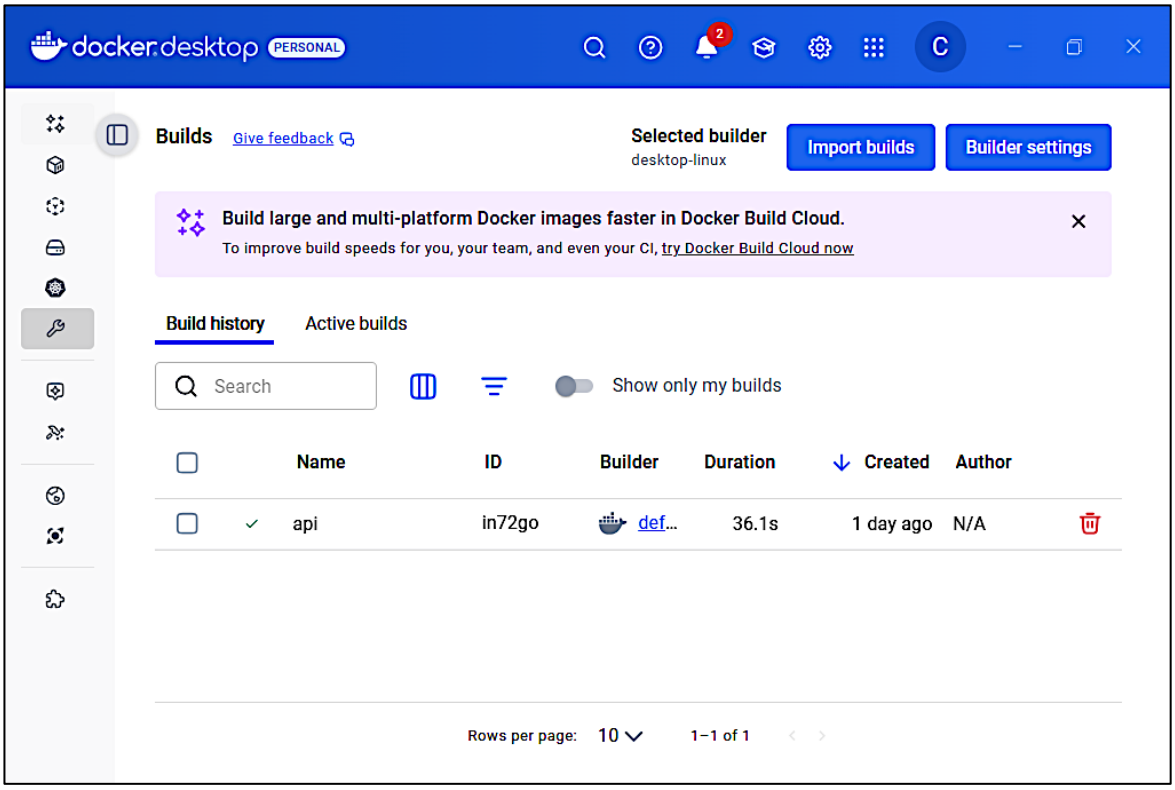

Figure 3. 4: Containerization Process of the API

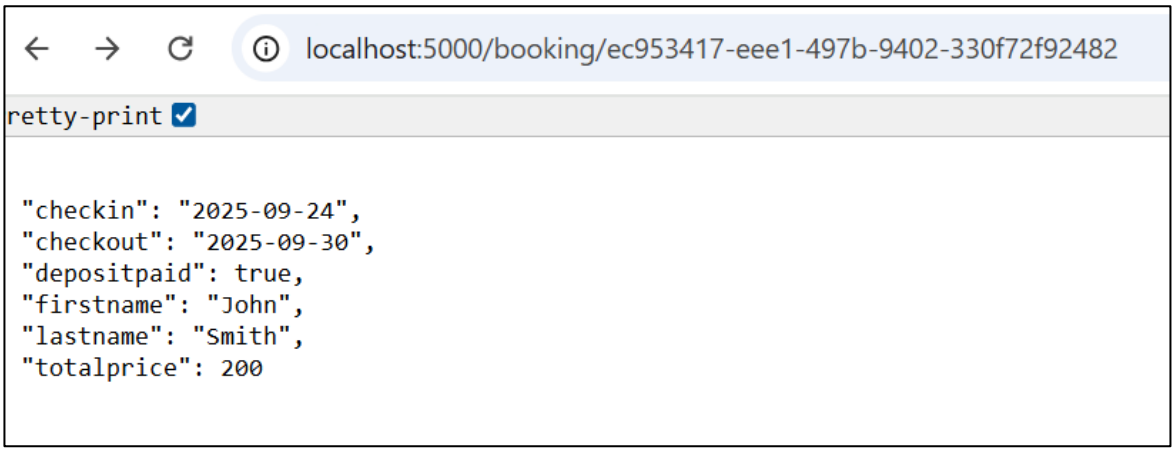

Figure 3. 5: Sample output on localhost for the API use for booking

3.2.4. Execution of Testing Suites

After the environment is up, the pipeline is put under testing process and the three testing methodologies that constitute the core of this methodology are applied.

This study has conducted three categories of automated API test suites, which are Contract Testing (Pact), Integration Testing (REST Assured), and End-to-End Testing (Postman). The suites are aimed at another level of the API testing pyramid and are provided with a specific testing tool.

Contract Testing (Pact – Consumer-Driven Contracts)

Contract tests ensure that both the consumer and provider of API agree on a common interface. The consumer application was used to generate mock interactions with the help of Pact, which are saved in the form of a contract file (.json) as shown in Figure 3.5. In the course of the execution, the provider application checked the conformance of its responses to the expectations stipulated in the contract (status codes, JavaScript structure, mandatory fields).

- Procedure:
    - The Pact file is created through consumer tests.
    - The contracts are stored at Pact Broker.
    - Provider verification step is used to compare actual output of the provider with the stored contract.
- Output: Pass/Fail (no UI/network workflow).

Integration Testing (Java Rest Assured -Java API Automation Framework)

Integration testing provides general confirmation of the cooperation between several internal modules (controller - service - repository) in the context of actual execution of real HTTP requests.

Requests were made to specific API endpoints with the assistance of the REST Assured tool and the accuracy of response was asserted on the basis of the real system logic.

- Procedure:

  - Tests are performed against actual deployed database access.
  - Statements made against: HTTP status code, headers, JSON response body and data persistence.

- Output: API-level functional verification, no interaction with the UI.

E2E testing (Postman + Newman in CI/CD)

End-to-End tests model the entire workflow that a real user would use through chain calls of the API.

Postman In Postman, a library was developed to verify, authenticate, generate resources, update, and test output of several API modules. The suite has been conducted using Newman in CI/CD.

- Procedure:
  - The information sent out is dynamically chained to each request.
  - The environment variables are used to store id and tokens.
- Output: Makes sure that the entire system is functioning properly by request - processing - database - response.

The test cases and the details of injected defects/injected bugs are detailed and analysed in the Results and Evaluation chapter.

Environment Teardown

Once all tests' suites are completed the runner runs docker-composed down and stops the containers and removes resources. This will reduce the amount of leakage that resources experience and make every run of the pipeline start with a clean environment.

3.2.5. Storage and Reporting Results

During testing phases, the results of the testing process such as execution time, pass/fail, and defects identified, and maintainability notes are continually uploaded to a central metrics repository. This store is like the basis of further analysis, as the three testing strategies can be comparatively assessed.

Lastly, the CI/CD orchestrator documents the artifacts and the health back to the repository. With the help of detailed logs and reports, developers will be able to immediately decide whether their commit was successful or not. This is the loop that cannot be avoided in continuous delivery as it ensures that only verified code is passed on to deployment phases.

3.2.6. Importance of the Pipeline Execution Process

Pipeline execution process per commit makes the process automated, repeatable and scalable. All commits go through the same intensive process, which facilitates the early identification of errors, minimizes the price of bugs, and improves reliability of the system. The approach implements a self-contained validation framework by combining Docker, CI/CD coordination, and multi-test strategies, the approach is cloud-native development-specific.

3.3. Methodology Summary

This is the chapter that outlines the methodology that has been developed to conduct a comparative evaluation of the API testing approaches such as contract testing, integration testing, and E2E testing in the context of cloud-based continuous integration/continuous deployment pipelines. The research used an experimental design ensuring that it is reproducible through containerized APIs, Docker-based deployment, and complete automated CI/CD workflows. The process started with the choice of the target API and containerization followed by the pipeline setup and automated Docker image building. All the testing strategies were utilized in a systematic manner: service contracts were tested using contract tests; inter-component tests were used to test inter-component interactions; E2E tests were used to test complete system workflows. Measures, such as the execution time, the rate of defect detection, and the maintainability score, were automatically extracted by the logs of CI/CD and test reporters. The chapter also expounds on the process of evaluation where the results were accumulated and examined to identify trade-offs between the approaches. The methodology provides a strong platform by incorporating automation, containerization and stringent assessment to accomplish effective and dependable comparative testing.

4. Results

This chapter includes the experimental findings on the comparative study of contract, integration, and E2E API testing strategies in the framework of a cloud-based CI/CD pipeline. The results are caused by a series of controlled test execution on a containerized RESTful API through automated processes. All testing methods have been tested in the same environment to be fair and reproducible. The dimensions that are being analyzed include three, execution time, ability to detect defects and maintainability. These findings offer both quantitative and qualitative information about the performance and trade-offs of the individual testing strategy as the groundwork to further discussion and the following recommendations to the DevOps teams.

4.1. Evaluation measures

The analysis of the three API testing methods was done based on three primary metrics, which include execution time and rate of defect detection(Table 4.1). Execution time is the duration of time taken by the given testing suite to run in the CI/CD pipeline which demonstrates its efficiency. Defect detection rate (DDR) is a measure of the reliability and accuracy of the test by determining the percentage of seeded defects and real defects detected during the test.

Table 4. 1: Evaluation Metrics Details

Metric	**Definition**	**Source**
Execution Time	How long each test suite takes to run	CI/CD logs
Defect Detection Rate	% of injected/seeded defects detected	Test reports

Table 4. 2: Injected Defect Details and their Endpoint

ID	**Type of Bug**	**Description / Change Made**	**Affected Endpoint**

D1	Authentication bypass	Accepts any credentials and always issues a token, removing authentication enforcement.	/auth (POST)
D2	Response schema bug	Returns token under the wrong key name 'access_token' instead of 'token' for valid login.	/auth (POST)
D3	Status code bug	Returns HTTP 201 Created instead of 200 OK on successful authentication.	/auth (POST)
D4	Response payload bug	Returns a message without a token field even when credentials are correct.	/auth (POST)
D5	Credential handling bug	Only accepts username 'Admin' (case-sensitive), causing valid 'admin' credentials to fail.	/auth (POST)
D6	Type/serialization bug	Stores totalprice as a string instead of a numeric value after validation.	/booking (POST)
D7	Response schema bug	Create booking response omits bookingid and only returns the booking object.	/booking (POST)
D8	Status code bug	Returns HTTP 201 Created instead of 200 OK for successful booking creation.	/booking (POST)
D9	Data corruption bug	Modifies the returned booking to use fake firstname/lastname values in the response.	/booking (POST)
D10	Persistence bug	Does not persist the newly created booking in the in-memory database.	/booking (POST)

D11	Validation/schema bug	Makes 'depositpaid' optional, so bookings missing this required field are accepted.	/booking (POST)
D12	Content-type/schema bug	Returns a plain text bookingid string instead of a JSON response object.	/booking (POST)
D13	Not-found handling bug	Returns HTTP 200 OK with an empty JSON object when the booking ID does not exist.	/booking/<bid> (GET)
D14	Response shape bug	Wraps the booking under a 'booking' key instead of returning the booking object directly.	/booking/<bid> (GET)
D15	Response shape bug	Returns the booking inside a single-element JSON list instead of an object.	/booking/<bid> (GET)
D16	Status code bug	Returns HTTP 204 No Content on successful GET instead of 200 OK.	/booking/<bid> (GET)
D17	Status code bug	Returns HTTP 200 OK with an error JSON body when the booking is not found instead of 404.	/booking/<bid> (GET)
D18	Existence/semantics bug	Implements upsert behaviour for PUT, creating a booking even when the ID does not exist.	/booking/<bid> (PUT)
D19	Data loss bug	Drops the 'depositpaid' field when updating a booking via PUT.	/booking/<bid> (PUT)
D20	Response consistency bug	Updates the booking but returns the old (pre-update) booking data in the response.	/booking/<bid> (PUT)

D21	No-op update bug	Ignores the PUT request body and leaves the stored booking unchanged while returning it.	/booking/<bid> (PUT)
D22	Status code bug	Returns HTTP 201 Created instead of 200 OK for a successful full update.	/booking/<bid> (PUT)
D23	Existence/semantics bug	PATCH creates a booking if it does not exist instead of returning 404.	/booking/<bid> (PATCH)
D24	Patch semantics bug	PATCH overwrites the entire booking with the patch body instead of merging fields.	/booking/<bid> (PATCH)
D25	Patch semantics bug	PATCH drops fields with null or empty values instead of updating them as provided.	/booking/<bid> (PATCH)
D26	Status code bug	Returns HTTP 200 OK instead of 204 No Content for a successful delete.	/booking/<bid> (DELETE)
D27	Existence/semantics bug	Returns 204 No Content even when the booking ID does not exist (delete is silently successful).	/booking/<bid> (DELETE)
D28	Data integrity bug	Deletes a different booking (first in DB) rather than the specified booking ID.	/booking/<bid> (DELETE)
D29	Health-check response bug	Returns body 'PONG' instead of 'OK' for the ping endpoint.	/ping (GET)
D30	Health-check status bug	Returns HTTP 500 Internal Server Error instead of 200 OK for the ping endpoint.	/ping (GET)

4.2. Contract Testing Analysis

The contract testing experimental setup was the creation of a consumer-provider architecture to test the API interactions with Pactman and Flask. The consumer tests have been created in Python and tested using pytest to produce the Pact contract files containing the anticipated requests and responses of two main endpoints: /auth and /booking (Table 4.2). The provider service, which was run using Flask framework, acted as a simulation of a real booking API that was locally running on the host http://localhost:5000. Pactman Verifier was then used to compare the live provider with the defined consumer contracts stored so that there is consistency in the expectation and the actual response. All the experiments were run in an isolated python virtual environment to ensure control of dependencies. Measures of performance like execution time, defect detection rate, maintainability and stability were measured over several runs. This device created a controlled and repeatable context to determine the dependability and precision of contract-based API test in a continuous integration procedure.

4.2.1. Verification Results

The experiment of contract testing was able to test the interactions between the consumer and provider APIs with Pactman successfully (Table 4.3). The endpoints that were tested were two, namely /auth and /booking, and their Pact contracts. The testing was conducted on the locally deployed Flask API (http://localhost:5000) so that the expectation of the consumer and the response given by the provider match.

Table 4. 3: Contract Testing for Booking API

End point	**Consumer Pact File**	**Verification Command**	**Result**	**Remarks**
/auth	pacts\booking_consumer_auth-booking_provider-pact.json	pactman-verifier booking_provider http://localhost:5000 http://localhost:5000 -l pacts\booking_consumer_auth-	Passed	Validated token generation; provider state “valid creds” warning ignored.

		booking_provider-pact.json		
/booking	pacts\booking_consumer_booking-booking_provider-pact.json	pactman-verifier booking_provider http://localhost:5000 http://localhost:5000 -l pacts\booking_consumer_booking-booking_provider-pact.json	Passed	Validated booking creation; provider state "booking can be created" warning ignored.

These pre-tests were done without injected defects to initially ensure that the consumer provider communication was operating properly, and the contracts were being fulfilled. It is necessary to create a baseline of bug-free interaction, since only after confirming that the system operates the way it is expected to, can we meaningfully place failures in the later phases to determine how much the system detects defects and is stable. Both tests succeeded with 200 ok answers, which testified to the fact that the provider met the specifications specified in the contract. There were minor warnings about provider states (e.g., valid creds and booking can be created) but this did not influence the general validation (Figure 4.1 and 4.2). These findings support the fact that contract testing is an effective tool that can verify API compatibility prior to integration, which provides reliability and stability in the communication of microservices. The experiment showed that Pactman is a credible framework to automate the process of testing consumer-driven contracts with great accuracy and minimum execution time.

```
s> pactman-verifier booking_provider http://localhost:5
000 http://localhost:5000 -l pacts\booking_consumer_aut
h-booking_provider-pact.json
Consumer: booking_consumer_auth
Request: "login" ... PASSED
 Invalid provider state [{'name': 'valid creds', 'param
s': {}}]
```

Figure 4. 1: Contact Testing Passed for the Consumer Authorization

```
s> pactman-verifier booking_provider http://localhost:5
000 http://localhost:5000 -l pacts\booking_consumer_boo
king-booking_provider-pact.json
Consumer: booking_consumer_booking
Request: "create booking" ... PASSED
 Invalid provider state [{'name': 'booking can be creat
ed', 'params': {}}]
```

Figure 4. 2: Contract Test Pass for the Booking process

Table 4.4 indicates that Contract testing identified three defects (D2-D4) which were all connected with the /auth response structure, e.g. incorrect key name or incorrect status code. These modifications violated the consumer-provider contract and led to the failure of Pact verification and demonstrated that contract testing is useful at identifying some schema-level defections on covered interactions. The majority of other injected bugs (D1, D5-D30) were however undetected since they did not disrupt the pact interactions but instead maintained the anticipated response form. This proves that contract testing in itself cannot unveil numerous logical, state-based or multi-endpoint defects hence the need to complement it with integration and E2E testing.

Table 4. 4: Defect Injection Result for the Contract Testing

BUG ID	**Tests Passed**	**Detected by Contract Testing**
D1	TRUE	FALSE
D2	FALSE	TRUE
D3	FALSE	TRUE
D4	FALSE	TRUE
D5	TRUE	FALSE
D6	TRUE	FALSE
D7	TRUE	FALSE
D8	TRUE	FALSE
D9	TRUE	FALSE
D10	TRUE	FALSE
D11	TRUE	FALSE
D12	TRUE	FALSE
D13	TRUE	FALSE

D14	TRUE	FALSE
D15	TRUE	FALSE
D16	TRUE	FALSE
D17	TRUE	FALSE
D18	TRUE	FALSE
D19	TRUE	FALSE
D20	TRUE	FALSE
D21	TRUE	FALSE
D22	TRUE	FALSE
D23	TRUE	FALSE
D24	TRUE	FALSE
D25	TRUE	FALSE
D26	TRUE	FALSE
D27	TRUE	FALSE
D28	TRUE	FALSE
D29	TRUE	FALSE
D30	TRUE	FALSE

The outcomes of the contract testing shown in Table 4.5. The recorded time of the contract testing run was 55 seconds which demonstrates that Pact-based verification is fast and lightweight, even when tested against several injected defects. The rate of detection of defects is, however, only 10% (3 out of 30 bugs), which implies that the majority of injected issues did not lead to the failure of contracts. It is so since the pact addresses only the /auth interaction and is concerned with the structure of responses and schema (status code, field names, presence of token). Still, it passed the contract tests with bugs that maintained the same shape with the difference in internal logic, data persistence or cross-endpoint behavior. In general, contract testing is effective and useful in the detection of schema-level regressions, but it has limited coverage and will need to be supplemented with integration and end-to-end testing.

Table 4. 5: Evaluation Metrics Score for Contract Testing

Type	**Sub Type**	**value**
Execution Time	test execution	55 sec

Defect Detection Rate	-	10%

4.3. Integration Testing Analysis

The integration testing was carried out to ensure that several API endpoints interacted with each other within the entire workflow. It started by running all the test cases in the test's integration directory with pytest. The tests were conducted one right after another and engaged in activities such as authentication through /auth, booking through /booking, and the validation of field requirements. The test suite was run with reporting on (--junitxml and --html flags) and formatted results have been obtained. There were 12 test cases that were executed, which consisted of authentication, creation of bookings, and validation. Every test was passed, and this is an indication of uniform integration among components. To determine the functional correctness and cross-endpoint data flow integrity, execution time measures were studied and endpoint interactions.

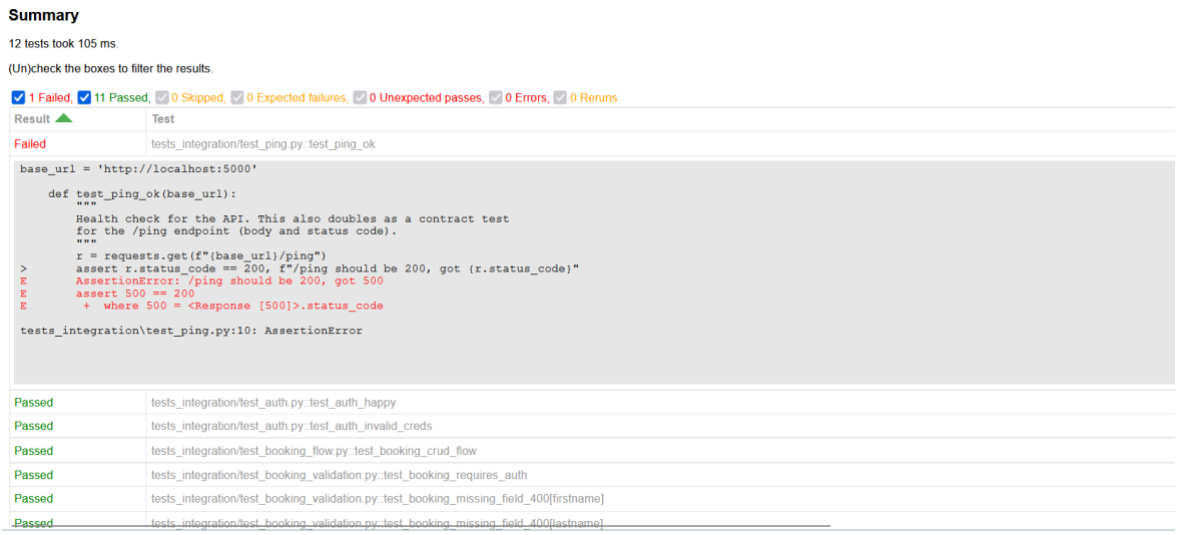

Figure 4. 3: Sample Integration Testing Report for Defect 29

4.3.1. Verification Results

The results of integration testing show that the majority of defects were identified as shown in (Figure 4.3), which proves the success of integration testing in revealing the communication and configuration problems between services (Table 4.6). The results of the integration testing indicate that the detection capability is very good with 24 of the 30 injected bugs (80%) being detected. Most of the schema- and behavior-related problems,

including incorrect status codes, missing fields, and different response structures within the booking and auth flows (i.e. D1-D4, D6-D17, D19-D22, D24, D26, D29-D30) were rightfully identified. This affirms that the integration suite is efficient, in that it validates endpoint contracts, cross flows, and configuration among components. Nevertheless, 6 defects (D5, D18, D23, D25, D27, D28) were not identified. These are less noticeable logical or state-based anomalies, e.g. upsert like updates, wrong delete semantics, patch behaviors that do not invalidate the high-level assertions. This evidences that integration testing is potent and will have to be accompanied by other checks of more profound business logic and data semantics.

Table 4. 6:Defect Injection Results for the Integration Testing

BUG ID	**Tests Passed**	**Detected by Integration Testing**
D1	FALSE	TRUE
D2	FALSE	TRUE
D3	FALSE	TRUE
D4	FALSE	TRUE
D5	TRUE	FALSE
D6	FALSE	TRUE
D7	FALSE	TRUE
D8	FALSE	TRUE
D9	FALSE	TRUE
D10	FALSE	TRUE
D11	FALSE	TRUE
D12	FALSE	TRUE
D13	FALSE	TRUE
D14	FALSE	TRUE
D15	FALSE	TRUE
D16	FALSE	TRUE
D17	FALSE	TRUE
D18	TRUE	FALSE
D19	FALSE	TRUE
D20	FALSE	TRUE

D21	FALSE	TRUE
D22	FALSE	TRUE
D23	TRUE	FALSE
D24	FALSE	TRUE
D25	TRUE	FALSE
D26	FALSE	TRUE
D27	TRUE	FALSE
D28	TRUE	FALSE
D29	FALSE	TRUE
D30	FALSE	TRUE

The integration test was determined based on efficiency of the execution and level of detecting the defects. The overall duration of the execution of the entire suite on all the defects injected (155 seconds) shows that the overall API workflow (authentication, booking creation, update, deletion, and verification) was quite efficient, even with repeated end-to-end interactions as shown in (Table 4.7). This indicates that component and network overhead communication did not cause any major bottleneck in performance. The defect detection rate of 80% indicates that out of 30 defects that had been introduced deliberately, the integration tests detected 24 of them. These were largely interface and flow issues, including discrepancies in status codes, missing or bad fields and improper behavior among booking lifecycle operations. Nonetheless, some minor group of less obvious logical and state-related problems survived, which points to the possibility of reinforcing integration statements concerning complicated business regulations and edge cases.

Table 4. 7:Evaluation Metrics Score for the Integration Testing

Type	**Sub Type**	**Score**
Execution Time	test execution	155 seconds
Defect Detection Rate	-	80%

4.4. E2E Testing Analysis

E2E testing does the overall test of the application in a start-to-end way, with all integrated components operating in a specific manner. Newman was used to carry out E2E testing in

this project by running Postman API collections in a Dockerized environment. This was started by the deployment of the Flask API container and then the Newman container was run to emulate the user interactions like authentication, booking creation, retrieval, update and deletion. Response to each request was tested as of proper status codes, response structure, as well as the consistency of data. This was a holistic method that had to be used to make APIs work perfectly on every endpoint and in a way that imitated the real-world user behavior (Figure 4.5).

	executed	failed
iterations	1	0
requests	7	0
test-scripts	7	0
prerequest-scripts	0	0
assertions	13	0
total run duration: 202ms		
total data received: 778B (approx)		
average response time: 11ms [min: 3ms, max: 58ms, s.d.: 19ms]		

Figure 4. 4: Sample E2E Testing Report for the Bug-ID-1

4.4.1. Verification Results

In the E2E testing verification, there is a moderate yet significant capability to validate the end-to-end API workflow. At 30 injected defects, 17 defects were found that shows Defect Detection rate of 56.7% as shown in (Table 4.8 & 4.9). The majority of bugs that were caught (D2-D4, D7-D10, D12-D17, D20-D22, and D26) concerned either clarity of functional or flow related issues, such as wrong status codes, broken response schemas, and failures throughout the booking lifecycle. This proves that the Newman based E2E suite is efficient in identifying observable API level failures that destroy the primary happy path cases. But several defects (e.g. D1, D5, D6, D11, D18, D19, D23-D25, D27-D30) were not spotted since they did not disrupt the main flow but corrupt internal logic or state. This

underscores the necessity to enhance assertion and to add more cross endpoint and data-driven checks to enhance E2E coverage.

Table 4. 8: Defect Injection Result for the E2E Testing

BUG_ID	**Tests Passed**	**Detected by E2E**
D1	TRUE	FALSE
D2	FALSE	TRUE
D3	FALSE	TRUE
D4	FALSE	TRUE
D5	TRUE	FALSE
D6	TRUE	FALSE
D7	FALSE	TRUE
D8	FALSE	TRUE
D9	FALSE	TRUE
D10	FALSE	TRUE
D11	TRUE	FALSE
D12	FALSE	TRUE
D13	FALSE	TRUE
D14	FALSE	TRUE
D15	FALSE	TRUE
D16	FALSE	TRUE
D17	FALSE	TRUE
D18	TRUE	FALSE
D19	TRUE	FALSE
D20	FALSE	TRUE
D21	FALSE	TRUE
D22	FALSE	TRUE
D23	TRUE	FALSE
D24	TRUE	FALSE
D25	TRUE	FALSE
D26	FALSE	TRUE
D27	TRUE	FALSE

D28	TRUE	FALSE
D29	TRUE	FALSE
D30	TRUE	FALSE

Table 4. 9: Evaluation Metrics Result for E2E Testing

Type	Sub Type	Score
Execution Time	test execution	230s
Defect Detection Rate	-	56.7%

4.5. Cross-Comparison

The cross-comparison indicates that there are distinct differences in strength of the three methods. The best method is integration testing, as it identifies 24/30 bugs, and nearly all schema, status-code, and flow-related end-point bugs as shown in (Table 4.10). E2E testing identifies 17/30 bugs, and all the defects it identifies are also identified by integration tests, which means that it adds realism (full workflow via Newman) but no special detects in this experiment. The smallest is contract testing which only identifies 3/30 bugs (D2-D4) all involving the /auth response schema, and it never finds a defect that is missed by integration or E2E. Six bugs (D5, D18, D23, D25, D27, D28) elude all three, meaning that there are minor logical or business-rule flaws that need more specific or detailed test oracles.

Table 4. 10: Defect Injection Results Comparison

BUG ID	**Contract Testing**	**Integration Testing**	**E2E testing**
D1	FALSE	TRUE	FALSE
D2	TRUE	TRUE	TRUE
D3	TRUE	TRUE	TRUE
D4	TRUE	TRUE	TRUE
D5	FALSE	FALSE	FALSE
D6	FALSE	TRUE	FALSE
D7	FALSE	TRUE	TRUE

D8	FALSE	TRUE	TRUE
D9	FALSE	TRUE	TRUE
D10	FALSE	TRUE	TRUE
D11	FALSE	TRUE	FALSE
D12	FALSE	TRUE	TRUE
D13	FALSE	TRUE	TRUE
D14	FALSE	TRUE	TRUE
D15	FALSE	TRUE	TRUE
D16	FALSE	TRUE	TRUE
D17	FALSE	TRUE	TRUE
D18	FALSE	FALSE	FALSE
D19	FALSE	TRUE	FALSE
D20	FALSE	TRUE	TRUE
D21	FALSE	TRUE	TRUE
D22	FALSE	TRUE	TRUE
D23	FALSE	FALSE	FALSE
D24	FALSE	TRUE	FALSE
D25	FALSE	FALSE	FALSE
D26	FALSE	TRUE	TRUE
D27	FALSE	FALSE	FALSE
D28	FALSE	FALSE	FALSE
D29	FALSE	TRUE	FALSE
D30	FALSE	TRUE	FALSE

The comparative analysis shows that there is a definite trade-off between the execution time and the coverage of defects among the three approaches as shown in (Table 4.11). Contract testing has the shortest time (55 seconds), but the lowest defect detection rate (DDR) (10%) because it focuses on schema- and interface-level defects on a small number of interactions. Integration testing is slower (155 seconds) but has the best DDR of 80 percent, as it checks the interaction between endpoints and components as well as is good at identifying structural, behavioural, and cross-endpoint faults. End-to-end (E2E) testing takes the longest (230 seconds) and has a DDR of 56.7 and is realistic in workflow validation, although still

has a few minor logical bugs. All in all, the findings substantiate a stratified approach: quick contract verification, more profound integration testing and E2E flows coupled with more robust assurance.

Table 4. 11: Evaluation Score Comparison

Type	Contract testing	Integration Score	End 2 End Testing
Execution Time	55 sec	155 seconds	230s
Defect Detection Rate	10%	80%	56.7%

5. Discussion

5.1. Trade-off Analysis

The reported results are a clear indication of a trade-off between speed and covering defects across the three testing methods. The quickest was contract testing, which took approximately 55 seconds to complete but had a low defect detection rate of 10% as it primarily tests API schemas and communication contracts and most errors between endpoints. Integration testing took more time (155 seconds) and provided best coverage (80% DDR) by testing both endpoints interaction and exposing structural, behavioral, and flow defects. End-to-end (E2E) testing was the slowest (230 seconds) with a DDR of 56.7 but offered realistic testing of overall user processes, business logic, and lifecycle functions. In general, the tradeoff underlines a layered approach, which involves quick contract validation and vigorous integration testing and supplements it with infrequent E2E testing to provide functional certainty.

5.2. Strengths

The study has provided a strong and organized analysis of API reliability with the help of a layered testing approach. The systematic defect-injection approach on 30 controlled bugs (D1-D30) was also a strong point of the study, as it provided an opportunity to compare quantitatively the Contract, Integration, and End-to-End (E2E) testing, which is based on a quantitative measure of Defect Detection Rate (DDR) and execution time. Its findings established that contract testing is useful in detecting a small proportion of schema- and contract-level inconsistencies (10% DDR), whereas integration testing offered the best coverage (80% DDR) by testing inter-endpoint interactions, as well as response behaviour. E2E testing also confirmed realistic user workflows at a moderate DDR (56.7 percent) which validated its worthiness in the inspection of entire business procedures. The other strength was the provision of automated Python runners and Docker-based surroundings which allowed reproducible, isolated, and repeatable experimentation of every BUG_ID. Lastly, the use of objective metrics (execution time and DDR) on consistency gave a clear and

measurable picture of the trade-offs between speed, depth, and coverage of every testing technique.

5.3. Limitations

Although the research delivered significant information, a few limitations should be noted.. To start with, even though 30 defects had been injected, they remained hand-crafted, artificial bugs that were centered around HTTP status codes, schema structure, simple logic defects and lifecycle semantics of just one API, namely booking style. Bigger classes of problems like concurrency problems, race conditions, heavy-load bottlenecks, security vulnerabilities, and upstream dependency failures are not addressed in the experiment and are prevalent in practice in a microservice ecosystem. Second, all defects were switched using a BUG_ID switch in only one service, as opposed to being spread out in many independent microservices with asynchronous communication or unreliable networks. This is to say that the environment remains a simplified API architecture, and not an entirely realistic enterprise-scale landscape.

Third, although experiments performed automated Contract, Integration and E2E tests on all BUG_IDs, test oracles are quite narrow: they focus on a pre-determined set of flows and assertions. Consequently, several minor logic and state-related bugs (such as D5, D18, D23, D25, D27, D28) were not identified through any method, which means that business rules and edge cases coverage is still not complete. Also, it was only measured at the total level of the suite time (55 s, 155 s, 230 s) not at different loads or scaling conditions, hence, only conclusions are restricted to the context of single-user and single-instance.

On the whole, although the results are informative in the context of the behavior of the three testing methods on a controlled set of defects, further research should involve: (i) automated or fuzz-based defect generation, (ii) experiments on a multi-service, event-driven architecture, (iii) richer, more data-driven assertions about the integration and E2E layers, and (iv) non-functional faults (performance, resilience, security) in order to enhance the generalizability and realism of the findings.

5.4. Future Work

The future study ought to be directed at the expansion of this framework in the direction of AI-assisted automated defect prediction and adaptive testing orchestration. The machine learning models may be trained using the results of previous tests and thus can dynamically give more priority to high-risk areas. System reliability testing would be enhanced with the incorporation of non-functional tests dimensions, including performance, load and security validation. Also, it is possible to install the contract evolution testing to make API resilient in case providers change schemas or introduce new endpoints. Generalizability could be improved by increasing the number of injected defects and performing research in multi-service microservice setups. Lastly, by incorporating this framework into a continuous testing pipeline, this may integrate the process of regression detection and would enable faster feedback loops and incentivize DevOps-based quality assurance throughout the software lifecycle.

6. Conclusion

The work provides an adhering analysis of API reliability by considering a systematically developed multi-layer test system of Contract Testing, Integration Testing, and End-to-End (E2E) Testing. The aim was to determine the input of each layer towards defect detection, performance during execution and general system validation.

The performance and coverage gradient results demonstrate a very clear difference in results: Contract Testing took about 55 seconds, with a very low Defect Detection Rate (DDR) of 10%, mostly limited to interface-level schema and response structure validation. The Integration Testing took 155 seconds with highest DDR of 80% and thus successfully verified cross-endpoint behavior and data consistency. The longest was the E2E Testing which took 230 seconds with a DDR of 56.7% and involved the execution of entire user workflows and business rules in a realistic setting.

Through a systematic injection of thirty controlled defects (D1-D30), the study has shown that contract testing is useful to a small number of schema/contract violations, but integration testing is the workhorse to identify most of the structural and behavioral defects during the booking lifecycle. E2E tests also provide a greater level of confidence, as they verify the entire workflow, especially at logic as well as process level defects that can only be detected during multi step interactions, although they do not outperform integration testing in raw detection rate. Notably, there were a small number of undetected minor flaws of logical nature which all three layers failed to detect, highlighting that no method of testing can be undertaken independently.

This study highlights the need to implement a multi-level testing approach that provides a balance between rapidity, precision, and thoroughness. Integration testing is a combination of contract and E2E testing, which allows organizations to attain greater applicability of their systems, identify flaws earlier, and ensure stable performance of their systems over changing API-driven frameworks.

References

Adeleke, Temitope Oluwafunmike Sanyaolu, Christianah Pelumi Efunniyi, Lucy Anthony Akwawa & Chidimma Francisca Azubuko (2024) API integration in FinTech: Challenges and best practices. *Finance & Accounting Research Journal*, 6(8), 1531–1554. Available online: https://doi.org/10.51594/farj.v6i8.1506.

Arcuri, A. (2019) RESTful API Automated Test Case Generation. *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017*, 9–20. Available online: https://doi.org/10.1109/QRS.2017.11.

Atlidakis, V., Godefroid, P. & Polishchuk, M. (2020) Checking Security Properties of Cloud Service REST APIs. *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation, ICST 2020*, 387–397. Available online: https://doi.org/10.1109/ICST46399.2020.00046.

Bai, X., Li, M., Chen, B., Tsai, W.-T. & Gao, J. (2011) Cloud testing tools. In *Proceedings of 2011 IEEE 6th International Symposium on Service Oriented System (SOSE)*. IEEE, 1–12. Available online: https://doi.org/10.1109/SOSE.2011.6139087.

Barboni, M., Morichetta, A. & Polini, A. (2022) Smart contract testing. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, 21–24. Available online: https://doi.org/10.1145/3528226.3528370.

Ehsan, A., Abuhaliqa, M.A.M.E., Catal, C. & Mishra, D. (2022) RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences*, 12(9), 4369. Available online: https://doi.org/10.3390/app12094369.

Eisty, Nasir U, Upulee Kanewala, ·, Carver, Jeffrey C, Eisty, N U, Kanewala, U. & Carver, J C (2025) Testing Research Software: An In-Depth Survey of Practices, Methods, and Tools. https://arxiv.org/pdf/2501.17739v1 [Accessed 14 Oct 2025].

Felício, D., Simão, J. & Datia, N. (2023) RapiTest: Continuous Black-Box Testing of RESTful Web APIs. *Procedia Computer Science*, 219, 537–545. Available online: https://doi.org/10.1016/j.procs.2023.01.322.

Gao, J., Bai, X. & Tsai, W. (2011) Cloud Testing- Issues, Challenges, Needs and Practice.

Golmohammadi, A., Zhang, M. & Arcuri, A. (2024) Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology*, 33(1), 1–41. Available online: https://doi.org/10.1145/3617175.

Gu, J.T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O. & Xu, T. (2023) Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, 96–112. Available online: https://doi.org/10.1145/3600006.3613161.

Hong, X.J., Sik Yang, H. & Kim, Y.H. (2018) Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. *9th International Conference on Information and Communication Technology Convergence: ICT Convergence Powered by Smart Intelligence, ICTC 2018*, 257–259. Available online: https://doi.org/10.1109/ICTC.2018.8539409.

Hu, Z., Zhu, L., Ardi, C., Katz-Bassett, E., Madhyastha, H. V., Heidemann, J. & Yu, M. (2014) The need for end-to-end evaluation of cloud availability. *Springer*, 8362 LNCS, 119–130. Available online: https://doi.org/10.1007/978-3-319-04918-2_12.

Indrasiri, Kasun. & Suhothayan, Sriskandarajah. (2021) Design Patterns for Cloud Native Applications . https://dokumen.pub/design-patterns-for-cloud-native-applications-patterns-in-practice-using-apis-data-events-and-streams-1nbsped-1492090719-9781492090717.html [Accessed 14 Oct 2025].

Iyer, G.N. (2016) Cloud Testing. In *Encyclopedia of Cloud Computing*. Wiley, 327–337. Available online: https://doi.org/10.1002/9781118821930.ch27.

Izzat, S. & Nada, N.S. (2023) Software Testing Techniques and Tools: A Review. *Journal of Education and Science*, 32(2), 30–44. Available online: https://doi.org/10.33899/edusj.2023.137480.1305.

Jani, Y. (2023) Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development. *International Journal of Science and Research (IJSR)*, 12(6), 2984–2987. Available online: https://doi.org/10.21275/SR24716120535.

Kim, M., Corradini, D., Sinha, S., Orso, A., Pasqua, M., Tzoref-Brill, R. & Ceccato, M. (2023) Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd*

ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 1232–1243. Available online: https://doi.org/10.1145/3597926.3598131.

Master Thesis of Lasse Kajavalta & Tempere Univeristy (2022) *REST API SECURITY: TESTING AND ANALYSIS*.

Leotta, M., Clerissi, D., Ricca, F. & Tonella, P. (2016) Approaches and Tools for Automated End-to-End Web Testing. In, 193–237. Available online: https://doi.org/10.1016/bs.adcom.2015.11.007.

Lomotey, R.K. & Deters, R. (2013) Reliable Consumption of Web Services in a Mobile-Cloud Ecosystem Using REST. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. IEEE, 13–24. Available online: https://doi.org/10.1109/SOSE.2013.10.

Lu, Q., Zhu, L., Bass, L., Xu, X., Li, Z. & Wada, H. (2013) Cloud API issues. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. ACM, 23–32. Available online: https://doi.org/10.1145/2465478.2465481.

Mohammed, B. & Kiran, M. (2015) Analysis of Cloud Test Beds Using OpenSource Solutions. *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, 195–203. Available online: https://doi.org/10.1109/FICLOUD.2015.106.

Nguyen, H.T. (2021) *Microservices, RESTful API and a use case - Theseus*. https://www.theseus.fi/handle/10024/500286 [Accessed 14 Oct 2025].

Nurhayati, E. & Agussalim, A. (2023) Rancang Bangun Back-end API pada Aplikasi Mobile AyamHub Menggunakan Framework Node JS Express. *Jurnal Sistem Dan Teknologi Informasi (JustIN)*, 11(3), 524. Available online: https://doi.org/10.26418/justin.v11i3.66823.

Panahande, M. & Miller, J. (2023) A Systematic Review on Microservice Testing. Available online: https://doi.org/10.21203/RS.3.RS-3158138/V1.

Petrillo, F., Merle, P., Moha, N. & Guéhéneuc, Y.-G. (2016) Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study. In, 157–170. Available online: https://doi.org/10.1007/978-3-319-46295-0_10.

Pogudin, V. (2023) End-to-end test implementation for a SaaS platform. https://www.theseus.fi/bitstream/handle/10024/797211/Pogudin_Vladislav.pdf?sequence=2 [Accessed 25 Sep 2025].

Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H. & Cai, Y. (2021) Empirical evaluation of smart contract testing: what is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 566–579. Available online: https://doi.org/10.1145/3460319.3464837.

Sangannagari, S.R. (2024) Design and Implementation of a Cloud-Native Automated Certification Platform for Functional Testing and Compliance Validation. *International Journal of Technology, Management and Humanities*, 10(02), 34–43. Available online: https://doi.org/10.21590/K5R28W48.

Shahin, M., Ali Babar, M. & Zhu, L. (2017) Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, 3909–3943. Available online: https://doi.org/10.1109/ACCESS.2017.2685629.

Simosa, M. & Siqueira, F. (2023) Contract Testing in Microservices-Based Systems: A Survey. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 312–317. Available online: https://doi.org/10.1109/ICSESS58500.2023.10293058.

Sotiriadis, S., Lehmets, A., Petrakis, E.G.M. & Bessis, N. (2017) Unit and Integration Testing of Modular Cloud Services. *International Conference on Advanced Information Networking and Applications*, 1116–1123. Available online: https://doi.org/10.1109/AINA.2017.57.

Suthendra, J.A. & Pakereng, M.A.I. (2020) Implementation of Microservices Architecture on E-Commerce Web Service. *ComTech: Computer, Mathematics and Engineering Applications*, 11(2), 89–95. Available online: https://doi.org/10.21512/comtech.v11i2.6453.

Trad, T. (2023) Integration testing for enterprise web applications. https://webthesis.biblio.polito.it/28677/ [Accessed 14 Oct 2025].

Tsai, W. T., Bai, X., Paul, R., Shao, W. & Agarwal, V. (2001) End-to-end integration testing design. *Proceedings - IEEE Computer Society's International Computer Software and*

Applications Conference, 166–171. Available online: https://doi.org/10.1109/CMPSAC.2001.960613.

Tsai, W.T., Xiaoying Bai, Paul, R., Weiguang Shao & Agarwal, V. (2001) End-to-end integration testing design. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE, 166–171. Available online: https://doi.org/10.1109/CMPSAC.2001.960613.

Ullah, F., Raft, A.J., Shahin, M., Zahedi, M. & Babar, M.A. (2017) Security Support in Continuous Deployment Pipeline. http://arxiv.org/abs/1703.04277 [Accessed 14 Oct 2025].

Venkatraj, S., Vincent, R., Vijayakumar, V., Vengatesan, K. & Rajesh, M. (2019) Development of Test Automation Framework for REST API Testing. *Journal of Computational and Theoretical Nanoscience*, 16(2), 453–457. Available online: https://doi.org/10.1166/jctn.2019.7749.

Viggiato, M., Terra, R., Rocha, H., Valente, M.T. & Figueiredo, E. (2018) Microservices in Practice: A Survey Study. http://arxiv.org/abs/1808.04836 [Accessed 14 Oct 2025].

Wolde, B.G. & Boltana, A.S. (2021) REST API Composition for Effective Testing the Cloud. *Journal of Applied Research and Technology*, 19(6), 676–693. Available online: https://doi.org/10.22201/icat.24486736e.2021.19.6.924.

Yatini, I., Nurwiyati, F.W. & Anam, K. (2023) PERFORMA MICROFRAMEWORK PHP PADA REST API MENGGUNAKAN METODE LOAD TESTING. *Jurnal Informatika Komputer, Bisnis Dan Manajemen*, 19(2), 12–20. Available online: https://doi.org/10.61805/fahma.v19i2.55.

Zendeli, B., Luma, A., Selimi, B. & Rexha, B. (2025) Identifying Limitations in End-to-End Testing: A Systematic Review of Coverage Analysis Techniques. *ICHORA 2025 - 2025 7th International Congress on Human-Computer Interaction, Optimization and Robotic Applications, Proceedings* [Preprint]. Available online: https://doi.org/10.1109/ICHORA65333.2025.11016980.