# Testability and Testing of Microservices - complex challenge

DANI ALMOG
Department of Software Engineering
SCE - Shamoon College of Engineering
Bialik/Basel Streets, Be'er Sheva 84100
ISRAEL
Almog.dani@gmail.com

HADAS CHASSIDIM
Department of Software Engineering
SCE - Shamoon College of Engineering
Bialik/Basel Streets, Be'er Sheva 84100
ISRAEL
hadasch@sce.ac.il

SHLOMO MARK
Department of Software Engineering
SCE - Shamoon College of Engineering
84 Jabotinsky St. Ashdod, 77245 Israel
ISRAEL
marks@sce.ac.il

*Abstract:* - Microservices has become one of the most popular software engineering approach for modern web and IoT applications. Nevertheless, the testability and the testing processes require a comprehensive study that expands the testability model to a wider approach, focusing on the "not-testable" perspective. This paper presents the issues areas and proposes a conceptual framework for testability and testing of applications created with Microservices architecture - a multi-dimension testing approach for improving the process and the outcome. As the transfer from monolithic to Microservices architecture led to a major shift of complexity from the actual basic code into the implementation. This paper explores testing levels and dimensions suggesting a new holistic framework to address the existing challenges.

*Keywords:* - Microservices, testing services, Multi dimension testing. IoT

## 1 Introduction

It's been a long road software development has been taken since the beginning with the first computers, from procedural thinking via object orientation and many different methodologies and buildup. Software architecture was always the signal representation of the actual building blocks that enable the fulfillment of the transformation from an abstract idea to a functioning machine drives instructed and manifested from a symbolic set of commands and interactions (software). From one chunk dependent code and data structure, evolved during the time a more sophisticated structure and build of this set (as specific software) [1]. Recently, we tend to increase imitating the actual life behavior in the pattern of executing this code. Service-Oriented Architecture (SOA) presenting this behavioral pattern so each element in the code will perform (independently) a service to support the full system.

The term "microservices" was first introduced in 2011 at an architectural workshop as a way to describe the participant's common ideas in software architecture patterns. Microservices are small autonomous services that work together, modeled around a business domain [2]. "The foundation of microservices architecture (MSA) is about developing a single application as a suite of small and independent services that are running in its own process, developed and deployed independently".

In most of the definitions of microservices architecture, it is explained as the process of segregating the services available in the monolith into a set of independent services. However, It is more than just about splitting the services within the monolith application into independent services. SA is based on a "share-nothing philosophy" [3]. This architectural style structures a system as a set of loosely coupled small services which are isolated in small coherent and autonomous units.

The key idea is that by looking at the functionalities offered from the monolith, we can identify the required business capabilities. Then those business capabilities can be implemented as fully independent, fine-grained, and self-contained (micro)services. They might be implemented on top of different technology stacks and each service is addressing a very specific and limited business scope.

In their research Di Francesco et al [3], review the most recent research trends regarding microservices applications and their industrial potential. It is clear that very little research was done on testing (10/70) suggesting, it will become a significant field for further investigations. This work proposes a conceptual framework with clear practical implications.

One of the main challenges facing quality assurance and the longtime resilience of applications based on Microservice is the testability issue. Chapter 2 follows the theoretical background and description of Microservices architecture (MSA), proposing a new way to look at all testing activities on the microservice-based application. Chapter 3 elaborates and provides another innovative view for testability.

## 2 Microservices Architecture

Microservice is an SOA interpretation using a set of specific principles and patterns that claim to be in harmony with agility and introduce the new name (Microservice) to break the misunderstandings and incomprehension of SOA. If SOA is a “victim” of its success and that microservice will “blow” its new life, we confirm that it's the opposite, and you must go back to SOA principles to govern your microservices. SOA will give the company the missing enterprise scaling dimension which is explicitly absent in microservice architecture. As an extension of the SOA approach to developing an application as a set of small independent services. Each of the services is running in its own independent process [4], and development on Micro-services Architecture may own a set of drawbacks. In practice, the micro-services approach means for the developers the additional complexity of creating a distributed system [5]. Testing is more difficult for distributed systems. The inter-service communication mechanism is probably one of the main challenges that should consider, including the specific form of the required distributed transactions. Multiple services will require strong coordination among developers within the team or between the teams of developers. Consequently, the deployment complexity will increase as well as the management of different service types. The micro-services approach leads to increased memory consumption, due to the own address space for each service. Therefore, one of the significant challenges is deciding how to split (partition) the system into micro-services. One obvious approach is to partition services by use cases.

Based on Newman [6] and Fowler [2], microservices are about functional decomposition often in a domain-driven design context. They are characterized by well-defined and explicitly published interfaces. Each service is fully autonomous and full-stack. Consequently, changing a service implementation should have no impact on other services, as communication takes place using interfaces only. Functional decomposition of an application and the team is the key to building a successful microservices architecture. This achieves loose coupling (probably by REST interfaces) and high cohesion (multiple services can compose with each other to define higher-level services or applications). Functional decomposition enables for instance agility, flexibility, scalability. These could be achieved by the microservices ecosystems. The overall system containing the microservices and infrastructure can be divided into four layers [6]:

**Layer 1:** Hardware servers, databases, OS, resource isolation, resource abstraction, configuration management, host-level monitoring, host-level logging, etc.

**Layer 2:** Communication (network, Dynamic Service Registries (DNS), Remote Procedure Calls (RPCs), endpoints, messaging, service discovery, service registry, load balancing, etc.)

**Layer 3:** App Platform self-service (dev tools, dev environment(s), test, build, pkg, release, deployment pipeline, app-level logging, app-level monitoring, etc.)

**Layer 4:** Microservices and microservices-specific configs

According to Savchenko et al [7] the features of microservices are:

**Open Interface** - microservices should provide an open description of interface and communication messages format (either API or GUI).

**Specialization** - each microservices provides support for an independent part of the application's business logic.
**Containerization** - isolation from the execution environment and other microservices based on a container virtualization approach. Technologies like OpenVZ, Docker, or Rocket became a de-facto standard for the implementation of such an approach.
**Autonomy** - microservices can be developed, tested, deployed, destroyed, moved, or duplicated independently and automatically. Continuous integration is the only option to deal with such development and deployment complexity. We may add to that - Internal containment of logics – all the logics of the service are contained internally so the output of the service will be very uniform and standard.
In microservice architectures, applications are built and deployed as simple, highly decoupled, focused services. They connect over lightweight language-agnostic communication mechanisms, which often means simple APIs and message queues [2]. Services implement a self-contained, well-defined, and documented set of functionalities, which they expose only via versioned APIs [8]. Implementing microservices, are polyglot in terms of programming languages, frameworks, and data stores used. Lastly, microservices are resilient, which means they are immutable artifacts that are designed to fail and to be elastic in scale [9] [10].

## Microservices and IoT

It seems that a vision of applying microservices architecture in IoT systems is becoming widely spread. wherein, one can expect that microservices can be additionally associated with a device [19]. Recent work [21] testify that most of the testing used by practitioner's unit and end-to-end testing and other levels and aspects of testing do not have dedicated solutions, which poses challenges during the testing of microservices. Are we facing the familiar challenge that was reported in our previous work? [22] in which, the tendency of the industry to lean more on unit code base testing? – we believe this merits additional research.

# 3 Testability and Testing of Microservices Applications

Software testability is one of the important concepts in design and testing of software programs and components. Building programs and components with good testability always simplifies test operations, reduces test cost, and increases software quality. One way to improve the maintainability of a software system is the design for testability, which can address various aspects of software including size, complexity, system structure, built-in-test facilities, distribution, and non-determinism.
Design for testability is a strategy to align the development process so that testing is maximally effective under either a reliability-driven or resource-limited regime. There are several sets of program characteristics that lead to testable software including operability, observability, controllability, and understandability. Software testability analysis may be useful to examine and estimate the quality of software testing using an empirical analysis approach. Testability is important for both ad-hoc developers and organizations with a high level of process maturity. It reduces cost in a reliability-driven process and increases reliability in resource-limited processes. It refers to ‘the inherent ability, or extent of the ease with which software undergoes through testing’.
Testability has been defined by ISO as one of the attributes of software that bare on the effort needed to validate the software product. ISO25010 – 2011: "degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met".
Several characteristics have been identified in the literature that contributes to a software system’s testability. Later, [11] describe the term practical testability of a product as: "how easy it is to test " saying, this is a function of five other “testabilities”: project-related testability, value-related testability, subjective testability, intrinsic testability, and epistemic testability (also known as the “risk gap”). Earlier Binder [12], describes testability using a fish-bone diagram, showing a range of typical factors that are expected to facilitate testability. Later in his book (page 93) Binder provides a more specific fish-bone model regarding the capabilities of built-in tests. We find this binder fish-bone model too general and not detailed enough. Following Binder González et al. [13] suggested looking at the runtime testability with a new fish-bone diagram that describes qualitative factors that affect runtime testability. In their model, they separate the components into four different considerations that manifest oppositely – sensibility - isolation: State fullness - State separation, Interaction – interaction separation, Resource limits – resource monitoring, and Availability – scheduling.
Another attempt to reduce the challenges of the interactions among services was proposed by Chen [14] using four practices: (1) using test-first mind-set and practices. (2) consumer contract-driven testing

(3) Online system Integration test environment as part of the CD pipeline (4) test in production and monitoring. Waseem [15] aimed to deepen understanding at of how microservices are developed and tested in industry. They found that unit and end-to-end testing defined as the most used testing strategies. However, the complexity of microservices systems poses challenges for their design, monitoring, and testing, for which there are no dedicated solutions

Recent works introduced software tools that help to deliver high-quality microservices. For example, Schrieber [16] proposed an approach to deploying and composing containerized microservices as reviewable applications. Another study compared several tools that can be used to test microservices at different levels including end-to-end testing and regression testing [17]. The main challenge was that the tool configuration files need to be written manually and match the environments in the containers of the services. Other challenges include the resources needs to run both the tools and the prototype (e.g., RAM; unreliable network connectivity). Hernández [18] evaluated two tools that support end-to-end (E2E) testing of microservices.

To fit the micro-service architecture, we suggest expanding the testability model to a wider approach, focusing on the "not-testable" perspective. For the testing layers of microservices architecture we have chosen to present the concept negatively – "Non-Testability" fish-bone diagram, pointing out various aspects and characteristics which may lead to exposing the difficulties in relate to the ability to test an artifact.

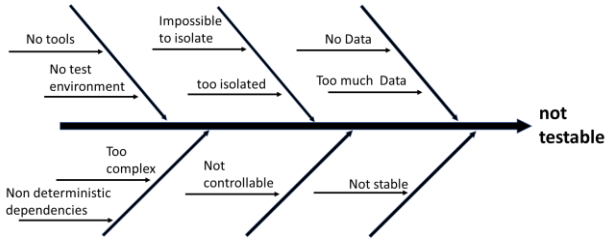


Figure 1 "Non – testability" fish-bone model

Many of these risks expose alternatively each in two opposite extremes for a single issue:

- The technical ability to test (tools and environment)
- Enable isolation –but retain the accessibility
- Data-driven facilitation
- Complexity
- Controllability
- A stable situation

We believe that in order to deliver quality software within time and budget, reducing effort in measuring the testability of Microservice and IoT is necessary. Later, we'll try to address these aspects and produce a valid testing model for covering and assuring each of these limitations and risks by presenting a multi-dimension model concept for testing microservices-related applications. Facing the transition from a Monolithic application into Microservices may raise serious issues of complexity and control. Figure 2 presents the transformation from monolithic to Microservices architecture.

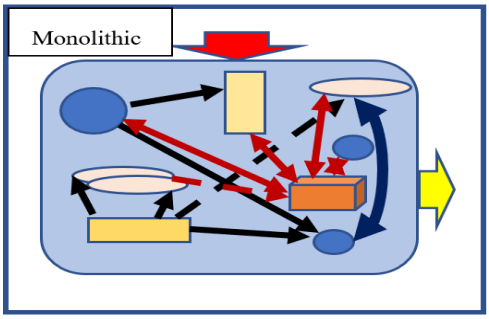


Figure 2 Internal complexity

Figure 2 suggests that all internal interaction between the artifacts is maintained, so the complexity is contained within the monolithic application. The new architecture on another hand enforces additional external interaction which creates

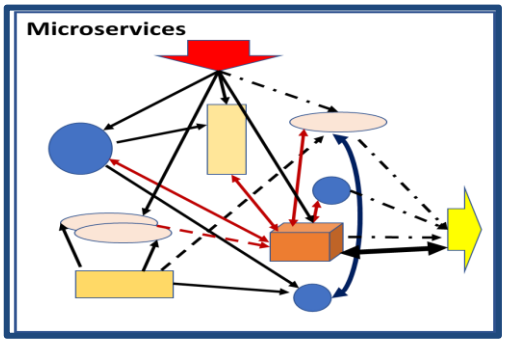


Figure 3 external chaos

enormous complexity (Figure 3). Also, each service is easier to maintain/replace the full picture is of much more chaotic behaviour. There is a shift of complexity from the space of code design and implementation into system operations [19].

While describing testing of MJOLNIRR-Based Microservices [7], Savchenko considers the most

significant stages of the validation process, addressing a full software package and particular microservices systems, defining which features of the validation items to be considered for microservice systems. Our work provides a wider view of microservices testing and attempts to cover more general aspects and validation activities and as a result may provide better quality results.

# 4 Multi dimension (plane) Testing Levels Approach for Microservices

In this study, we offer a holistic perspective that enables a conceptual framework for the testing activities requirements that meets the main challenges mentioned above.
Justification for differentiating the testing level dimensions may be demonstrating by address the following: Who are the right candidates to perform these activities? What are their needed skills and qualifications? When this activity should be exercised? Are there dependencies or constraints attached to the implementation of the microservices? Examining the full context of testing, produce new challenges, the following is a new attempt to address the testing of microservice differently; multi-dimension levels for testing starting the traditional well versed testing levels.

### 4.1 Technical/code plane testing levels

Following the traditional testing levels [20], technical levels represent the chronological buildup of the software starting with the testing of the basic unit of code, assemble it into components, orchestrate them into product framework and integrate it into a system that has other products to integrate with. The affiliation to the code is a major aspect of these testing level dimensions. Having Microservices architecture will probably place unit tests as an internal service activity. The integration testing may also appear internally within the service (when a complex service was developed). Other technical levels will manifest as external to the actual service itself.

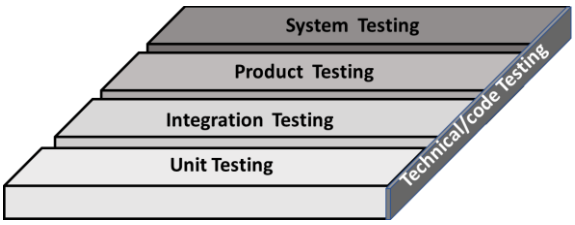


Figure 4 Technical/code testing levels

Technical/code testing is a procedural serial buildup process; you cannot fully test an element before you have fully implemented all its ingredients.

### 4.2 Functional/business plane testing levels

Testing the functional/business context of an element requires knowledge and familiarity with the usage domain. Most of the time, tools and the mechanism do not demand a real insight of the software internals. Testing functionality starts with validating each feature. The second level will be testing the most basic functional artifact, the microservices. The next level is the testing of a complex service (which was granulated from several Microservices). The most advanced level ought to be testing the actual usage of the service in its own domain and natural execution environments.

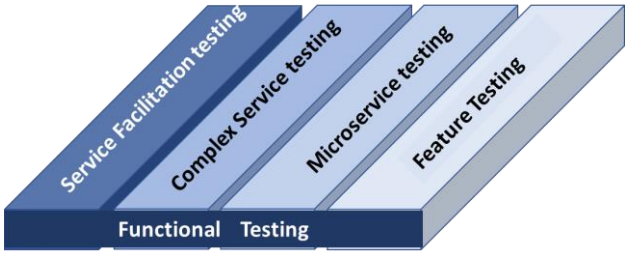


**Figure 4: Functional testing levels**

We tend to treat service in more loosely affiliation, therefore the order of implementation these testing levels are not as strict as the software actual buildup. The Microservices ought to supply isolation and independence so testing a single feature may be performed separated. Next is most challenging – and present all difficulties we find testing SOA [21] [4], where the complexity and flexibility of business service functionally discovered to its fullness. Facilitation testing represents its reusability and independents nature which in many cases may appear as an endless combination of possible situations to be tested – so prioritization and risk assessment is essential.

### 4.3 Nonfunctional plane testing levels

In the nonfunctional plane testing levels, the differentiation between the levels is not so clear. We rather separate it by various aspects and types of non-functional testing. Some of these testing activities could start and be performed on the different development stages, some of them should be evaluated prior to the actual implementation of the system altogether – for example, the Security element must be assured as early as a coding standard.

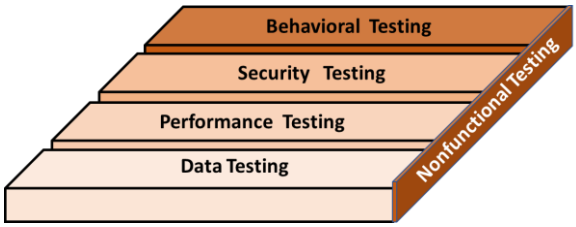


Figure 5 Nonfunctional testing levels

Figure 5 presents a possible list of non-functional tests. The actual nonfunctional testing capacity is much wider than described in Figure 4, it will be varied depending on the needed business requirements. For distributed a system, we have to consider also Network latency, fault tolerance, message serialization, unreliable networks, synchronicity, versioning, varying loads within our application tiers, etc. [22]. New Performance engineering might be demanded [23], saying that existing performance engineering techniques - focusing on testing, monitoring, and modeling - cannot simply be re-used.
The behavioral testing should focus on addressing the chaos creating during the operation of distributed systems. Idiomatic Microservices involves placing less emphasis on testing and more on monitoring so we can spot anomalies in production [22] a possible tool for addressing it, is the ChaosMonki tools [24].

### 4.4 Multi Dimension matrix plane for microservice approach

When examining the full context of testing levels in Microservices, we can also observe that some of these testing activities may be performing simultaneously (figure 6). This figure simplifies and clarifies the testing activities which seem to be separate and probably applied independently, our experience shows this demands a deeper analysis – interdimensional influence should be investigated and research.

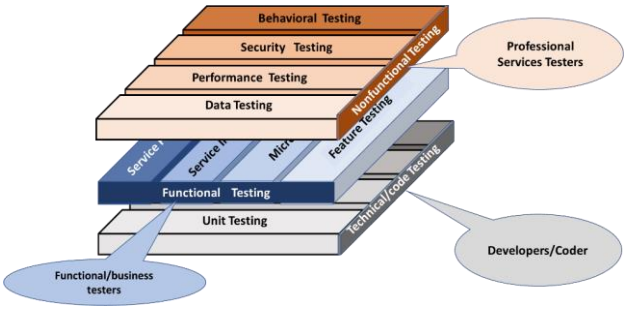


Figure 6 Multi-plane test levels

Multi dimension testing levels expose some essentials regarding who is doing each activity (skill-wise) and the exact timing to exercise it?
The following table (table 1) suggests a possible partition for these activities. Table 1 floods a new research question addressing the needed skills needed for each testing plane.

| Dimension (plane) | Technical /code | Functional /business | Non-functional |
|---|---|---|---|
| **Who** | Developer | Business tester | Professional services |
| **When** | During development | Development and integration | Flexible, musty during integration and production |
| **Outcome** | All code is sufficiently tested | All services are accurately functioning | Better assurance for expected behavior |
| **Tools** | Unit test tools | Domain related test tools | Professional targeted tools – for each type of testing |

Table 1 activities assignment for testing dimensions

The more we'll explore these dimensions we may realize that the affiliated domain may present another dimension. (for example, the agriculture microservices domain may present another implementation and user acceptance plane of testing).

## 5. Testing the Implementation of Microservices with a Chaotic Behavior Approach

Using Chaos Engineering is the discipline of experimenting on a distributed system to build confidence in the system's capability to withstand turbulent conditions in production. Another aspect affecting the testability of application Microservices is that even when all of the individual services in a distributed system are functioning properly, the interactions between those services can cause unpredictable outcomes. Unpredictable outcomes, compounded by rare but disruptive real-world events that affect production environments, make these distributed systems inherently chaotic – complexity is way beyond control therefore it is almost impossible to test [25] [18].

To achieve better stability and confidence with the microservices systems, we propose the following steps:

1. Defining ‘steady state’ as some measurable output of a system that indicates normal behavior. To do so we must hypothesize that this steady-state will continue.
2. Introduce new or substitute microservices that reflect real-world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.
3. Recheck the 'steady state' hypothesis by looking for a difference in a steady state between the different papulations of your testing environments.

The harder it is to disrupt the steady-state; the more confidence we have in the behavior of the system. If a weakness is uncovered, we now have a target for improvement before that behavior manifests in the system at large.

### 5.1 Dynamic behavior of microservices implementation – risk for testing

Another risk factor exposed while implementing microservice-oriented applications is the uncertainty of which microservice to be used since it by nature is selected dynamically according to a very specific service discovery from the microservice repository.

An additional concern is another level of uncertainty and risk at service scalability, modularity, and objects’ reusability for intelligence IoT service provisioning using Web of Objects (WoO) platform [26]

## 6. Summary

This paper presents a new conceptual approach toward testing Microservice-based applications. At the center of this approach is the formation of different dimensions (planes) of testing levels enabling to separate the different aspects needed to ensure the quality of the application. Using this strategy may separate the complexity exposed during the actual implementation of microservices. Further research is needed to test in practice the proposed framework. This future evaluation may compare the matrix multi-dimension testing approaches to alternatives.

## References:

[1] A. Bertolino, G. De Angelis, A. Sabetta and A. Polini, "Trends and research issues in SOA validation. In Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions," *IGI Global*, pp. 98-115, 2012.

[2] M. Fowler, "Microservices.," 2017. [Online]. Available: http://martinfowler.com/articles/microservices.html.

[3] P. Di Francesco, I. Malavolta and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption.," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017.

[4] G. A. Lewis, D. B. Smith and K. Kontogiannis, "A research agenda for service-oriented architecture (SOA): Maintenance and evolution of service-oriented systems.," Carnegie-Mellon University Pittsburgh PA Software Engineering Inst., 2010.

[5] O. Zimmermann, "Microservices tenets. , ," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 301-310, 2017.

[6] S. Newman, Building microservices: designing fine-grained systems, O'Reilly Media, Inc., 2015.

[7] D. I. Savchenko, G. I. Radchenko and O. Taipale, "Microservices validation: Mjolnirr platform case study. In 2015 38th International convention on information and communication technology, electronics and microelectronics (MIPRO) (pp.," 2015.

[8] C. Carneiro and T. Schmelmer, Microservices from day One., Apress. Berkeley, CA., 2016.

[9] P. Abbassi, "What are microservices?," 2017. [Online]. Available: https://www.quora.com/What-are-microservices/answer/Puja-Abbassi..

[10] C. Santana, L. Andrade, B. Mello, E. Batista, J. V. Sampaio and C. & Prazeres, "A reliable architecture based on reactive microservices for IoT applications.," in *In Proceedings of the 25th Brazillian Symposium on Multimedia and the Web*, October, 2019.

[11] J. Bach and M. & Bolton, "Rapid Software Testing Appendices.," Version (1.3. 2), wwvv. satisficc. com, 2007.

[12] R. V. Binder, "Design for testability in object-oriented systems.," *Communications of the ACM*, vol. 37, no. 9, pp. 87-101., 1994.

[13] A. González, E. Piel and H. G. Gross, "A model for the measurement of the runtime testability of component-based systems.," in *2009 International Conference on Software Testing, Verification, and Validation Workshops* , 2009.

[14] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," *IEEE International conference on software architecture (ICSA)*, pp. 39-397, 2018.

[15] M. Waseem, P. Liang, M. Shahin and A. &. M. G. Di Salle, "(2021). Design, monitoring, and testing of microservices systems: The practitioners’ perspective.," *Journal of Systems and Software*, vol. 111061, p. 182, 2021.

[16] M. Schreiber, "Prevant (Preview servant): composing microservices into reviewable and testable applications.," in *In Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, Schloss Dagstuhl-Lei, 2020.

[17] J. P. Sotomayor, S. C. Allala, P. P. J. Alt and T. M. &. C. P. J. King, "Comparison of runtime testing tools for microservices," *In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC) (Vol. 2, pp. 356-361*, vol. 2, pp. 356-361, 2019.

[18] C. M. Hernández, A. Martínez and C. &. J. M. Quesada-López, "Comparison of End-to-End Testing Tools for Microservices: A Case Study," in *International Conference on Information Technology & Systems*, Cham, 2021, February.

[19] I. Nadareishvili, "Microservices shift complexity to where it belongs.," 2016, [Online]. Available: https://www.oreilly.com/ideas/microservices-shift-complexity-to-where-it-belongs.

[20] A. &. L. B. Kramer, Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level, John Wiley & Sons, 2016.

[21] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24-27., 2014.

[22] B. Wootton, 2014. [Online]. Available: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html.

[23] R. Heinrich, A. van Hoorn, H. Knoche and et. al., "Performance engineering for microservices: research challenges and directions," *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion.*, no. Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., ... & Wettinger, J. (2017, April). Performance engineering for microservices: research challenges and directions. In Proceedings of the, pp. 223-226, 2017.

[24] "https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey," 2017. [Online].

[25] C. Rosenthal, "Principles of Chaos Engineering.," 2017. [Online]. Available: https://www.usenix.org/conference/srecon17americas/program/presentation/rosenthal.

[26] M. A. Jarwar, M. G. Kibria and S. &. C. I. Ali, "Microservices in web objects enabled iot environment for enhancing reusability," *Sensors*, vol. 18, no. 2, p. 352., 2018.