Software Quality Journal (2020) 28:1147–1171
https://doi.org/10.1007/s11219-020-09505-2

RETORCH: an approach for resource-aware orchestration of end-to-end test cases

Cristian Augusto[1] · Jesús Morán[1] · Antonia Bertolino[2] · Claudio de la Riva[1] · Javier Tuya[1]

Published online: 2 March 2020

Abstract
Continuous integration practice mandates to continuously introduce incremental changes into code, but doing so may introduce new faults too. These faults could be detected automatically through regression testing, but this practice becomes prohibitive as the cost of executing the tests grows. This problem is preponderant in end-to-end testing where the whole system is requested for test execution. However, some of these test cases could be executed with fewer resources (e.g., memory, web services, computation, Cloud instances, among others), by deploying only the subsystems needed by each test. This paper is focused on the optimization of the resources employed in end-to-end testing by means of a resource-aware test orchestration technique in the context of continuous integration practices in the Cloud. The RETORCH approach proposes a novel way to identify the resources required by end-to-end test cases and to use this information to group together those tests requiring equivalent resources. Besides, the approach proposes to deploy the grouped tests in isolated and elastic environments, so that their execution can be scheduled in parallel on several machines. RETORCH is exemplified with a real-world application, and its performance evaluation shows promising savings in terms of resource usage and time.

Keywords Software testing · Continuous integration · Continuous testing · Testing in the Cloud · End-to-end testing · Test orchestration

1 Introduction

Continuous integration practices are based on incremental changes in the code to improve quality or add new functionalities (Meyer 2014). However, while introducing new features in the code, new faults can be introduced as well. Detection of these failures early in the code may reduce between 25 and 40% the amount of time and cost in fixing them compared to fix

✉ Cristian Augusto
augustocristian@uniovi.es

Extended author information available on the last page of the article

them in production (Shull et al. 2002). To ensure that the modifications and the new code do not endanger the existing functionality, regression testing (Yoo and Harman 2012) is standard practice. In modern agile processes, though, in which new versions of software are continuously and frequently delivered within very short cycles, regression testing may face many challenges as the efficient execution of the test cases, the reliability improvement of the tested system, or the reduction of time between different releases.

One emerging practice to shorten the validation of newly released versions is *continuous testing* (Fitzgerald and Stol 2017). Continuous testing consists of automating the test cases and re-executing them before any new release in the source code repository. However, a well-known problem is that as the number of tests increases, re-executing all of them at each change may not be possible due to the extent of resources that should be employed, such as the computational execution cost, the time required, or the number of instances needed. As a solution to partially address this problem, many test minimization and prioritization techniques (Yoo and Harman 2012) have been proposed, to identify a minimal subset of test cases or optimize their order of execution, respectively. The objective of these techniques is to look for a trade-off between the probability of discovering the faults potentially added with modifications and the resources employed for regression testing. The prioritization techniques permute the execution order of the test cases aimed to firstly execute the most relevant test cases, but the whole execution of the test suite remains expensive unless the tester decides to execute only a subset of the more relevant test cases through a minimization technique. The latter techniques reduce the execution time by not re-executing all test cases, but they neither optimize the resources of the test executed nor alleviate the thoroughly use of resources in the whole test suite.

One of the testing levels that requires a large amount of physical-logical-computational resources is end-to-end testing (from now onward referred to as E2E). E2E testing includes the interaction between system components, from the user interaction with the system to low-level layers like databases. The execution of the E2E test cases requires large amounts of resources due to the high execution time, the cost of replicating resources, or the setup of the system, among others. Therefore, the application of techniques such as prioritization minimization or reduction may not be effective enough for cost reduction in E2E testing.

During the execution of E2E test cases, the resources are usually oversubscribed because the test cases tend to deploy more resources than they need for execution. For example, suppose a user interface test case that requires a web and database server, but the setup of this test case deploys the whole system including an email server. As a consequence, the test case oversubscribes resources because the deployment of the whole system also includes an email server that this test will not use.

A proper setup of the whole system is relevant not only to optimize the resources deployed during E2E testing but also to decrease the execution time. Thus, if this setup requires a large amount of time compared with that employed in test execution, parallelizing the test cases in separate instances without a proper strategy would not solve the problem: for the test cases that share the usage of heavy resources, parallelization would be inefficient, and the best solution would be to setup the test environment once and execute them in a sequential way. Therefore, to optimize the cost of E2E testing, the detection of the dependencies between the test cases and the resources is a crucial aspect which may achieve cost savings (Herzig et al. 2015).

Moving testing to the Cloud (Bertolino et al. 2019) is commonly acknowledged as a solution to reduce the cost of testing, especially to exploit the potential of unlimited resources and scalability delivered on demand. One open-source platform to support Cloud testing and simplify the E2E testing process has been developed within the European Project ElasTest (Bertolino et al. 2018). The solution avoids several testing dependence problems by providing dependency isolation through the containerized execution of the tests. This is done through the TJobs that are the tests together with the Docker containerized system under test (SUT) customized to provide not only the production environment but also utilities to execute, monitor, and collect testing information.

Containerization has provided new advantages in the virtualization field, reducing the amount of both resources and time required to deploy a service in an isolated environment. The SUT instantiation can take advantage of the containerization in order to be deployed several times in the same machine, avoiding common problems like dependencies. However, in the current version, the ElasTest containerized execution presents the problem that it needs the instantiation of the resources required for each container causing oversubscribing (under usage) of those resources.

Our proposal is intended to reduce the number of resources used in the containerized execution of the test during E2E testing, and it may be integrated into the ElasTest platform to support resource-aware Cloud testing orchestration: we call the approach RETORCH (**Re**source-aware **E**2E **T**est **ORCH**estration). RETORCH aims at decreasing the execution time and optimizing the resources used in E2E testing through the identification of the resources used during testing and groups the tests based on the resources they need to avoid unnecessary redeployments and running of the identified test groups in parallel to reduce the execution time.

This article extends an earlier work (Augusto et al. 2019) by introducing a number of new concepts that are useful to identify the resources used by the test cases, a complete reorganization and extension of the state-of-the-art, and the application and evaluation of the approach on an ElasTest demonstrator. More precisely, this article includes the following contributions:

1. Definition of the RETORCH framework to perform the E2E resource identification, the test grouping, and scheduling.
2. An illustrative application scenario of RETORCH usage.
3. An evaluation of the RETORCH approach in a real-world application.

The remainder of the article is organized as follows. The related work is described in Section 2. The orchestration approach proposed in this article is defined in Section 3. Section 4 describes a working example related to a teaching online service (*FullTeaching* application using the *OpenVidu* Streaming Engine). Section 5 contains the evaluation of the approach proposed using a real-world application, and Section 6 describes the future implementation of the approach. Finally, the conclusions and future work are in Section 7.

2 Related work

An inspiring work to RETORCH is the Multi-Objective Regression Test Optimization approach (Harman 2011). In his work, Harman discussed several cost- and value-based objectives for testing, supporting the point of view that testing optimization should be performed by considering a combination of the several different types of resources needed. Our work is also focused on the same problem and proposes a specific solution for the case of end to end testing, considering a number of resources and the time spent during the testing. This section discusses different lines of orthogonal works that are related to RETORCH: (1) the reduction, prioritization and minimization techniques, (2) test dependency detection, (3) the resource optimization techniques, and finally (4) the orchestration approaches.

2.1 Test reduction, prioritization, and minimization techniques

Despite the recent advances in both efficiency and effective usage of resources during the testing, there are several open challenges (Bertolino 2007) to be addressed when performing test prioritization, selection, and minimization. Test reduction, prioritization, and minimization have been widely discussed in the literature. Yoo and Harman (Yoo and Harman 2012) have surveyed some works about minimization, prioritization, and selection, comparing the results of these techniques in terms of failure detection effectiveness and discussing open problems and future directions of them. Several authors have studied approaches to optimize these techniques considering both cost and rate of fault detection (Engström et al. 2008; Rothermel et al. 2002; Wong et al. 1998). These techniques are also used in big companies like Google (Memon et al. 2017) that executes a subset of the test cases according to both the failure rate and the historical number of modifications. Another line of research combines these approaches with other techniques, such as machine learning (Lachmann et al. 2017), prioritizing the test cases according to the requirements coverage, execution cost, and the historical number of failures detected by the test cases (Yoo and Harman 2012).

Our proposal has some aspects in common with the arrangement of the test cases of prioritization techniques (Yoo and Harman 2012).

2.2 Test dependency detection

During the test prioritization, one relevant issue to consider is the test dependencies. Some authors have proposed techniques and tools to detect these dependencies between test cases. Bell et al. (2015) provide a tool to detect dependencies (Electrictest) and compare it with other state-of-the-art tool achieving similar fault detection rate but with lower slowdown. The Electrictest tool was evolved by Gambi et al. (2018) and tested empirically achieving good results: they discover several dependencies previously known and also another one never discovered by the previous tests and tools. Gyori et al. (2015) have introduced the concept of the test pollution problem and present a technique (called POLDET) which was implemented into a tool that detects in execution time the "polluting" tests.

Our article proposes a framework to optimize the resources of the test executions avoiding unnecessary system redeployments by grouping those test cases that have no dependencies between them. The test dependencies play a key role to discover the relationships between the test cases and their resources. Our approach is intended to introduce a dependence detection mechanism to improve the test efficiency though the aggrupation of those test cases that they do not interfere with their execution.

2.3 Test resource optimization

The optimization of the test resources has been widely covered in the literature. Several works attempt to choose between different objectives as optimize the time, cost, or a mix of both

(Gambi et al. 2017) or optimize the resources in testing at the same time that they comply with time constraints (Liu et al. 2017). Other authors (Chakraborty and Shah 2011) have focused on the cost optimization, via different processes that partition, group, and redistribute the test cases in order to parallelize them. This aggrupation or partition of test cases is also present in clustering techniques applied to the test optimization problem (Yu et al. 2009), on which the resources are linked with the test cases in order to discover the underlying dependencies and execute them.

Unlike the previous works, our article proposes to optimize the resources not only focused on time or cost but also on other resources used during the testing based on both the test dependencies and resource usage. García et al. (2018) also propose to orchestrate the test cases through a proper selection and sequencing based on the outcome of test execution (verdict-driven) or on the produced output (data-driven).

In the field of Cloud services, several approaches have been proposed to face similar issues related to resource optimization. The Microsoft CloudBuild (Esfahani et al. 2016) faces dependences issues extracting dependency graphs and deriving the dependencies on them automatically. Based on these dependency analyses, Microsoft CloudBuild optimizes the testing execution through the deployment and execution of only the test cases that change the code. We propose a future line of work that aims at a similar automated detection of the test resources into the containers.

2.4 Orchestration

Depending on the field, the term orchestration has different meaning and usage. In general, orchestration is a jargon term that refers to the action of coordinating and scheduling several components improving their execution. In network field, Giotis et al. (2015) propose to orchestrate the network functions virtualization (NFV) with the goal of managing a policy-based traffic engineering. In the Cloud field, there are a number of orchestration systems, as for example, Kubernetes (Burns et al. 2016), Borg (Burns et al. 2016), Swarm (Docker Inc. 2019), Fuxi (Zhang et al. 2014), and System Center–Orchestrator (Microsoft n.d.). These previous orchestration systems are focused on Cloud following different architecture like TOSCA (Cloudify and Kubernetes) (Draft 2014).

The orchestration in Cloud is performed via an orchestrator (e.g., Docker Compose (Docker Inc. 2017)) that monitors and deploys the Jobs focused on optimizing the usage of the instances (Casalicchio 2017) or providing a determined QoS (Singh and Chana 2015).

Closely related to Cloud services, Fog and Edge computing also address several new open challenges related to resource optimization (Velasquez et al. 2018). These challenges are usually addressed via different architectures that present an orchestrator acting as both allocator and scheduler of the resources available (De Brito et al. 2017; Velasquez et al. 2017). Several of these works were analyzed according to the resource scheduling, allocation, sharing, or optimization by Toczé et al. (Toczé and Nadjm-Tehrani 2018) proposing a taxonomy of resource management in the Edge.

These works propose orchestration techniques to optimize the resources according to their specific domain. Instead, our approach aims to optimize the resources employed in the execution of the E2E testing, thus decreasing the execution time while achieving savings in the usage of resources.

3 RETORCH overview

The RETORCH framework aims at optimizing the cost/usage of resources orchestrating the E2E test cases in different machines based on the resources needed to execute each test. Figure 1 depicts the core concept of the orchestration starting from the E2E test cases to their execution in several machines/instances grouping those tests that use homogeneous resources in order to optimize both resources and execution time.

As the first step, resources used by each test case are identified to detect which test cases require the same resources (resource identification). According to the resources identified, some tests can be executed together while others cannot because of incompatibilities in their allocated resources or in the way in which these tests access the resources. Then, those test cases that can be executed together are grouped to arrange their execution and reuse their resources to optimize their cost (grouping). These groups of tests are called TGroups. Test cases that belong to different TGroups can be executed independently because the resources they employ are different. Finally, each TGroup may be split and allocated in several instances (scheduling) to optimize both the cost/usage of resources and the test execution time. The test cases of these TGroups are split into several disjoint subsets, which are smaller than the previous TGroups, which are called TJobs. Each TJob contains not only the code of a subset of test cases but also the environment with the dependencies isolated in a container that allows easy deployment of the test cases in a Cloud instance.

In the following subsections, the above key concepts are detailed. Subsection 3.1 details the resources and their attributes. These resources can be characterized according to their category (Subsection 3.1.1), the static attributes (Subsection 3.1.2), and the dynamic attributes that change during test execution (Subsection 3.1.3). Finally, the processes that orchestrate the E2E test cases are defined in Subsection 3.2.

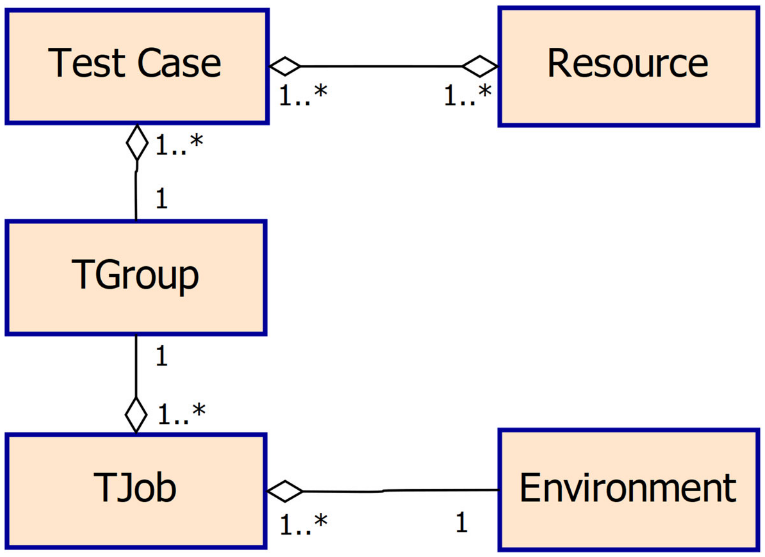


Fig. 1 Key concepts of RETORCH

3.1 Main concepts of RETORCH

The core of RETORCH is based on four main concepts defined below: the resources required by the test cases, the groups of these test cases that can reuse the same resources (TGroups), the disjoint subsets of these TGroups in which each subset can be allocated independently in parallel to decrease the execution time (**TJobs**), and an environment to isolate the dependencies to allow the scheduling in elastic Cloud instances.

- Resources are physical, logical, and/or computational entities required by the execution of one or more test cases. Examples of resources are databases with their tables, web servers, mobile phones, and services such as a payment gateway or a pool of containers provided by a Cloud carrier.
- TGroup (Test Group) is a set of test cases that use homogeneous resources and can be deployed together in the same environment. For example, a TGroup can contain the test cases that query the same database with the same initial load and without modifying the information. These test cases can use the same database setup in the same instance. In contrast, if two test cases modify the database causing side effects like flakiness or other issues related to dependencies, then these two test cases must be on different TGroups and therefore deployed in different environments. Each TGroup settles the environments needed by the test execution in the whole system or also considering scaffolding and test harness through the mocks, stubs, or other simulators that can alleviate the cost of resources that are not mainly needed for the tests of the TGroup. The test cases of the TGroup can be also divided to not only optimize the cost/usage of resources but also the execution time through a distributed scheduling.
- TJob (Test Job) is a subset of a TGroup containing several test cases inside a Docker container that also deploys the environment composed by the system under test isolating the dependencies and customized to provide utilities to execute, monitor, and collect testing information. The TGroup are split into several TJobs which are scheduled into different Cloud instances to reduce the execution time due to the parallel execution. Each TJob contains a subset of test cases that also reuse the same resources to optimize the resource usage avoiding unneeded redeployments of the environment.
- Environment is the set of resources requested and which they interact with a test case during testing. Examples of environments are a web application composed by a web server and a database or several mobile phones required to test an android application.

The following subsections define the categories of the resources (Subsection 3.1.1), their static attributes (Section 3.1.2), and the attributes relative to the usage of the resources during E2E testing (Subsection 3.1.3).

3.1.1 Resource categories

Resources are classified according to three different categories described below:

1) Physical: The tangible resources that are used during E2E testing. For example, the smartphones used to test a mobile application, the router that allows us to configure a virtual network into a system test, or other peripherals like printers and sensors, among others.

2) Logical: The non-tangible resources used during E2E testing that are provided in the traditional way. For example, a web server on which the application under test is deployed, the operative system on which the test is performed, or one flight emulator that simulates a plane in air traffic management (ATM).
3) Computational: The logical resources that are provided or consumed as a service. This category consists of those resources served on-demand in Cloud models like IAAS (infrastructure as a service), PAAS (platform as a service), or SAAS (software as a service), on which the computation, software, network, or storage may be provided as needed by the tester.

3.1.2 Static attributes

Regardless of the category, the resources are also characterized according to certain static attributes. These static attributes do not change during the E2E execution and provide additional information about the resource and how it can be used during the testing. We consider the following static attributes:

- Elasticity: A resource is elastic when it can be instantiated and made available for the test cases on the fly (e.g., a database running in a container, a software simulator). Conversely, a resource is not elastic when only a fixed maximum number of instances are available (e.g., a sensor, a camera, a hardware emulator).
- Hierarchy/partitioning: A resource may contain sub-resources or partitions that are also resources (e.g., a database may be partitioned into several tables or sets of tables). These sub-resources and partitions characterize the resources.
- Sharing: Shared resources may be used simultaneously by more than one test case without interfering into the test result.
- Replaceable: One resource is replaceable if may be interchangeable by a new instance (or another equivalent resource) with no penalty for a given test case. For example, one service that only provides tokens is replaceable if we can replace it by a simple mock that also provides continuously the same token for all requests in a similar way than the original resource.
- Life cycle: All resources have a life cycle composed of different phases like the setup of resource, test execution using the resource, and disposal of the resource. In the setup phase, the resources are deployed and initialized according to the test data (e.g., initial load of the database or configuration data). Once the resources are ready, the test cases use them during the test execution. Finally, after the test execution has finished, the resources are disposed and released, making them available for other test cases into a disposal phase.

For example, suppose the E2E testing of an air traffic management (ATM). When testing the operations that an air traffic controller makes to manage their assigned flights, we need a resource that is the control working position (CWP), which is itself a complex non-elastic physical system. The CWP may become a shared resource if we partition the flight area (a logical resource) into hierarchical clusters of sectors, provided that each test case will manage only flights belonging to a cluster. Moreover, when testing a transfer of flights between controllers will need two CWP, either exclusive or shared. The CWP is a not replaceable resource because it is a highly coupled-complex system that makes even more complicated to deploy partially its functionalities or change it by a mock. This resource also has his own life cycle, with a setup (prepare all the CWP and flight plans), a test phase, and finally a release and disposal.

3.1.3 Access mode and dynamic attributes

The resources can also be characterized according to their usage during the E2E testing considering how each test case accesses the resource (access mode) and how the resource changes due to the test execution.

Each test case uses the resources through different operations characterized by two properties: safety and idempotency. Safe operations are those whose execution does not modify the resource, for example, a SELECT or a JOIN operation in a database query because it does not change the information of the database and does not introduce dependencies between test cases. Idempotent operations are those that can be performed several times consecutively producing the same result.

Different test cases may have different usage patterns when using the same resource. Each pair of test case and resource is associated according to an access mode that determines if the operations performed during the test execution modify the resource or not and how. The access modes are enumerated below:

- Read-only: The test case performs both safe and idempotent operations allowing other test cases to read the resource at the same time (e.g., a test case that queries the master tables of a database without any change allows that other test cases query the same resource).
- Read-write: The test case performs operations that are neither safe nor idempotent. Then, other test cases may not use this resource simultaneously to avoid unexpected erroneous executions (e.g,. all half-duplex communication channel, on which two devices can emit or receive, but not at the same time in the same channel).
- Write-only: The test case performs operations that are neither safe nor idempotent similar to those “read-write” but allows that more than one test case update the resource simultaneously, restricting reads to only assertions that check the expected results (e.g., a centralized log system that acts as a sink for several test cases provided that, if we need to check the logs, there is a mechanism that allows identifying the logs produced by each test case).
- Dynamic: The test case performs operations that are safe but not idempotent. The resource is partitioned on the fly allowing that each test case creates and accesses each partition independently from other test cases (e.g., when testing several test cases that issue orders, more than one test can place an order at the same time, but in dynamic access, each test case must only use the orders that it has created).
- No access: this access mode is banally safe because the operations of the test case do not make use of the resource (e.g., when using a simple mock that does not require any resource).

The previous characterization provides insights about how the E2E tests use the resources. During the test execution, other attributes may change dynamically due to the usage of the resource. These attributes are called dynamic and are the following:

- Allocated: Location of each resource must be known to make possible their identification (e.g., the environment over where is deployed). Allocation is crucial when an effective use and measure of the resource performance during testing is considered.
- Measurable: Each resource must have indicators to allow measuring how many of them are deployed and their performance (e.g., RAM, processor usage or heartbeat latency received by a sensor network).
- Elasticity cost: The elasticity cost measures the expenses incurred during the resource life cycle. This cost may be a combination of money, time, processing power, memory, and energy, among others.
- Traceability: Each resource must be always traceable, allowing to know its state at every time of the test execution according to the life cycle (e.g., ready, running, disposing of, or testing over it).
- Test instance: The resources and test cases must be deployed in an instance that isolates the dependencies and avoids wrong executions/accesses with a properly setup.
- Availability: According to the number of instances available, resources are classified into renewable and nonrenewable. Resources are renewable if can be re-instantiated without any kind of limitations. On the other hand, a resource is nonrenewable when only may be instantiated a fixed amount of times.
- Granularity: Each resource has its own granularity depending on how the scope is focused over it and its underlying sub-resources. For example, one mobile phone that is used as a physical device for the testing may be considered with more granularity as a set of sub-resources (camera, microphone, screens, among others).

For example, in the previous scenario (ATM), the flight plans in an air traffic simulator are usually shared and renewable resources, because they are created on the fly as needed when the test is performed. On the other hand, the operation logs that are kept for legal requirements are a write-only resource because they do not use it for anything other than saving the different usage traceback.

3.2 Processes

RETORCH has three different processes, namely, resource identification, grouping, and scheduling. The resource identification provides insights about the resources required by the test cases and their dependencies. Next, the grouping is performed to group together those test cases that can be executed together to reuse resources. Finally, the scheduling optimizes the execution of the test cases providing a parallel schedule that reduces the test execution time. These processes are represented in Fig. 2 and described below:

Resource identification This process identifies the resources that each test case needs to be executed properly. To determine how the test case uses the resource, each association of a

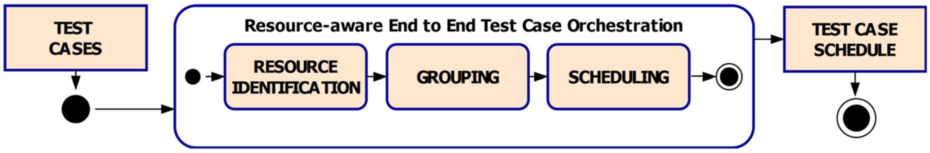


Fig. 2 Scheme of the main RETORCH processes

resource and test case is labeled with an access mode and the attributes (Subsection 3.1). With all this information, the test cases are characterized obtaining all the resources and their attributes, which will be the basis for grouping and scheduling the test cases.

Grouping This process aims at optimizing the usage of resources through an aggrupation of those test cases that can reuse the same resources to avoid unneeded redeployments. The test cases are arranged together into TGroups (Subsection 3.1) based on the compatibility of the attributes of the resources employed by these tests. The result of this grouping is a set of test cases together with all scaffolding required for the execution. The main goal of the grouping process is to avoid the oversubscription of the resources when one test case requests more resources than needed. For example, if two tests perform an operation with a safe access mode, they can be grouped together. However, if two test cases perform a non-safe operation on the same resource, they are candidate to be placed in the same or separate groups depending on the access mode.

Scheduling Although the grouping process achieves some optimization on resource usage, the whole test process may be further optimized by ordering and splitting the TGroups into a TJobs (Subsection 3.1). For instance, TJobs may be distributed in parallel to achieve better use of the test infrastructure and reduce the execution time. Not all schedules are aimed to minimize both execution time and the resource usage (one possible objective may be to maximize the usage of several instances, minimizing the idle time or another possible objective may be minimizing the execution time using more resources).

4 Working example

To illustrate RETORCH, we present an example of its application on a real-world open-source application called *FullTeaching* (Pérez 2017). *FullTeaching* is an educational platform that provides many features for organizing the teaching material, courses, and structuring classes; it provides also means for interacting with students, e.g., calendars, dashboards, and forums.

Resource identification The *FullTeaching* system is a resource that can be partitioned in hierarchical way by several sub-resources, including the *OpenVidu* videoconference server (University R. J. C. 2017), the *Kurento* media server (Technologies 2014), and the *MySQL* DBMS (Oracle 2019). In particular, for online teaching, *FullTeaching* includes features enabling real-time video conferencing that are supported by *OpenVidu* via W3C Web-RTC (Uberti and Thatcher 2018) open-source API. For testing the E2E functionality, the testers should consider the underlying infrastructure and the usage of resources, especially for the most expensive one (*OpenVidu*).

Deploying one instance of the *OpenVidu* resource per each test case that requires this resource is too expensive due to heavy resources for storage and graphical processing evolved in the video streaming. Despite we can group test cases to reuse the *OpenVidu* deployments, the *OpenVidu* resource is replaceable because it can be changed by simple mock in some test cases. For example, the test cases that only use the *OpenVidu* resource to acknowledge the connection, they do not need the full *OpenVidu* resource and can replace it by mock resource to be more efficient. Considering that the *OpenVidu* resource can be replaced based on how the test cases use this resource, we identify the following three replacements of the *OpenVidu* resource with different elasticity costs:

1. Light *OpenVidu* resource: This resource is a mock that just provides a random number as session-id, whenever any client requires it. Precisely, this resource has a no-access mode meaning that the requests from the test do not access the real *OpenVidu* resource, but a mock resource. This resource may be used by the test cases that only require the session-id from *OpenVidu*.
2. Medium *OpenVidu* resource: This resource is a simple implementation of the real *OpenVidu* resource, but with only basic functionalities and without any storage to record the session. This resource will be employed by the test cases that only need to check functionalities without storage like online chats between users or the navigation in the classroom menu.
3. Heavy *OpenVidu* resource: This resource provides all the functionalities of *OpenVidu* besides several video lessons recorded. This will be used in those test cases that require these video streaming recording functions or require all the functionality of the engine for their execution.

Once identified the previous three resources, we proceed to arrange all the test cases available depending on their resource usage requirements. Test cases assigned to a Light *OpenVidu* are the cheapest in term of elasticity cost: they can be available for testing on the fly and can be shared between multiple tests. The life cycle of this resource differs that it does not require additional setup or disposal, getting an improvement in terms of cost.

Test cases assigned to a Medium *OpenVidu* resource require the deployment of a simple container that consumes a small number of resources in terms of elasticity cost, and it allows sharing between multiple tests (although with some performance penalty). In this case, the setup/dispose life cycle phases are more expensive than the Light *OpenVidu* resource, so the aggrupation of the test cases (grouping) can reduce the usage of resources sharing this setup between several tests. Test on this resource has a read-write access mode.

Last, test cases assigned to the Heavy *OpenVidu* resource should be executed in a sequential way because they access to the resource in a read-write mode and the high elasticity cost that does not allow the deployment of more than one instance. This resource has this type of access mode because the test cases use the resource to create and modify videos at the same time. As a consequence, the test cases that use the Heavy *OpenVidu* resource should be executed sequentially to avoid issues due the concurrent access/modification of the same videos.

Grouping Once the resources are identified and characterized, we proceed to group these test cases into TGroups considering the test dependencies with the resources used. Figure 3. depicts the mapping between test cases and the groups (TGroup) that they belong to. The test cases that use the Light *OpenVidu* resource are represented in blue color, the test cases that use the Medium *OpenVidu* resource in red color, and the test cases that use the Heavy *OpenVidu* resource in black color.

Let us suppose that we have nine test cases and determine three TGroups as indicated below:

TGroup 1 (Light *OpenVidu*): test cases 1, 5, 8, and 9

TGroup 2 (Medium *OpenVidu*): test cases 2, 6, and 7

TGroup 3 (Heavy *OpenVidu*): test cases 3 and 4

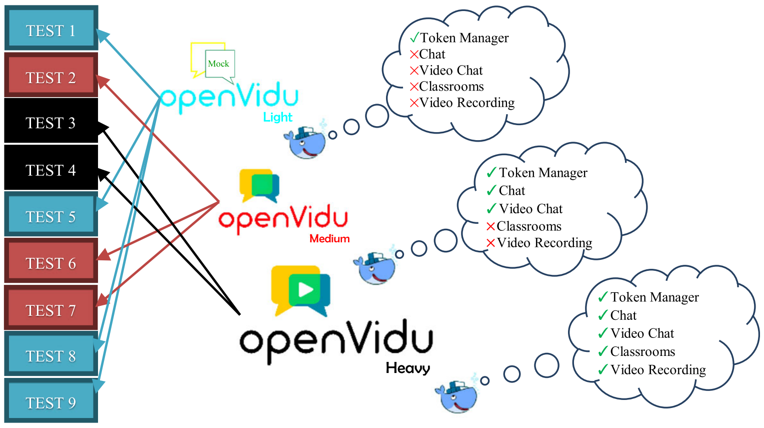


Fig. 3 Resource identification and grouping

The previous aggrupations can improve resource usage deploying minimal resources and avoiding unnecessary redeployments. For example, the TGroup 1 instead to deploy the *OpenVidu* resource just deploys a mock resource (Light *OpenVidu* resource) that is more efficient in terms of resource usage. However, there are test cases that require the *OpenVidu* resource like test cases 3 and 4, and they deploy this whole resource (Heavy *OpenVidu* resource), but they can be executed in the same group (TGroup 3) to deploy the resource one time and reuse again avoiding a new unneeded redeployment.

Scheduling Once the grouping is done, we divide the TGroups into TJobs to schedule them and optimize both objectives: the resource usage and execution time. Figure 4 represents four different schedules. The TJobs derived from TGroup 1 are represented in blue (Light), the TJobs of TGroup 2 in red (Medium), and the TJobs of TGroup 3 in black (Heavy):

4. Figure 4a only creates one TJob from each TGroup. In this schedule, the TJobs are executed in parallel over three instances, but the test cases of the same TJobs are executed sequentially. This schedule provides a baseline, giving the worst execution time, but using the minimal number of instances required to keep the TGroups isolated. All TGroups are deployed in separate instances sharing the same setup between them.
5. Figure 4b also creates one TJob from each TGroup. However, the test cases of the TJob 1 are deployed in parallel over the same instance because they use the Light *OpenVidu* resource that allows the parallel execution of the test cases at the same time. As consequence of the parallel execution of the test cases of TJob 1, the schedule reduces the execution time in comparison with the previous one that executes them sequentially (Fig. 4a). Each test case of the TJob 1 employs individually more execution time than by executing them sequentially due to the overload caused by the concurrent access. Nonetheless, in this case, it is not relevant because the critical execution time corresponds with the TJob 2 execution (TGroup 2).
6. Figure 4c as the opposite of TGroup 1, the execution time cannot be reduced executing the test cases of the TGroup 2 in parallel inside of the same instance because they cannot

access concurrently to the same instance of the Medium *OpenVidu* resource. However, the test cases can use this resource in parallel if it is deployed in several instances. Therefore the schedule of the Fig. 4c creates three TJobs from the TGroup 2 that are deployed in a parallel way in three instances. This schedule reduces the execution time, and the critical execution time corresponds with the TJob 3 execution (TGroup 3). However, the test cases of the TGroup 3 cannot be executed in parallel at the same time neither in the same instance nor in several instances because they use the Heavy *OpenVidu* resource that has high elasticity cost. This schedule reduces the execution time but increases the use of resources because employs five instances.

7. Figure 4d instead to create three TJobs from TGroup 2, it creates only two TJobs to reduce the number of instances. This schedule maintains the same execution time than the schedule of Fig. 4c but also reduces the resources employed avoiding the deployment of one more instance: the schedule of Fig. 4c deploys five instances (three Medium *OpenVidu* resources), whereas the schedule of Fig. 4d only four instances (two Medium *OpenVidu* resources).

As shown in this working example, there are several features and constraints considered during the optimization of the test scheduling based on resource usage through test orchestration. The critical step is the proper identification of which resources are needed by the test cases and their dependencies.

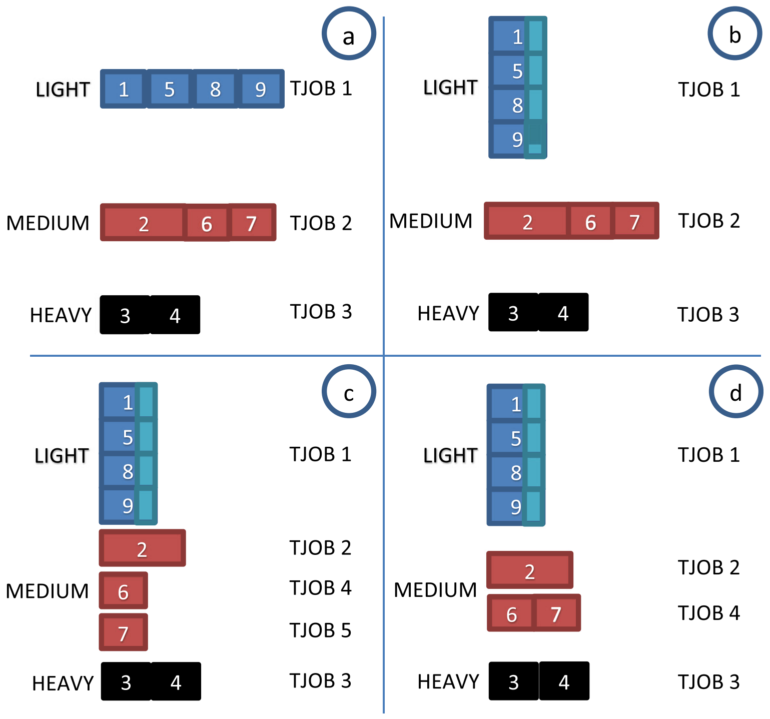


Fig. 4 Different proposed TJob Scheduling

5 Evaluation

In order to assess whether RETORCH improves the execution time and saves resources during E2E testing, we perform an empirical evaluation of the application described before. To evaluate how the resources and time may be optimized via a better distribution, we attempt to answer the following research questions:

RQ1: Does RETORCH yield an efficient execution of the E2E tests in terms of time?

RQ2: How do the schedules proposed by RETORCH affect to the use of other resources?

Test suite We tested the *FullTeaching* application with a test suite composed of 20 test cases available in different git repositories (ElasTest Developers Team 2017, 2018) of the *ElasTest/FullTeaching* community. These test cases employ JUnit and Selenium Web driver to emulate the user interactions checking the main functionalities of the application: classrooms, file uploading, and comment creation, among others.

Setup In order to evaluate RETORCH, the test cases are executed using up to 5 HyperV virtualized instances of *Ubuntu* Server 18 LTS into a *Ryzen* 8-core, 32 gigabytes of RAM, and solid-state drive computer. To analyze the efficiency of the test schedules provided by RETORCH, we measure the execution time and different memory indicators from the system monitor: physical memory required by each instance and total physical memory required.

Resource identification In the *FullTeaching* system, we identify three different resources as indicated in Section 4: *OpenVidu* videoconference server (University R. J. C. 2017), the *Kurento* media server (Technologies 2014), and the *MySQL* DBMS. According to the resource identification detailed in Section 4, the *OpenVidu* resource can be replaced by another three resources depending on how the test cases use the resource: Light *OpenVidu* resource (mock), Medium *OpenVidu* resource (implementation with basic functionalities), and Heavy *OpenVidu* resource (full *OpenVidu* functionalities). When it is possible, it is preferable to execute the test cases in the Light *OpenVidu* resource to save resources in comparison with both Medium and Heavy *OpenVidu* resources, and also because the last two resources have more elasticity cost. All the resources are allocated into a Cloud server by means of a Docker orchestrator (Docker compose) that also deploys the aggrupation of test cases according to the schedules proposed by RETORCH.

Grouping The test cases are grouped into TGroups based on the resources identified. As we detailed in Section 4, we create three TGroups based on the functionality of the test cases and their usage of the *OpenVidu* videoconference system. The TGroup 1 is composed of the test cases that only need a session-id from the *OpenVidu*; then these test cases can use the Light *OpenVidu* resource (mock). The TGroup 2 is composed by the test cases that need *OpenVidu* functionality without storage, and then they can use the Medium *OpenVidu* resource. Finally, the TGroup 3 is composed of the test cases that require the full functionality of the *OpenVidu* including the storage, then they can use the Heavy *OpenVidu* resource.

All of the test cases of the TGroup 1 use the Light OpenVidu resource with read-only access because they only request a session-id, so they should be executed sequentially/parallelized either in one instance or in several instances. In contrast, the test cases of the TGroup 2 use the Medium *OpenVidu* resource with read-write access because they modify information that other test cases can access if they are executed in the same instance. To avoid issues between test cases due the concurrent access, the test cases of the TGroup 2 should be executed either sequentially in the same instance or parallelized through different instances of the Medium *OpenVidu* resources. However, the test cases of the TGroup 2 should not be executed in parallel way in the same instance of the Medium *OpenVidu* resource.

Finally, all the test cases of TGroup 3 also use *OpenVidu* with read-write access, but in contrast, they use the Heavy *OpenVidu* resource because they need storage to modify and access to the data. To avoid issues between the test cases due the concurrent access, the test cases of the TGroup 3 should be executed sequentially in the same instance. However, the test cases of the TGroup 3 should not be executed in several instances of the Heavy *OpenVidu* resources because the resource has a high elastic cost.

Table 1 summarizes the number of test cases per each TGroup and their possible executions. The TGroup 1 has 5 test cases, TGroup 2 has 11 test cases, and the TGroup 3 has 4 test cases. The test cases of these TGroups will be executed in TJobs according to the scheduling.

Scheduling We have executed the four schedules that are detailed in Section 4 and represented in Fig. 4. The first schedule A creates one TJob per each TGroup, that means that the TJob 1 is created with all test cases of TGroup 1 (5 test cases), the TJob 2 with all of TGroup 2 (11 test cases), and the TJob 3 with all of TGroup 3 (4 test cases). The three TJobs of schedule A are executed in parallel each one in one instance. Despite the TJobs are executed in parallel, the schedule A proposes that the test cases of each TJob should be executed sequentially in the instance of the TJob. In contrast, schedule B proposes that the test cases of the TGroup 1 should be executed in the same TJob using one instance but the test cases in parallel. The schedule C proposes to parallelize the execution of the TGroup 2 in 3 TJobs executing each one in one instance. Finally, the schedule D, instead to parallelize the execution of the test cases of TGroup 2 in three instances, parallelizes them in two instances. Once these four schedules are executed, we answer the research questions analyzing the efficiency of the test execution: memory required by the virtualized instances and execution time of the test cases.

RQ1: To evaluate how RETORCH may optimize the execution time of the test suite, we have executed the schedules and obtained the execution time on the up to five virtualized instances. Figure 5 depicts the execution time of the E2E test cases with the four schedules represented in Fig. 4 (A, B, C, and D). The execution time is obtained from the test log timestamps. The blue lines represent the execution time of the TGroup 1, the red about TGroup 2, the black about the TGroup 3, and the green/white the total execution time of the test suite according to the schedule.

RETORCH reduces the execution time of the test suite by a 62% (from 149 s in the schedule A to 57 s in the schedule D). The schedule A employs 149 s because the test cases of TGroup 1 are executed sequentially during these 149 s. The schedule B execute these test

Table 1 *FullTeaching* test cases

No. of TGroup	Resource name	No of test cases	Access mode	Execution
1	Light *OpenVidu*	5	Read-only	Sequentially in one instance Parallel in one instance
2	Medium *OpenVidu*	11	Read-write	Sequentially in one instance Parallel in three instances Parallel in two instances
3	Heavy *OpenVidu*	4	Read-write	Sequentially in one instance

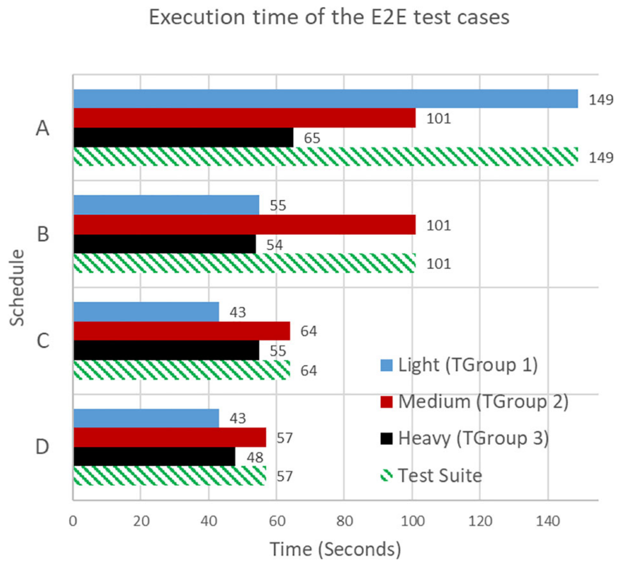


Fig. 5 Execution time different schedules

cases in parallel inside of the same instance reducing the execution time of the test cases of TGroup 1 to 55 s. However, the execution time of schedule B is 101 s because the sequential execution of the test cases of TGroup 2 takes 101 s. The schedule C executes the test cases of the TGroup 2 in parallel in three instances reducing the execution time to 64 s. Finally, schedule D executes the test cases of the TGroup 2 in two instances to optimize the resource usage employing 57 s. According to the *FullTeaching* system and the four schedules evaluated, RETORCH is able to reduce the execution time more than half through the identification of the resources, grouping of the test cases, and the sequential/parallel scheduling according to test dependencies.

RQ2: To evaluate the performance in terms of resource usage, we monitor the physical memory used in all the virtualized instances during the testing. This measurement was obtained via the *HyperV* performance monitor that provides the memory requested by each instance together with the percentage of memory used at each moment.

With these two values, we obtain the memory used by the different virtualized instances at every moment multiplying the percentage of use by the memory requested. The resource usage in the four schedules is represented in Fig. 6. The x-axis represents the execution time and y-axis the total amount of the memory used by all the instances in gigabytes. Note that the four schedules represented in the figure employ different execution times. Schedule A takes 149 s, but the schedule D finishes at 57 s. During these seconds, the memory usage varies in different ways, sometimes with peaks and other times flatter depending on the sequential/parallel execution of the test cases and instances. The schedule A executes the test cases in 3 instances, the schedule B also in 3 instances, the schedule C in 5 instances, and the schedule D in 4 instances.

RETORCH reduces the execution time executing the test cases in parallel either in one instance or several instances. The increasing of one instance also increases the memory usage, but RETORCH provides a schedule that at the same time that reduces the execution time also optimizes the usage of memory. We can observe from the schedule A to schedule D that

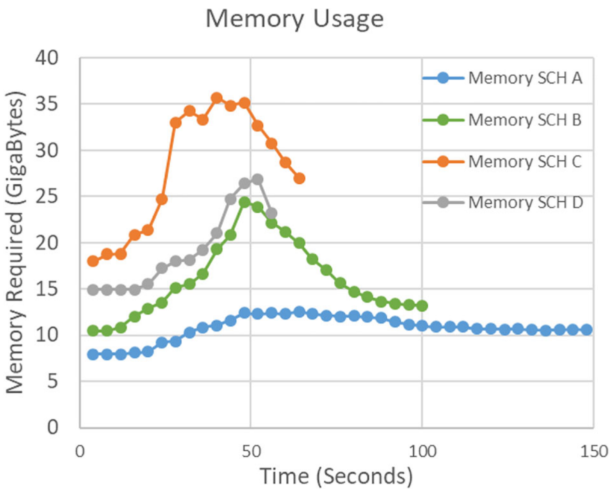


Fig. 6 Memory usage in the different schedules

RETORCH reduces the execution time by 62% (RQ1), whereas the memory usage only increases at ~× 2 (from ~ 8 to ~ 15 GB in the lower values and from ~ 12 to ~ 26 GB in the peak). Despite the memory increases, RETORCH optimizes the memory usage at the same time that aims to reduce the execution time.

The schedule A executes in parallel three TJobs in three instances employing ~ 15 GB. All test cases of each TJobs are executed sequentially according to the schedule A, so the memory usage is more or less flat (the lower value is ~ 8 GB, but usually is around ~ 15 GB). Schedule B instead to execute the test cases of TGroup 1 sequentially in one instance proposes to execute them in parallel in the one instance.

The parallel execution of these test cases not only reduces the execution time but also increases the memory achieving a peak of ~ 24 GB when the test cases of TGroup 1 are executed in parallel at the same time. Schedule C proposes to execute the test cases of TGroup 2 in parallel in three instances. The parallel execution of the test cases again decreases the execution time but also increases the memory achieving some peaks of ~ 36 GB. The memory has increased a lot because each instance of the three deployed for the test cases of TGroup 2 contains the Medium *OpenVidu* resource, *Kurento* media server resource, and *MySQL* database resource. To optimize the resources, the schedule D proposes to reduce one instance for the test cases of TGroup 2, that is, to execute two instances for the test cases of TGroup 2, one instance for TGroup 1, and another instance for TGroup 3. The reduction of one instance from schedule C to D not only improves the execution time but also reduces/optimizes the memory usage from ~ 18 to ~ 15 GB in lower values and from ~ 36 to ~ 26 GB during the peak. According to the *FullTeaching* system and the four schedules evaluated, RETORCH not only reduces the execution time of E2E testing through the sequential/parallel execution but also optimizes the resource usage varying the number of instances considering the resources deployed.

Threats to validity The above evaluation shows promising results of the RETORCH approach. However, there are several issues that may threaten the validity of these results. Regarding the internal threats, the evaluation analyzes the memory usage and execution time on which it is easy to introduce noise into the measures by other system processes. In order to mitigate this issue, we performed the experimentation into the same dedicated computer inside virtualized instances with the same specifications. Regarding the external threats that may limit the ability to generalize the results, our evaluation is related to only one case of study with a limited set of resources. Despite this, the results provide us insights that by carefully arranging the resources used by the test cases, the overall efficiency of the test execution may be improved, although more experimentation should be done with other systems and different kinds of resources. Another issue is related to the size of the test suite. Although the test suite is not large, it contains a variety of typical tests scenarios in E2E testing, which have been taken from a real-world application. Finally, regarding the construct validity, we handled the most representative variables (e.g., overall test execution time and memory consumption). Other measures have not been considered (e.g., processor load). To mitigate this problem, we have monitored the other resources, observing that the other resources remain with a low usage rate compared with the memory.

6 RETORCH implementation

The automatization of RETORCH faces several challenges in order to identify the testing resources, grouping the test cases considering their system/testing dependencies, and schedule them efficiently in several machines. The first challenge is related to deal with the complexity of the resource identification process and the dependencies between test cases resources. The detection of those dependencies and the problems related to the test cases has been widely treated in the literature in different ways (Gambi et al. 2018; Bell et al. 2015). In order to identify the testing dependencies in RETORCH, we plan to adapt/modify state-of-the-art approaches taking into account the particular characteristics of the E2E test cases based on the categories and attributes described into the Section 3. Once the testing resources are identified, RETORCH arranges the test cases in groups avoiding dependencies between the test cases and the abovementioned resources. We plan to take advantage of the state-of-the-art focused on make groups, as the clustering algorithms. More concisely, we plan to apply a hierarchy clustering technique (Guha et al. 2001) that allows us to divide/aggregate the groups of test cases (TGroups) in different machines.

Finally, into the scheduling process, the increasing complexity and heterogeneity of the test cases/resources in real test suites is a challenging problem. We plan to devise or develop a solver based on a scheduling algorithm, such as Job-Shop, Multi-Objective Task, Genetic, or Shortest-Job-Next algorithms. These algorithms had been widely used to orchestrate resources and processes in the Cloud and are beginning to be used to schedule test cases (Xie and Yang 2018).

7 Conclusions and future work

This article proposes an approach called RETORCH to orchestrate the execution of the end-to-end test cases (E2E) through the identification of resources required to run an E2E test case, the grouping of the test cases based on the minimization of the resources to be deployed, and on the parallel scheduling of the tests in several instances. We performed an evaluation of RETORCH with a real-world application in a Cloud test environment.

The results show that RETORCH improves the efficiency of the E2E test execution optimizing their resources and execution time. The execution time is decreased through the scheduling of the test cases in several instances considering the test dependencies and other issues related to parallelize the test execution. Not only does the execution of the tests in several instances increase the usage of resources like memory, but RETORCH can optimize these resources avoiding the oversubscription that may cause redeployments through the aggregation of similar tests in the same instance.

As future work, we plan to integrate RETORCH in the *ElasTest* platform to orchestrate efficiently the execution of the E2E test cases. This would require the automatic identification of both resources and dependencies between the tests in the Cloud systems. Another line of research is to thoroughly evaluate the grouping and scheduling methods in the context of optimizing the E2E test executions. Part of this work is currently on progress, and we plan to develop a tool that allows selecting the resources and arranging testing with the test cases. With this tool, we ought to autogenerate the scripting or pipelining code required by ElasTest or other application to deploy the test cases with their required resources, avoiding the resource oversubscription.

Funding information This work was supported in part by the Spanish Ministry of Economy and Competitiveness under TestEAMoS (TIN2016-76956-C3-1-R) project and ERDF funds and by the European Project ElasTest in the Horizon 2020 research and innovation program (GA No. 731535).

References

Augusto, C., Morán, J., Bertolino, A., de la Riva, C., & Tuya, J. (2019). RETORCH: Resource-aware end-to-end test orchestration. In M. Piattini, P. Rupino da Cunha, I. García-Rodríguez de Guzmán, & R. Pérez-Castillo (Eds.), *12th International Conference on the Quality of Information and Communications Technology (QUATIC 2019)* (p. 14). https://doi.org/10.1007/978-3-030-29238-6_22.

Bell, J., Kaiser, G., Melski, E., & Dattatreya, M. (2015). Efficient dependency detection for safe Java test acceleration. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, 770–781. https://doi.org/10.1145/2786805.2786823.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *FoSE 2007: Future of Software Engineering*, 85–103. https://doi.org/10.1109/FOSE.2007.25.

Bertolino, A., Calabró, A., De Angelis, G., Gallego, M., García, B., & Gortázar, F. (2018). When the testing gets tough, the tough get ElasTest. Proceedings - International Conference on Software Engineering (pp. 17–20). https://doi.org/10.1145/3183440.3183497.

Bertolino, A., de Angelis, G., Gallego, M., García, B., Gortázar, F., Lonetti, F., & Marchetti, E. (2019). A systematic review on cloud testing. *ACM Computing Surveys, 52*(5), 1–42. https://doi.org/10.1145/3331447.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM, 59*(5), 50–57. https://doi.org/10.1145/2890784.

Casalicchio, E. (2017). Autonomic orchestration of containers: Problem definition and research challenges. In *ValueTools 2016 - 10th EAI international conference on performance evaluation methodologies and tools* (pp. 287–290). https://doi.org/10.4108/eai.25-10-2016.2266649.

Chakraborty, S. S., & Shah, V. (2011). Towards an approach and framework for test-execution plan derivation. In *2011 26th IEEE/ACM international conference on automated software engineering, ASE 2011, proceedings* (pp. 488–491). https://doi.org/10.1109/ASE.2011.6100106.

De Brito, M. S., Hoque, S., Magedanz, T., Steinke, R., Willner, A., Nehls, D., et al. (2017). A service orchestration architecture for fog-enabled infrastructures. In *2017 2nd international conference on fog and mobile edge computing, FMEC 2017* (pp. 127–132). https://doi.org/10.1109/FMEC.2017.7946419.

Docker Inc. (2017). Overview of Docker compose | Docker documentation. Retrieved October 14, 2019, from Docker Inc. website: https://docs.docker.com/compose/.

Docker Inc. (2019). Swarm mode overview | Docker documentation. Retrieved October 15, 2019, from https://docs.docker.com/engine/swarm/.

Draft, W. (2014). *TOSCA Simple Profile in YAML Version 1.0*. (March), 1–83. Retrieved from http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html.

ElasTest Developers Team. (2017). ElasTest: Full-teaching. Retrieved October 28, 2019, from https://github.com/elastest/full-teaching.

ElasTest Developers Team. (2018). ElasTest: FullTeaching-experiment. Retrieved October 28, 2019, from https://github.com/elastest/full-teaching-experiment.

Engström, E., Skoglund, M., & Runeson, P. (2008). Empirical evaluations of regression test selection techniques: A systematic review. *ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 22–31. https://doi.org/10.1145/1414004.1414011.

Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., et al. (2016). CloudBuild: Microsoft's distributed and caching build service. In *Proceedings - international conference on software engineering* (pp. 11–20). https://doi.org/10.1145/2889160.2889222.

Fitzgerald, B., & Stol, K. J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software, 123*, 176–189. https://doi.org/10.1016/j.jss.2015.06.063.

Gambi, A., Gorla, A., & Zeller, A. (2017). O!Snap: Cost-efficient testing in the cloud. Proceedings - 10th IEEE international conference on software testing, verification and validation, ICST 2017, 454–459. https://doi.org/10.1109/ICST.2017.51.

Gambi, A., Bell, J., & Zeller, A. (2018). Practical test dependency detection. Proceedings - 2018 IEEE 11th international conference on software testing, verification and validation, ICST 2018, 1–11. https://doi.org/10.1109/ICST.2018.00011.

Garcia, B., Lonetti, F., Gallego, M., Miranda, B., Jimenez, E., De Angelis, G., … Marchetti, E. (2018). A proposal to orchestrate test cases. Proceedings - 2018 international conference on the quality of information and communications technology, QUATIC 2018, 38–46. https://doi.org/10.1109/QUATIC.2018.00016.

Giotis, K., Kryftis, Y., & Maglaris, V. (2015). Policy-based orchestration of NFV services in software-defined networks. *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT* 2015, 1–5. https://doi.org/10.1109/NETSOFT.2015.7116145.

Guha, S., Rastogi, R., & Shim, K. (2001). CURE: An efficient clustering algorithm for large databases. *Information Systems, 26*(1), 35–58. https://doi.org/10.1016/S0306-4379(01)00008-4.

Gyori, A., Shi, A., Hariri, F., & Marinov, D. (2015). Reliable testing: Detecting state-polluting tests to prevent test dependency. *2015 International Symposium on Software Testing and Analysis, ISSTA 2015 - Proceedings*, 223–233. https://doi.org/10.1145/2771783.2771793.

Harman, M. (2011). Making the case for MORTO: Multi objective regression test optimization. Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011, 111–114. https://doi.org/10.1109/ICSTW.2011.60.

Herzig, K., Greiler, M., Czerwonka, J., & Murphy, B. (2015). The art of testing less without sacrificing quality. Proceedings - International Conference on Software Engineering, 1, 483–493. https://doi.org/10.1109/ICSE.2015.66.

Lachmann, R., Nieke, M., Seidl, C., Schaefer, I., & Schulze, S. (2017). System-level test case prioritization using machine learning. Proceedings - 2016 15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016, 361–368. https://doi.org/10.1109/ICMLA.2016.163.

Liu, C. H., Chen, S. L., & Chen, W. K. (2017). Cost-benefit evaluation on parallel execution for improving test efficiency over cloud. Proceedings of the 2017 IEEE International Conference on Applied System Innovation: Applied System Innovation for Modern Technology, ICASI 2017, 199–202. https://doi.org/10.1109/ICASI.2017.7988384.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., & Micco, J. (2017). Taming google-scale continuous testing. Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16.

Meyer, M. (2014). Continuous integration and its tools. *IEEE Software, 31*(3), 14–16. https://doi.org/10.1109/MS.2014.58.

Microsoft. (n.d.). Orchestrator overview | Microsoft Docs. Retrieved October 15, 2019, from https://docs.microsoft.com/en-us/system-center/orchestrator/learn-about-orchestrator?view=sc-orch-2019.

Oracle. (2019). MySQL. Retrieved November 3, 2019, from https://www.mysql.com/.

Pérez, P. F. (2017). *Fullteaching: A web application to make teaching online easy.* Retrieved from https://github.com/pabloFuente/full-teaching.

Rothermel, G., Harrold, M. J., Von Ronne, J., & Hong, C. (2002). Empirical studies of test-suite reduction. *Software Testing Verification and Reliability, 12*(4), 219–249. https://doi.org/10.1002/stvr.256.

Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., … Vinter, O. (2002). What we have learned about fighting defects. *Proceedings - International Software Metrics Symposium*, *2002-Janua*, 249–258. https://doi.org/10.1109/METRIC.2002.1011343.

Singh, S., & Chana, I. (2015). QoS-aware autonomic resource management in cloud computing: A systematic review. *ACM Computing Surveys, 48*(3), 1–46. https://doi.org/10.1145/2843889.

Technologies, K. (2014). *Kurento*. Retrieved from https://www.kurento.org/.

Toczé, K., & Nadjm-Tehrani, S. (2018, June 4). A taxonomy for management and optimization of multiple resources in edge computing. *Wireless Communications and Mobile Computing, 2018*, 1–23. https://doi.org/10.1155/2018/7476201.

Uberti, J., & Thatcher, P. (2018). *WebRTC Home*. Retrieved from https://webrtc.org/.

University, R. J. C. (2017). *OpenVidu*. Retrieved from https://openvidu.io/.

Velasquez, K., Abreu, D. P., Goncalves, D., Bittencourt, L., Curado, M., Monteiro, E., & Madeira, E. (2017). Service orchestration in fog environments. Proceedings - 2017 IEEE 5th International Conference on Future Internet of Things and Cloud, FiCloud 2017, 2017-Janua, 329–336. https://doi.org/10.1109/FiCloud.2017.49.

Velasquez, K., Abreu, D. P., Assis, M. R. M., Senna, C., Aranha, D. F., Bittencourt, L. F., Laranjeiro, N., Curado, M., Vieira, M., Monteiro, E., & Madeira, E. (2018). Fog orchestration for the internet of everything: State-of-the-art and research challenges. *Journal of Internet Services and Applications, 9*(1), 14–23. https://doi.org/10.1186/s13174-018-0086-3.

Wong, W. E., Morgan, J. R., London, S., & Mathur, A. P. (1998). Effect of test set minimization on fault detection effectiveness. *Software - Practice and Experience, 28*(4), 347–369. https://doi.org/10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L.

Xie, P., & Yang, D. (2018). Research on scheduling of software cloud testing. 2017 International Conference on Computer Systems, Electronics and Control, ICCSEC 2017, 1311–1314. https://doi.org/10.1109/ICCSEC.2017.8446709.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability, 22*, 67–120. https://doi.org/10.1002/stv.430.

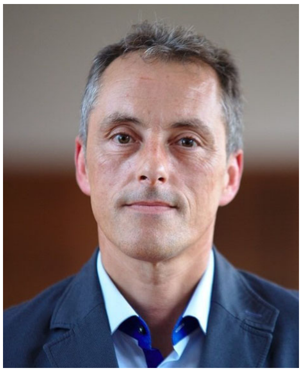
Yu, L., Su, Y., & Wang, Q. (2009). Scheduling test execution of WBEM applications. Proceedings - Asia-Pacific Software Engineering Conference*,* APSEC, 323–330. https://doi.org/10.1109/APSEC.2009.27.

Zhang, Z., Li, C., Tao, Y., Yangy, R., Tang, H., & Xu, J. (2014). Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment, 7*(13), 1393–1404. https://doi.org/10.14778/2733004.2733012.

Cristian Augusto received the degree in Computer Science in Information Technology from the University of Oviedo, Gijon, Spain in 2018. He is currently finishing his master's degree in Computer Engineering into Oviedo University. His interest research areas in the field of Software Engineering are Big Data, privacy-preserving techniques and Software Testing mainly focused on the efficient use of resources in the test process. He has also been part since 2018 of the Software Engineering Research Group (GIIS) at the Oviedo University.

Jesús Morán received the Ph.D. degree in computing from the University of Oviedo, Spain, in 2019. He is a Lecturer of the Computer Science Department with the University of Oviedo, Spain. He is a member of the Software Engineering Research Group. His research interests include software testing, big data technologies, and distributed programming.

Antonia Bertolino received the M.S. degree in electronic engineering from the University of Pisa, Pisa, Italy, in 1985. She is a Research Director with the Italian National Research Council—Institute of Information Science and Technologies (ISTI), Pisa, Italy. Her research focuses on software and service testing. Ms. Bertolino is an Associate Editor for Transactions on Software Engineering and Methodology, Empirical Software Engineering Journal, and Journal of Software: Evolution and Process. She also serves as Senior Editor for the Journal of Systems and Software. She has been the General Chair of the 2015 International Conference on Software Engineering, Florence, Italy.



Claudio de la Riva received the Ph.D degree in computing from the University of Oviedo, Spain, in 2004. He is an Associate Professor of the Computer Science Department with the University of Oviedo, Spain. He is a member of the Software Engineering Research Group. His research interests include software verification and validation, software quality and software testing, mainly focused on testing database applications and services.

Javier Tuya received the Ph.D. degree in engineering from the University of Oviedo, Oviedo, Spain, in 1995. He is a Professor with the University of Oviedo, Oviedo, Spain, where he is the Research Leader of the Software Engineering Research Group. He is the Director of the Indra-Uniovi Chair, a member of the ISO/IEC JTC1/SC7/WG26 Working Group for the recent ISO/IEC/IEEE 29119 Software Testing Standard, and a Convener of the corresponding UNE National Body Working Group. His research interests in software engineering include verification, and validation and software testing for database applications and services.

Affiliations

Cristian Augusto[1] • **Jesús Morán**[1] • **Antonia Bertolino**[2] • **Claudio de la Riva**[1] • **Javier Tuya**[1]

Jesús Morán
moranjesus@uniovi.es

Antonia Bertolino
antonia.bertolino@isti.cnr.it

Claudio de la Riva
claudio@uniovi.es

Javier Tuya
tuya@uniovi.es

1 Computer Science Department, University of Oviedo, Gijón, Spain

2 ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy