# Methods for Live Testing of Cloud Services

Oussama Jebbar, Ferhat Khendek, Maria Toeroe

# Methods for Live Testing of Cloud Services

Oussama Jebbar[✉1], Ferhat Khendek[1] and Maria Toeroe[2]

[1] Gina Cody School of Engineering and Computer Science, Concordia University, Montreal, Canada
ojebbar@encs.concordia.ca
ferhat.khendek@concordia.ca
[2] Ericsson Canada Inc., Montreal, Canada
maria.toeroe@ericsson.com

**Abstract.** Service providers use cloud to reduce the cost of their services and speed up their time to market. There is a huge gap between the environment where the services are developed and first tested, and the environment where they operate. Live testing is defined as testing a service in its production environment without causing any intolerable disruption to its usage. It is considered as one of the solutions to overcome the impact of the difference between the development environment and the production environment on the reliability of the test results. Test interferences are a major challenge for live testing as they may lead to the system mishandling the production traffic. Existing solutions to alleviate the risk of interferences have limited applicability. In this paper we propose a set of test methods, applicable in different situations, for live testing to improve the reliability of test results without causing any intolerable disturbance to the cloud services. To reduce the complexity of these test methods we define the concept of boundary environments and a set of coverage criteria to aim at during live testing.

**Keywords:** Live testing, Cloud, Test method, Boundary environment, Runtime configuration state

## 1 Introduction

Cloud is a commonly adopted paradigm due to the flexibility of resource provisioning and the improved resource utilization it enables. Whether it is dedicated (private cloud) or shared (public cloud), a cloud is considered a complex system due to its scale and the configurations involved in its setup and management. This raises the question of how the differences between the test environment (lab, staging, dev.) and the cloud as production environment may impact software engineering activities such as software testing. Indeed, testing in the test environment is not enough anymore. In [2] for instance, Google reports a 43 minutes outage of the compute engine service due to a bad configuration. Although the configuration was tested pre-deployment, unexpected errors manifested once it was deployed in production. [7] presents a similar yet more elaborated issue encountered by Microsoft Azure as a configuration and code change errors that were not detected prior to the deployment and which lead to an almost 3

hours outage of several services. These errors manifested as a result of the new code and configuration being exposed to a specific traffic pattern not known/considered pre-deployment. To reveal such scenarios, services hosted in clouds need to be re-tested in their production environments as 1) the multiple configurations involved in a cloud system lead to differences between the configurations used in the test environment and the ones deployed in production [3]; and, 2) clouds may be subject to unexpected scenarios (requests or traffic patterns) that may not have been covered by testing activities. Live testing, as we define it [24], is testing a service in its production environment without causing an intolerable disruption to its usage. Live testing becomes more challenging as the tolerable disruption constraints become more stringent as in the case of carrier grade services. Interferences between test traffic and production traffic are among the main challenges of live testing as they result in violations of one or more of the functional or non-functional requirements of the cloud service. Test interferences may be due to internal characteristics of the services being tested (statefulness, nature of interactions, etc.), or the shared resources among services hosted in the same environment. Test interferences can manifest at the level of the services being tested or at the level of other services that share resources with them. A concrete example of such cases can be found in [4] where Google describes how testing an experimental feature for Java and Go applications in the Google App Engine caused instances of Java, Go, and Python applications to malfunction for almost 20 minutes.

The countermeasures taken to alleviate the risk associated with test interferences are known as test isolations. Some test methods provide isolation by using only components that can be tested live, i.e. they incorporate specialized modules called Built-In Test modules (BITs) that provide isolation [12]. Other test methods rely on other mechanisms such as cloning (used in canary releases and gradual rollout [6] test methods), snapshot and restore, resource negotiation, or scheduling tests when interferences are less likely to happen [13]. Canary releases and gradual rollouts are two test methods that are commonly used for testing a system in production. They rely on the use of production traffic to test new versions of existing features. In addition to the limitation of not being able to address new features, the reliability of the test results they provide needs to be improved. [1] reports an incident in which a canary tested configuration turned out to be erroneous after being propagated to the rest of the system. The failure, according to [1], was due to the fact that the feature was canary tested in network locations where some of its scenarios were not executed. These un-covered scenarios have been executed in other network locations after the change was propagated system wide. Had the new feature been tested in all possible network locations, this problem could have been avoided. Similarly, [5] reports on an issue caused by a bug in a feature that was not detected using gradual rollout. Although we cannot confidently claim that this could have been avoided if the feature was tested in other locations under different conditions, we cannot rule it out as a possibility either.

In this paper, we propose a set of test methods, applicable under different conditions, for live testing. These methods offer more flexibility compared to the existing ones and help alleviate the risk of potential test interferences. Testing of cloud services can be very complex and resource/time consuming. Therefore, we define the concept of

boundary environment and different coverage criteria to reduce the cost of proper testing of cloud services and avoid situations such as in [1] and [5].
The rest of this paper is organized as follows. In Section 2 we provide the background knowledge. We present the test methods we propose in Section 3. We introduce the boundary environment concept and its related coverage criteria in Section 4 and discuss how they can be integrated with our test methods as well as with other methods. Before we conclude in Section 6, we review the related work in Section 5.

## 2 Background and definitions

Cloud service providers accommodate tenants with a varying range of requirements. To reduce the cost of their services, cloud service providers realize their systems using configurable software which can be configured differently to satisfy different requirements. Configurations can be of different types, *tenant configuration, application configuration,* or *deployment configuration*. Tenant requirements are realized using services. A *service* consists of one service instance or multiple service instances chained together to compose the service. A *service instance* is the workload which is assigned to a *single configured instance*. Service providers create service instances using configurations of all the aforementioned types. These configurations play various roles in tuning the behavior of the configurable software. *Tenant configurations* for instance are used to parametrize the service instances composing the service of a specific tenant. *Application configurations* are used to expose/refine features of the configurable software which may be required to be parameterized differently for different tenants. When instantiated, a configured instance yields a set of *components* which are providing the actual service instance. The number of such components, their locations, their interactions with components of other configured instances, the policies that govern the number of such components, etc. are aspects set using *deployment configurations*.
A *configured instance* may be deployed on several physical or virtual nodes, i.e. its *components* may be running on any of the nodes on which it is deployed. Such design is usually used for capacity and/or fault tolerance purposes. Therefore, at any moment of the system's lifespan, components of configured instances may be running on same nodes, on different nodes, bounded to the same or different components of another configured instance, etc. The locations of those components, their numbers per configured instance, and their binding information is called a *runtime configuration state*. The set of runtime configuration states in which a system can be depends on the system's configuration. This is also called the viability zone for self-adaptive systems [22]. When the system is in a given runtime configuration state, each component is located on a specific node, in a specific network, sharing that node with a set of components from other configured instances. The location information (node and network) and collocation information define the *environment* under which the component is actually serving. Therefore, a runtime configuration state is identified by the set of environments under which the components of the configured instances are serving when the system is in that runtime configuration state. Furthermore, we can also identify runtime configuration states by the environments under which the service instances that compose each

service are provided. For each service, such a combination of environments is called the *path* through which the service is provided. Note that for services that are composed of a single service instance the concept of path coincides with the concept of environment as there are no combinations of environments to consider at this level. As a result, the concept of path, as we define it, is not to be confused with path in white box testing which may refer to control flow path or data flow path. To evaluate the compliance of the services with the requirements, cloud service providers use test cases as needed. These test cases may involve one or more configured instances depending on the realization of the requirements the test case covers.

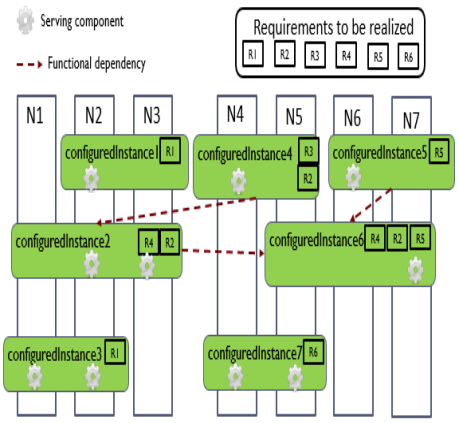


**Fig. 1.** Example of systems under consideration.

Fig. 1 depicts an example of a system we are interested in and addressing in this paper. Some requirements such as R3 are realized using services composed of a single service instance (assigned to ConfiguredInstance4). Other requirements such as R2 are realized through services that are compositions of more than one service instance (service instances assigned to ConfiguredInstance2, ConfiguredInstance4, and ConfiguredInstance6). The figure also captures a runtime configuration state where ConfiguredInstance2, ConfiguredInstance3, and ConfiguredInstance7 have two components running, while the rest of the configured instances have only one. These numbers of components can change as a result of configured instances scaling in or out. We can identify and describe the environment under which the component of ConfiguredInstance4, for instance, is serving as {location: N4, collocation: comps of {ConfiguredInstance7}}. We can also describe the paths taken by the service realizing R2, for instance, when the system is in this runtime configuration state. These paths are as follows:

- Path1: Component of ConfiguredInstance4: {location: N4, collocation: comps of {ConfiguredInstance7}}, Component of ConfiguredInstance2: {location: N2, collocation: comps of {ConfiguredInstance1, ConfiguredInstance3}}, and Component of ConfiguredInstance6: {location: N7, collocation: none}.
- Path2: Component of ConfiguredInstance4: {location: N4, collocation: comps of {ConfiguredInstance7}}, Component of ConfiguredInstance2: {location: N3, collocation: none}, and Component of ConfiguredInstance6: {location: N7, collocation: none}.

## 3 Test methods

This section describes the various live testing methods we propose to avoid the interferences between the test traffic and the production traffic of different conditions, i.e. configured instances with different characteristics.

### 3.1 Goals and assumptions

We aim for solutions that are capable of covering all runtime configuration states. Therefore, the coverage of runtime configuration states must be incorporated in our test methods. Furthermore, our solution should be independent of the configured instances under test (not assuming any capabilities of the configured instances under test); and, independent of the test cases being executed and the features being tested (like testing only features for which we only have production traffic). To meet these goals, we assume that the environment in which the configured instances run has the capability: 1) to snapshot the components of the configured instances composing the system. The snapshot image that is taken should be enough to clone these components. The environment has also the capability to clone a component from a snapshot; and, 2) to relocate service assignment from one component to another. These assumptions are aligned with the cloud paradigm. Due to containerization, snapshotting and cloning, for instance, can be done independent of the technologies used to realize the configured instances. Tools such as CRIU enable the snapshotting and cloning of processes running in various container technologies such as Docker [8] and LXC [9]. Furthermore, production like setups containerize even infrastructure services (kubelet and kubeproxy for Kubernetes [10], nova and neutron for openstack [11], for instance) which makes this assumption applicable also to infrastructure services. Service relocation is a feature also supported by cloud orchestrators. Such feature is usually needed to meet QoS requirements such as availability and service continuity. However, not all orchestrators may support service continuity.

### 3.2 Illustrative example

Fig. 2 shows a setup we will use to explain some concepts used in our test methods. It is composed of two configured instances which share three out of five nodes. The two configured instances share nodes N1, N2, N3, while nodes N4 and N5 can only be used by ConfiguredInstance1. At the time of the testing, ConfiguredInstance1 and ConfiguredInstance2 have two components each.

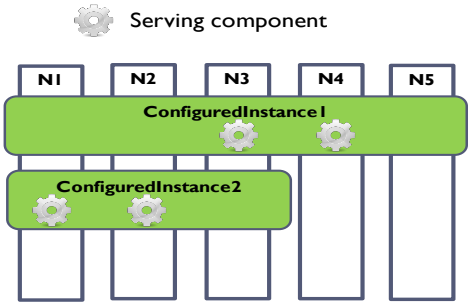


**Fig. 2.** Setup for the illustrative example.

To cover all runtime configuration states of the system, one should reproduce these states during testing. Therefore, during testing a component can be either a serving component, a component under test, or a test configuration component. A *serving component* is a component that handles production traffic. A *component under test* is a component that receives the test traffic. A *test configuration component* is a component, which is not under test, which receives a duplicated production traffic, but does not

handle it. Such components are used to recreate the runtime configuration state corresponding to the component being tested. The set of paths that need to be covered for testing a service composed of a single service instance provided by ConfiguredInstance2 includes but is not limited to:

- Path1: {location: N1, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Fig. 2, this will imply instantiating a test configuration component for ConfiguredInstance1 on N1, while having the component of ConfiguredInstance2 on N1 as the component under test.
- Path2: {location: N2, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Fig. 2, this will imply instantiating a test configuration component for ConfiguredInstance1 on N2, while having the component of ConfiguredInstance2 on N2 as the component under test.
- Path3: {location: N3, collocation: comps of {ConfiguredInstance1}}. If we start from the runtime configuration state illustrated in Fig. 2, this will imply instantiating a component of ConfiguredInstance2 on N3 as the component under test while ConfiguredInstance1 already has its component on N3 handling production traffic, so it is a serving component.

In the rest of this section we will describe various test methods that can be used to test configured instances in production while avoiding the risk of interferences. Note that these test methods are applicable at configured instance level and not at path level. That is, to test a service which is a composition of two or more service instances that are provided by different configured instances, we can use different test methods for the different configured instances on the path providing the service. Furthermore, we need to test the service for all the paths through which it can be provided.

### 3.3 Single step

The single step test method can be used for configured instances with no potential risk of interferences, i.e. when the testing activities have no impact on the configured instances behavior. Executing a test case for a configured instance using the single step

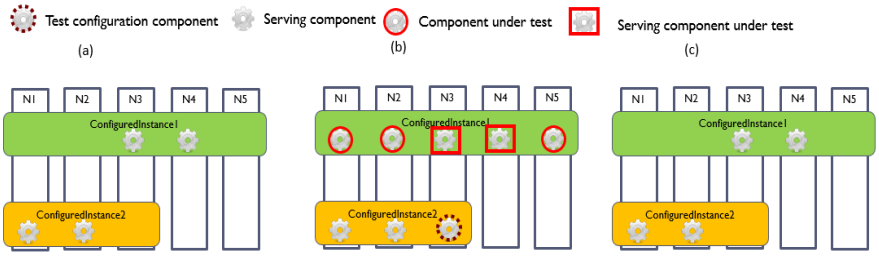


**Fig. 3.** Testing iteration for the single step test method applied to ConfiguredInstance1.

test method goes as follows:

1. Instantiate components under test to setup the paths that will be taken by the test traffic.

2. Instantiate test configuration components as needed to complete the creation of the environment under which the test case is to be executed.
3. Hit all the paths.

Fig. 3 illustrates the iteration for using the single step method to test a service composed of one service instance which is provided by ConfiguredInstance1. From the configuration of ConfiguredInstance1 one can deduce that the service can be provided through one of the eight paths (that means also eight environments as we are in the case of a service composed of a single service instance). As shown in Fig. 3, one can exercise five paths at a time using this test method. The figure shows five out of the eight paths tested. Since there is no potential risk of interferences, components of ConfiguredInstance1 are allowed to handle test traffic and production traffic at the same time. Therefore, test components are instantiated on an as-needed basis. Starting from the runtime configuration state in Fig. 3(a), in (b), for instance, for ConfiguredInstance1 we had to instantiate test components on nodes N1, N2, and N5, but for N3 and N4 we use the components which already handle production traffic, i.e. these components play two roles: serving component and component under test (marked as serving component under test). Similarly, the instantiation of test configuration components is not necessary as existing serving components help to create the environments under which one wants to test. However, in the absence of such components, one needs to instantiate test configuration components as it is the case for ConfigurationInstance2 on node N3 in Fig. 3(b). Fig. 3(c) shows the runtime configuration state after completing the single step test method. It is the same as in Fig. 3(a).

### 3.4 Rolling paths

The rolling paths test method can be used in situations where testing may impact the configured instance to be tested (therefore, serving components under test cannot be used), and the total cost of isolation mechanisms such as snapshot-and-clone and service-relocation is not too high in terms of causing intolerable disruption to the service. The rolling paths test method applied to a single configured instance as follows:

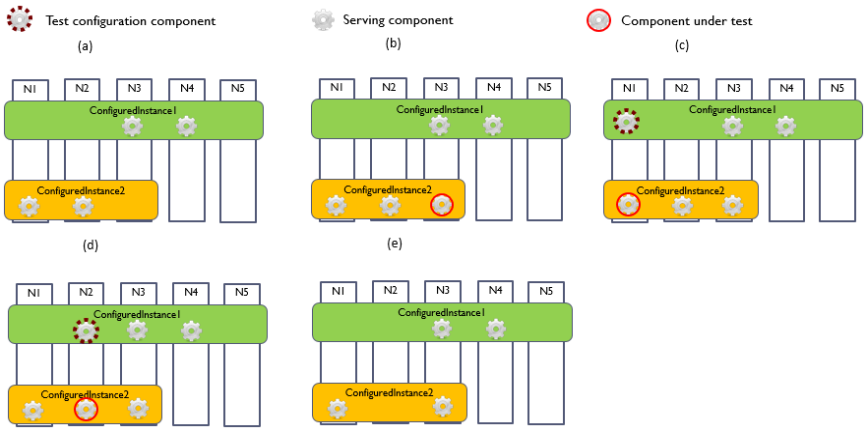


**Fig. 4.** Testing iterations for the rolling paths test method applied to ConfiguredInstance2.

1. Instantiate a component under test to create the path that will be taken by the test case.
2. Instantiate test configuration components as needed to setup the environments under which the test component should undergo testing.
3. Execute the test case on the created path.
4. Snapshot a serving component and replace the already tested component under test with a serving component cloned from the snapshot. Relocate the service to the new serving component.
5. Replace the snapshotted serving component with a component under test to create a new path to be tested.
6. Repeat from the second step until all the paths are exercised by the test case.

Fig. 4 illustrates a few iterations of the rolling paths in the case of a service composed of a single service instance provided by ConfiguredInstance2. There are six paths to be exercised by the test case. Because of the potential risk of interferences, test traffic has to be isolated from production traffic. As a result, unless there are enough resources, one cannot test multiple paths at the same time. A single path is created at each iteration by instantiating the necessary components under test and test configuration components as shown in Fig. 4(b), (c) and (d). After executing the test case in each iteration the service is relocated, and a next path is created for it to be exercised by the test case until all paths have been exercised. Fig. 4(e) shows that the runtime configuration state after the testing differs from the one in which the testing activity has started, i.e. Fig. 4(a).

### 3.5 Small flip

The small flip test method can be used to test configured instances when there is a risk of potential interferences; and, for which the number of currently required components is less than the number of unused nodes that can be used by this configured instance. For instance, the small flip can be used for a configured instance which can have components on six nodes and that, at the time of testing, requires only two components.

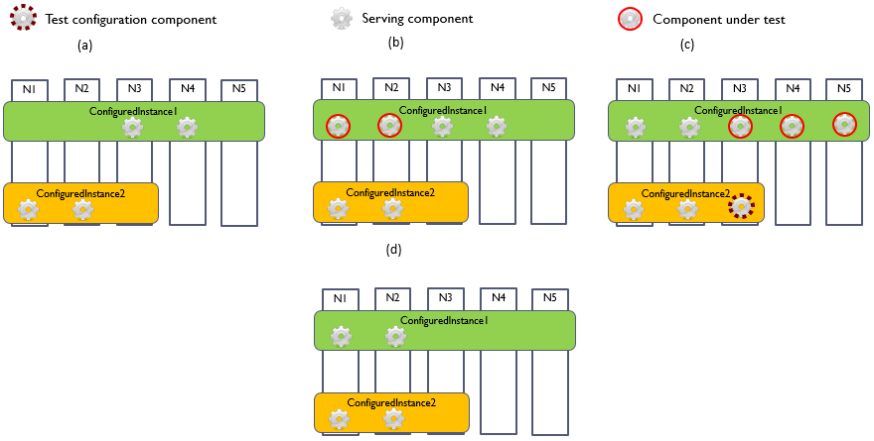


**Fig. 5.** Testing iterations for the small flip test method applied to ConfiguredInstance1.

Assuming we are testing a configured instance that, at the time of testing, requires k components out of the maximum n, where k <= n/2, the small flip test goes as follows:

1. Instantiate k components under test for the configured instance being tested on k available nodes.
2. Instantiate the test configuration components to create the test environments under which the k components under test are to be tested.
3. Test all the paths which can be exercised and which involve the k instantiated components under test until all the paths have been covered
4. Take snapshots of the k serving components, and replace the k components under test with k serving components cloned from the snapshots. Relocate the production traffic to the k new serving components.
5. Run the test cases for components on the rest of the nodes using the single step test method.

Fig. 5 shows the iterations of the small flip test method in the case of a service composed of a single service instance that is provided using ConfiguredInstance1. The small flip is used to test configured instances that present a potential risk of interferences, however, the resources available allow for exercising at least k paths in each iteration. In the first iteration shown in Fig. 5(b) two paths are exercised. The small flip results in one service relocation shown in Fig. 5(c) when used for only one configured instance involved in providing the service being tested. It shows the application of the single step test method to the remaining nodes. Fig. 5(d) shows the runtime configuration state after completion of the test. Note that the test of a service composed of more than one service instance and, therefore more than one configured instance (of the configured instances providing these service instances) is being tested using the small flip, may result in more than one service relocation for some configured instances.

### 3.6 Big flip

The big flip is the test method which induces the least service disruption and takes the shortest time. However, among the test method discussed here it has the highest resource consumption. The big flip test method of a configured instance goes as follows:

1. Create a new configured instance, the test configured instance, that has the same configuration as the configured instance to be tested. The components of the test

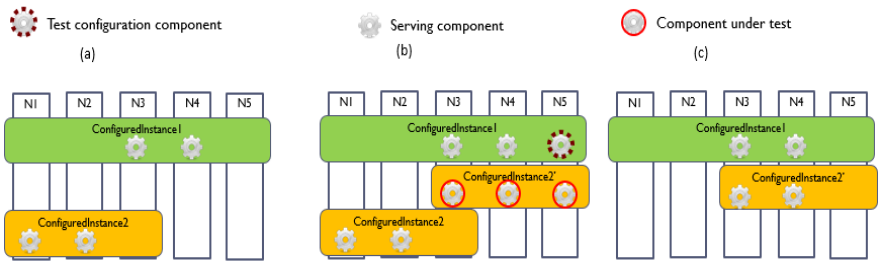


**Fig. 6.** Testing iterations for the big flip test method when applied to ConfiguredInstance2.

configured instance are the components under test, while the components of the configured instance to be tested remain as serving components during testing activities.
2. Create test configuration components required to test each path and run the test case.
3. After completing the tests, take a snapshot of the original configured instance to be tested, replace the test configured instance with an instance cloned from the snapshot, relocate the production traffic to the test configured instance (which will make it the configured instance with the new serving components); and remove the original configured instance from the system.

Fig. 6 shows how the big flip test method is used to test a service composed of one service instance provided using ConfiguredInstance2. As shown in Fig. 6(b), a new instance of ConfiguredInstance2 (ConfiguredInstance2') is instantiated on nodes that can host its environment. Such nodes are identified by taking into consideration information from the system configuration such as, software installed on each node, anti-affinity rules, etc. The new instance is then used to perform the tests under different paths (which are environments in this case as the service is composed of only one service instance). After the tests for all paths pass, the tested configured instance is replaced by one cloned from the original configured instance, which becomes the one with the serving components (service relocation from the original instance to the new one), and the original configured instance is terminated as shown in Fig. 6(c).

### 3.7 Compatibility between the test methods

The test methods described in this section apply to a single configured instance, however, a test case often involves more than one configured instance. It is possible to combine our proposed test methods throughout the path traversed by the test case keeping in mind: 1) The big flip method can be combined with any test methods; 2) the single step can only be used if throughout the paths being tested only the single step or the big flip are used; and, 3) The rolling paths and the small flip methods can be used together, and even combined with the big flip method.

## 4 Boundary environments and coverage criteria

A cloud service needs to be tested against all possible runtime configuration states of the system providing it. For cloud services we redefine the notion of "*test case passed*" to "*a test case passed only when it passed in all possible applicable runtime configuration states*". Testing a service against all its applicable runtime configuration states is necessary, however it is very costly. Let us consider a service composed of only one service instance protected by a configured instance where the components can be on any one of ten dedicated nodes (not shared with any other configured instance). For a normal functional test case, one will have to run this test case ten times to test this service against all its applicable runtime configuration states (once per node). In the case of a stress test case, and if we assume the maximum size of the configured instance is four components, the test case will need to be executed 210 times ($C_{10}^{4}$ times, without

considering the scaling steps). These numbers increase with the number of service instances. Thus, covering all the runtime configuration states may be impossible for complex and large systems such as cloud systems.

### 4.1 Boundary environments

To tackle the aforementioned problem we propose testing against a representative set of runtime configuration states. A runtime configuration state is described via the environments in which each service instance is provided; as a result, identifying the representative set of runtime configuration states consists of identifying the environments that describe the runtime configuration states in this set. Because any environment has two elements, i.e. location and collocation, such environments can be derived in two mutually non-exclusive ways:

- Collocation-wise: for a given set of locations on which a configured instance is deployed, one can identify the environments with the largest collocations per location which we call *boundary environments*. In other words, the collocation set of an environment that has that same location is a subset of the collocation set of the boundary environment. Two environments are said to have the same boundary environment if they have the same maximum collocation and equivalent locations, i.e. same network, and hosts of identical specifications.
- Location-wise: for a given set of collocations of components of a given configured instance and a maximum number of N components, one can identify various assignments of N collocations to N locations as allowed by the configuration. Such assignments are what we call *mixtures of environments*. Such assignments may or may not allow the reuse of collocations as the configuration allows. Two mixtures are said to have equivalent assignments if their assignments involve the same set of collocations in equivalent locations with the same numbers of occurrences of each collocation per location class.

By identifying the boundary environments in a system one can group the nodes of the system into groups that have the same boundary environment. Similarly, by identifying mixtures of environments one can group them into mixtures which involve the same set of collocations with the same number of occurrences of each collocation per location class. Our main idea is to use such groupings (location-wise and collocation-wise) to group runtime configuration states into equivalence classes taking into consideration the environments they involve. In other words, test runs should cover runtime configuration states that involve: 1) boundary environments, and 2) mixtures that were derived from collocations of boundary environments and that involve as many boundary environments as possible. The rationale behind this method is based on the following assumptions:

- Boundary environments present the worst case of resource sharing under which one can put the component under test; therefore, if a property holds under the boundary environment it will hold under all its sub-environments.

- Boundary environments allow for grouping nodes into equivalence classes. As a result, one node that replicates the boundary environment is considered representative of all the equivalent nodes that can host that boundary environment. This assumption enables us to reduce the number of paths, and as a result the number of runs the test case should go through.

### 4.2 Coverage criteria

Using the boundary environments, one can define the set of paths that should be exercised by the test case. Such set depends also on the nature of the test case itself. A functional test case will only need to target boundary environments of the configured instances. However, for stress test, for instance, one needs to use mixtures of boundary environments as well as check how the service behaves when these are chained with various mixtures of the other configured instances involved in the test case. Fig. 7 shows an example in which a service composed of two service instances that are provided (and protected) by two configured instances (ConfiguredInstance1 and ConfiguredInstance2) undergoing a stress test. Fig. 7(a) shows that ConfiguredInstance1 can have up to three components and ConfiguredInstance2 can have up to two components. If we assume all nodes are identical and are in the same network, ConfiguredIntsnace1 components may serve under environments with any of four collocations (for instance, collocation: comps of {}). Same applies to ConfiguredInstance2. Furthermore, ConfigureInstance1 has three different collocations for the boundary environments, namely one shared with only ConfiguredInstance2 (for example on N1), one shared with only ConfiguredInstance3 (for example on N4), and one shared with both (on N3). Similarly, ConfiguredInstance2 has two collocations for the boundary environments, one shared with only ConfiguredInstance1 and one shared with both ConfiguredInstance1 and ConfiguredInstance3. Fig. 7(b) to (g) capture some mixtures under which the service has to be tested (as it may be provided under these mixtures).
One can use various coverage criteria of the boundary environments as well as their mixtures in order to define the paths a test case has to exercise. Among these coverage criteria we believe the following are the most relevant. They are ordered in the descending order of their respective error detection power:

- *All boundary environments mixtures (resp. boundary environments) paths*: in this coverage one identifies first all possible mixtures of boundary environments (resp. boundary environments), then tests on all the paths that chain the mixtures (resp. boundary environments) of the configured instances being tested.
- *Pairwise boundary environments mixtures (resp. boundary environments)*: in this coverage one identifies all possible mixtures of boundary environments (resp. boundary environments), then generates a set of paths such as each pair of identified mixtures (resp. boundary environments) is in at least one path. To do so, one can generate a covering array of strength two for the identified mixtures (resp. boundary environments) considering each configured instance as a factor and each mixture (resp. boundary environments) of a configured instance as a level of the factor representing that configured instance.

- *All boundary environments mixtures (resp. boundary environments)*: in this coverage one aims at testing a set of paths in which each mixture of boundary environments (resp. boundary environments) is used at least once.

Based on the assumptions we made, one can use any of these criteria to reduce the number of runs of a test case while maintaining an acceptable level of error detection power. To run a functional test case against the service composed of the service instances provided by the configured instances in Fig. 7, one needs twenty runs for that test case to cover all runtime configuration states (without varying the location, taking into consideration the location one will end up with ninety six runs), as compared to six runs using all boundary environments paths and pairwise boundary environments coverage criteria, or three runs for all boundary environments criterion. Similar reduction in the number of runs can be achieved for test cases that may involve environments mixtures such as stress tests.

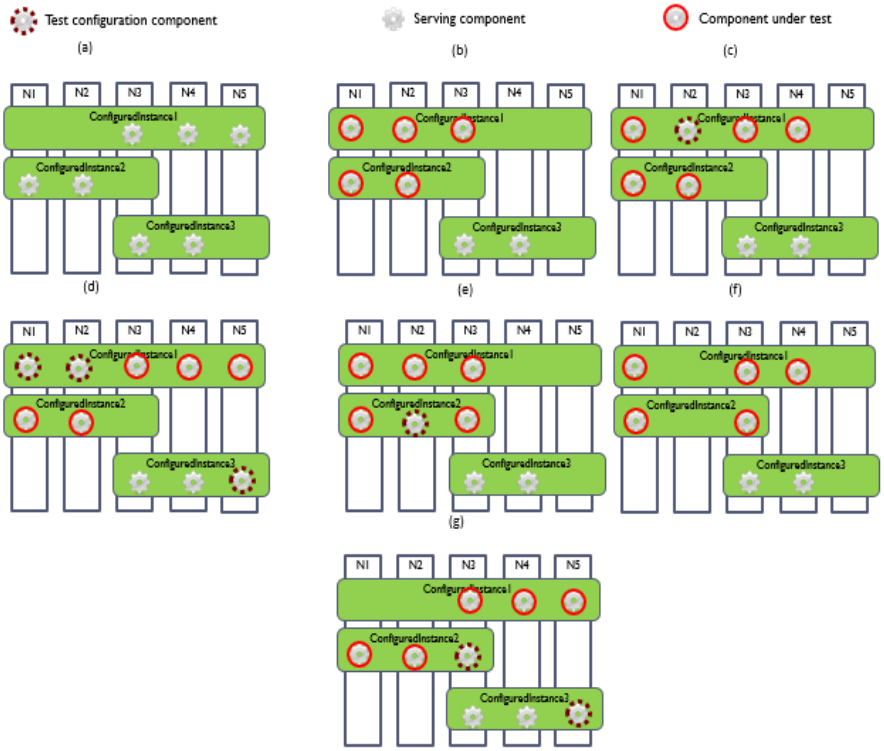


**Fig. 7.** Some mixtures and their combinations against which a service instance provided by ConfiguredInstance1 and ConfiguredInstance2 should be tested.

### 4.3 Revisiting the test methods

The test methods as described in Section 3 run a test case against all possible paths while reducing the impact of testing activities on the production traffic. We have shown

that this can be time consuming and may induce some intolerable service disruption due to service relocation for some methods. We proposed the concept of boundary environments along with the coverage criteria to reduce the number of paths under which a test case is to be run. Using these concepts the proposed test methods can be revisited to run the test cases only on paths generated given an environments-coverage criterion. Such enhancement help reduce the disruption induced by live testing on the one hand by reducing the number of runs of the test case regardless of which test method is used for isolation, which reduces the time needed for testing. On the other hand, by reducing the number of service relocations when the small flip or rolling paths is used which reduces the service disruption.
Other test methods can be enhanced as well. Canary releases can be enhanced by placing the components that expose the new version of the feature under test on nodes that represent different boundary environments. Furthermore, as the rollout is progressing, the placement should aim to cover relevant mixtures of boundary environments as new users are being redirected to the new version. Had such approach been used, the problem in [1] could have been detected earlier before retiring the older version and the damage could have been contained. The same applies to the gradual rollout method.

## 5 Related work

The author in [13] proposed a set of test methods to alleviate the risk associated with interferences. The main focus was to assess the runtime testability of the SUT provided a set of test methods that can provide isolation, and propose methods for test case selection that can balance between reducing the cost of runtime testability and the cost of runtime diagnosis. The authors in [12, 14] propose alleviating the risk of interferences using BITs to test a system in production. [15] proposes the use of the methods in [12, 13, 14] to avoid interferences. Moreover, [15] extended the TTCN Test System Reference Architecture [23] to orchestrate test case execution in production. Although this solution allows for the automation of test case execution and test configuration deployment, it remains limited to the use of TTCN as a language for test case specification in specific environments, namely OSGi managed JAVA systems. Canary releases [6], gradual rollouts [6] and dark launches [17] leverage the use of production traffic for testing purposes. In canary releases a new version of an existing feature is released to a subset of customers for a period of time, the new version will be released to all the customers only if it does not show any problem during the test period. Unlike canary releases, in gradual rollouts the percentage of customers using the new version increases in small steps, 5% for instance, until the new version takes over provided no problems are revealed while it is being tested. Dark launches consist of deploying the new version without releasing it to any customers. The production traffic is duplicated, and the behavior of the new version is compared to the behavior of the old version. If no problems are detected during the test period, the new version will be made visible to the users. As they rely on duplicating the production traffic, canary release, gradual roll outs, and dark launches have limited applicability as they can only be used to test new versions of existing features, i.e. they are not applicable for new features for which

there is no production traffic to duplicate. Same applies for methods such as Simplex [16] which is used for dependable live upgrade and testing of real time embedded systems. Blue-Green deployment technique [18] is used to enable zero downtime live upgrade and testing. It consists of maintaining two identical production environments, Blue and Green. One of them is used to handle the production traffic while the second remains idle. When it is time for an upgrade, the idle environment will be upgraded and tested. If the new setup passes all the tests the production traffic will be redirected to the idle environment and the active environment will become idle. The drawback of this approach is that it comes at a high cost resource wise as it poses the challenge of maintaining two environments in synch, which is not acceptable for large scale systems. In addition to testing, software upgrade is yet another management activity performed on live systems. The main challenge of live upgrade is the potential impact on the system's availability. Our test methods follow similar patterns as some state-of-the-art live upgrade methods. The rolling paths that we propose is similar to the rolling upgrade method [19] as this latter consists on rolling on batches of nodes (of a given size) and upgrade them iteratively in order to upgrade the system while maintaining service availability. Methods proposed to deal with software incompatibility include the split mode upgrade [20] and the delayed switch upgrade [21]. They consist of upgrading half of the nodes providing the service, redirecting the traffic to the upgraded nodes, then upgrade the rest of the nodes; which makes the principles used in the small flip method similar to the principles on which these methods are based. Blue/Green deployment [18] is also nothing more than an upgrade method, the parallel universe, used for testing activities. The gradual rollouts can also be considered as a rolling upgrade in which the tests are performed as the upgrade progresses.

## 6 Conclusion

Testing cloud services in the production environment has implications that cannot be addressed while testing in the development environment. On one hand, the complexity and the heterogeneity of the system should be taken into consideration in order to obtain reliable test results. On the other hand, one has to alleviate the risk of test interferences. In this paper we proposed a set of test methods, applicable in different situations, that will enable live testing of cloud services. We discussed how covering all the runtime configuration states of the system is costly and infeasible in real life systems. We proposed the concept of boundary environments and a set of coverage criteria that can reduce the cost of testing cloud services when combined with our proposed test methods as well as existing ones. As future work, we plan to validate with real case studies the methods and concepts introduced in this paper.

## Acknowledgement

This work has been partially supported by Natural Sciences and Engineering Research Council of Canada (NSERC) and Ericsson.

## References


1. https://status.cloud.google.com/incident/compute/16012. Last accessed June, 18th, 2020
2. https://status.cloud.google.com/incident/compute/15046. Last accessed June, 18th, 2020
3. https://status.cloud.google.com/incident/appengine/19001. Last accessed June, 18th, 2020
4. https://status.cloud.google.com/incident/appengine/16002. Last accessed June, 18th, 2020
5. https://status.cloud.google.com/incident/cloud-networking/18012. Last accessed June, 18th, 2020
6. D. G. Feitelson, E. Frachtenberg, K. L. Beck. Development and Deployment at Facebook. IEEE Internet Computing, 2013
7. November, 20th, 2019 incident. https://status.azure.com/en-us/status/history/. Last accessed June, 18th, 2020
8. Docker. https://www.docker.com. Last accessed June, 18th, 2020
9. Linux containers. https://www.linuxcontainers.org. Last accessed June, 18th, 2020
10. Kubernetes. https://www.kubernetes.io. Last accessed June, 18th, 2020
11. Openstack. https://www.openstack.org. Last accessed June, 18th, 2020
12. D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, D. Suliman. Reducing verification effort in component-based software engineering through built-in testing. Inf. Syst. Front. 9(2–3) (2007) 151–162
13. A. G. Sanchez. Cost Optimizations in Runtime Testing and Diagnosis. PhD Thesis, Delft University of Technology, September 2011
14. D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, R. Malaka. The MORABIT Approach to Runtime Component Testing. In the proceedings of the30th Annual International Computer Software and Applications Conference, COMPSAC'2006
15. M. Lahami, M. Krichen, M. Jmaiel. Safe and efficient runtime testing framework applied in dynamic and distributed systems. Science of Computer Programming, 2016, Volume 122, pp. 1-28
16. K. Lee, L. Sha. A Dependable Online Testing and Upgrade Architecture for Real-Time Embedded Systems. In the proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'2005
17. C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, R. Karl. Holistic Configuration Management at Facebook. In the proceedings of the 25th Symposium on Operating Systems Principles, SOSP '2015
18. Blue-green. https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html. Last accessed June, 18th, 2020
19. E. T. Roush. Cluster Rolling Upgrade Using Multiple Version Support. In the proceedings of IEEE International Conference on Cluster Computing, ICCC'2001
20. T. Das, E. T. Roush, P. Nandana. Quantum leap cluster upgrade. In the proceedings of the 2nd Bangalore Annual Compute Conference. COMPUTE'2009
21. X. Ouyang, B. Ding and H. Wang. Delayed Switch: Cloud service upgrade with low availability and capacity loss. In the proceedings of the 5th International Conference on Software Engineering and Service Science, ICSESS'2014
22. R. de Lemos et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science, 2013, Volume 7475
23. ETSI ES 201 873-5 v4.8.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part5: TTCN-3 Runtime Interface (TRI)

24. O. Jebbar, F. Khendek, M. Toeroe. Architecture for the Automation of Live Testing of Cloud Systems. In the proceedings of the 20th IEEE International Conference on Software Quality, Reliability, and Security, IEEE QRS ‘2020