Misconfiguration Discovery with Principal Component Analysis for Cloud-Native Services

Alif Akbar Pranata, Olivier Barais, Johann Bourcier, Ludovic Noirie

Misconfiguration Discovery with Principal Component Analysis for Cloud-Native Services

Alif Akbar Pranata, Olivier Barais, Johann Bourcier
Univ Rennes 1, Inria, Irisa
Rennes, France
alif-akbar.pranata@inria.fr, olivier.barais@irisa.fr, johann.bourcier@inria.fr

Ludovic Noirie
Nokia Bell Labs
Nozay, France
ludovic.noirie@nokia-bell-labs.com

***Abstract*—Cloud applications and services have significantly increased the importance of system and service configuration activities. These activities include updating (i) these services, (ii) their dependencies on third parties, (iii) their configurations, (iv) the configuration of the execution environment, (v) network configurations. The high frequency of updates results in significant configuration complexity that can lead to failures or performance drops. To mitigate these risks, service providers extensively rely on testing techniques, such as metamorphic testing, to detect these failures before moving to production. However, the development and maintenance of these tests are costly, especially the oracle, which must determine whether a system's performance remains within acceptable boundaries. This paper explores the use of a learning method called Principal Component Analysis (PCA) to learn about acceptable performance metrics on cloud-native services and identify a metamorphic relationship between the nominal service behavior and the value of these metrics. We investigate the following research question: Is it possible to combine the metamorphic testing technique with learning methods on service monitoring data to detect error-prone reconfigurations before moving to production? We remove the developers' burden to define a specific oracle in detecting these configuration issues. For validation, we applied this proposal on a distributed media streaming application whose authentication was managed by an external identity and access management services. This application illustrates both the heterogeneity of the technologies used to build this type of service and its large configuration space. Our proposal demonstrated the ability to identify error-prone reconfigurations using PCA.**

***Keywords*-Reconfigurations; Metamorphic testing; Principal component analysis; Cloud-native services**

I. Introduction

A growing number of network services and applications are now being deployed in the form of so-called "cloud native" applications or services. [1]. These services are characterized by the following features:

- A service-based architecture. The style can be any architectural model that is modular and loosely coupled.
- A common packaging model and a self-contained execution environment that provides portability as well as isolation using container, VM and/or uni-kernel solutions.
- A continuous and automated process to develop, build and deploy these services.

As a side-effect of this trend, the frequency of services reconfigurations is significantly increased [2], [3], [4]. These reconfigurations are related to: changes in the source code of services or third-party libraries of these services, a change in the parameters of one of these services, or a change in the underlying network parameters. The software complexity is now overwhelming and it becomes difficult to guarantee, a priori, that a new service configuration will be correct or less error-prone [5]. Each of these new, modified configurations has the potential to introduce errors at the service level. While some reconfigurations or updates can significantly change the service behavior, most of these reconfigurations do not fundamentally change the business logic. Still, business outputs are affected by this complexity due to unexpected errors.

Each reconfiguration is seen as a network state over time. Given that errors and failures are caused by reconfiguration management, therefore, from a high abstraction level, the expected service behavior should remain "*close*" to the previous configurations, meaning that for each particular network state should have relations to the $state - 1$ configuration as the oracle definition to embrace failures. Software testing techniques are well-known techniques to address reconfiguration management issues and to ensure performance validity [6]. In such techniques, each test defines an oracle, characterized by input-output definitions to check whether a test has passed or failed. However, defining oracle and maintaining the testing code base is burdensome and costly.

A metamorphic testing method has been proposed [7] to test programs removing the need for oracle. It employs properties of the target function, known as metamorphic relations, to generate follow-up test cases and verify the outputs automatically. The intuition behind using metamorphic testing to alleviate the oracle problem is as follows: Even if we cannot determine the correctness of the actual outputs for individual input, it may still be possible to use relations among the expected outputs of multiple related inputs (and the inputs themselves). Following the principle of metamorphic testing, it has been suggested that indi-

cating a service reconfiguration validity is possible if there are metamorphic relations between the previous behaviors of the service and the recent behavior of the service with the new configuration [8]. Yet, since it is nearly impossible to have an exact characterization of the behavior of a service, the main challenge is to find an efficient method to characterize the service behavior and compare several executions of a particular system.

In this paper we study the discovery of error-prone reconfigurations to understand the complex nature of native cloud services and identify the actions that may lead to erroneous behavior due to misconfigurations. Motivated by the ability of metamorphic testing to avoid oracle definition for testing software systems, we rely on performance metrics as an input to this analysis to identify potential misconfigurations. Two types of metrics are introduced: generic metrics and business metrics that must be kept within acceptable limits to avoid user experience degradation and maintain continuous delivery

We applied a learning method called principal component analysis (PCA) [9], [10] on cloud-native applications metrics. As a proof of concept, we tested our approach in a media streaming application, simulating a real-world use case scenario of cloud-native systems. Given the advantage of no oracle definition and flexible metrics selection, our experiment shows that the use of generic and business metrics, plotted as the metamorphic relations of multiple system executions, as the inputs of the PCA are effective to discover potential valid and invalid system reconfigurations in the cloud environment. We also confirm such validity of each reconfiguration by measuring the proximity of metamorphic relations of each system execution.

The remainder of the paper is as follows. Section II explores real-world implementation examples that motivate this paper, discusses the concept of PCA, and directs a research question, followed by the proposed approach in section III. We show our implementation scenario and the experiment setup in IV, then we evaluate and validate our approach in section V. Section VI discusses the related works. Finally, section VII concludes the paper and foresees our future works.

II. Motivation & Background

In this section, we provides examples of today's network system complexity from companies and our specific implementation. Then, we discuss principal component analysis (PCA) method for our evaluation technique. To understand the purposes and constraints of our work, we develop a research question and raise the hypothesis.

A. Motivating Examples: Software Complexity

The software is complex [5], [11]. It has the inherent characteristic of having several degrees of dependencies that exist and work together to form a whole system to meet customer needs. A software system is therefore the result of the spontaneous composition of a set of modules.

The trend towards the "softwarization" of everything allows IT to move gradually to software-based implementation, usually with API technology for service dependencies and communication. An example of this evolution is the current trend of software-defined networks/network functions virtualization (SDN/NFV) in a native cloud service environment [12]. A case of Amazon networks is formed by software-based microservices, creating full connectivity through which API communicates information data over network protocols in the cloud environment[1].

The complexity and dynamic of software module interaction makes system management difficult. Failures are inevitable, especially in the production environment, when engineers and experts are not always available to maintain and manage the systems. As a motivating example, consider a large-scale distributed system with microservices deployed in cloud servers to provide seamless data exchange and processing around the world. We take a more concrete faulty event by Google Clout Platform (GCP)[2]. GCP has Memcache, an app engine service that speeds up response to data queries by storing them in cache memory. Google utilizes well-built, automatic failover to ensure a seamlessly switch from one failed data center to the other to prevent failure after setting up Memcache for a consistent view of data center serving every application. Yet, utilizing a well-maintained failover mechanism does not mean that systems are invulnerable. Failures can occur due to the unforeseen circumstances at any time during runtime due to a single failure of a module.

In this paper, we take an example of a simple media streaming application (built for our use case implementation in section IV) where some instances of video server deliver video data to their clients. The implementation complexity increases when an access management server and a reversed proxy server sit between the video servers and clients to provide authentication and authorization mechanisms as well as a load balancing scheme. Considering only the video server application, we can have at least 174 application modules. From those modules, we have eleven core dependencies with the software version of each dependency as listed in table I. When we deploy and run our media streaming application system in a cloud environment, the system may evolve during the runtime and it becomes hard to manage background events in the execution. If, at any moment, we upgrade the version of one of the dependencies, other services that depend on it may have to adapt their behavior and may fail.

[1]A presentation by Chris Munns at Amazon.com: *I love APIs 2015: Microservices at Amazon*. Reference: https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258.

[2]An incident happened in 06/11/2017 for 1 hour 50 minutes. Reference: http://status.cloud.google.com/incident/appengine/17007.

B. Principal Component Analysis (PCA)

Principal component analysis (PCA) [9], [10] is an unsupervised learning method, multivariate data analysis of large data sets. It reduces the dimensionality to increase interpretability without losing essential information at the same time. The core idea behind PCA is that it creates from the original data using orthogonal transformation new uncorrelated variables that successively maximize variance. This new variable is called the principal components (PCs). Finding PCs reduces solving an eigenvalue/eigenvector problem. It accomplishes such task at hand, not a priori, making PCA an adaptive tool for data analysis. The adaptivity increases as a result of its capability to be tailored to various data types and structures.

The main objective of PCA is to reduce the dimensionality of the original data after obtaining the maximum amount of variance with the selected minimum number of PCs. The process of PCA is explained as follows.

1) Given the original 2 x n (n is the number of metrics), PCA computes the eigenvectors and the eigenvalues of the covariance matrix. The eigenvectors are used to project the data from n dimensions to decrease representation, and the eigenvalues provide the data variance in the eigenvector direction.
2) PCA iterates the calculation of finding PCs using the number of eigenvector. The first eigenvector determines the direction of the highest data variance and the first PC is obtained from the data with the greatest possible variance in the data set.
3) The subsequent PCs (second, third, and so on) are obtained in the same manner using the respective eigenvector value inside the iteration. However, there is a condition for each PC: it is uncorrelated with the previous one and it accounts for the next highest variance.
4) Each of the eigenvector and the eigenvalue is associated with the direction of each PC. The eigenvector associated with the largest eigenvalue has the same direction as the first PC. The subsequent PCs have the same association rule.

PCA plots data points after its calculation into a PCA space. PCA can show data that have extreme points, called outliers, which do not follow the model of the majority of the data at the boundaries of the space.

C. Research Question

We formulated the following research question: **is it possible to combine metamorphic testing technique with unsupervised learning methods on service monitoring data to detect error-prone reconfigurations before moving to production?** Our hypothesis arises for the answer.

Metamorphic testing is a comparison testing technique of two programs that have a relational pattern called metamorphic relations (MRs) for checking and comparing the correctness of attributes in programs. In common cases, the MRs are known, for example when we perform metamorphic testing in functional programming (such as NumPy library[3] in Python) which relies mostly on mathematical functions computations and thus are comparable. But there may be other cases, such as in our implementation context, which is difficult to find relations and patterns as we avoid oracle definitions. This leads to insufficient tooling for comparison and analysis. The combination of metamorphic testing with unsupervised learning method is foreseen as the solution. Given neither the metamorphic relations nor the oracle definitions, our proposed approach automatically creates relations among different sets of input and output.

Table I
MODULE DEPENDENCIES IN VIDEO SERVER

Names of dependency	Version
body-parser	1.19.0
cors	2.8.5
express	4.17.1
express-http-proxy	1.6.0
express-session	1.17.0
keycloak	1.2.0
keycloak-connect	9.0.3
keycloak-js	9.0.3
request	2.88.2
videojs-resolution-switcher	0.4.2
xmlhttprequest	1.8.0

III. The Proposed Approach

We propose a technique to discover misconfiguration in cloud-native services. Our approach was motivated by the metamorphic testing technique that removes the need to define oracle in advance. As a replacement, we define our oracle with an unsupervised learning method called principal component analysis (PCA) method. We choose PCA among similar machine learning and dimensional reduction technique such as ICA, LDA, and SVD as it suit our requirement: measuring uncorrelated and unsupervised metrics. We monitored and analyzed our approach in our experiment using monitoring agents and statistical distance measurements.

A. General Approach

Our approach was motivated by the metamorphic testing technique, which compares and analyze multiple program executions with each of them may have different performance behavior due to adjusted configurations, or reconfigurations, during the runtime. We define a reconfiguration as an action trigger that affects the execution behavior yet does not change the application business outputs. Reconfiguration aimed to cause intentional system disruption. The examples of reconfigurations are: changing parameter,

[3]NumPy library. Reference: https://numpy.org/

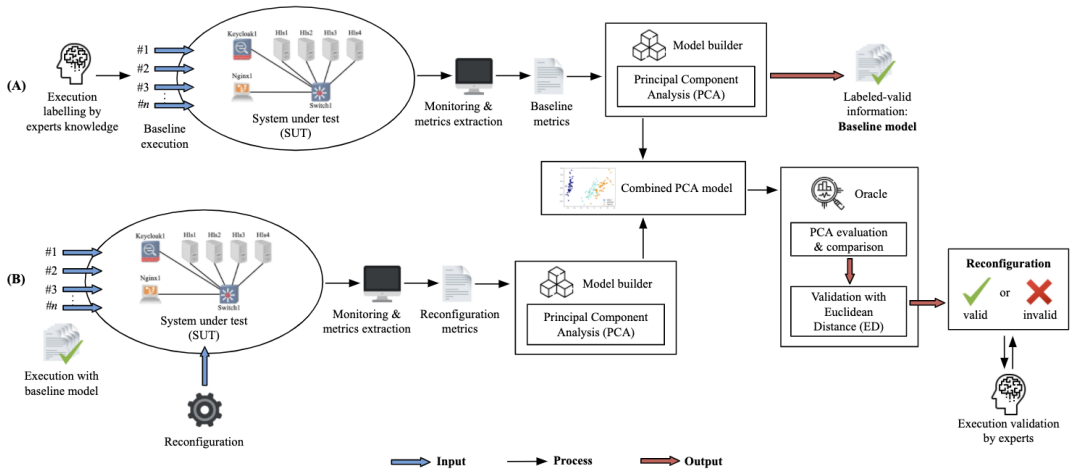

Figure 1. The proposed approach of misconfigurations discovery in cloud-native services: (A) Baseline execution; (B) Reconfiguration scheme.

revoking access, shutting down, or restarting a service (The exact reconfigurations in section V-B). Our approach tested reconfigurations and discovered misconfigurations that may fail cloud-native services execution.

Metamorphic testing also offers flexibility in removing a priori oracle definition and choosing any metrics for evaluation. The metrics are useful for the PCA to create relations among different program executions. In the metamorphic testing technique, these relations are called metamorphic relations (MRs). Our approach used the MRs as our oracle definition for PCA. We compared and analyzed the MRs to check the correctness of the executions under testing.

Figure 1 illustrates the general approach of our proposal. We had two testing steps, denoted by (A) and (B), which differ in the context of execution. Step (A), called baseline execution, is defined as the initial execution on which we had experts knowledge about the normal expected execution. To avoid manual and repetitive effort for generating baseline execution, we automate the process, starting from creating nodes, connecting links, configuring the networks attributes (ip, gateway, etc.) per nodes, launching services in the nodes, until destroying the nodes. We tested our system using the knowledge for the (A) input in a system under test (SUT) consisting of network architecture that runs our use case scenario with running services in a cloud environment (detailed description of use case scenario in section IV). During the execution, we monitored the execution behavior, extracted and obtained the performance metrics data for building the model of execution behavior using PCA. Our model builder run the PCA method for each baseline execution to obtain this testing output, namely the baseline model.

With the baseline model in (A), we then executed another set of executions (B) in the SUT. Here in (B), we applied reconfigurations to the SUT to change the execution behavior. The subsequent processes were similar to (A) until we built the (B)'s PCA model. From here, we combined the PCA model from (A) and (B) to obtain our oracle by evaluating and comparing the PCA model. We obtained the PCA space, consisting of each data point representing each execution behavior by the PCA calculation. We then evaluated and compared each data point and validated their relations using euclidean distance (ED). More specifically, we measured the euclidean distance between each data point and the centroid value of the major cluster in the PCA space. The euclidean distance also served as the MRs in the perspective of the metamorphic testing approach. Finally, we decided the correctness of each execution, whether it is valid and invalid reconfigurations based on the ED relations of each execution in the PCA space. This decision is the final output of our proposed approach. Further validation can be made by the experts, whether a particular execution has the correct decision of valid or invalid reconfigurations.

By the euclidean distance (or MRs), Our approach categorized and defined two sets of output representing the system execution correctness as follows:

1) **valid reconfiguration**, which did not affect the normal delivery of data after the trigger was applied.
2) **invalid reconfiguration**, which produced error messages and disrupted the data delivery after the trigger was applied.

We define error messages as the report of error obtained by monitoring agent in the form of bad requests/responses in data exchange. Here, we obtained the number of HTTP error code messages.

B. Monitoring & Analysis

We deployed monitoring agents in our network architecture for monitoring purposes. Our monitoring agents collected the metrics data and exported the data into

a comma-separated values file format. For analysis, we obtained and compared a set of two scenario outputs of baseline execution and reconfigurations. Specifically, we selected metrics for observation, extracted the metrics data from each scenario, and constructed our data set from the data for PCA method processing.

We obtained the result abstraction of valid and invalid reconfigurations with the PCA method. We affirmed the valid reconfigurations as the correct system behavior. At the same time, we also discovered system miscofigurations caused by invalid reconfigurations. In order to validate the classification of reconfiguration validity, we calculated the euclidean distance (ED) of each data points to the centroid of the baseline execution scenario in the PCA space. Data points representing each execution behavior may be scattered in the PCA space in far proximity that we considered outliers. To determine the outliers, we computed the 97.5%-Quantile Q of the Chi-square distribution as a cutoff value of the ED ($\sqrt{X^2_{a,0.975}}$). According to the table of the Chi-square distribution[4], the value is equal to $\sqrt{7.38} = 2.71$. Any ED value bigger than that is identified as the outlier, which means that the reconfiguration is invalid. Otherwise, the system running behavior is valid after the reconfigurations.

IV. Implementation

We implemented our proposed approach in a cloud-native system running a real-world business application. The application in our experiment had a use case scenario with a system under test (SUT), which is defined as the set of particular services as the testing targets. The implementation was a media streaming system with services that integrated processing and communicated media data.

A. Use Case Scenario

Application domains range widely in cloud environments, such as data stream processing. Such processing operates a series of continuous data delivery to achieve outputs. These domains include media/video streaming application, delivering video information and playing it on-demand as per user needs [13]. The common protocol in this application is HTTP live streaming (HLS), which has gained popularity over the past years due to its fine rate adaptation and workload sharing [14].

Media streaming in a large-scale system can create service and delivery quality issues to clients, mostly when reconfigurations occur during the runtime. For example, when a service is down or changes the parameter value, other dependant services may lose its states to continue functioning correctly. Due to the complex nature of such application systems, we devised an experiment technique

[4]Chi-Square distribution table. Reference: https://store.fmi.uni-sofia.bg/fmi/statist/education/Virtual_Labs/tables/tables3.html

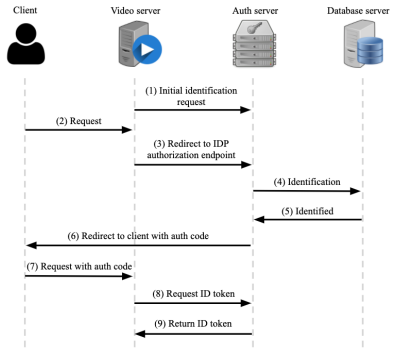

Figure 2. The workflow of identity and access management mechanism by the video server instances and clients.

in a media streaming application to discover dynamic system behavior and detect misconfigurations resulting from reconfigurations.

We applied our approach to a real-world video server-clients communication with an identity and access management service. There were four objects in our implementation: video server, client nodes (end users), identity and access management server (or auth server for brevity), and database server. Figure 2 illustrates the workflow of these objects. Initially, the video server executed an identification mechanism by asking the auth server to obtain a secure token (1). Then, clients requested media stream to the video server (2). Before holding its secure token from the auth server, the video server could not directly grant valid access to the clients for the requested video data. Instead, it redirected the request to an authorization endpoint so the clients can log in to the video server for authorized access (3). Once logged in, the auth server communicated with its database server to ask for client identification (4). In our implementation, the database server sat in the same machine with the auth server. If not identified, the video servers access was blocked; otherwise, the auth server identified the request (5) and redirected the client responses with auth code (6). This code contains an access token that has a limited lifespan. The detail of short lifespan benefit is out of the scope in our discussion (in short, it is beneficial for enhanced security: to keep short validity duration for frequent checking of attackers in the system). Due to the short lifespan of access token, the clients had to refresh the auth code with a refresh token when the access token reached the timeout. Upon obtaining the auth code, the clients requested the video data with its code to the video server (7). The Video server checked once again for the code validity and requested a valid session with an ID token to the auth server (8). The auth server returned the ID token (9), and finally, the valid connection between clients and video server was established.

To evaluate the implementation, we chose two kinds of native system metrics: generic and business. The generic

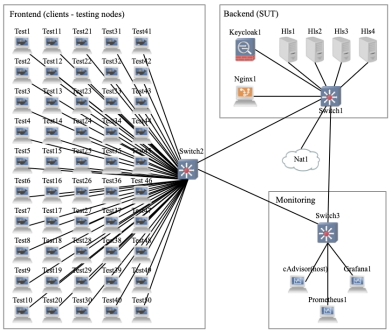

Figure 3. The network architecture of our media streaming application.

metrics were a quantitative measurement that dealt with values inherent from the general system running performance, such as traffic rate, CPU usage, and memory usage. The business metrics were the application-specific metric that became the key parameter in running such an application in real-world execution. In our implementation, the business metric was the number of error messages caused by reconfiguration. We elaborate our metrics selection in section V-A.

B. Experiment Setup

We set up and run our network architecture consisting of nodes that run services in container applications. Our services in the architecture communicated and constructed our media streaming application. We experimented and evaluated our approach in GNS3 network emulation environment. The GNS3 server runs in our private cloud lab with computing resources that can accommodate load consumption with as many as thousands of nodes: Intel(R) Xeon(R) Gold 6238 CPU with 2.10GHz 88 Core, 187GB memory, and 11 TB disk storage size.

We show the network architecture of media streaming application in our experiment in figure 3. We had four objects in the experiment as described in section IV-A. For video server and clients, we deployed multiple instances of each of them. We also deployed a reversed proxy server to serve load balancing for requests sent to each video server instance. We separated service nodes based on service layering and function into the system under test (SUT) cluster, testing cluster, and monitoring cluster. Following this setup, we implemented our approach to the use case scenario, evaluated our experiments and validated the results.

V. Evaluation & Validation

We evaluated our proposed approach in the experiments for various system reconfigurations on system under test (SUT) and discovered misconfigurations in cloud-native systems. Our experiments compared and analyzed a series of initial execution, which was labeled as the baseline by experts, and executions with reconfiguration schemes that changed the system runtime behavior. The baseline is defined as a test scenario with expected correct behavior due to the absence of execution changes during runtime. In monitoring, we obtained execution metrics data and collected them into a data set for the evaluation with principal component analysis (PCA).

A. Metrics and Data Collection

For the evaluation, the PCA allows us to select arbitrary metrics. We introduced metrics selection from generic metrics in various application domains, and business metrics, which are systems-specific metrics. With PCA, we processed the metrics data and produced data points in the PCA space. The correlations of each data point were regarded as the metamorphic relations (MRs), which help to determine the system performance correctness. We chose the generic and business metrics as follows (the bold names inside parentheses refer to the column names in table II).

1) Number of HTTP error codes (**err**). We obtained and counted the total number of 400 error codes for showing which reconfigurations produce invalid configurations. The number of error codes is one of the business metrics, representing the failures in a media streaming execution.
2) CPU usage. In container application, each service has CPU usage which denotes its workload. We measured the average (**cpu_avg**) and standard deviation (**cpu_std**) of the CPU usage in percentage.
3) Sent traffic. Requests from clients were sent to the video server and subsequently were routed to the auth server for access management mechanism. The sent traffic also included video data. We measured the average (**sent_avg**) and standard deviation (**sent_std**) of the sent traffic in megabyte per thirty seconds.
4) Received traffic. Being authorized, clients received the responses of valid access from the auth server. The received traffic also included video data. We measured the average (**rcvd_avg**) and standard deviation (**rcvd_std**) of the received traffic in megabyte per thirty seconds.
5) Memory usage. Memory denotes the memory consumption of the container application used by the service. We measured the average (**mem_avg**) and standard deviation (**mem_std**) of the memory usage in megabyte (MB).
6) Uptime (**uptime**). We counted the uptime from the first client requested media streaming until the last client closed the streaming. During the uptime, media streaming processing and communication occur, such as token exchange, media data delivery, and error message exchange. We observed the uptime per

Table II
PCA DATA SET OF EXECUTION SCENARIOS

Execution scenario	err	cpu_avg	cpu_std	sent_avg	sent_std	rcvd_avg	rcvd_std	mem_avg	mem_std	uptime
baseline	0	6.216	2.291	8.173	3.308	7.998	3.233	126.923	6.191	13
baseline, *kcRestart*	50	2.683	3.383	2.723	4.573	2.657	4.496	124.980	4.033	13
baseline, *wrongUri*	0	7.194	2.259	9.377	3.472	9.182	3.400	130.037	5.355	13
baseline, *webOrigin*	2152	4.111	2.123	3.405	3.443	3.330	3.383	128.050	5.683	13
baseline, *revokeToken*	50	3.585	3.720	3.893	5.059	3.807	4.977	128.929	5.578	12

Note: We prune this table, only showing one entry/test case per execution scenario.

Table III
THE NUMBER OF TEST CASES PER EXECUTION SCENARIO

Execution scenario	Number of test cases
baseline	30
baseline, *kcRestart*	3
baseline, *UriRedirect*	3
baseline, *webOrigin*	3
baseline, *revokeToken*	3

minute. The uptime is another business metric in our evaluation.

We collected data from the above metrics into a two-dimensional matrix, which then formed our data set, with **features** indicating the metrics in column, and **components** denoting our test cases in row. This matrix is shown in table II.

We tested multiple execution times in test cases, which is defined as a unique system execution and can be performed multiple times per execution scenario (forming a series of test cases). The purpose of multiple test cases is to obtain data consistency, which formed a historical data series of each execution scenario. The historical data builds a behavioral execution trend for the comparison with other execution scenarios. Table III shows the number of test cases in each execution scenario. In the table, we executed especially the baseline executions more in quantity to build data confidence for the basis of comparison with execution with reconfiguration schemes.

B. Reconfiguration Schemes

Reconfigurations aimed to cause intentional system disruption during runtime. To achieve this goal, we triggered each reconfiguration in the SUT at the time the last client has started the streaming until the system runtime has ended. The following explains the types of reconfiguration in our experiment and their behavior.

1) restarting auth server (*kcRestart*). We restarted the server when all clients had run the streaming and observed that all clients could not manage token expiration when the access token must be refreshed at a particular interval. The clients failed to receive streaming data from the video server. The number of error messages was equal to the number of clients.
2) misredirecting URI (*UriRedirect*). A valid redirect URI is a scheme in the auth server that allows the server to redirect responses to the requested application after successful login. In baseline execution, we set this value with a wildcard (*) and intentionally reconfigured it with a false URI at runtime to disrupt the redirection scheme. Although valid responses from the video server were missing, the clients retrieved the video information from its cache, allowing normal streaming.
3) removing web origin value (*webOrigin*). We removed the web origin wildcard and obtained error messages by this reconfiguration. We observed this since the video server failed to refresh the access token in the auth server. We also noticed this produced a great number of error messages compared to other reconfigurations, although it only stopped the media streaming and did not crash the video player instance.
4) revoking token (*revokeToken*). The auth server periodically sent two kinds of token: access and refresh token to maintain valid authentication once clients were authorized. We destroyed this valid authentication by revoking the token and observed failures in the token exchange mechanism.

Table IV summarizes our observation in executing reconfiguration scheme, including the categorization of valid or invalid reconfiguration referring to section III-A.

C. Results & Discussion

1) Results: We evaluated our implementation using the PCA method. The PCA result is shown in figure 4. Data points scatter and have distance among each of them in a PCA space in the figure. In most experimentations, any execution contexts share similar output if they have similar actions, triggers applied to them in the experiment. Specific to our experimentation, they follow a trend in performance behavior in the PCA space and form a cluster. The differentiator can be seen among different execution scenarios; They indicate different trends, thus classifying themselves in the space and constituting a cluster. If they are far from the trend of the dominant execution due to different reconfigurations, they are considered as an outlier.

The baseline execution is the major trend and the data points from its test cases form a cluster (figure 4, green). Other execution scenarios in the experiment with reconfigurations, in this case *kcRestart* (blue), *webOrigin* (purple),

Table IV
THE OBSERVATION SUMMARY OF EXECUTION SCENARIOS

Execution scenario	Description	Observation	Output category (section III-A)
Baseline	Running normal exection	No impact, normal streaming delivery	Valid reconfiguration
baseline, *kcRestart*	Restarting auth server	- Access token is expired at clients node - Application crashes, re-login is required	Invalid reconfiguration
baseline, *UriRedirect*	Misredirecting URI from auth server to video server	- Normal streaming delivery - The clients retrieve the video information from its cache	Valid reconfiguration
baseline, *webOrigin*	Removing webOrigin wildcard value	- Response from valid URI redirection is not permitted - Failed CORS request	Invalid reconfiguration
baseline, *revokeToken*	Revoking access and refresh token	- Access and refresh token are missing - Authentication is no longer valid for the clients	Invalid reconfiguration

and *revokeToken* (magenta), are scattered and some are seen as outliers in the PCA space. We call these outliers as invalid reconfigurations. These outliers may affect the system performance due to its deviation found in their metrics value. With this information of outliers, we argue that obtaining the decision of invalid reconfigurations can help to notify the system developers and managers for potential weaknesses of the systems as discovered by our proposed approach with PCA analysis.

However, we also find a particular reconfiguration, *UriRedirect* (brown), that does not have errors during execution yet has similar behavior to the baseline execution. This kind of reconfiguration produces a small principal component correlation output (by the PCA) and locates close to the baseline. We categorize this phenomenon as valid reconfigurations (along with the baseline execution). Yet, this may become a seemingly-fine execution with subtle potential to fail the system as it may have hidden, suspected deviations in the performance. For system developers and managers, we report this phenomenon as a warning for potential system weaknesses. Having our PCA result, we proceeded to measure each data point correlation in the PCA space by calculating the distances.

To validate the determination of valid and invalid reconfigurations, we calculated the euclidean distance between each data point in the PCA space and the baseline cluster centroid. In the view of metamorphic testing, this distance measurement identifies the MRs for our built oracle (figure 1). Any data point that does not follow the trend of the baseline execution was suspected as the outliers. We show the results of our outliers detection method in figure 5. We set the threshold of 2.71 in the y-axis, following the 97.5%-Quantile Q of the Chi-square distribution commonly used in the literature [8], [15], [16]. Modifications in this threshold value may change the determination of the reconfigurations validity; The lower the value, the more data points will be detected as the outliers. In figure 1, data points number 31-33 and 37-42 is considered as the outliers. They confirm that such points belong to the execution scenarios of the baseline execution with *kcRestart*, *webOrigin*, and *revokeToken* reconfigurations (figure 4). By this calculation, we also confirm that the baseline execution with *UriRedirect* reconfiguration (data points 34-36 in figure 1) has the correct behavior, being close to the baseline scenario (figure 4), thus regarded as valid reconfiguration.

2) Discussion: The phenomena of data points scatteration after PCA show that each system execution assembles into clusters and differs based on the execution scenario applied to them. There were observable patterns in which we remarked four types of relational patterns (as symbolized by the MRs in metamorphic testing) among different phenomena:

1) baseline and all reconfigurations. In this relational pattern, the baseline serves as the guideline and comparator of the system execution to each reconfiguration. In this pattern, we disregarded the decision of valid and invalid reconfigurations.
2) baseline and invalid reconfigurations. In this pattern, invalid reconfigurations were identified as the system misconfigurations. They were seen as outliers and informed an explicit warning for the experts and developers about system weaknesses that can fail the system execution.
3) baseline and valid reconfigurations. Valid reconfigurations showed delicate performance and can be reported to the system developers for potential failures that can be hard to manage in the execution and may fail the system.
4) valid and invalid reconfigurations. Both phenomena should be marked as potential system weaknesses, regardless of the severity of the system failures.

The combination of metamorphic testing and multivariate data learning method effectively detects potential weaknesses of service reconfigurations due to its absences of a priori oracle requirement and flexible metrics data selection. By our approach, the four relational patterns are adequate to discover misconfigurations for further reports and analyses by cloud business experts.

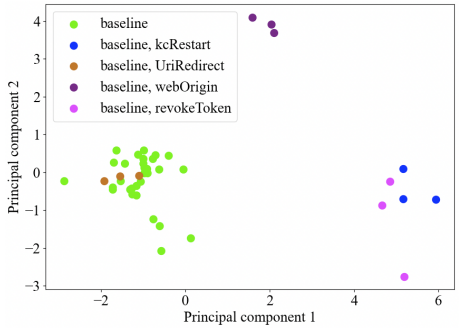

Figure 4. The data points (from test cases) of each execution scenario in a PCA space: baseline execution (the green points) and various reconfiguration scheme.

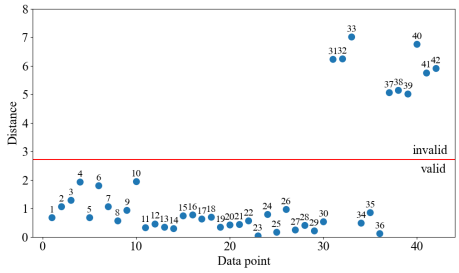

Figure 5. The euclidean distance plot of each data point to the centroid of baseline execution and the threshold which separates valid and invalid reconfiguration.

VI. Related Works

Reconfiguration Verification

Large-scale data centers and cloud computing have turned system configuration into a challenging problem. Several widely-publicized outages have been blamed not only on software bugs but also on configuration bugs. To face this challenge, Shambaugh *et al.* [17] provide a verification tool for Puppet configurations (DSL to describe cloud configurations). In [4], Duran *et al.* propose a decentralized and robust protocol for runtime reconfiguration of cloud applications with failures. The proposed protocol focuses on VM instantiation and destruction as well as component start-up and shutdown. In the same trend, Jarraya *et al.* [18] provide a formal framework for the specification of virtual machine migration and its security policy updates.

These three approaches represent so-called white-box techniques [19] for which the evolution of a configuration has a precise semantics in the meaning of programming language and logic. In this paper, we address reconfigurations with a finer granularity (for example, version changes of a third party library) of which the semantics are not defined. Therefore, we use so-called black-box testing techniques to validate the correctness of a reconfiguration.

Closer to our proposal, ConfAdvisor [20] provides a performance-centric configuration tuning framework for containers on Kubernetes. Parts of this framework can be used to implement automatic configuration exploration using our approach to pre-detect invalid configuration. Yet, as our proposed approach does not match with the implementation scenario in Kubernetes environment where oracles can be found in advanced, we opt for our approach with metamorphic testing and PCA in GNS3 cloud environment.

Metamorphic Testing

As discussed in the introduction, the core of our approach is to investigate the possibility of building a metamorphic relationship following a reconfiguration that does not change the business logic of a cloud-native service by looking at a set set of generic and business metrics. A recent survey on metamorphic testing [21] highlights that this approach is used in lots of domains [22], [23] and can be applied when we can define a clear metamorphic relation between inputs and outputs [24].

In [25], *Chen et al.* discuss the next challenges for metamorphic testing, which is to identify and select systematic metamorphic relations. The authors identified three opportunities that match the context of our approaches (cloud, big data, and agile development). Based on their initial idea, this paper proposes a mechanism to benefit from learning metamorphic relations between the correct behavior of services and these monitoring data after reconfigurations, from agile development and a vast volume of monitoring data production services. In this paper, the metamorphic testing approach combines the PCA to provide metamorphic relations that are useful for determining the correctness or validity of system executions.

VII. Conclusion & Future Works

This paper proposes a misconfiguration discovery technique of cloud-native services using PCA on monitored service data. Our approach was motivated by metamorphic testing that removes a priori oracle definition when passing functional tests of cloud-native services execution for discovering the service misconfiguration. Instead, we created our oracle using data point correlations in the PCA space from the arbitrary selection of generic and business metrics. Applying the proposed approach to a distributed media streaming application shows that the approach effectively detects error-prone reconfigurations through scenario execution without any associated oracle definition. After the PCA method and the euclidean distance analysis, business expert judgments help to validate the correctness of particular reconfiguration and understand the complex nature of cloud-native systems.

For future works, we keep working on implementing several efficient heuristics to detect misconfigurations that do not impact a priori the business logic of the application. The simple heuristic proposed in this paper is relevant, but it is necessary to compare several heuristics to detect these

specific changes. Finding a useful heuristic is a way to include the approach in a continuous integration pipeline. We are also interested to not only change the system execution with reconfigurations, but also with network failure injections, for example putting link latency, adding packet losses, or changing link bandwidth. Finally, we are keen to obtain the ability to re-execute workflows on updates of services in a test environment to directly exploit the monitored metrics in production.

Acknowledgment

This work was done in the framework of the joint research lab between Inria and Nokia Bell Labs on Smart, Automated Programmable Infrastructures for End-to-end Network Services (SAPIENS).

References

[1] N. Kratzke and P.-C. Quint. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journ. of Systems & Software*, 126:1–16, 2017.

[2] S. Wang, F. Du, X. Li, Y. Li, and X. Han. Research on dynamic reconfiguration technology of cloud computing virtual services. In *2011 IEEE Int. Conf. on Cloud Computing and Intelligence Systems*, pages 348–352. IEEE, sep 2011.

[3] L. Assuncao and J. C. Cunha. Dynamic Workflow Reconfigurations for Recovering from Faulty Cloud Services. In *IEEE Int. Conf. on Cloud Computing Technology and Science*, pages 88–95, dec 2013.

[4] F. Durán and G. Salaün. Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122:524 – 537, 2016.

[5] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo. Software Configuration Engineering in Practice Interviews, Survey, and Systematic Literature Review. *IEEE Transactions on Software Engineering*, 46(6):646–673, jun 2020.

[6] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry. A snowballing literature study on test amplification. *Journ. of Systems & Software*, 157:110398, nov 2019.

[7] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

[8] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry. Leveraging metamorphic testing to automatically detect inconsistencies in code generator families. *Software Testing, Verification and Reliability*, 30(1), jan 2020.

[9] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[10] I. T. Jollife and J. Cadima. Principal component analysis: A review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065), 2016.

[11] K. S. Lew, T. S. Dillon, and K. E. Forward. Software Complexity and Its Impact on Software Reliability. *IEEE Trans. on Software Engineering*, 14(11):1645–1655, 1988.

[12] R. Bruschi, F. Davoli, P. Lago, A. Lombardo, C. Lombardo, C. Rametta, and G. Schembra. An SDN/NFV Platform for Personal Cloud Services. *IEEE Transactions on Network and Service Management*, 14(4):1143–1156, dec 2017.

[13] A. Alasaad, K. Shafiee, H. M. Behairy, and V. C.M. Leung. Innovative Schemes for Resource Allocation in the Cloud for Media Streaming Applications. *IEEE Transactions on Parallel and Distributed Systems*, 26(4):1021–1033, apr 2015.

[14] T. C. Thang, H. T. Le, A. T. Pham, and Y. M. Ro. An Evaluation of Bitrate Adaptation Methods for HTTP Live Streaming. *IEEE Journal on Selected Areas in Communications*, 32(4):693–705, apr 2014.

[15] D. P. Enot, W. Lin, M. Beckmann, D. Parker, D. P. Overy, and J. Draper. Preprocessing, classification modeling and feature selection using flow injection electrospray mass spectrometry metabolite fingerprint data. *Nature Protocols*, 3(3):446–470, 2008.

[16] M. Hubert, P. Rousseeuw, and T. Verdonck. Robust PCA for skewed data and its outlier map. *Computational Statistics and Data Analysis*, 53(6):2264–2274, 2009.

[17] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: A configuration verification tool for puppet. *SIGPLAN Not.*, 51(6):416–430, June 2016.

[18] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. Cloud calculus: Security verification in elastic cloud computing platform. In *2012 Int. Conf. on Collaboration Technologies and Systems (CTS)*, pages 447–454, 2012.

[19] S. Nidhra and J. Dondeti. Black box and white box testing techniques-a literature review. *Int. Journal of Embedded Systems and Applications*, 2(2):29–50, 2012.

[20] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam. Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. In *2019 IEEE Int. Conf. on Cloud Engineering (IC2E)*, pages 168–178, 2019.

[21] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Trans. on Software Engineering*, 42(9):805–824, 2016.

[22] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Trans. on Software Engineering*, 44(11):1083–1099, 2018.

[23] A. F. Donaldson and A. Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing*, MET '16, page 44–47, New York, NY, USA, 2016. Association for Computing Machinery.

[24] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *2012 12th Int. Conf. on Quality Software*, pages 59–68, 2012.

[25] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. 2018.