Automated Testing Framework for Microservice Architecture

Berkay Dinç
Computer Engineering Department
Dokuz Eylül University
İzmir, Türkiye
berkay.dinc@ogr.deu.edu.tr

Güney Söğüt
Computer Engineering Department
Dokuz Eylül University
İzmir, Türkiye
guney.sogut@ogr.deu.edu.tr

Yusuf Gassaloğlu
Computer Engineering Department
Dokuz Eylül University
İzmir, Türkiye
yusuf.gassaloglu@ogr.deu.edu.tr

***Abstract*—Microservice architectures offer modularity, scalability, and independent service deployment, but their distributed nature introduces unique testing challenges. This study presents an automated testing framework tailored to microservices, leveraging Kubernetes for orchestration and integrating tools such as Jenkins for continuous integration, Harbor for secure container image management, and Locust for performance testing. The framework incorporates a multi-level testing strategy that includes unit, integration, and performance tests to ensure service reliability under realistic conditions. Critical design considerations, such as addressing pipeline limitations with Kaniko and implementing a gated workflow, promote efficiency and security throughout the testing process. A representative microservice application built with Flask was used to validate the framework's capabilities, demonstrating its practical applicability. While the framework provides a robust foundation, future work could explore enhancements such as chaos engineering, cross-service dependency analysis, and support for hybrid-cloud environments to address evolving demands in microservice ecosystems.**

I. Introduction

Microservice architecture has emerged as a transformative approach in modern software development, offering enhanced modularity, scalability, and independent deployment of individual services. By breaking down monolithic systems into smaller, independently deployable units, organizations can achieve faster development cycles, better fault isolation, and streamlined scalability. However, these advantages come with inherent complexities, including managing distributed systems, ensuring seamless inter-service communication, and maintaining system reliability. The testing of microservices, therefore, requires a comprehensive and automated framework capable of addressing the unique challenges posed by their distributed nature.

In this study, we propose an automated testing framework designed specifically for microservice architectures, leveraging Kubernetes for cluster management and orchestration. Kubernetes provides a robust platform for deploying, scaling, and managing containerized applications, forming the backbone of our framework. Jenkins, a widely used automation server, is employed to create and manage a continuous integration and testing pipeline. To store and manage container images efficiently, we incorporated Harbor, a secure and scalable container image registry. However, during implementation, we encountered a critical challenge: Jenkins, running as a Kubernetes-managed container, was unable to access commands included in the local PATH, which disrupted the pipeline's workflow. To address this limitation, we integrated Kaniko, a tool specifically designed for building container images within containerized environments. This solution eliminated the dependency on a Docker daemon and allowed the pipeline to function seamlessly in the Kubernetes environment.

To validate the proposed framework, we developed a representative microservice-based application using Python and Flask. This application provided a testbed to evaluate the framework's capability to execute diverse testing methodologies. The testing process included unit tests to verify the correctness of individual components, integration tests using the Pytest framework to ensure seamless interactions between microservices, and performance tests using Locust to evaluate the system's ability to handle concurrent requests under varying loads. These tests were automated within the Jenkins pipeline, ensuring a streamlined and repeatable process for quality assurance.

Our results demonstrate the efficacy of the proposed framework in addressing the challenges associated with microservice testing. By automating the testing process and leveraging Kubernetes' capabilities, the framework provides a scalable and reliable solution for validating the functionality, integration, and performance of microservices. This study highlights the importance of combining container orchestration tools, robust testing strategies, and automation pipelines to meet the demands of modern microservice architectures.

II. Related Works

Sundaram [1] discusses the trends, techniques, and challenges in software testing, highlighting its evolution alongside advancements in technology. The paper explores testing methodologies across various domains, including traditional models like the Waterfall approach, and modern trends such as Agile, Automation, and DevOps. Testing phases and types, such as black-box, white-box, and grey-box testing, are visualized in figures. Emerging technologies like AI, IoT, Big Data, and Cloud Computing are reviewed for their impact on testing, while challenges like test automation complexity, regression testing in Agile, and test oracle problems in AI are underscored. Specific testing methods for mobile applications, Big Data, and IoT are also detailed. The study emphasizes the critical role of software testing in ensuring quality and suggests that generalized models could address fragmented platform challenges.

Rodrigues [2] conducted a study evaluating the relevance and impact of 12 critical factors of success (CFS) in the basic software test automation lifecycle (BSTAL). The research aimed to bridge the gap between academic and practitioner perspectives on software test automation. They surveyed 33 practitioners, representing an 84% confidence level. The BSTAL consists of four phases: Selection, Modeling, Execution, and Analysis, each impacted differently by CFSs. Results showed that Feasibility Assessment (83.59%) and Automation Planning (80.92%) had the highest impact on the Selection phase, while Dedicated and Skilled Team (80.01%) was most relevant in Execution. Their findings suggest prioritizing CFSs specific to each lifecycle phase to enhance automation success

Giamattei et al. [3] introduced the MacroHive framework, which automates functional and robustness testing for microservice architectures (MSA). This framework supports five phases: API specification collection, test specification definition, test suite generation, test execution, and test reporting. It employs gray-box monitoring to observe both edge and internal microservices, a unique feature among state-of-the-art tools like EvoMaster, RestTestGen, RESTler, and bBOXRT. Experimental results on the TrainTicket benchmark system (41 microservices) demonstrated that MacroHive achieved 80% internal microservices coverage for deeply nested invocation chains, with 14.9% failure detection rates—significantly outperforming RESTler and bBOXRT in robustness testing. Additionally, MacroHive's causal inference engine identified failure propagation and masked failures, offering a novel approach to pinpoint critical failures within MSA

Sharmila et al. [4] explore AI/ML-driven automated testing methodologies for microservices to address the inherent complexities and limitations of traditional approaches. They emphasize key AI techniques, such as neural networks for test case generation, natural language processing for deriving test cases from requirements, and reinforcement learning to optimize testing strategies. The study highlights the scalability of these methods, particularly through distributed testing frameworks and AI-enhanced performance analysis. Case studies show significant improvements, such as a 30% reduction in test execution time and a 25% increase in defect detection rates, illustrating the real-world efficacy of these advanced testing frameworks.

Camargo et al. [5] present an architecture designed to automate performance tests on microservices. Their approach embeds a test specification directly into each microservice, allowing automated test execution without human intervention. They utilize the HTTP `OPTIONS` method to expose the test specification, ensuring seamless integration with external testing applications. The evaluation, conducted on a financial application deployed on Amazon EC2, demonstrated no performance impact from the framework. Key metrics included a response time mean of 118 ms with the framework (WF) and 120.5 ms without the framework (NF). Throughput results averaged 50.35 requests per second (WF) compared to 48.91 (NF), affirming the architecture's efficiency and practicality for real-world environments.

Söylemez et al. [6] conducted a systematic literature review (SLR) on microservice architecture (MSA), analyzing 3842 papers from 2014 to 2022 and selecting 85 primary studies. They identified nine major challenge categories with 40 subcategories, including testing, performance prediction, and service orchestration. Notable findings include 18 studies addressing monitoring, tracing, and logging, and 33 focusing on service orchestration. Proposed solutions ranged from an automated regression testing framework to novel orchestration strategies. They found that 82.3% of studies demonstrated good reporting quality, with 63.5% being directly relevant, offering valuable insights into advancing MSA adoption.

Venkat et al. [7] examine automated testing strategies tailored for microservices within a DevOps framework, emphasizing unit, integration, and end-to-end testing. They explore the integration of tools like Docker, Kubernetes, and Jenkins to enhance CI/CD pipelines. The paper highlights the importance of achieving high test coverage and introduces best practices, such as test-driven development (TDD) and behavior-driven development (BDD), to improve testing efficiency. Case studies demonstrate a reduction in deployment time by 30% and a 25% improvement in defect detection rates, underscoring the practical benefits of these methodologies in improving the scalability and reliability of microservices architectures.

III. Design and Implementation

The design (Fig. 1) of our "Automated Testing Framework for Microservice Architecture" centers on ensuring seamless coordination among multiple services, robust testing at every development stage, and secure, performant deployment. We developed a Flask server to act as the primary target for testing, providing a simple yet representative microservice environment. This server encapsulates standard HTTP endpoints and data-processing routes that closely resemble real-world microservice patterns. By hosting the server in a Kubernetes cluster, we simulate production-like conditions, enabling more realistic evaluations of our testing pipeline. This setup also benefits from Tailscale, an operator that provides secure, private networking capabilities, thereby ensuring that only authorized services and users can interact with the testing environment.

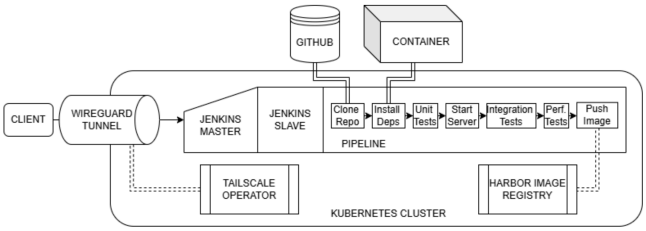

Fig. 1: The pipeline design

A key component of this design is our multi-level testing strategy, which includes unit tests, integration tests, and performance tests. Unit tests, written in Python using Pytest, allow us to isolate individual microservice components and validate their functionality. This approach enables rapid iteration by identifying flaws early and reducing the complexity of debugging later stages. Once individual microservices pass unit testing, they progress to integration tests, also facilitated by Pytest. Integration testing confirms that the various microservices and external dependencies communicate correctly, preserving the contract-driven nature of a microservice architecture. Finally, performance testing with Locust helps us assess the resiliency and throughput of the system under simulated load conditions. By quantifying response times and resource usage, Locust aids in detecting performance bottlenecks and ensuring that each service meets its scalability requirements.

To execute these tests continuously and consistently, we employed Jenkins as our primary automation server. Jenkins orchestrates the entire process through a pipeline that comprises seven stages: cloning the repository from GitHub, installing dependencies, running unit tests, starting the Flask server, executing integration tests, performing Locust-based performance tests, and finally pushing the Docker images to the Harbor registry. This stage-gated pipeline ensures that only code passing each level of verification advances to the next, thereby mitigating the risk of introducing regressions into production. By automating these steps, Jenkins reduces manual effort and promotes a DevOps culture of rapid feedback, which is crucial for maintaining high-quality software in a dynamic microservice ecosystem.

Once tests have validated the microservices' functionality and performance, container images are built and pushed to Harbor, an on-premises container registry hosted within the Kubernetes cluster. Harbor offers enhanced security features such as vulnerability scanning and role-based access controls, ensuring that only authorized users can pull or deploy container images. Coupled with Tailscale, this setup provides both secure storage of images and encrypted connections between the development team and the cluster. This arrangement bolsters our zero-trust security posture by restricting unauthorized access to critical infrastructure.

The chosen toolset—Flask, Pytest, Locust, Jenkins, Harbor, and Tailscale—brings multiple benefits to a microservice testing environment. Flask offers a light and flexible framework for creating services that mirror real-world use cases, while Pytest simplifies the process of writing and maintaining test suites. Locust facilitates proactive identification of performance issues before they escalate in production. Jenkins provides a proven, extensible platform for continuous integration and continuous delivery (CI/CD), ensuring that each test stage executes reliably and in order. Meanwhile, Tailscale secures network communication among distributed components, and Harbor serves as an enterprise-grade container registry with advanced security checks. Overall, integrating these tools within a Kubernetes environment forms a robust pipeline for delivering software updates rapidly and safely while maintaining the performance and reliability requirements that define a successful microservice architecture.

IV. Future Work

The current framework lays the groundwork for automated testing in microservice environments, but further refinements could improve its scope and depth. Incorporating test prioritization techniques could optimize resource usage by focusing on the most critical or frequently failing components. Future work could also involve integrating chaos engineering principles to simulate real-world failures and assess system resilience under unexpected conditions. Another promising direction is the inclusion of cross-service dependency analysis to understand the impact of changes in one service on the overall system. Moreover, extending the framework to support multi-cloud or hybrid-cloud environments could address the growing demand for distributed deployments. These developments would enhance the framework's relevance and effectiveness in managing the complexities of large-scale, dynamic microservice ecosystems.

V. Conclusion

This paper outlines the development of an automated testing framework designed for microservice architectures. By leveraging Kubernetes for orchestration and tools such as Jenkins, Locust, and Harbor, the framework integrates multiple testing stages, including unit, integration, and performance tests, into a streamlined pipeline. The design supports secure, scalable, and efficient testing processes that align with the requirements of distributed systems. Although the framework addresses many practical challenges, opportunities for improvement remain. Future enhancements, such as integrating test prioritization, chaos engineering, and cross-service dependency analysis, could extend its functionality and adaptability. These refinements would allow the framework to remain relevant and effective in evolving software environments.

References

[1] A. Sundaram, "TECHNOLOGY BASED OVERVIEW ON SOFTWARE TESTING TRENDS, TECHNIQUES, AND CHALLENGES," vol. 6, pp. 94–98, 2021, doi: 10.33564/IJEAST.2021.v06i01.011.

[2] A. Rodrigues and A. Dias-Neto, "Relevance and Impact of Critical Factors of Success in Software Test Automation lifecycle: A Survey," in *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, in SAST '16. Maringa, Parana, Brazil: Association for Computing Machinery, 2016. doi: 10.1145/2993288.2993302.

[3] L. Giamattei, A. Guerriero, R. Pietrantuono, and S. Russo, "Automated functional and robustness testing of microservice architectures," *Journal of Systems and Software*, vol. 207, p. 111857, 2024, doi: https://doi.org/10.1016/j.jss.2023.111857.

[4] S. R. Sudharsanam, P. Sivathapandi, and D. Venkatachalam, "Enhancing Reliability and Scalability of Microservices through AI/ML-Driven Automated Testing Methodologies," *Journal of Artificial Intelligence Research and Applications*, vol. 3, no. 1, pp. 480–514, Jan. 2023, [Online]. Available: https://aimlstudies.co.uk/index.php/jaira/article/view/195

[5] A. de Camargo, I. Salvadori, R. d. S. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services*, in iiWAS '16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 422–429. doi: 10.1145/3011141.3011179.

[6] M. Söylemez, B. Tekinerdogan, and A. Kolukısa, "Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review," *Applied Sciences*, vol. 12, p. 5507, 2022, doi: 10.3390/app12115507.

[7] V. R. R. Alluri, P. Katari, S. Thota, S. Ganesh Reddy, and A. K. P. Venkata, "Automated Testing Strategies for Microservices: A DevOps Approach," *Distributed Learning and Broad Applications in Scientific Research*, vol. 4, pp. 101–121, May 2018, [Online]. Available: https://dlabi.org/index.php/journal/article/view/92