Appendix-B (Summarization)

- PR1 adapts a *learning-based* testing (*LBT*) approach to the microservice architecture in order to extract a model of the system under test (SUT). Then, model checking techniques are applied to the generated model and counter examples are produced as fault scenarios. This work is the only investigation which has leveraged machine learning concepts and model checking techniques in microservices testing. However, the generated model in this study is not convergence to the system which means that the low-likelihood behaviors or indeterministic states of the SUT might remain uncovered in the model generation process.

- PR2 provides a tool called Gremlin for the test of microservice resiliency at the network level. This tool includes components for translating the failure scenarios to network layer messages, orchestrating failure according network proxies to inject and emulate tests, and observing data from the proxies using an assertion checker to identify results. The results from Gremlin provide feedback about why and how the system has failed. This feedback allows the developer to test the faulty part with more specified tests rather than random failure injection. This characteristic allows the definition of the failure-recovery measures in the development environment and before occurring failures in production setup. However, the number of supported faults in PR2 is limited to fail-stop/crash failures, performance/omission failures, and crash-recovery failures in the network layer [69]. Moreover, this work does not provide any measuring of the fault coverage or percentage of various detected faults in each failure model.

- PR3 is one of the few investigations which talks directly about non-functional requirements testing as the primary goal of the research. This article creates queries containing the specifications of the system and expected behavior. Then, queries are executed and the microservice performance is studied using the response time and the throughput. The composed queries are test cases which validates services' functionalities. In fact, in contrast to the title of this research article, it presents a functional testing framework and performance analysis rather than microservice performance testing. The performance metrics are assessed to show SUT conditions not to test the microservice performance against of accepted performance behaviors or expected criteria.

- PR4 provides a pattern language using adapting design patterns from software engineering to the cloud system development. This pattern language is composed of development, deployment, supervision, execution, discovery and communication, and monitoring tasks patterns in the microservice granularity. Developing a system using applying sound principles, i.e. patterns, in all phases of the development life cycle is a unique idea adapted from software engineering for the microservice development. PR4 lacks method evaluation and experimental studies to validate the idea.

- PR5 proposes an approach for quantitative assessment of microservice deployment alternatives. This research measures the scalability of an architectural configuration as a performance metric and compares it with alternative configurations. This approach proposes four main tasks containing 1) observing the operational data and analyzing them to estimate the probability of occurring each workload situation. 2) Generating load tests cases based on a defined performance baseline as the expected test result, given an architectural configuration under test, i.e. tests are generated by sampling the distribution of workload satiations according to their probability of occurrence and an expected criterion such as response time as a performance baseline. 3) Computing the microservice scalability using the baseline performance of services as a quantitative requirement. And, 4) Executing the load test and concluding an architecture ability in the satisfaction of scalability requirements in each profile testing. The idea of defining and computing a baseline performance for services and conducting a quantitative metric representing the scalability requirement of a microservice are innovations of this work. This research article automatically generates load test cases for the scalability testing of different configurations. These tests are generated considering the high-level architecture model of the configuration and the quantitative definition of the scalability as the expected result of the test. Furthermore, more tests are assigned to the most likely operational profiles, which are estimated using AI algorithms. A potential point of this work is suggesting the best configuration design. It can compare the possible streams of the operational data for adding resources in terms of performance which can be exploited in automatic scaling and adding resources in an efficient way.

- PR6 proposes a visualized monitoring approach for microservices. This research retrieves a dependency graph from the communication data and validates this graph based on some metrics. Introducing these validation metrics retrieved by reviewing microservice features and challenges can be considered as an innovation. The metrics are ten quantitative measures for evaluation of microservice dependencies. Moreover, this study determines hot spots or high risky services and communications of a system by analyzing the dependency graph and visualizing the results within the graph, which make monitoring a system easier for the developer. PR8 calls the extracted dependency graph an architecture model while the generated graph only includes service communication for the involved services within monitoring the system.

- PR7 presents an anomaly and root cause detection algorithm in the performance test of microservices. This algorithm, first, collects the network connections as well as the average of response time as a Service Level Objective (SLO) metric. Then, a causality graph is generated by detecting any violation in SLO monitoring. The causality graph presents service

communications decorated by notations to show root causes. The root cause is inferred by traversing the graph and checking all of the suspicious nodes. The positive point of the presented algorithm is that the causality graph is reconstructed or updated automatically by detecting any new anomaly in the front-end services. Thus, the method can detect the most recent root causes. Also, this approach suggests ranking the candidate root causes based on the measured SLO, which provides a prioritizing method in performance monitoring.

- PR8 recommends a declarative method for the test of a new version in the deployment line. This paper presents a metamodel for deployment modeling and testing. A deployed version is modeled conformed to the presented metamodel. Then, the model is annotated with tests and automatically executed to check the correctness of the new version. The proposed idea can control deployment versions by providing a possibility of rolling a new version back to the previous version when deployment testing faces any failure. This research article is a declarative test method in which a new version is tested in the business perspective rather than as a technical test of a deployment process. In fact, in contrast to validating the deployment process's steps, it assesses satisfying the desired results of the end user for the new version. However, there are some pointers to improve this idea. For instance, according to the application scope of this research idea, it needs to be sensitive to changes and detect them during the system execution to test any version automatically. PR8 lacks outdating models detection and modeling environment parameters also which might impact on the result of deployment testing.
- PR9 offers a Microservice Adaptive Reliability Testing (MART) algorithm for microservice reliability estimation. This research suggests partitioning the on-demand space of microservices, i.e. partitioning the space of a microservice's methods call, and specifying the probability of failure on demand (PFD) for the partitions. The PFDs are initiated and carried on with updating using monitoring the failed/passed tests over the demands. MART, an adaptive sampling design, is exploited to generate and select tests. During every iteration, before exhausting the budget test, MART selects a test frame with a higher likelihood of failure and generates test cases for the selected frame by deriving demands within the frame. Finally, the reliability of the microservice is estimated using a presented formula. Monitoring real data in change detection and updating probability of failure on demand iteratively makes the idea more accurate and faster in adapting to changes. Moreover, MART is sensitive to the previous tests results in test selection. Therefore, it draws desired test cases in a few steps which makes it more efficient in runtime environments, especially under a scarce testing budget. However, PR9 initiates and updates PFD for test frames derived from on-demand space while service communications might be uncovered in the partitions, which is not discussed in this work.

- PR10 employs three gateways to extract microservice topologies and performance metrics automatically. Derived information is exhibited as statistical graphs or an abstract relational model to provide an easy view for the developer's analysis. However, this monitoring approach might miss some information in the collected data due to the suggested way that registers only the origin and destination of requests. Although it causes a lower overhead to the system rather than usual tracing models, messages tracing contains more details for revealing the causalities in anomaly detection. Moreover, the monitoring task of this approach needs an overlay network which is implemented and configured in Docker and SWARM manager. Thus, the published tool of this work is only deployable in microservices equipped with Docker and SWARM.

- PR11 presents a quality evaluation method based on model drive engineering (MDE) approaches. This research uses recovery engines to extract an architecture model of a microservice. This task can be performed continuously to adapt the architecture model to changes. Next, it specifies a quality model containing quality attributes and their constraints defined based on a customized Domain Specific Language (DSL) provided for quality specifications. A quality evaluation engine serves the architecture and quality models and produces two output models. These two models are evaluation quality model and a weaving model. The first one represents quality attributes with their values. The latter one contains links between the quality and architecture models. Using MDE approaches is the principle novelty of this paper which results in a reusable idea independent of the architecture languages. In fact, the proposed metamodel as a DSL for the quality specification provides an extendable and reusable way for defining quality attributes. Furthermore, the generated weaving model keeps the definition of the quality attributes independent of the architecture platform. The weaving model gets the developer out of dealing with the architecture model platform via storing the links between the quality and architecture models instead of keeping architecture model.

- PR12 presents a new automatic approach for testing a deployed version in a production environment. This approach sends a portion of the load request to the deployed version and checks the correctness and performance of the system in handing the received load request. The load portion is increased if the new version's behaviour satisfies the user's expectations. This research article transforms the log data into meaningful and comparable metrics and decides about the passing/failure of a version using these metrics. This idea employs a statistical test to increase the accuracy of the workload control for the new version, and decrease the impact of the small sample size, i.e. load request size. Using statistic tests for analyzing the performance metrics and deploying a version makes the method more precious. However, while PR12 detects changes in the behaviour of a recently deployed version, some of these changes do not mean a failure of deployment. For instance, a change in hardware may impact user satisfaction and is recognized as a failure in the proposed approach though it is one of the modifications of the new version. Furthermore, this work is one of the research studies which has not published its experimental effort for evaluating the approach and its performance.

- PR13 present a new white-box approach for integration and system test of microservices with APIs conformed to REST. This research results in generating test cases, i.e. a sequence of HTTP calls, within the development phase. The proposed approach is a combination of several algorithms which make it more powerful. It extracts domain knowledge by analyzing the URIs1, used in test template generations. A smart sampling algorithm is employed to speed up the search for producing test cases according to the prerequisite resources. Moreover, a genetic algorithm is applied and mutates test cases to cover as much as specified test targets. The fitness function of this test approach is an optimizing function mixed by a branch distance algorithm which searches branch coverage for advancing the test target coverage. Moreover, using an evolutionary algorithm, namely the Many Independent Objective (MIO), increases the test generation effectiveness of the method against adding test targets. Introduced and customized algorithms in this work have provided a unique test method for microservices though they are well-known in the testing of other software architectures. PR13 is dependent to the communication format of RESTful. The current presented tool supports JSON while there are other languages or formats which are common in the REST style. To test a RESTful application with uncovered languages, some changes should be applied to the proposed approach. As a limitation of PR13, it considers the number of failing endpoints as the number of detected faults while different bugs can result in failing an endpoint or one bug in a test case might cause failing of various endpoints. Therefore, the number of detected faults as a test target is not specified precisely. Furthermore, this research is not able to test databases content and interaction with a database as well since generated test cases do not include database commands. This issue can impact on the code coverage. Moreover, most of the detected faults are generated by test cases with invalid inputs while less attention has been paid to testing application states.

- PR14 adjusts the Delta debugging algorithm from monolithic software architectures to the microservice infrastructure and presents an automatic approach to this end. This approach extracts some circumstances from the failed test cases and executes Delta debugging on them to find the minimum of changes in the circumstance which can cause passing the test case. Since this investigation reveals the cause of problems, the proposed approach can be categorized as research concerning the identification of the root cause of defects. In addition to instantiating Delta debugging, this research provides a layer with an API which facilitates configurations and infrastructure manipulations. This layer automates the system settings and resource allocations required for the executing test cases. In this research article, the Delta method is fed with test cases while the generation of test cases has not been addressed. Test case generation is one of the effective factors in this approach performance. The proposed method can be more robust in fault disclosing if variety of test targets are covered by the input test cases.

- PR15 presents an approach based on generating a service dependency graph for analysis, test, and reuse of microservices. This research idea automatically generates a dependency graph for microservices developed in Java-based Spring Boot framework and tested by Pact. It employs the reflection mechanism for extracting service calls and endpoints to visualize them and their relationship in a graph. Moreover, the generated graph includes dependency cycles as critical areas, the status of service call tests, and the test coverage measure of each service. The method, namely GSMART, detects untested services in the directory code and generates tests for them using reflection in Pact. Therefore, the generated dependency graph is an informative graph for anomaly detection, analysis, and test of the microservice. This investigation also extracts traceability among services and between scenarios and services by appending BDD (Behavioural Driven Development) scenarios to the dependency graph. This feature is used for identifying and periodizing test cases related to a system modification in the regression test phase. Another novelty of this research work is using NLP and Vector space model (VSM) for training a model which expresses the similarity of the existing microservices and a new requirement scenario defined in BDD format. The trained model is delivered to the developer to be used in retrieving the best existing microservice which address the new demanded requirement. Even though the proposed approach presents a potential dependency graph for analysis and regression test, it is specific to following some principles in microservice specification and development. However, the positive point of this research is elaborating a guideline which provides required principles for developing a microservice conformed to the proposed test idea.

- PR16 is one of the works which can be identified as a Netflix innovation in the test of microservices. This article describes “chaos engineering” which is a known and generalized by Netflix for the fault injection in test of microservices. Chaos engineering is a process which finds steady-states of a system and monitors them while disorders the system by injecting faults. Though many studies address similar idea, Netflix is the concessionaire of chaos engineering in microservice testing. The positive point of proposed method in PR16 is that the idea is a general concept adjustable for both hardware and software test. However, PR16 is a magazine article which has not presented implementation and validation in details.

1 UNIFORM RESOURCE IDENTIFIER

- PR17 has performed comparative research on the various microservice platforms such as Spring Boot[2], WildFly Swarm[3], Payara Micro[4], and SilverWare[5] to select the best one for extending to a fault tolerance microservice platform. To this end, various Java libraries including JRugged[6], Hystrix[7], Javaslang CircuitBreaker[8], and Failsafe[9] have been also studied to be used in adding resiliency features to the frameworks. Finally, SilverWare has been taken to be extended to a fault tolerance microservice platform using annotations generated by the Hystrix library. The performed comparative study is a good reference for future research. However, PR17 results are limited to the microservices developed in the Silverware platform. Furthermore, the implemented failure scenarios are limited to those that are producible by the Hystrix library. All the covered faults in this research are crash, circuit breaker, fail, thread pool, and timeout.

- PR18 adapts fault prediction methods of software and network testing to the microservice scope. The purpose of this thesis is studying the effects of services executed on a physical host in the fault prediction. In order to consider the effective metrics in microservice fault prediction, the proposed method extracts the host architecture in addition to capturing other metrics. Faults are forecasted by detecting unusual behavior in monitoring the performance data and considering the host internal architecture. Extracting a hierarchical level of component and service interactions for fault prediction is novelty of this thesis. Furthermore, PR18 compares various fault prediction algorithms in software and network areas, composing of ARIMA [70], HOLT- winters [71], and HORA [72], to analyze what can be customized for the data observed in microservices. The performed studies on these predictors are a potential point of the study which has provided a literature for future research. Moreover, statistical concepts, such as T-tests, have been employed in this study which makes the evaluation results more reliable. However, PR18 has not implemented the idea entirely. The suggested hierarchical predictor, which is the novelty of this work, has not been developed. Therefore, it not available as a reusable predictor in the microservice testing area. Furthermore, concerning fault coverage of the prediction process, PR18 is limited only capturing and analyzing abnormal CPU and memory usage.

- PR19 proposes several meta-models for modeling a microservice development, deployment, presenting dependencies, and monitoring data. According to the introduced meta-models, a developed microservice conforms to these development and dependency meta-models. The deployment model exhibits the architectural context of the underlying hosts, which has been supposed fixed in this article. The dependency model shows operations association in the microservice linking to their version; and the monitoring model represents (time-series) observed data associated to the microservice model's items. Development, dependency, and deployment models are used in system production and deployment. In addition to the generating and managing a microservice in a model-driven approach, this work proposes an anomaly detection method (novelty contribution). In fact, the monitoring model is used in anomaly detection procedure. This thesis has implemented this idea executable on Kubernetes[10] clusters. Evaluating the results using statistical concepts such as accuracy and effect size has enriched the validation process. Moreover, PR19 has compared different software anomaly detection tools such as ΘPAD [73, 74], Event-aware Anomaly Revisor (EAR) [52], and a customized version of RanCorr [75]. The results can be employed future works. However, some parts of the introduced idea have not been implemented. For instance, the automatic extraction of a model from monitored data conformed to the presented meta-model is not developed. Furthermore, the proposed extensions for EAR, suggested for capturing the anomalies related to deploying add-ins, and some proposed details for improving the results are missing in the published tool.

- PR20 provides a parallel test method for the testing of each service by routing the request traffic into two paths, the original path and the resiliency path. The parallel tests improve the performance and flexibility of service tests without affecting each other. In this method, requests in the resiliency path encounter an injected failure. Then, a fallback behavior is implemented and transmitted to the downstream. Then, the result is compared to the success rate determined by transmitting the requests to the original path. Since this method is run for each service, it can reveal causalities. This patent identifies a failure and/or validates an implemented fallback by comparing the generated responses between the general and the resiliency paths. However, this work does not declare the acceptable behavioral variation in the alternative stream or resiliency path compared to the original stream.

- PR21 is a control plane microservice test, which takes test scripts including failure scenarios and expected system behaviours and runs them through the system. This control plane contains three modules. First, a translator module for

2 HTTP://SPRING.IO/PROJECTS/SPRING-BOOT

3 HTTPS://THORNTAIL.IO/

4 HTTPS://WWW.PAYARA.FISH/SOFTWARE/PAYARA-SERVER/PAYARA-MICRO/

5 HTTP://SILVERWAREIO.BLOGSPOT.COM/

6 HTTPS://GITHUB.COM/COMCAST/JRUGGED/WIKI

7 HTTPS://GITHUB.COM/NETFLIX/HYSTRIX/WIKI

8 HTTPS://GITHUB.COM/RESILIENCE4J/RESILIENCE4J

9 HTTPS://GITHUB.COM/JHALTERMAN/FAILSAFE

10 HTTPS://KUBERNETES.IO/

interpreting the fault scenario into fault injection rules at the network level. Second, an orchestrator module for the execution of faults. Third, an assertion checker module compares the observed behavior, extracted from the event log database, with the expected behavior mentioned in the test scripts. The assertion checker verifies the existence of resiliency patterns. However, the resiliency patterns of the assertion checker are limited to timeouts, retries, circuit breakers, and bulkheads.

- PR22 produces test inputs from the passed interface event by the user using an API call graph generation. The API call graph is an annotated graph generated by traversing the user interface event log and the server side log files. This graph is iteratively mutated until meet all scenarios extracted from the user interface. Executing the idea in a live environment makes it sensitive to changes while generating the API graph in a real time setup causes an overhead for the running microservice.

- PR23 generates a state transition graph by traversing the user interface events and the server side logs. This graph is a model used for the test of microservice resiliency which is labeled using an API call graph. The salient point of this idea is that subgraphs are prioritized based on the impact of a failure. This prioritization provides an order for the test experiments. Then, resiliency patterns are injected into the subgraphs based on the provided priority. The higher priority a subgraph has, the more prevalent impact it causes. Therefore, high risky subgraphs are tested first by inserting faults to the graph. The path is captured if it cannot handle the fault. The output of PR23 is a failure scenario, i.e., a path in the transition graph, which bears a failure. Moreover, PR23 annotates the sub-paths of the graph after passing a resiliency test or injecting a failure pattern. This can decrease the test redundancy. However, PR23 suggests a method with the ability to inject a restricted number of resiliency patterns such as circuit breaker, time out, bulkhead, and bounded patterns which follows crash-recover, fail-stop/crash failures, and performance/omission failures.

- PR24 is a patent which presents a validation deployment approach for data driven applications, designed with microservices. The proposed method monitors data schema and detect updates. Then, by detecting an update, the liveness probe associated to the updated microservice is upgraded with the data schema as test data. If the updated microservice can resolve the test data, the microservice and test data are packaged in a service dataplan container. Otherwise, the previous version of the microservice is substituted. This patent is a unique idea in change detection using monitoring data in data driven applications. P24 is limited to the data driven application. Change or update detection in the proposed method is dependent on the data schema. The approach cannot find an update in a microservice structure if the monitoring data does not represent that new feature. Therefore, generating test data is an important part of the proposed idea which is not addressed in the study. To detect updates, the data schema should be updated by changes due to the new version of the microservice.

- PR25 presents a problem detection method using the monitoring of performance metrics. This method traces information through the network and creates a deployment profile for microservices cooperating in a request. Then, it classifies the created profile as an existing, extension, update, or a subset profile by comparing it with existing profiles. The performance metrics is tracked for the classified profile. This information helps the developer to analyze performance and decide about the deploying a new version or rolling back to the previous one. Using the classification of microservice profiles in analyzing the performance of an application is a unique idea.

- PR26 presents a test method for managing container testing. Emphasizing on the monitoring and evaluating containers has not been addressed in other studies, and this is one of the noticeable contributions of this work. A container crash during the test can cause a need for restarting the software container. Moreover, the container filesystem may be lost after a crash, which extends fixing time. The proposed idea in PR26 comprises two components, a test manager and a test controller which respectively deals with creating and monitoring the test container tasks. The test manager creates a test container image on the host with test artifacts and dependencies. Then, the manager populates a directory with sets of tests and mounts the directory to the test container. The manager is also responsible of running, starting and killing the container. The other component is test controller associated with the test container. The test controller executes the set of tests and applications inside of the container, monitors the test results, stores results in the directory, and provides feedbacks to the developer. Since the directory is not located inside of the test container, container crashes have no impact on the test scripts and changes in the tests are ineffective in the container structure as well. Therefore, the container does not need to be rebuilt for any change or update in the tests. However, the detail of test scripts is not discussed in this research. In fact, this study does not suggest test types which can reveal container root causes or anomalies.