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 to 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 [1]. 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 a performance testing DSL and a model-driven framework for executing generated model of the proposed DSL. The developer can specify the performance test intent including the goal, environment configuration, workload, etc. using the proposed DSL entities. The generated instance provides advanced fea tures which are according to the microservice architecture features. For instance, the generated test instance can be integrated into a continuous software development pipeline; or it includes concepts for collecting data and controlling the configuration of each service, such as workloads and timeouts. Overall, the presented DSL provides easy ways for the user to employ advanced performance test concepts. Moreover, for automating the execution of test instance generated by the proposed DSL, a model-driven framework is presented by this idea. The framework defines the syntax and semantic of scheduling and executing performance test instances, in YAML format. The benefit of this framework is its ability to be integrated into the other parts of the development pipeline via a REST API. As the proposed DSL includes different concepts of a continuous development pipeline, the generated test scripts by this method cover testing various layers of a microservice including testing a container with specific features, s ervices, and a system. However, to our knowledge, the current version of PR5 does not share the proposed DSL as an open-source language. Therefore, using some concept such as testing the integration level is not possible with the current published DSL.

- PR6 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.

- PR7 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. PR9 calls the extracted dependency graph an architecture model while the generated graph only includes service communication for the involved services within monitoring the system.

- PR8 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.

- PR9 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 points 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. PR9 lacks detecting outdated versions model and modeling environment parameters also which might impact on the result of deployment testing.

- PR10 proposes a test case generation method for regression testing. This idea employs execution log files, obtained from a real-time execution, to extract test cases and mock objects for the new version of a microservice. The execution logs are analyzed for extracting the microservice request messaged and I/O dependencies. The microservice requests are used as test cases and I/O dependencies are synthesized as mocked services. Considering the mock objects in the test case generation is a strong point of this idea. However, mocks are only generated for I/O dependencies. Moreover, the method is limited to the stateless microservices with REST API and Mongo DB. Also, the log files are assumed that proved synchronous messages while it is not appliable to real case studies. Moreover, this idea is limited to generating test cases for observable changes or scenarios in the log files. Therefore, more improvement can be applied to make it stronger in testing all possible scenarios.

- PR11 proposes the same idea with PR13 for exploring the fault scenario space to generate scripts with the ability to sever the failure detection process. However, instead of using LDFI, PR11 explores the integration test cases to generate the higher priority fault scenarios. Integration test cases are executed and visualized using DOM and a visual test case extractor [2]. Moreover, distributed traces are collected using monitoring the test execution setup by Jaeger, provided by Istio. PR11 extracts injection point from test execution traces. To generate fault scenarios, PR11 generates a fault scenario for each test case randomly. Then, its impact on the system and test execution is evaluated and scored. Next, PR11 algorithm selects a previously fault test with the higher impact and mutants it by producing changes in its injection point, fault type, and/or its request message call. Each generated fault test is again assessed in terms of its impact on the system behaviour and the test execution result. Test selection process and generating new fault is continued until a user-defined time is met or a severe bug is detected. PR11 is different from other fault injection methods in the level tested scenarios that is testing system functionalities. However, this method needs the system integration test as a requirement to explore them. Moreover, using the service mesh of the microservice system, which provides detailed information about the execution, is another strong point of PR11. Using service mesh allows PR11 to find injection points and simulating faults as well.

- PR12 proposes a BDD version of the contract acceptance test for microservices, which make acceptance tests reusable and more maintainable. This article suggests using one repository in BDD framework architecture, called Automated Acceptance Test (AAT), instead of using different repositories with repetitive features. There are the same acceptance test features or common steps in BDD scenarios for testing different functionalities. These common concepts are repeated in each source code repository. This redundancy makes maintenance, audit, and reusability of tests difficult. PR12 solves the mentioned challenges by structuring the BDD test features in one repository, AAT. Microservice's feature files and BDD scenarios are organized in a shared directory with all groups of developers, testers, and product owners. This facilitates test audits for different groups. Moreover, ATT has a directory including both implementations of common steps of BDD tests and service-specific tests. It increases the reusability of common steps as they are accessible for all BDD tests. To improve maintainability, PR12 organizes source code for the BDD framework, i.e., testing methodology, in another directory of AAT. This solution makes upgrading the BDD framework easy. However, the main problem of this idea is keeping directories updated and consistent with the system changes. Furthermore, generating tests and their execution as an important part of a test method is delegated to the BDD framework. The paper also lacks experimental evaluation.

- PR13 describes a Netflix test method, an adapted version of LDFI method. LDFI is a fault injection method that guides search of possible fault scenarios, supposed to be injected into the system. This method is initialized by a correct state of the SUT and generates a system lineage graph using a backward and recursive method. Then, the lineage graph by the granularity of data computation is translated to formal phrases. Candidate fault scenarios are obtained by solving the formal graph. These fault scenarios are a combination of failures that invalidate all alternative paths for a correct solution to the system. Then, the generated fault scenario is injected into the system. When the SUT fails, i.e., when there is no alternative path for handling the injected failure, the user will be notified by sharing the failed execution trace. Passed injected scenarios are used to improve the lineage graph as LDFI merges the corresponding executed path to the previous graph and enhances the correct model of the SUT. PR13 discusses how LDFI prototype, MOLLY, has been adjusted to the Netflix environment. Netflix adapts LDFI to generate a call graph out of service interactions, in the level of service nodes. Moreover, Netflix uses learning methods to identify the equivalent classes of execution results and find redundancies, alternative path, for each request. Although this idea is a practical and industrial test method, the published article lacks experiments, discussion, and results. Generally, LDFI is a similar approach to PR1 idea in solving the formal phrases for finding the optimized fault scenarios. However, the generated graph and the outcomes are different in these two articles. PR1 produces a system model using the system specification and employs learning approaches to generate test cases whilst LDFI results in a lineage graph from the system execution using a recursive approach and exploit it to generate fault scenarios.

- PR14 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, PR14 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.

- PR15 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.

- PR16 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 principal 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.

- PR17 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 behavior 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 PR17 detects changes in the behavior 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.

- PR18 proposes a deep learning algorithm, deep Bayesian network, for modelling the expected service call traces and their response times. This method first collects Remote Procedure Calls (RPC) and response times. Then, a service trace vector including service call trace ID and corresponding response time to each trace ID is extracted to learn the Bayesian model. This vector is usable for other learning algorithms as well. The unsupervised Bayesian network algorithm generates a pattern for expected request calls and their response times. This model is produced in an offline environment and suggested to be periodically built to support new changes. PR18 uses the generated model in anomaly detection in a live environment. New traces are represented in the vector format and are compared with the generated model. An unseen path with a low likelihood to happen is known as an anomaly. PR18 suggests the longest anomalous path as a detected root cause.

- PR19 propose a method for detecting anomalies of container granularity level in the production setup. This method collects tracing data including sender Id, receiver Id, container Id of running service, request message result tag, start time, and elapsed times. Data collection can be performed by any monitoring tool in real-time system execution. Tracing data are grouped by sender Id and form tracing chains, which represent all tracing data from the same client's request. Next, PR19 classifies tracing chains based on quantity and type of containers. Anomalies are detected in a window time when a result tag shows a failure or when the elapsed time is three times greater than the average of elapsed times for the tracing chains passing through the same container. Fault localization is performed by Spectrum Based Fault Localization (SBFL) [3], a statistic algorithm. SBFL localizes anomalies by finding the difference between successful and failed tracing chains. Finally, a ranked list of suspicious containers is provided to the tester. PR19 is providing an anomaly and root cause detection in the level of containers which makes this idea different from similar works. Moreover, since the distributed tracing data in a real setup are collected, PR19 doesn't need to extract services or calls' dependencies and it makes it a lighter approach. However, PR19 has made some statistical assumptions that are not always true.

- PR20 proposes an anomaly detection approach using clustering response time series by BIRCH [4]. Response time besides other performance metrics such as resources metrics of containers are collected using various monitoring tools from components including servers, microservices, and containers. PR20 employs Node-explorer to monitor the operating system of server nodes, Cadvisor is used for collecting containers' resources metrics, Istio service mesh is gathered for tracing network communication and response times, and Prometheus monitors the microservice under test. An anomaly is detected when multiple clusters show an SLO violation or an increase in a response time, i.e., clustered times series meet a pre-defined threshold. By detecting an anomaly, a graph representing anomaly propagation across the components is generated, i.e., a component dependency graph. Then, anomaly propagation across dependent components' metrics is assessed. This assessment generates a causality graph or an anomaly path that shows metrics of the components affecting each other by an anomaly. PR20 weighs the metrics causality graph, infers the culprit metric, and ranks weights using a graph centrality algorithm. Finally, the highest-ranked metric has the highest probability to be the root cause. PR20 approach provides a significant precision in anomaly detection and root cause but, it requires a predefined threshold in running the unsupervised clustering algorithm and is limited in detecting anomalies that cause a change in response time.

- PR21 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 URIs[1], 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. PR21 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 PR21, 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.

- PR22 proposes an iterative testing approach in which test cases are iteratively executed to find a desired characteristic that fails the system under test. PR22 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

1 Uniform Resource Identifier

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.

- PR23 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 (Behavioral 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.

- PR24 is an anomaly detection approach employing the knee-point indicator [5]. The knee-point shows performance bottleneck when throughput is in a stable condition while a sudden increase happens in the latency time. PR24 collects latency time and throughput using an API proxy. Then, when a knee-point is observed for a service, the service dependencies or anomaly propagation path from the detected service is extracted. The extracted path is called a service impact graph generated by an unsupervised PC algorithm [6]. The service impact graph represents services dependencies using metric dependency. PR24 defines Pearson correlation between services' metrics as the service dependency. Traversing service impact graph is used for root cause detection. Because of the reliability design patterns, the anomaly propagation path might change in a microservice. Therefore, PR24 tries to calibrate metrics when it detects a network design pattern. To calibrate metrics, PR24 transforms metrics to features representing the patterns. Then, by detecting the network design pattern, metrics in the interval of the detected pattern are calibrated using their neighbour metrics. Then, calibrated metrics are replaced in the graph, and the graph is traversed for detecting the root causes. More likely visiting services in the graph traverse are ranked as higher scored root causes. This work suggests promising approaches for extracting service impact factors and considering the impact of the network design patterns on anomaly propagation. However, the calibration has been implemented and validated for only one pattern, the circuit-break pattern.

- PR25 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 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 PR25 is that the idea is a general concept adjustable for both hardware and software test. However, PR25 is a magazine article which has not presented implementation and validation in details.

- PR26 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, PR26 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.

- PR27 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, PR27 compares various fault prediction algorithms in software and network areas, composing of ARIMA [7], HOLT- winters [8], and HORA [9], 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, PR27 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, PR27 is limited only capturing and analyzing abnormal CPU and memory usage.

- PR28 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, PR28 has compared different software anomaly detection tools such as ΘPAD [10, 11], Event-aware Anomaly Revisor (EAR) [89], and a customized version of RanCorr [12]. 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.

- PR29 is a control plane microservice test, which takes test scripts including failure scenarios and expected system behaviors and runs them through the system. This control plane contains three modules. First, a translator module for 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

2 http://spring.io/projects/spring-boot
3 https://thorntail.io/
4https://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/

of resiliency patterns. However, the resiliency patterns of the assertion checker are limited to timeouts, retries, circuit breakers, and bulkheads.

- PR30 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.

- PR31 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.

- PR32 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 data-plan 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. PR32 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.

- PR33 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 provi des 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 PR33 is a failure scenario, i.e., a path in the transition graph, which bears a failure. Moreover, PR33 annotates the sub-paths of the graph after passing a resiliency test or injecting a failure pattern. This can decrease the test redundancy. However, PR33 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.

- PR34 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.

- PR35 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 PR35 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.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," Distributed computing, vol. 13, no. 2, pp. 99-125, 2000.

[2] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 503-514.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007), 2007: IEEE, pp. 89-98.

[4] A. Gulenko, F. Schmidt, A. Acker, M. Wallschläger, O. Kao, and F. Liu, "Detecting anomalous behavior of black-box services modeled with distance-based online clustering," in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018: IEEE, pp. 912-915.

[5] K. K. Ramakrishnan and R. Jain, "A binary feedback scheme for congestion avoidance in computer networks," ACM Transactions on Computer Systems (TOCS), vol. 8, no. 2, pp. 158-181, 1990.

[6] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, Causation, prediction, and search. MIT press, 2000.

[7] Y. Khandakar, M. U. D. o. Econometrics, and B. Statistics, Automatic ARIMA Forecasting. Monash University, 2009.

[8] C. Chatfield, "The Holt-winters forecasting procedure," Journal of the Royal Statistical Society: Series C (Applied Statistics), vol. 27, no. 3, pp. 264-279, 1978.

[9] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, "Hora: Architecture-aware online failure prediction," Journal of Systems and Software, vol. 137, pp. 669-685, 2018.

[10] T. C. Bielefeld, "Online performance anomaly detection for large-scale software systems," Kiel University, 2012.

[11] T. Frotscher, "Architecture-based multivariate anomaly detection for software systems," Kiel University, 2013.

[12] N. Marwede, M. Rohr, A. v. Hoorn, and W. Hasselbring, "Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems Based on Timing Behavior Anomaly Correlation," in 2009 13th European Conference on Software Maintenance and Reengineering, 24-27 March 2009 2009, pp. 47-58, doi: 10.1109/CSMR.2009.15.