Optimizing End-to-End test execution: Unleashing the Resource Dispatcher - WiP

Cristian Augusto [*1], Jesús Morán [2], Claudio de la Riva [3] and Javier Tuya [4]
Computer Science Department, University of Oviedo
Campus de Viesques, Gijón, Asturias
[1] augustocristian@uniovi.es
[2] moranjesus@uniovi.es
[3] claudio@uniovi.es
[4] tuya@uniovi.es

Abstract—**Continuous integration practices have transformed software development, but executing test suites of modern software developments addresses new challenges due to its complexity and its huge number of test cases. Certain test levels, like End-to-end testing, are even more challenging due to long execution times and resource-intensive requirements, moreover when we have many End-to-end test suites. Those E2E test suites are executed sequentially and in parallel over the same infrastructure and can be executed several times (e.g., due to some tester consecutive contributions, or version changes performed by automation engines). In previous works, we presented a framework that optimizes E2E test execution by characterizing Resources and grouping/scheduling test cases, based on their compatible usage. However, the approach only optimizes a single test suite execution and neglects other executions or test suites that can share Resources and lead to savings in terms of time and number of Resource redeployments. In this work, we present a new Resource allocation strategy, materialized through a Resource Dispatcher entity. The Resource Dispatcher centralizes the Resource management and allocates the test Resources to the different test suites executed in the continuous integration system, according to their compatible usage. Our approach seeks efficient Resource sharing among test cases, test suites, and suite executions, reducing the need for Resource redeployments and improving the execution time. We have conducted a proof of concept, based on real-world continuous integration data, that shows savings in both Resource redeployments and execution time**

End-to-End Testing; E2E Testing; Test Suite Optimization; System Testing; Testing Resources; Test Optimization; Testing

I. INTRODUCTION

Continuous integration (CI) practices that integrate and test software code automatically are widely adopted in both academia and industry, reducing the duration of development cycles from years/months to weeks, or even days [1]. These shortened cycles have impacted critical development stages such as software testing, where validating modern software developments is even more complex, due to longer and costly test suites comprised of thousands of test cases that are executed frequently [2], [3].

Apart from the challenges faced by CI, the testing level End-to-End testing (E2E) present additional challenges due their high cost. E2E testing validates from the user iteration to the low-level layers like persistence or networks and are costly due to long execution times, expensive test Resources[1] [4] or requiring the entire system up for their execution. The Resources are the physical (e.g. a mobile device or a physical sensor), logical (e.g. a database or a webserver) or computational (e.g. a lambda function or a container provided in Azure containers) entities that are required by a test suite during its execution. Although there are techniques that aim to optimize the test suite execution, such as test prioritization, selection, and minimization [5] that are effective in other testing levels [6], [7]. However, in E2E testing these traditional techniques are less effective because they continue to require the same expensive Resources/system for their execution.

In previous works, we introduced RETORCH: Resource-Aware End-to-End Test Orchestration [4], a framework that optimizes the E2E test execution through a characterization of the Resources used, a grouping and scheduling of the test cases according to their usage. The groups of compatible test cases are scheduled and executed against their Resources, exclusively deployed and tear down for them. This grouping and scheduling of test cases reduces the execution time and the number of unnecessary Resource redeployments to execute the suite as it enables test parallelization and concurrent execution over the Resources. However, when several executions of the same test suite (e.g., some repository changes committed closely, pull requests opened to test several configurations or dependency updates) are executed in the same CI system, there is still room for improvement.

Throughout the life of a software project, these additional Resource deployments during hundreds or even thousands of CI executions impact the total project budget, especially in a Cloud environment where you only pay for what you use—but you pay for everything you use. [8].

In this paper, we propose an approach that extends the RETORCH framework with a Resource Dispatcher that enables

[1] Henceforth, we will use the term "Resources" (capitalized) when referring to the ones required by the E2E test suite.

E2E test Resource sharing between different test cases and suite executions (e.g., two consecutive commits or pull requests opened in the repository). The objective of this Resource Dispatcher is to take advantage of already deployed Resources and share those Resources between the test cases of different test suites or test suite executions, whenever the tests perform a compatible Resource usage (e.g., test cases that do not modify the Resource or restore its original state after its execution). To achieve this, we propose a Resource Dispatcher, that is integrated with the RETORCH approach, centralizing the Resource management, deploying and tearing down the necessary Resources for the entire continuous integration system.

The use of this Dispatcher has several scenarios. For example, it can assign an already deployed Resource in the CI (e.g., an ELK stack or a Selenoid Instance), one Resource that previously belonged to another TJob and make a compatible usage (e.g., a database that was cleaned before its usage). Another feasible strategy could be allocate a Resource that is being used concurrently by other TJobs belonging to other execution plans. The Execution plans [4] are TJobs scheduled in sequential or parallel aimed to reduce the execution time and the number of Resource redeployments during the E2E test execution. The Resource Dispatcher is applied over a case study with the CI data of a real demonstrator that showcases differences in Resource redeployments and execution time.

The rest of the paper is structured as follows: Section II provides the necessary background, Section III presents the approach, Section IV presents the proof of concept with the real demonstrator CI data, and finally, Section V presents the conclusions.

II. BACKGROUND

The RETORCH orchestration approach [4] is composed of four processes: Resource identification, grouping, scheduling, and deployment. In the Resource identification process, the tester performs a smart characterization of the Resources with different static and dynamic attributes that describe the Resources and show how they are used by the test cases. For instance, examples of these attributes are the maximum number of Resources, their cost, the hierarchy relationships, or the specific access mode (e.g., read, write, read-write). The output of the Resource identification is used in the grouping and scheduling processes that group and schedule the test cases with the containerized SUT in the so-called TJobs that are arranged sequentially and in parallel in the Execution Plan. The Execution Plan is deployed during the deployment phase generating the necessary pipelining and scripting code (e.g., Jenkins Jenkinsfile, GitHub actions YAML files, or Travis travis.yml).

The TJobs execution follows a lifecycle composed of different phases. First, a set-up is performed in which various actions are required to deploy and configure the environment. This set-up is followed by test execution (onwards exec), during which one or several test cases with compatible Resource usage are executed together. Finally, the tear-down phase performs cleaning and release actions, as well as saving results and other debugging information, such as different logs.

In RETORCH, the Resource management is handled by each TJob, which is responsible for deploying and releasing Resources. This strategy is efficient if the Execution Plan is executed alone in the CI system and not executed frequently, but there is room for improvement when the CI system has already deployed Resources, other Execution Plans, or executions of the same plan sequentially or in parallel. Some of the Resources already deployed can potentially be used by other TJobs that belong to another Execution Plan executed later or in parallel.

III. RESOURCE DISPATCHER FOR E2E TESTING

The new approach aims to enable Resource sharing between different Execution Plans, minimize the number of Resource redeployments, and also to reduce the execution time, because uses deployed Resources and not wait for its instantiation.

To enable this Execution Plan Resource sharing, the concept of Resource is refined to include the possibility of sharing them between different TJobs. In other words, the TJobs can take advantage of already deployed Resources whereby their usage does not impact their or other TJobs execution.

We propose to decouple this Resource management (deployment and tear-down) of the TJob, through a Resource Dispatcher (henceforth referred to as Dispatcher). The Dispatcher introduces the role of test Resource manager who is responsible for managing the Resources used within the CI environment. The general process is depicted in Fig. 1, which gives as input several pull requests (PR) opened with their Execution Plans (that can be different or several executions of the same plan). When the CI starts with the plan, the different TJobs start their execution sequentially and in parallel, requesting different Resources from the Dispatcher. The CI executes the different TJobs until the last has ended, which continues until the last TJob has finished.

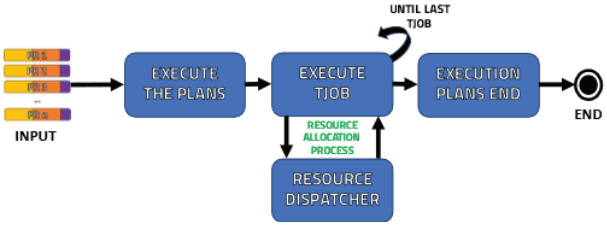

Figure 1 General overview of the process

The Resource allocation process using the Dispatcher is depicted in Fig. 2. *T*he process starts with the TJob set-up, on which the Resources are requested. If a compatible Resource is already available, it is allocated. If not, a new Resource is instantiated. Then the Resource is used during the test execution phase and released before the end of the TJob in the tear-down phase.

When a TJob starts its execution, it enters into the set-up phase (in yellow), which requests a Resource from the Dispatcher (1), who checks whether the Resource is already deployed by other Execution Plans (2) by reviewing the

Resource Pool (3). If the Resource is not deployed, the Dispatcher deploys a new Resource and registers it in the Resource Pool (5), along with the type of access mode performed and its attributes, e.g., if it is possible to be shared with other TJobs or it can be accessed concurrently.

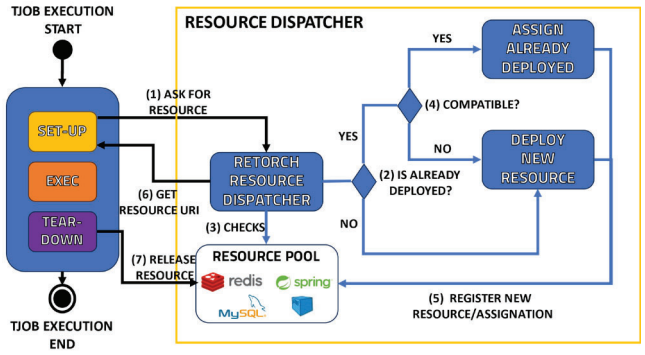

Figure 2 Resource Allocation process

If the Dispatcher has an already deployed Resource, it verifies whether the intended usage by the TJob is compatible with it (4). For example, two TJobs that use the same Resource without modifying it are compatible and can share it for their execution, saving the time of a new deployment. However, if TJob 1 modifies and pollutes the Resource, it must be executed with different Resources. If not, the Dispatcher proceeds with the instantiation of a new Resource (e.g. turns on the hardware device if it's a physical Resource, instantiates it on the air if it's a logical Resource, or asks the service provider for a computational Resource) and registers it within the Resource Pool with the usage that is performing the TJob (5). Conversely, if the Resource is compatible, it is assigned mapping this assignation in the Resource of the Resource Pool (5). The Dispatcher answers the TJob with the Resource (6), allowing the TJob to continue with its execution (in orange). When the TJob ends, during its tear-down phase (in violet), it notifies the Dispatcher (7) that this Resource is no longer required, giving to the Dispatcher the responsibility for the tear-down or its reasignation to another TJob.

IV. Evaluation

To assess the viability of the Resource Dispatcher, we carried out a proof of concept using the FullTeaching test suite [9], part of a demonstrator belonging to ElasTest Horizon 2020 European Project [10]. FullTeaching [11] is an online education platform designed to simplify the creation of courses and virtual classrooms, making remote teaching more accessible. FullTeaching is composed of several Resources, such as web and multimedia servers, relational databases, or web browsers.

RETORCH, with the information provided by the annotated Resources and access modes in the test cases, provides an Execution Plan composed of 12 TJobs deployed in parallel in groups of 5 TJobs. For this proof of concept, we focused on three different TJobs of this Plan: TJob-A, TJob-B, and TJob-C, whose Resources, number of test cases, and access modes performed are depicted in Table I:

TABLE I
TJobs and Resources access modes

	Access Modes		
TJob	**Database**	**Web Server.**	**Mult. Server**
A	R-W	R	R
B	R-W	R	R
C	R-W*	R	No-Access

All the TJobs **A, B, and C** modify the database, but **TJob-C** restores its state before concluding (R-W*), allowing the database to be used by subsequent test cases (albeit not concurrently). TJob C is also characterized by not accessing to the multimedia server, making it possible to mock it to provide only the health check, which is lighter than the entire Resource.

We employ the continuous integration data of the Friday 1st of March at 0:00 a.m. on which 5 pull requests were opened by DependaBot with different version updates in the GitHub repository [9]. This repository is integrated into our Jenkins continuous integration system, executing the Execution Plan for each new pull request created. The average times on which the different TJob lifecycle phases start and end all pull requests are shown in *Table II*:

TABLE II
Average times for PR Execution and deployment time (Resources)

	Resources			TJobs					
ID	**Database**	**Web Server**	**Multimedia Serv,**	**Set-up-start**	**Set-up-end**	**Exec-start**	**Exec-end**	**Tear-down-start**	**Tear-down-end**
TJob-A	29	30	46	1	48	49	120	121	126
TJob-B	29	30	46	1	47	48	149	150	153
TJob-C	29	30	46	1	49	50	121	122	124

Each Resource set-up individually takes on average in the different TJobs 28-29 seconds for the BD, 30.5 seconds for the multimedia server, and 46.36 seconds for the webserver. Fig. 3 depicts the difference in execution time with 3 parallel executions of the 5 different pull requests (from PR1 to PR5), using RETORCH's original approach (in blue) against RETORCH with the Dispatcher (in green).

For PR 2 and PR 3 executions, the Dispatcher alternative takes the same amount of time to set up. Part of this time is spent preparing the PR resources (indicated in yellow, with a 28-29 second BD setup wait), while the remaining 17-18 seconds are spent waiting for the readiness of the PR1 Resources (Multimedia Server). The reuse of the same Resources in TJob A only requires instantiating/cleaning the database in the different TJobs. PR 4 and PR 5 do not require waiting and the execution time is reduced. On the other hand, the RETORCH alternative requires instantiating the Resources for each TJob and tearing them down when it finishes, using more time than the other alternative in both phases and leading to a higher total

execution time, saving 21 seconds. These time savings are even more important in the CI systems where the test suites are executed hundred or even thousands of times at each repository change.

Figure 3 Duration of the different PR

Fig. 4 depicts the number of Resource redeployments of both alternatives during the whole PR, in blue RETORCH, and in green the RETORCH + Dispatcher alternative.

The Dispatcher alternative redeploys 70% fewer Resources (45 Resource re-deployments with RETORCH against 14 with RETORCH + Dispatcher) to execute the PRs because the TJob B and C share the multimedia and web servers of the first PR and clean the database for each TJob, while the TJob A instantiates its Resources in the first execution and reuses them in the subsequent executions. The reduction of Resource instantiations is useful when the Resources are limited (e.g. physical limitations like the number of available devices) or when the testing is carried out over the Cloud and each instantiation impacts the total project budget.

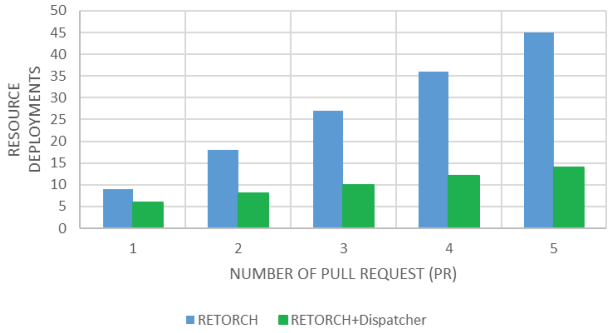

Figure 4 Number of Resource redeployments

V. CONCLUSIONS

We have proposed the Resource Dispatcher, that evolves the current RETORCH Resource allocation approach into a centralized way to manage the Resources. The Dispatcher enables Resource sharing between different Execution Plans or executions of the same Execution Plan itself, which can be performed when a pull request or several consecutive contributions arrive at the repository.

Through a proof of concept using real-world continuous integration data, we show the feasibility of our proposed approach, showcasing benefits in terms of Resource savings and execution time. This remarks the practical applicability of our solution in addressing the challenges posed by modern software development practices, where the complexity of test suites and the need for frequent testing require innovative solutions to streamline the testing process and continue improving and enhancing the software quality.

As future work, we plan to evaluate our approach in more demonstrators and test suites. Additionally, we intend to combine RETORCH with other optimization techniques such as test-batching. Another research line involves incorporating the Resource Dispatcher into a bot engine and integrating it with the RETORCH platform

ACKNOWLEDGMENTS

This work was supported by the project PID2022-137646OB-C32 under Grant MCIN/AEI/10.13039/501100011033/FEDER, UE

REFERENCES

[1] M. Meyer, "Continuous integration and its tools," *IEEE Softw.*, vol. 31, no. 3, pp. 14–16, 2014, doi: 10.1109/MS.2014.58.
[2] H. Esfahani *et al.*, "CloudBuild: Microsoft's distributed and caching build service," in *Proceedings - International Conference on Software Engineering*, in {ICSE} '16. ACM, 2016, pp. 11–20. doi: 10.1145/2889160.2889222.
[3] A. Memon *et al.*, "Taming google-scale continuous testing," in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*, IEEE, May 2017, pp. 233–242. doi: 10.1109/ICSE-SEIP.2017.16.
[4] C. Augusto, J. Morán, A. Bertolino, C. de la Riva, and J. Tuya, "RETORCH: an approach for resource-aware orchestration of end-to-end test cases," *Softw. Qual. J.*, vol. 28, no. 3, pp. 1147–1171, Sep. 2020, doi: 10.1007/s11219-020-09505-2.
[5] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, no. 2. John Wiley and Sons Ltd., pp. 67–120, Mar. 2012. doi: 10.1002/stv.430.
[6] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "Empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Conference on Software Maintenance*, 1998, pp. 34–43. doi: 10.1109/icsm.1998.738487.
[7] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application," *J. Syst. Softw.*, vol. 48, no. 2, pp. 79–89, 1999, doi: 10.1016/S0164-1212(99)00048-5.
[8] A. Eivy, "Be Wary of the Economics of 'Serverless' Cloud Computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, 2017, doi: 10.1109/MCC.2017.32.
[9] C. Augusto, J. Morán, C. de la Riva, and J. Tuya, "FullTeaching E2E Test Suite." 2023. [Online]. Available: https://github.com/giis-uniovi/retorch-st-fullteaching
[10] B. Garcia *et al.*, "A proposal to orchestrate test cases," in *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, 2018, pp. 38–46. doi: 10.1109/QUATIC.2018.00016.
[11] ElasTest EU Project, "Fullteaching: A web application to make teaching online easy." Universidad Rey Juan Carlos, 2017. Accessed: Aug. 10, 2023. [Online]. Available: https://github.com/pabloFuente/full-teaching