



Rational decomposition and orchestration for serverless computing

Deliverable 3.5

Continuous testing tool II

Version: 1.0

Publication Date: 31-March-2021

Disclaimer:

The RADON project is co-funded by the European Commission under the Horizon 2020 Framework Programme. This document reflects only the authors' views. EC is not liable for any use that may be done of the information contained therein.

Deliverable Card

Deliverable	D3.5
Title:	Continuous testing tool II
Editor(s):	André van Hoorn (UST) and Pelle Jakovits (UTR)
Contributor(s):	Thomas F. Düllmann (UST), André van Hoorn (UST), Mainak Adhikari (UTR), Pelle Jakovits (UTR), Ahmad Alnafessah (IMP)
Reviewers:	Hans Georg Næsheim (PRQ), Stefania D'Agostini (ENG)
Type:	Report
Version:	1.0
Date:	31-March-2021
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

As a part of the RADON framework, the Continuous Testing Tool (CTT) provides the functionality for defining, generating, executing, and refining continuous tests of application functions, data pipelines, and microservices, as well as for reporting test results. This document presents the final version of CTT. In particular, this deliverable (i) outlines the CTT tool's final architecture and integration into the RADON workflow and tool ecosystem, (ii) presents the improvements introduced to the extensible CTT framework since the previous deliverable, (iii) summarizes the research approaches developed in the context of DevOps-oriented FaaS/microservices load testing, and (iv) details the new CTT module for data pipeline testing.

Glossary

AI	Artificial Intelligence
APM	Application Performance Management
API	Application Programming Interface
AWS	Amazon Web Services
CI/CD	Continuous Integration/Delivery
CSAR	TOSCA Cloud Service Archive
CTT	Continuous Testing Tool
EC2	Elastic Compute Cloud
FaaS	Function-as-a-Service
GMT	Graphical Modelling Tool
IDE	Integrated Development Environment
SaaS	Software-as-a-Service
SUT	System Under Test
TI	Test Infrastructure
TOSCA	Topology and Orchestration Specification for Cloud Applications
URL	Uniform Resource Locator

Table of contents

1 Introduction	7
1.1 Deliverable objectives	7
1.2 Overview of main achievements	8
1.3 Structure of the document	8
2 CTT workflow and architecture overview	9
2.1 Continuous testing workflow	9
2.1 CTT architecture and integration in RADON	12
3 CTT framework extensions and improvements	13
3.1 Updates to CTT modeling types and artifacts	13
3.2 Revised and new functionalities and enhancements	14
3.2.1 xOpera	14
3.2.2 Configuration file	16
3.2.3 Execution of multiple tests	17
3.2.4 Radon IDE plugin	18
3.2.5 CLI Tool	19
3.2.6 Logging/error handling	20
3.2.7 Monitoring	20
3.3 Software engineering practices, code quality, and documentation	20
4 DevOps-oriented microservices/FaaS load testing	25
4.1 Microservice-tailored workload generation	25
4.2 Context-tailored workload generation	27
4.3 Domain-based scalability testing	28
4.4 Automated testing to estimate interference between co-located microservices	30
5 Data pipeline testing	32
5.1 CTT data pipeline architecture including core design decision	32
5.1.1 Workflow	32
5.1.2 Technical architecture and interaction with the environment	33
5.2 Technological and Development Decisions	34
5.3 Implementation view of CTT data pipeline module	34
5.3.1 Modelling types	35
5.3.2 CTT data pipeline agents	38
5.3.3 Extension framework including load injector	39
5.4 Volume/velocity testing of CTT data pipeline module	40
6 Conclusions	43

Appendix: Compliance with requirements	44
Usage Scenarios	44
Requirements List	44
References	48

1 Introduction

Testing is the prevalent quality assurance technique in practice. To assess whether applications developed via the RADON methodology and framework meet their quality requirements, RADON includes the *continuous testing workflow*, introduced in the initial CTT deliverable [\[D3.1\]](#), which particularly aims to support software developers, QoS engineers, and release managers in producing high-quality applications. The core component implementing the continuous testing workflow is the Continuous Testing Tool (CTT), being the subject of this deliverable. CTT provides the functionalities for defining, generating, executing, and refining continuous tests of application functions, data pipelines, and microservices, as well as for reporting test results.

CTT enriches the TOSCA ecosystem by end-to-end support for continuous testing of microservice-based (including FaaS) and data pipeline applications in DevOps. It is the first tool of its kind that supports the whole workflow — from test specification over execution and reporting to automated updates based on production data — that is also extensible to custom needs, e.g., integration of other types of tests or tools. A particular innovation lies in the integrative test generation features for obtaining tailored tests, which fits into the constraints of DevOps-based development settings with separate teams and pipelines for microservices, and the goal of fast and frequent releases.

The initial CTT deliverable [\[D3.4\]](#) focused on a description of CTT's high-level architecture and implementation, covering the domain model, the workflow, the hierarchy of modeling types, the server and agent components, as well as its extension points. Moreover, the deliverable provided a step-by-step description of the CTT workflow for creating, deploying, and executing tests using one of the RADON demo applications, namely SockShop.

Building on the previous deliverable, this deliverable (i) outlines the CTT tool's final architecture and integration into the RADON workflow and tool ecosystem, (ii) presents the improvements introduced to the extensible CTT framework since the previous deliverable, (iii) summarizes the research approaches developed in the context of DevOps-oriented FaaS/microservices load testing, and (iv) details the newly added CTT module for data pipeline testing.

1.1 Deliverable objectives

The main objective of this deliverable is to document the final version of the CTT tool and the related approaches for continuous testing of microservices, FaaS, and data pipelines. In order to avoid an overlap with the detailed descriptions in the previous deliverable, we will focus on those aspects that are extensions to the previous deliverable:

- The final integration into the RADON workflow and ecosystem.
- Extensions to the CTT framework.

- Research approaches focusing on continuous load testing of microservices and FaaS and their integration with CTT.
- CTT data-pipeline module that allows continuous testing of data pipeline components.

1.2 Overview of main achievements

The main achievements of the work reported in this deliverable are as follows:

- A final description of the integration of CTT with the overall RADON framework, particularly including the integration into the RADON IDE, the integration with the monitoring tool, as well as the revised integration of the orchestrator using the orchestrator's new SaaS interface.
- A description of the new features of the core CTT framework, including refined CTT-related modeling types and artifacts, the IDE plugin and command-line interfaces, and details on the newly integrated RADON tools. Moreover, we provide information on CTT's software engineering practices, code quality, and documentation.
- A summary of the research on DevOps-oriented load testing, including microservice-tailored, context-tailored, and domain-based load and scalability testing.
- A description of the data pipeline module, comprising its architecture based on CTT's extensions mechanisms, new modeling types, a modified data pipeline agent, a data injection framework with components based on Apache JMeter and NiFi, as well as a summary of velocity/volume testing of the data pipeline module.

1.3 Structure of the document

The remainder of this deliverable report is structured as follows.

- Section [2](#) provides an overview of the final workflow and integration of CTT in the RADON context.
- Section [3](#) presents the extensions made to CTT's core framework.
- Section [4](#) summarizes the research on DevOps-oriented microservices/FaaS load testing.
- Section [5](#) presents the new CTT-data pipeline module.
- Section [6](#) concludes the document by summarizing its contents and giving an outlook on possible future CTT developments.
- The [Appendix](#) summarizes the level of compliance for the requirements at this stage.

2 CTT workflow and architecture overview

The initial deliverable [\[D3.4\]](#) provided a detailed documentation of CTT’s architecture. In order to make this deliverable self-contained and up-to-date, this chapter provides a summary of the final workflow for continuous testing as well as CTT’s final high-level architecture and integration in the RADON ecosystem, including the parts introduced in this deliverable. The contents are based on the methodology deliverable [\[D3.1\]](#), which is released at the same time as this deliverable.

2.1 Continuous testing workflow

The continuous testing workflow comprises three usage scenarios, namely *Define Test Cases (US 1)*, *Execute Test Cases (US 2)*, and *Maintain Test Cases (US 3)*.

Figure 1 depicts the activity diagrams that summarize all usage scenarios.

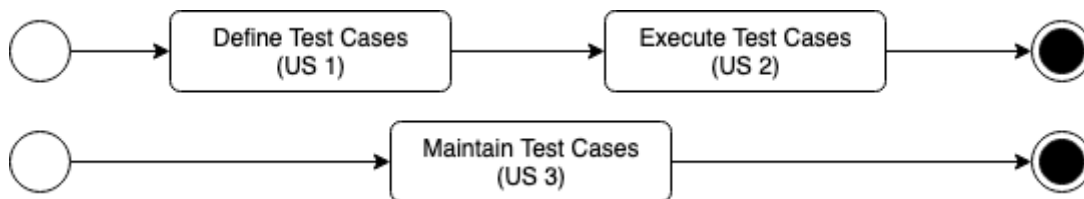


Figure 1. CTT usage scenario overview

Figure 2 shows the interaction of all tools that directly interact with CTT. Both, the IDE and CI/CD, have the ability to trigger the CTT workflow which then executes CTT’s internal workflow, including a deployment that uses the xOpera SaaS service. An optional step during the CTT workflow is the retrieval of operational monitoring data to update test parameters.

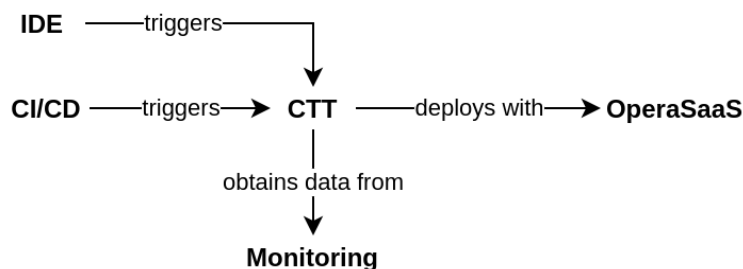


Figure 2. CTT’s tool interaction

The three usage scenarios can be summarized as follows:

- Define Test Cases (US 1).** In parallel to the regular development, the Software Developer or QoSEng can define test specifications (e.g., deployment and load tests) for their application, referred to as the system under test (SUT). The definition of test specifications is done via the RADON IDE by adding respective TOSCA policy types to the SUT's service template. Moreover, the developer defines an additional service template for the test infrastructure (TI). The resulting artifacts, comprising the SUT's and TI's CSAR files and input definitions, can be exported from the RADON IDE. Figure 3 shows an activity diagram for the usage scenario.

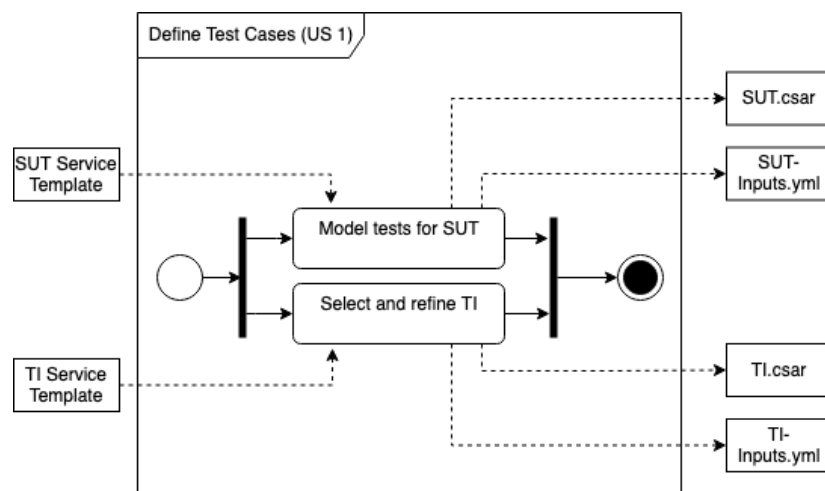


Figure 3. Activity diagram for *define test cases* usage scenario

- Execute Test Cases (US 2).** During development or before deploying to production, actors such as the Developer or Release Manager can manually trigger the execution via the RADON IDE or the standalone interface or automatically via CI/CD for integration into DevOps processes. In each case, the Continuous Testing Tool conducts a series of steps for each selected test case, namely preparing the project context, generating executable artifacts (CSARs), deploying the SUT and the TI via the Orchestrator, executing the tests, and collecting the results. Afterward, the test results can be inspected. Figure 4 shows an activity diagram for the usage scenario.

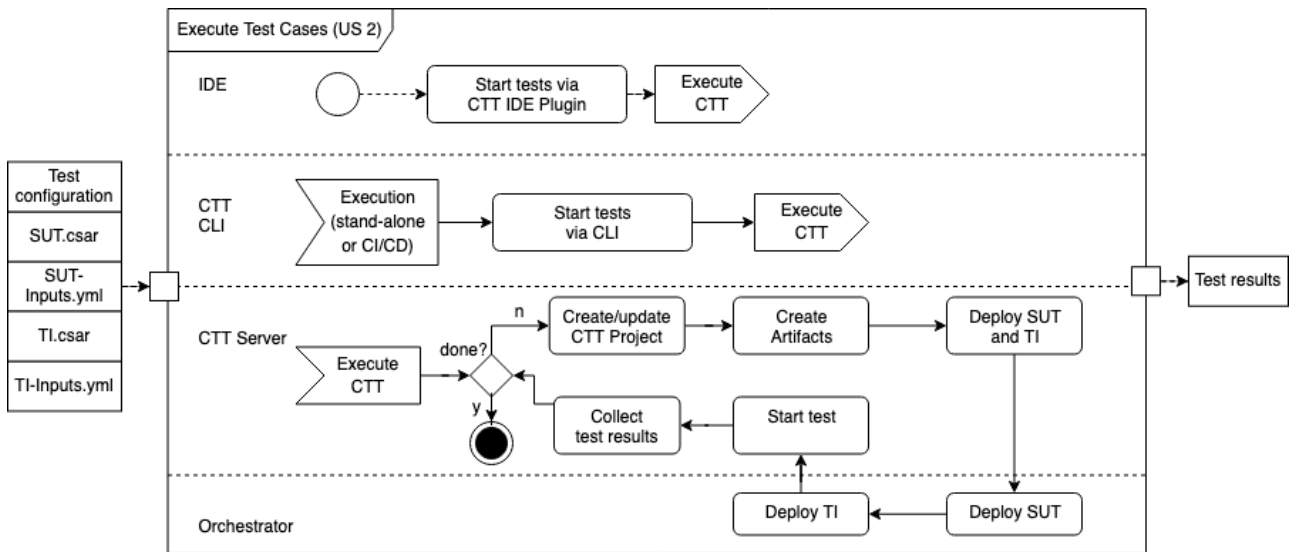


Figure 4. Activity diagram for *execute test cases* usage scenario

Maintain Test Cases (US 3). Once the application is deployed in a production environment, operational data can be used to generate, refine, and select test cases, to fit into DevOps contexts, such as evolving system usage, limited test budgets in CI/CD. Even though different approaches for maintaining test cases are provided in the continuous testing workflow, they share a similar process of analyzing the intended user request, querying the monitoring tool for the required monitoring data, and providing the generated/refined test artifacts. Figure 5 shows an activity diagram for the usage scenario.

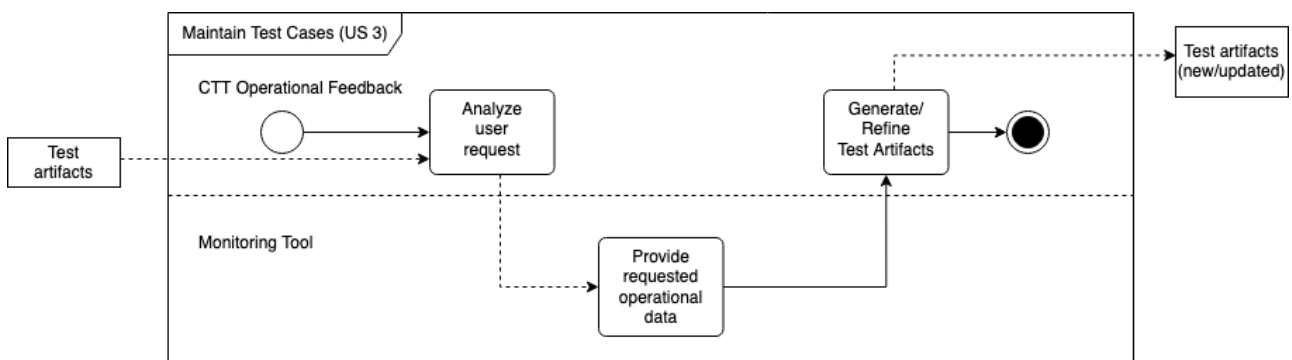


Figure 5. Activity diagram for *maintain test cases* usage scenario

2.1 CTT architecture and integration in RADON

Figure 6 depicts the CTT components and their integration with other entities and tools in the RADON ecosystem.

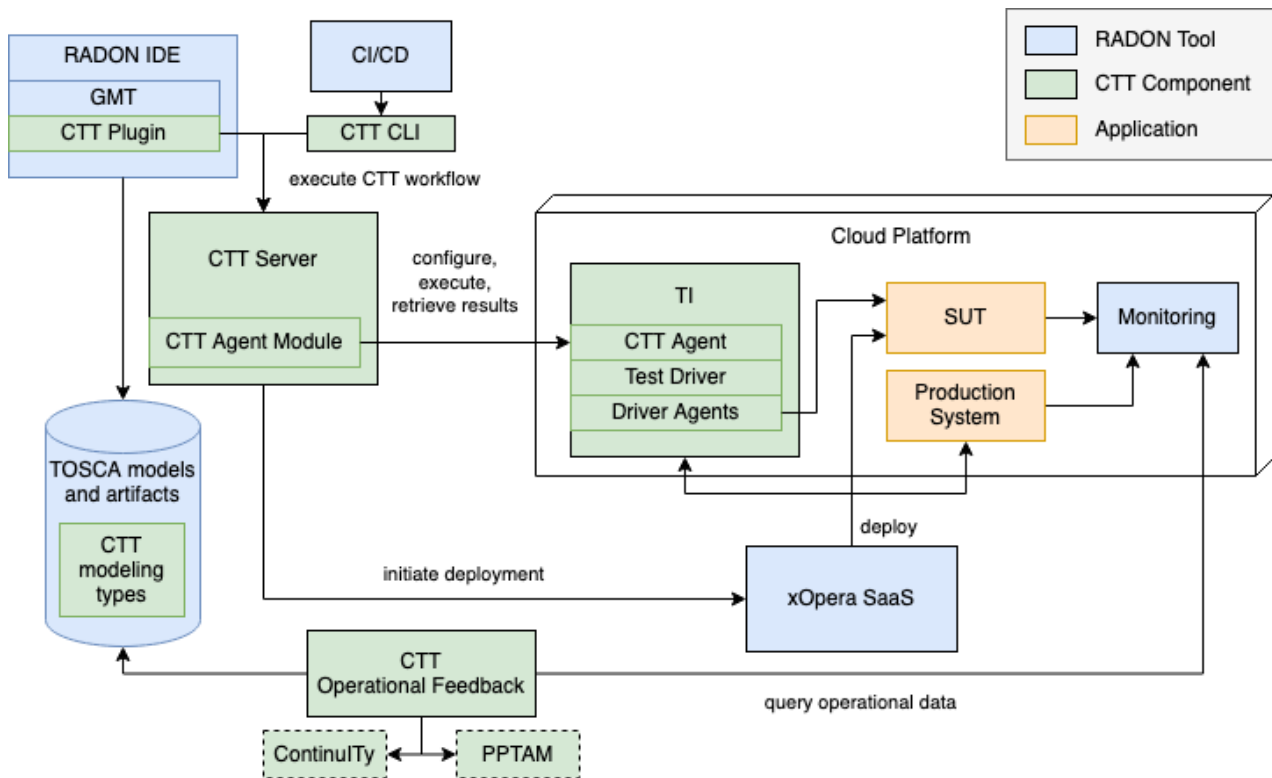


Figure 6. CTT components and their interaction with other RADON tools

Tests and test infrastructures are defined in TOSCA models, using CTT-based (extensible) modeling types, e.g., supporting performance or deployment tests. The CTT server is responsible for managing the test execution workflow, including the generation of executable TOSCA models, the deployment of the test infrastructure (TI) and the system under test (SUT) using the xOpera (SaaS) orchestrator, running the tests via CTT test agents, as well as collecting and providing the test results. A CTT agent executes the actual test, e.g., a load test with a load driver such as JMeter. The RADON IDE (including RADON's GMT) can be used to define the test-related information, such as the tests and the TIs, and to interact with the CTT server via the CTT IDE Plugin. CTT CLI provides an alternative command-line interface to interact with the CTT server. CTT CLI is the preferred way to use CTT in CI/CD pipelines.

Moreover, CTT provides various approaches for generating and updating load tests from operational data, particularly to obtain tailored tests for microservices, FaaS, and data pipelines. Therefore, it interfaces with a monitoring tool, such as the one included in the RADON ecosystem.

3 CTT framework extensions and improvements

This chapter builds on deliverable [\[D3.4\]](#), which provided a detailed description of CTT's extensible framework architecture and modeling type hierarchy, and focuses on the extensions and improvements made since the release of the previous deliverable to the modeling types and artifacts, as well as the framework and its integration. Moreover, we provide information on CTT's software engineering practices, code quality, and documentation.

3.1 Updates to CTT modeling types and artifacts

This section covers updates to modeling types and artifacts related to CTT.

There are several general types in the RADON Particles repository that CTT relies on. This can be either gathering information from types that are required for the execution of CTT (e.g., IP addresses or paths to access endpoints) or usage in CTT-related service templates like testing agents (e.g., `DeploymentTestAgent`). Another set of our contributions went towards showcase examples.

In the following, we will shortly explain what changes have been made to which modeling types and artifacts:

- **AWS API Gateway** (`radon.nodes.aws.AwsApiGateway`):
As functions in a FaaS setup do not have endpoints themselves, the information for the URL for reaching a function needs to be reconstructed from different entities. To achieve this, the base endpoint of an API Gateway needs to be exposed, which is added in the node type for the AWS API Gateway. The URL for an AWS API Gateway can be constructed as follows:
`https://<AWS API ID>.execute-api.<AWS Region>.amazonaws.com/<API Stage>`
After our change, this URL is populated with the respective values once all of them are known and can be retrieved from the node type as an attribute.
- **AWS EC2** (`radon.nodes.VM.EC2`)
In order to get the URL or IP of an AWS EC2 instance after deployment, the attribute for the public IP address of an EC2 instance needed to be accessible by CTT (e.g., to retrieve the IP address of a EC2-based SUT after its deployment). To achieve that, we added an output to the TOSCA model.
- **DockerEngine** (`radon.nodes.docker.DockerEngine`)
We implemented the Ansible playbook for the *DockerEngine* node type in order to be able to deploy the testing agents for CTT as Docker containers.
- **JMeterAgentEC2** (`radon.blueprints.testing.JMeterMasterOnlyEC2`)
Previously, we had a service template based on a local Docker deployment. Throughout the continuation of CTT development, we focused on remote deployments. That is why we added a service template that was based on an AWS EC2 instance with the deployment of the JMeter testing agent in Docker on top.

- **DeploymentTestAgent** (`radon.blueprints.testing.DeploymentTestAgent`)
In addition to a load test, we added a deployment test agent. This test sends an HTTP request to the specified end point of the SUT in order to check whether the deployment was successful. Similar to the architecture of the *JMeterAgentEC2*, this agent is also packaged as a Docker container and is based on an AWS EC2 instance with Docker installed in this configuration.
- **SockShopTestingExample** (`radon.blueprints.SockShopTestingExample`)
In order to give users an easy-to-use service template, we added a separate service template for the SockShop that includes a testing policy. That way, users can see a properly configured example that they can also adapt to their use case or fill with their custom parameters.
- **ServerlessToDoListExample** (`radon.blueprints.testing.ServerlessToDoListAPITestingExample`)
Similar to the *SockShopTestingExample*, we also created a testing example for the ServerlessToDoList that includes a ready to use testing policy.

3.2 Revised and new functionalities and enhancements

In this section we go into further details on what has been changed in terms of revised, new, or enhanced functionalities in the CTT tool implementation.

3.2.1 xOpera

Initially, the xOpera orchestrator has been used in the form of a module included in the CTT container. While this is still a viable approach, we also prepared the tooling to switch over to xOpera SaaS which allows the management of credentials on the website of xOpera SaaS.

The following example in Figure 7 shows an example for a deployment of a service template using *AmazonAWS* (e.g., EC2 instance). When the deployment endpoint of the *CTT Server* is triggered, it passes on all required parameters to the xOpera CLI tool which is installed in the environment of the *CTT Server*. The xOpera orchestrator then uses the credentials provided through environment variables and further parameters to deploy the provided service template using Ansible. Depending on the system to deploy to, xOpera and the underlying Ansible module have further dependencies (e.g., *aws-cli* for Amazon deployments) which need to be installed in the CTT environment too.

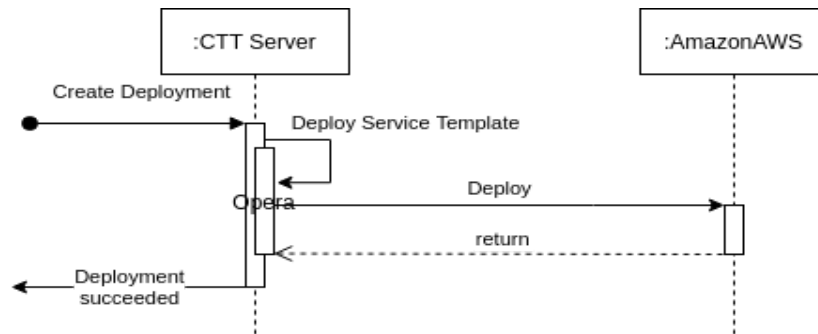


Figure 7. Example Deployment Process with xOpera CLI included in the *CTT Server*

With the migration to xOpera SaaS, CTT invokes the REST API endpoints of xOpera SaaS without the need to have xOpera CLI installed in the CTT Docker container. The credentials can be managed using the xOpera SaaS WebUI and CTT only requires one credential pair to access the respective xOpera SaaS workspaces of the user. Deployment-specific tools are no longer needed in the CTT Docker container. Figure 8 depicts the new process where the *xOpera SaaS* system is separated and removes the coupling with dependencies and credentials inside the *CTT Server* environment.

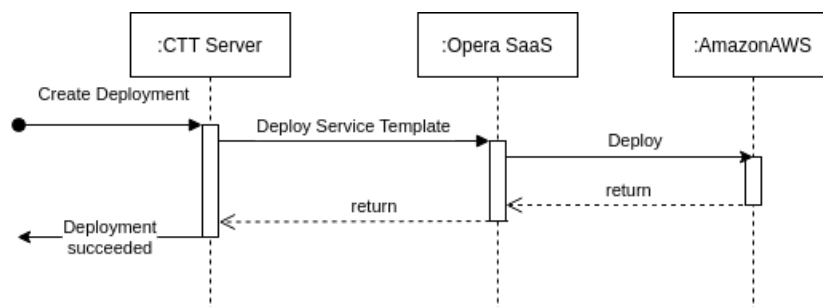


Figure 8. Example Deployment Process with xOpera SaaS as separate system

Implications of the migration

The migration from the old mode of operation to the new one had several implications on multiple levels of the process. This decision imposes positive and negative aspects which we detail below:

- *Clearer cut in responsibilities:* Having the functionality of managing the details of the deployment itself outsourced to xOpera SaaS makes a clearer cut in responsibilities of the involved tools.
- *Loose coupling:* With the deployment and credential management wrapped behind an external API allows the deployment using any orchestrator that is compatible with the same

API. This would make it easier for changing the orchestrator without the necessity of changes on the side of CTT.

- *Dependencies on other parties for operation of deployment:* Due to the outsourcing of the deployment functionality, the xOpera SaaS solution is a black box for the developers of CTT which makes the users dependent on the availability and proper functionality of the deployment service.

3.2.2 Configuration file

A configuration file for CTT includes the essential information required for executing the whole CTT workflow. Furthermore, it allows the user to save configurations in an organized way in single files. Configuration files are required for using the CTT integration in the RADON IDE (Section [3.2.4](#)), as well as the CTT CLI tool (Section [3.2.5](#)).

The configuration file combines all important information to deploy a TOSCA service template via CTT. Depending on the artifacts to be deployed, the number of required inputs may vary. In case a service template requires inputs, the file containing the inputs is required. If the service template does not require any inputs, the file is not needed. That is why we list inputs as optional fields.

Figure 9 depicts an example for a configuration file.

```

1  {
2    'name': 'ToDoListAPI',
3    'repository_url': 'demo-ctt-todolistapi',
4    'sut_tosca_path': 'todolist.csar',
5    'ti_tosca_path': 'deploymentTestAgent.csar',
6    'ti_inputs_path': 'inputs.yaml',
7    'result_destination_path': '/tmp/results.zip',
8    'test_id': 'deploymentTest_01',
9  }

```

Figure 9: CTT configuration file example

Mandatory fields:

- **name:** A custom name for the project.
- **repository_url:** Depending on the mode of operation, this field contains either the URL to the Git repository containing the project's artifacts or in case of a RADON IDE context, the folder where the project is located.
- **sut_tosca_path:** The path to the CSAR file for the SUT (relative to the location specified in `repository_url`).
- **ti_tosca_path:** The path to the CSAR file for the TI (relative to the location specified in `repository_url`).

- **result_destination_path**: The location where the results will be downloaded to (absolute path).
- **test_id**: The ID of the test policy to execute.

Optional fields:

- **sut_inputs_path**: The path to the inputs for the SUT (relative to the location specified in repository_url).
- **ti_inputs_path**: The path to the inputs for the TI (relative to the location specified in repository_url).

3.2.3 Execution of multiple tests

In order to execute multiple tests with CTT, the [configuration files](#) for the different executions need to be provided. These configuration files are accepted by a dedicated endpoint of the CTT server, which then executes the tests in separate environments. This allows the tests to take place without interference. In order to execute multiple tests, the user provides a set of configurations as shown in Listing 1 that can be passed to the CTT Server for bulk execution as depicted in Figure 4.

```
{
  {
    'name': 'ToDoListApi',
    'repository_url': 'https://github.com/radon-h2020/demo-ctt-todolist.git',
    'sut_tosca_path': 'test_01/sut.csar',
    'ti_tosca_path': 'test_01/ti.csar',
    'result_destination_path': 'test_01/result.zip',
    'test_id': 'test_01'
  },
  {
    'name': 'ToDoListApi',
    'repository_url': 'https://github.com/radon-h2020/demo-ctt-todolist.git',
    'sut_tosca_path': 'test_02/sut.csar',
    'ti_tosca_path': 'test_02/ti.csar',
    'result_destination_path': 'test_02/result.zip',
    'test_id': 'test_02'
  },
}
```

Listing 1: Set of CTT configurations for execution of multiple tests.

Similarly to a regular execution, each of the configurations is executed via CTT.

3.2.4 Radon IDE plugin

The CTT IDE plugin enables RADON users to execute CTT’s functionalities via the RADON IDE in a project workspace. Users specify the configuration of the CTT execution by creating a CTT configuration file (Section 3.2.2) in the project workspace of the SUT. A CTT execution can be started by right-clicking on a configuration file and selecting the respective menu entry (Figure 10). The IDE plugin then interacts with the CTT server in order to perform the usual CTT steps, i.e., deploying the TI and the SUT, executing the test, and collecting the test results. During the execution, the progress is logged in the output panel (Figure 11), and a progress bar appears on the bottom right of the RADON IDE. After a successful execution, users will find the test results in the configured location in their workspace (Figure 12).

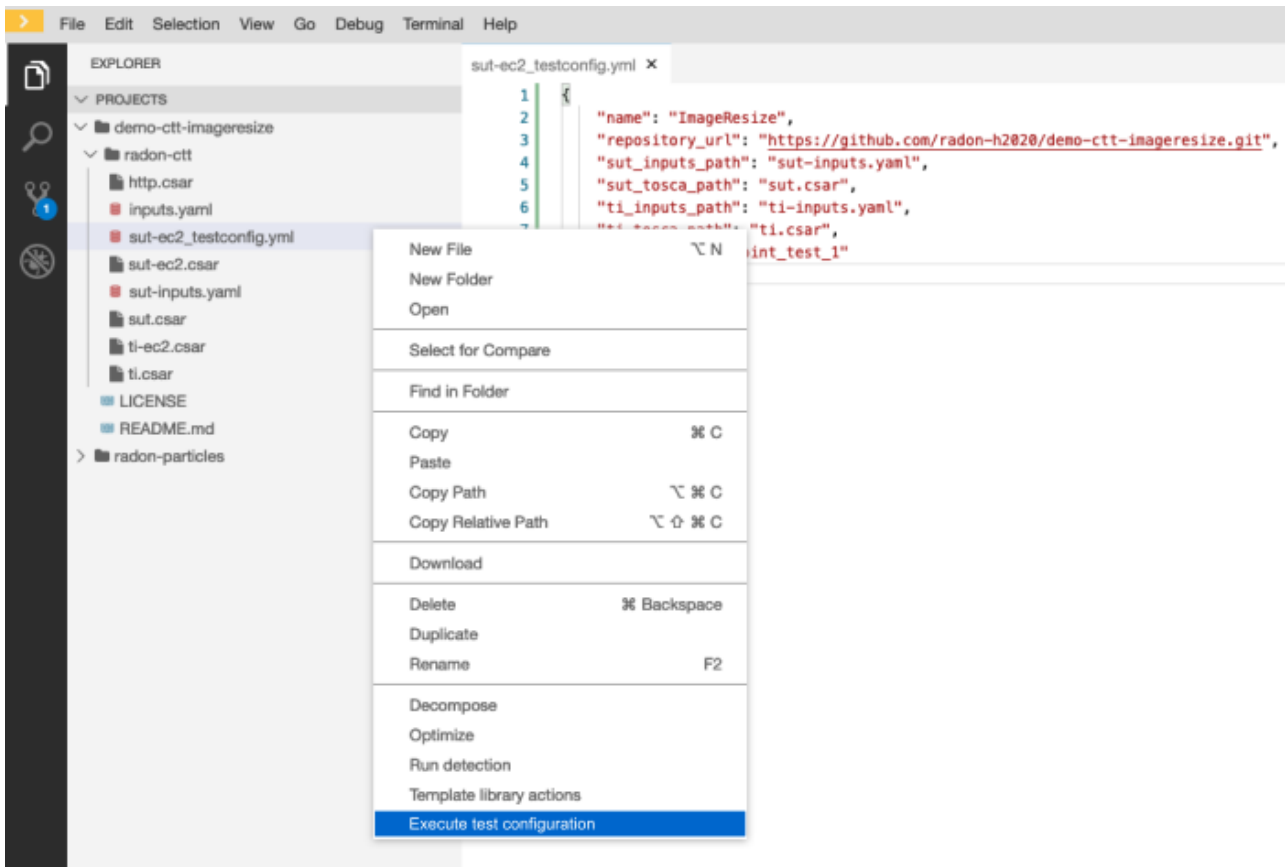


Figure 10. Integration of CTT in the RADON IDE

```

Output x
[1/6] Project created successfully ("570e0962-5a8d-4782-93d1-6723ad809c26").
[2/6] Test artifact created successfully ("68054f67-4c8e-471c-a2df-ecfbe933e79e").
[3/6] Deployment successful ("6f8f4428-ae94-4a3c-bfff-532e10c695a7").
[4/6] Test execution successful ("ca680e9d-04c2-4ba4-87b8-124fcedde679").
[5/6] Results successfully obtained ("211dfaa8-d1c8-4f7a-89b1-21695338392d").
[6/6] Results successfully downloaded to serverless-test-results.zip
  
```

Figure 11. Progress log in the output panel of the RADON IDE

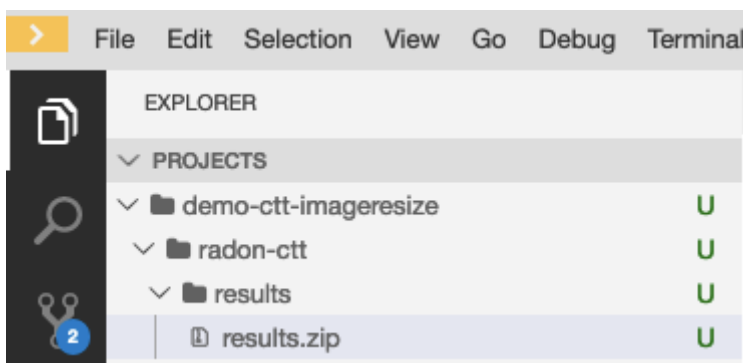


Figure 12. Test results in the RADON IDE

The CTT IDE plugin is written in Typescript, and it is integrated in the RADON IDE with a Che plugin and a Kubernetes component. A CTT server instance is automatically deployed into the RADON IDE. The communication between the plugin and the CTT server is performed via the REST API.

3.2.5 CLI Tool

The CLI Tool allows to make the execution more convenient by automating the sequential invocation of the API endpoints of the CTT Server. This makes it easier to integrate a deployment via CTT in a script or as part of a CI/CD pipeline.

As can be seen in Figure 13, two parameters are required to run the CLI tool: the URL of the running CTT server and the CTT configuration file as described in Section 3.2.2.

```

ctt-cli.py [PARAMS]
Mandatory parameters:
  -u, --url=CTT_SERVER_URL  URL of the CTT server (e.g., http://localhost:18080/RadonCTT)
  -c, --config=CTT_CONFIG   Path to the CTT configuration file

Other parameters:
  -v, --verbose             Be verbose
  -h, --help                Print this help
  
```

Figure 13. CLI Tool usage output

The CLI Tool is implemented in Python 3 and makes use of the CTT API [\[D3.4\]](#) to sequentially invoke the steps required for the whole CTT workflow execution. This includes the following steps: project creation, test artifact creation, deployment of SUT and TI, execution of the tests, creation of results, and retrieving the results.

3.2.6 Logging/error handling

In the first implementations of CTT, errors and exceptions have not been reported to the outside, which made it very difficult to debug when not having direct access to the running instance of CTT. With these changes, we improved in this matter by passing more details on occurring errors to the outside.

3.2.7 Monitoring

CTT uses RADON's Monitoring tool to gather monitoring data from production systems in order to update tests based on data from production workloads. RADON's Monitoring tool builds on the open-source monitoring tool Prometheus¹. CTT uses Prometheus's standard REST-based interface to request data from the monitoring tool using the Prometheus Query Language (PromQL). The type of queries depends on the specific use cases of operational data, as detailed in Section [4](#) and Section [5](#). The prerequisite for querying the data is that the data is being collected from the production system using suitable collectors. In addition to Prometheus-based monitoring, some approaches require additional monitoring sources (e.g., for detailed access logs and execution traces), which will be detailed in the respective sections of this document.

3.3 Software engineering practices, code quality, and documentation

In this section, we show the different activities and techniques we applied from a software engineering and development point of view.

As a basic infrastructure, we used state-of-the-art tools integrated as shown in Figure [14](#). The source code of all RADON artifacts is hosted on GitHub and our build system is a self-hosted Jenkins instance that retrieves the source code for further processing. Apart from building and testing, Jenkins also pushes those tools that are packaged as Docker containers to DockerHub after every successful build. To display the current build status, test status, and code coverage metrics, we use *shield.io*, which provides badges for inclusion on the respective repositories' README file. In addition, ReadTheDocs obtains documentation from the repositories and compiles the respective ReadTheDocs pages to well-structured documentation pages.²

¹ <https://prometheus.io/>

² <https://continuous-testing-tool.readthedocs.io>

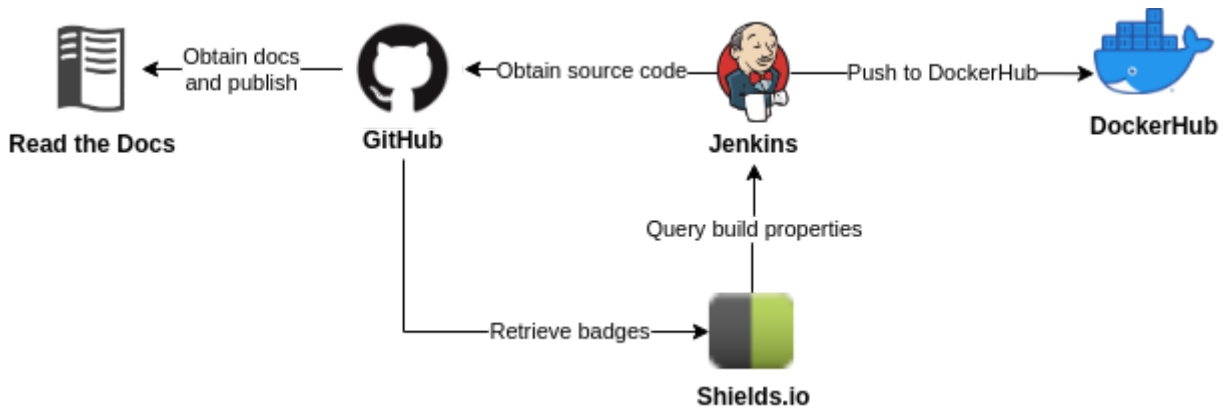


Figure 14. CI/CD and development workflow

The development of our contributions included several points of interactions, within our team, as well as our institution, work package, and other partners within the project.

We had regular stand-up meetings within our team and on-demand synchronization calls within our institution. On the work-package level, we attended the bi-weekly WP3 calls with regular status reports and discussions on the ongoing work. Apart from that, we used the project board and issue feature of GitHub to manage and assign tasks and track commits.³

In order to discuss topics and issues with other partners, crossing the boundaries of single tools, we used GitHub issues and pull requests, as well as the Slack instance that has been set up for the RADON project. In case a more thorough discussion was required, the discussions were moved to work package or use case calls, or were discussed in specifically set up meetings for that matter.

Table 1 lists all the CTT-related GitHub repositories. While `radon-ctt` and `radon-ctt-agent` have been mentioned in previous deliverables, the others have been added in order to structure and extend additional features and provide easy-to-use examples to get started with CTT.

<u>Tool Repositories</u>	
radon-ctt	<p>CTT Server GitHub: https://github.com/radon-h2020/radon-ctt DockerHub: https://hub.docker.com/r/radonconsortium/radon-ctt (Total pulls: ~2.1k)</p> <p>RADON CTT Server implementation.</p>
radon-ctt-agent	<p>CTT Testing Agent GitHub: https://github.com/radon-h2020/radon-ctt-agent DockerHub: https://hub.docker.com/r/radonconsortium/radon-ctt-agent (Total</p>

³ <https://github.com/orgs/radon-h2020/projects/2>

	<p>pulls: ~3.4k)</p> <p>CTT Testing Agent implementation that provides an API to control TI instances.</p>
radon-ctt-agent-plugins	<p>CTT Testing Agent Plugins GitHub: https://github.com/radon-h2020/radon-ctt-agent-plugins</p> <p>CTT Testing Agent Plugins extending the CTT Testing Agent by functionalities and additional endpoints specific to the testing tools used.</p>
radon-ctt-cli	<p>CTT CLI Client GitHub: https://github.com/radon-h2020/radon-ctt-cli</p> <p>CTT CLI Client interacting with a CTT Server instance to execute a CTT testing workflow based on a configuration file.</p>
radon-ctt-ide-plugin	<p>CTT RADON IDE Plugin GitHub: https://github.com/radon-h2020/radon-ctt-ide-plugin</p> <p>CTT IDE Plugin providing the interaction with a CTT Server instance from inside the RADON IDE to trigger the CTT testing workflow.</p>
radon-ctt-devops	<p>CTT DevOps Load Testing GitHub: https://github.com/radon-h2020/radon-ctt-devops</p> <p>CTT's functionality for the integration with the research approaches on DevOps-oriented load testing.</p>
<u>Demo Repositories</u>	
demo-ctt-imageresize	<p>CTT ImageResize Demo GitHub: https://github.com/radon-h2020/demo-ctt-imageresize</p> <p>Repository containing the ImageResize example including required artifacts for testing them with CTT.</p>
demo-ctt-sockshop	<p>CTT SockShop Demo GitHub: https://github.com/radon-h2020/demo-ctt-sockshop</p> <p>Repository containing the SockShop example including required artifacts for testing them with CTT.</p>
demo-ctt-todolistapi	<p>CTT ToDoList-API Demo GitHub: https://github.com/radon-h2020/demo-ctt-todolistapi</p>

	Repository containing the ToDoList-API example including required artifacts for testing them with CTT.
demo-data-pipeline-agent-testing	<p>Data Pipeline testing agent Demo GitHub: https://github.com/radon-h2020/demo_data_pipeline_agent_testing_project</p> <p>Repository containing a data pipeline for file transfer between two AWS S3 buckets for validating the customized data pipeline testing agent.</p>
demo-ctt-datapipeline-imageresize	<p>Data Pipeline for Thumbnail Generation Demo GitHub: https://github.com/radon-h2020/demo-ctt-datapipeline-imageresize</p> <p>Repository containing a data pipeline for the thumbnail generation demo lab application including required artifacts for testing them with CTT.</p>
demo-ctt-datapipeline-filetransfer	<p>Data Pipeline for File Transfer between different Cloud Storages demo GitHub: https://github.com/radon-h2020/demo-ctt-datapipeline-filetransfer</p> <p>Repository containing a data pipeline for the file transfer between different cloud storage (Amazon S3 and Google Cloud) demo lab applications including required artifacts for testing them with CTT.</p>

Table 1. Table of source code repositories and related resources

In addition to the integration tests, described in [D2.7], we use unit tests on the API level of CTT. That means that tests are executed against the API endpoints of CTT to ensure that the endpoints work as intended and additionally computes the test coverage. These tests are included in our Jenkins builds as can be seen in Figure 15.

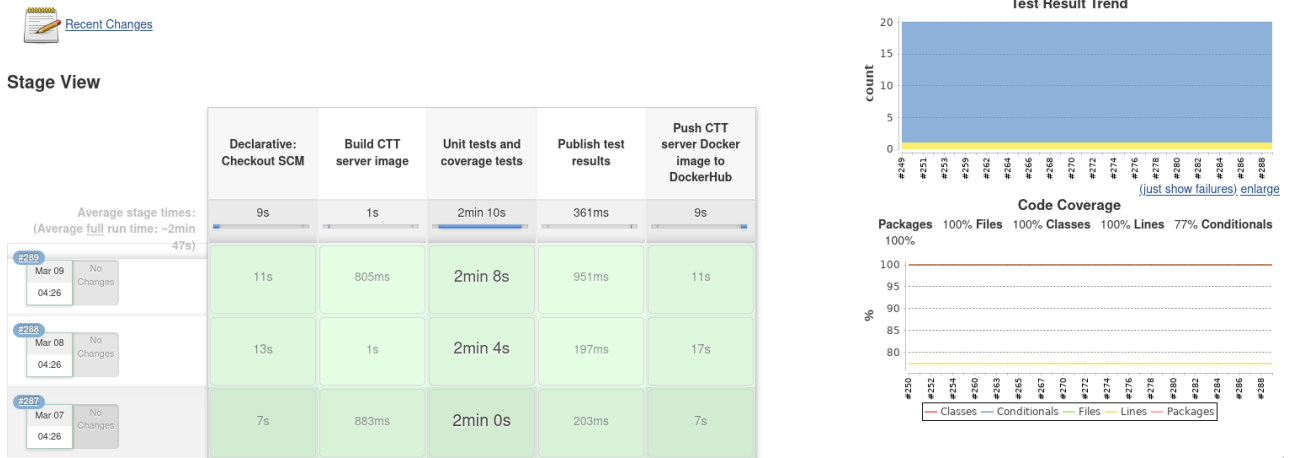


Figure 15. Test results and coverage displayed in Jenkins

4 DevOps-oriented microservices/FaaS load testing

In the context of the RADON project and the activities around CTT, we have developed novel research approaches for continuous testing of performance-related properties to be used in the DevOps context. This chapter aims to provide an overview of the approaches and describe their integration with CTT and RADON.

The approaches are aligned with our vision on automating representative load testing in continuous software engineering [SAH+18] and the findings of an industry survey [BEF+19] conducted with collaborators from the Standard Performance Evaluation Corporation Research Group (SPEC RG). The approaches are designed to fit into the wider goal of supporting users that are non-experts in load testing (e.g., [OBM+20], [SOH+19]).

Key characteristics of the approaches are:

- Inclusion of operational profile data on system usage in the production environment.
- Inclusion of architectural knowledge obtained from production monitoring data.
- Option to rely on (semi-)automatically extracted and evolved test scripts (building on [VHS+18], [SHW20]).
- Selecting and prioritizing relevant test cases with a focus on representativeness.
- Automation in executing tests.

Apart from the concrete approaches, we have jointly (again with SPEC RG) empirically studied benefits and challenges of load testing for microservices [EBS+20] and are currently conducting a longitudinal study for serverless applications.

The remainder of this chapter summarizes the approaches for microservice-tailored workload generation (Section 4.1), context-tailored workload generation (Section 4.2), domain-based scalability testing (Section 4.3), and automated testing to estimate interference between co-located microservices (Section 4.4), along with their relation to and technical integration with CTT. The following sections include contents from the respective publications.

4.1 Microservice-tailored workload generation

A representative load test uses workload characteristics according to the user behavior in production. Session-based systems have special workload characteristics as the system is used as sequences of inter-related requests. Approaches exist to automatically extract session-based workload models from production request logs. However, they focus on system-level testing, which is in stark contrast with modern development practices, where one development team is in charge of developing, testing, and deploying a single microservice. Hence, representative session-based workload models for testing single microservices and their integration are desirable. To deal with these issues, we propose a concept for tailoring a representative load test workload to target only certain services, instead of targeting the whole system. Our goal is to transform the workload for

one or more specified service(s) from the system-level workload collected in production. Using this approach, only a subset of the application’s microservices is deployed for a load test, specifically the targeted services and the services they depend on. We propose two algorithms. The log-based algorithm deals with extracting the workload for a specific service from collected production traces. The model-based algorithm performs the workload tailoring on the level of the workload model. Figure 16 illustrates the approach.

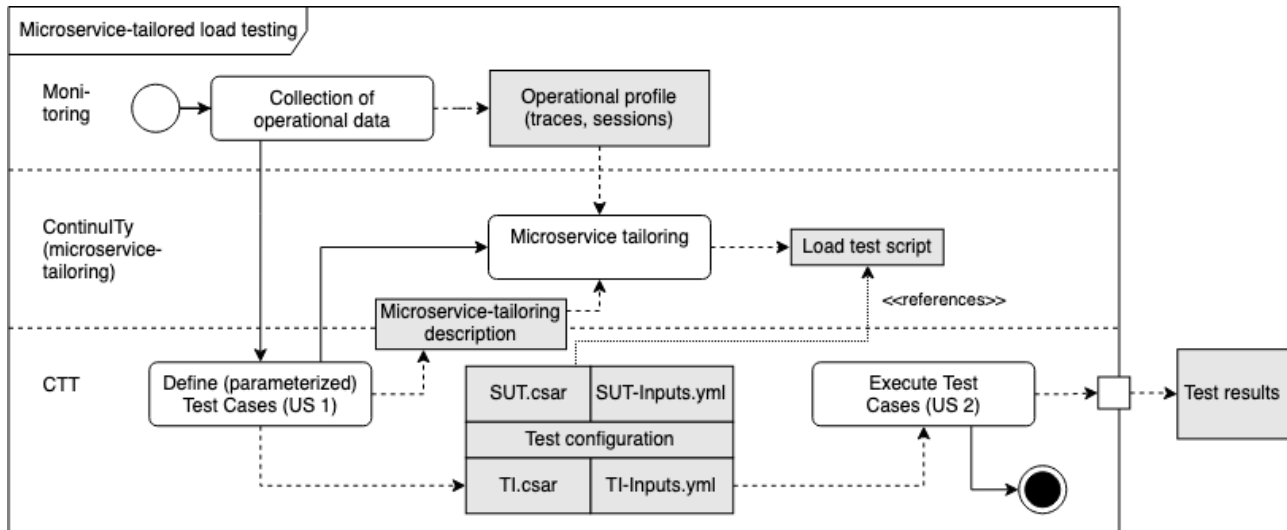


Figure 16. Microservice-tailoring approach integration in the CTT context

In an experiment series with the SockShop microservice application, we compare both algorithms with system-level and request-based workload models. The results show that when load testing a set of services, the tailored workload models outperform untailored workload models in terms of test duration and the capacity of the test infrastructure, and outperform request-based workload models in terms of representativeness.

The approach has been developed in a joint effort with colleagues from Novatec Consulting GmbH and the parts specific to context tailoring are implemented in tools in the Continuity ecosystem.⁴ Details on this approach can be found in a paper published at the MASCOTS conference [SAD+19].

The integration of the microservice tailoring approach in the RADON ecosystem can be sketched as follows, as depicted in Figure 16:

- The microservice-tailoring tool tailors the load test scripts following the presented approach and generates the load test scripts.
- RADON users equip the SUT’s model, which in this case targets a subset of services, with a load test definition, referencing the tailored load test script.
- CTT executes and reports the load tests.

⁴ <https://github.com/Continuity-Project>

4.2 Context-tailored workload generation

Load tests evaluate software quality attributes, such as performance and reliability, by e.g., emulating user behavior that is representative of the production workload. Existing approaches extract workload models from recorded user requests. However, a single workload model cannot reflect the complex and evolving workload of today's applications, or take into account workload-influencing contexts, such as special offers, incidents, or weather conditions. We propose an integrated framework for generating load tests tailored to the context of interest, which a user can describe in a language we provide. The framework applies multivariate time series forecasting for extracting a context-tailored load test from an initial workload model, which is incrementally learned by clustering user sessions recorded in production and enriched with relevant context information. Figure 17 illustrates the approach.

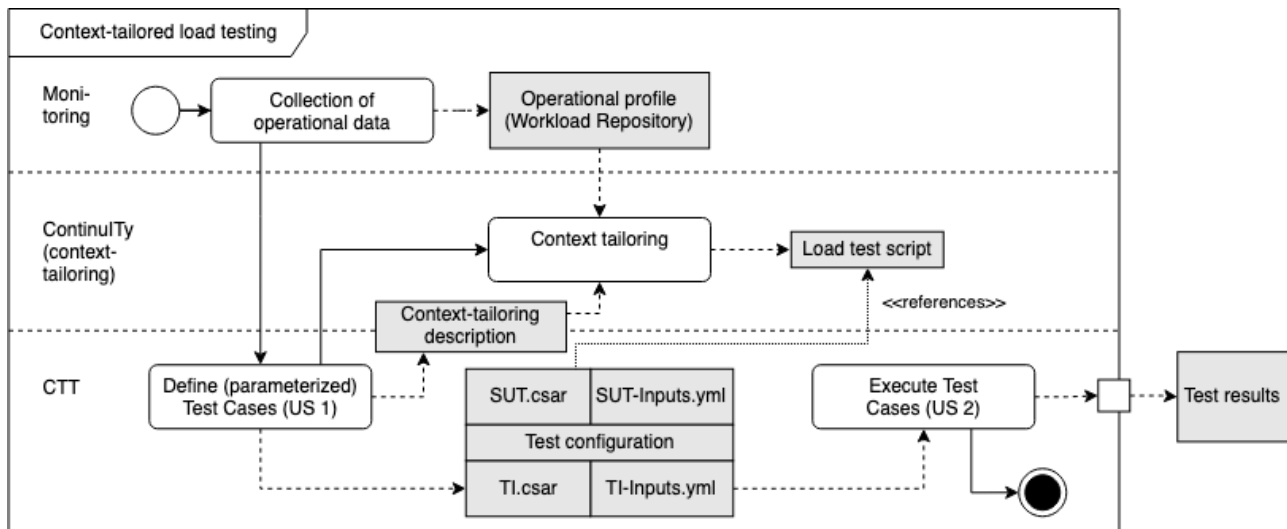


Figure 17. Context-tailoring approach integration in the CTT context

We evaluated our approach with the workload of a student information system. Our results show that incrementally learned workload models can be used for generating tailored load tests. The description language is able to express the relevant contexts, which, in turn, improve the representativeness of the load tests. We have also found that the existing workload characterization concepts and forecasting tools used are limited in regard to strong workload fluctuations, which needs to be tackled in future work.

The approach has been developed in a joint effort with colleagues from Novatec Consulting GmbH and the University of Prague and the parts specific to context tailoring are implemented in tools in

the ContinUITy ecosystem.⁵ Details on this approach can be found in a paper published at the ICPE conference [SOH+21].

The integration of the context tailoring approach in the RADON ecosystem can be sketched as follows, as depicted in Figure 17:

- RADON's Monitoring tool provides the raw operational profile data in form of time series, which can be used in the Workload Model Repository.
- RADON users equip the SUT's model with a load test definition, referencing a parameterized load test script and a context-tailoring description.
- The context-tailoring tool tailors the load test scripts following the presented approach.
- CTT executes and reports the load tests.

4.3 Domain-based scalability testing

Assessing the performance of architecture deployment configurations — e.g., with respect to deployment alternatives — is challenging and must be aligned with the system usage in the production environment. We introduced an approach for using operational profiles to generate load tests to automatically assess scalability pass/fail criteria of microservice configuration alternatives. The approach provides a Domain-based metric for each alternative that can, for instance, be applied to make informed decisions about the selection of alternatives and to conduct production monitoring regarding performance-related system properties, e.g., anomaly detection.

The first step is the collection of data about the system's operational profile, which can be obtained from common APM tools [HHM+17]. The operational data of interest are time series of workload intensities, e.g., number of concurrent sessions, requests rates to microservices or functions. This operational data is transformed into an empirical distribution of workload situations, which essentially models the probability of occurrence of the respective workload levels, e.g., 200-400 requests per second in 4% of the time. In parallel, the assumption is that load test scripts are (semi-)automatically created and evolved using the previously mentioned approaches [VHS+18], [SHW20]. The extracted scripts serve as load test templates, which are parameterized by the respective workload level to be tested. The load test tool handles the execution of the test series, comprising the loop of deploying the SUT and the TI, the actual load test execution, and the data collection. Finally, the domain-based metric is calculated from the test results and visually presented in a dashboard.

We have evaluated our approach using extensive experiments in a large bare-metal host environment and a virtualized environment using the SockShop application with operational profile data from a streaming service and Wikipedia. The findings support the need to carefully evaluate the impact of increasing the level of computing resources on performance. Specifically, we

⁵ <https://github.com/ContinUITy-Project>

observed that the evaluated Domain-based metric is a non-increasing function of the number of CPU resources for one of the environments under study.

The parts of the approach that are specific to the domain-based metric computation are implemented in the PPTAM tool⁶ [AFJ+20], which is a joint effort with collaborators from the University of Bozen-Bolzano and Esulab Solutions. Moreover, we collaborated with the University of Lugano on the integration of the BenchFlow [FP18] test orchestration tool, which in the RADON case is replaced by CTT. In the approach, we are supporting additional load test tools such as Faban and Locust.

Details on this approach and its evaluation can be found in a paper published in the JSS journal [AFJ+20].

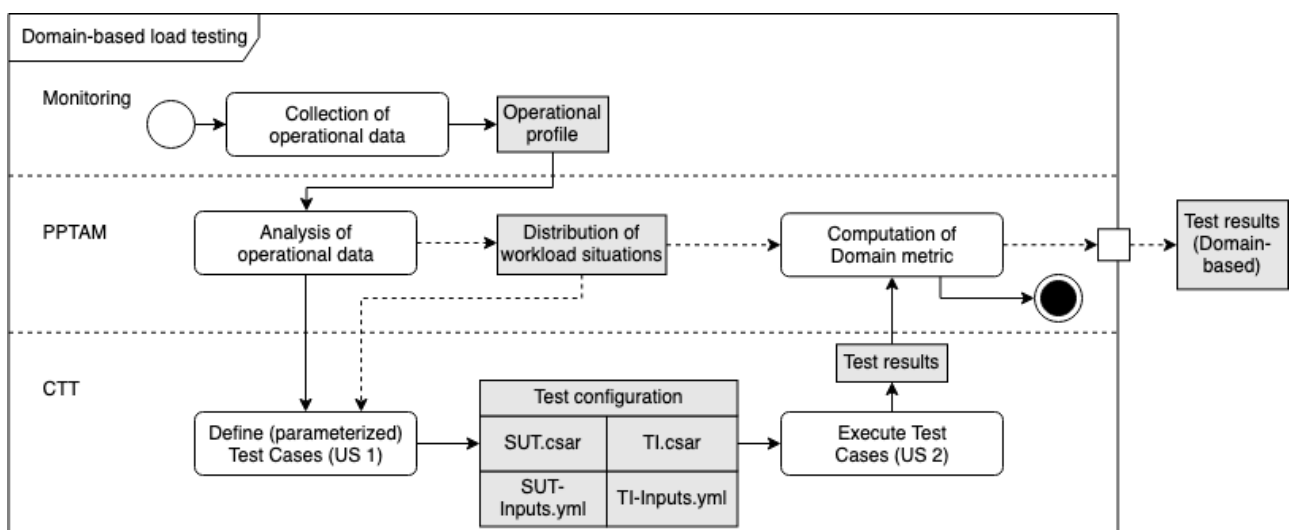


Figure 18. Domain-based scalability approach integration in the CTT context

The integration of the domain-based scalability testing approach in the RADON ecosystem can be sketched as follows, as depicted in Figure 18:

- RADON’s Monitoring tool provides the raw operational profile data in form of time series. The time series data needs to be imported into a format that is suitable for the PPTAM tool. CTT provides functionality to import the data from RADON’s Prometheus-based Monitoring tool. The PPTAM tool converts the source data into the distribution of workload situations, later used for the computation of the domain-based metric.
- RADON users equip the SUT’s model with a multi-iteration load test definition, using a parameterized load test script with the intended workload levels to be tested.
- After the execution of the load test(s), the results are imported into PPTAM to compute the domain-based metric.

⁶ <https://github.com/pptam>

In a subsequent series of experiments, we investigate the application of the approach to assess the impact of security attacks on the performance of architecture deployment configurations [AFJ+20]. In a follow-up work, we have leveraged the domain-based scalability testing approach for classifying performance anti-patterns [ABT+21]. Moreover, we have studied the applicability of the approach to other application domains [ACJ+21].

4.4 Automated testing to estimate interference between co-located microservices

In the production environment, the data center has obtained significant popularity as a cost-efficient platform. Nevertheless, the conventional data center has enormous over-provisioned computing resources for production applications that accommodate fluctuating workloads and peak demands. Most of the time, the conventional data centers in production suffer from overprovisioning and have low utilization within the computing resources.

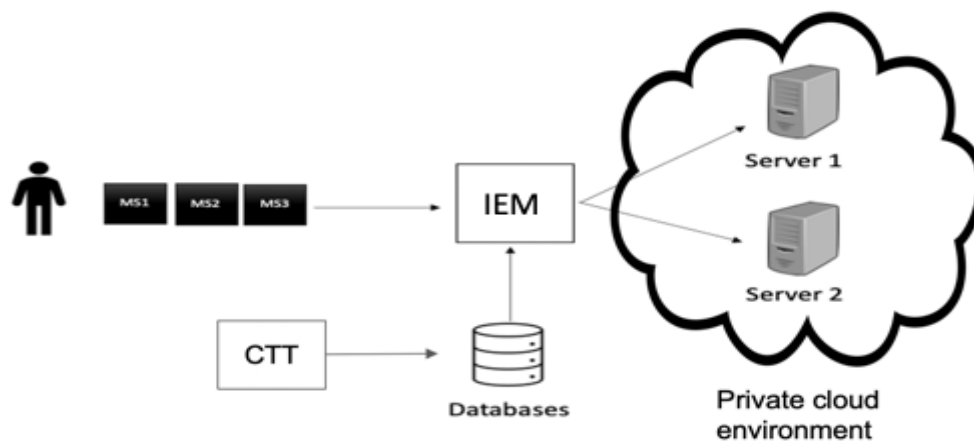


Figure 19. IEM model to estimate interference between co-located microservices

In this part of the RADON project, we develop an automated interference estimation modeling testing extension (IEM) to estimate interference between co-located microservices within the same computing environment. An AI-driven model is implemented to predict the services' interference, which may cause low system utilization or saturated system resources. This extension will alleviate and estimate the task slowdown affected by the interference among running services. The AI model is initially constructed through offline training datasets that are collected from the microservices system. The model estimates the job completion time to enhance the job placement within the microservices environment and predict any possible risk to take action before performance degradation arises. Many approaches in the literature collect all the possible performance metrics of each job and test each job behavior under different resource configurations to predict the system performance. These approaches are time-consuming for dynamic complex workloads to test all combinations of jobs behaviors needed to be profiled in advance under various system

configurations. In addition, the other co-located jobs may affect each other during the run time, which makes it hard to cover all the possible combinations of workload behaviors. The approach is depicted in Figure [19](#).

The proposed extension is agnostic of the business logic internal to each job. Instead, it is learned from data applying artificial neural networks. The target is to establish the completion time prediction error. We assume to have profiling data for individual jobs and two jobs, then attempt to predict the completion times when four jobs or more run simultaneously in the system. The prediction method works without any need for measurement from executions with four jobs; everything is predicted using only the profiling data and the AI model. Neural network algorithms with backpropagation and conjugate gradient are used to train the neural networks. Feature selection may include the following as input features to the neural networks for training purposes: hyperthreading, CPU nice, RunQ, throughput, type of job. Part of this work is published in ICCAIS 2021 [\[A21\]](#).

5 Data pipeline testing

The main features supported by the current version of the CTT module are defining, generating, executing tests, and reporting test results that can be directly applied. However, generating test data and collecting metrics of the data pipeline under test requires special attention, as data pipeline applications are rather different from serverless FaaS or microservice applications. In the RADON approach, designing and orchestrating data pipelines (discussed in deliverables [\[D5.5\]](#) and [\[D5.6\]](#)), the whole pipeline is divided into several sub-tasks, where each sub-task acts as an independent pipeline block. This represents that the whole application is composed of multiple independently deployable, scalable, and schedulable tasks, and it is important to extract metrics for each of them while testing the whole data pipeline to identify the performance accuracy and other issues located in the pipeline.

Further, the CTT data-pipeline module provides support for exposing monitoring data and generating metrics at the level of individual data pipeline blocks, efficiently generating data for load testing data pipelines, and defining and setting up corresponding test infrastructures. In this section, we provide a detailed description of the architecture and implementation of the CTT data-pipeline module.

5.1 CTT data pipeline architecture including core design decision

This section highlights the overview of the workflow (Section [5.1.1](#)) followed by the technical architecture and interaction with the environment (Section [5.1.2](#)) of the CTT-data pipeline module.

5.1.1 Workflow

In order to test the data pipeline application, the user is supposed to design a data pipeline service that she intends to test (SUT) in GMT. This SUT is then annotated with details about the tests by defining a testing policy in GMT. Based on the types of the tests included in the policy, the test infrastructure (TI) is created by designing in GMT again. The SUT and TI are TOSCA service templates that should be exported to a Git repository. The data pipeline testing workflow can be started when these artifacts are ready to execute.

The data pipeline workflow is otherwise very similar to a normal CTT workflow. The first step is to create a project using the Git repository information and a custom name for the project. In the second step, test artifacts are generated by selecting the previously created TOSCA models for the SUT and TI. After generating the test artifacts, in the next step, i.e., in the deployment step both the SUT and TI are prepared for the actual test. Further, the execution step is triggered for completing the CTT data-pipeline module workflow after the deployment step. Finally, the result set of the

tests is created by collecting the metrics, which can be downloaded for further processing and inspection.

5.1.2 Technical architecture and interaction with the environment

A high-level architecture of the CTT data-pipeline module and its interaction with the data pipeline application in terms of artifacts, tools, and infrastructures are depicted in Figure 20. The main purpose of the CTT data-pipeline module is to provide support for exposing monitoring data and generating metrics at the level of individual data pipeline blocks, efficiently generating data for load testing, and defining and setting up corresponding test infrastructures. Initially, a user defines a data pipeline test application by adding them to a TOSCA service template for the SUT. Further, CTT data-pipeline module specific TOSCA node types and policy types are defined in the RADON Particles template library for expressing different types of tests and TIs.

The current TOSCA pipeline models are based on Apache NiFi. For instance, CTT allows the definition of load test to be executed on the data pipeline module by a TI composed of a load driver such as JMeter or Apache NiFi load injector. CTT imports the TOSCA model of the data pipeline application and provides possibly refined TOSCA models with the SUT and TI. The refinement includes various approaches for testing the data pipeline blocks including performance testing and load testing by monitoring run-time metrics. To enable extracting fine-grained run-time metrics, we have utilized the RADON monitoring system (based on Prometheus). We have evaluated that it is possible to observe both Apache NiFi platform instance resource (e.g., CPU, RAM) metrics and data-pipeline run-time metrics (e.g., amount of bytes written, amount of byte read, amount of flowfile sent, amount of flowfile received) in real-time. Finally, the tests are executed after being deployed by the TOSCA orchestrator, i.e., xOpera, and the test results are made available to the user in the form of the raw test results or a revised test report.

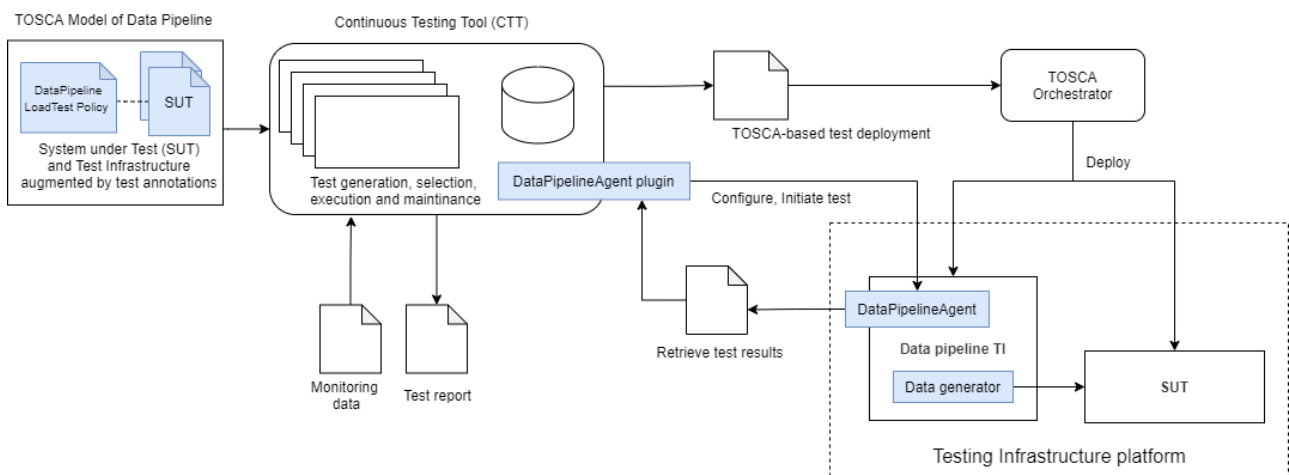


Figure 20. High-level architecture of CTT-data pipeline module [D3.4]

5.2 Technological and Development Decisions

- The demo lab application, thumbnail generation data pipeline, involved uploading images to the source S3Bucket of the data pipeline and hence, the JMeter script needed to be modified by adding a JSR233 Sampler. The agent is provided with the AWS SDK for uploading images into the source S3Bucket using JMeter for performance testing, discussed in Section [5.3.3](#).
- A new Test Infrastructure is designed using a NiFi pipeline, which is used to test the SUT of the data pipeline applications by injecting additional loads, discussed in Section [5.3.3](#).
- The CTT agent is modified to adapt to the new Test Infrastructure. Python is used as the main programming language for the agent, discussed in Section [5.3.2](#).
- For performance testing, the metrics of individual data pipeline components are monitored using a PrometheusReportingTask and extract the fine-grained run-time metrics, discussed in Section [5.4](#).
- Python scripts have been used for velocity/volume testing of the data pipeline module after injecting additional loads through a load injector (JMeter/NiFi pipeline), discussed in Section [5.4](#).
- Github is used as a development platform for Git repositories including the source artifacts.
 - Demo lab application of thumbnail generation data pipeline SUT and TI for testing with JMeter based CTT testing agent:
<https://github.com/radon-h2020/demo-ctt-datapipeline-module>
 - Demo lab application of data pipeline for file transfer between different cloud storage SUT and TI for testing with JMeter based CTT testing agent:
<https://github.com/radon-h2020/demo-ctt-datapipeline-filetransfer>
 - Demo application data pipeline SUT and TI for testing with the customized Data Pipeline CTT testing agent:
https://github.com/radon-h2020/demo_data_pipeline_agent_testing_project
 - Software component repositories are outlined separately in Table [1](#)

5.3 Implementation view of CTT data pipeline module

The CTT data-pipeline architectural structure is decomposed into the components modeling types, creation of a custom testing module using a NiFi-based agent for testing data pipelines, and a load injector for performance testing and load testing of the data pipeline using Prometheus servers. Section [5.3.1](#) to [5.3.3](#) describes the three components of the CTT data-pipeline module.

5.3.1 Modelling types

CTT depends on the TOSCA modeling concepts in order to augment the TOSCA models of the SUT and to define the TI. Therefore, we need to define a CTT data-pipeline specific modeling type hierarchy for testing the data pipeline components.

The TOSCA policy types are used to model the test cases. The existing component types for test infrastructure can be reused and are modeled as TOSCA node types. The reusable test infrastructures are modeled as TOSCA blueprints. The implementation of CTT types are done in RADON Particles, which is the RADON template library containing reusable definitions and extensions, in this case for testing the data pipeline components.

The remaining subsections give an overview of the CTT data-pipeline related type hierarchy in the form of policy types, node types, and blueprints.

CTT data pipeline policy type

The parent type for the CTT test type is a test case in `Test`. The type includes a reference to a CTT TI blueprint and a test identifier as attributes. The blueprint is a TOSCA service template that defines the infrastructure for executing the test. The class diagram in Figure [21](#) shows the policy type hierarchy for the data pipeline module.

The type hierarchy includes an abstract policy type `LoadTest` for load testing of the data pipeline module. In this scenario, `JMeterLoadTest` and `DataPipelineLoadTest` are the concrete policy types that are used for load testing. In the case of `JMeterLoadTest`, the test cases include properties such as a reference to load the test script or a single zip file containing all the resources and additional properties. For the `DataPipelineLoadTest`, the test cases include the following properties:

- **velocity_per_minute**: how many objects should be generated per minute
- **test_duration**: How long the test should last in seconds
- **performance_metric**: Performance metric which should be specifically tracked, such as CPU Load
- **lower_bound**: lower bound for the tracked performance metric
- **upper_bound**: upper bound for the tracked performance metric
- **resource_location**: path to the zip file including test artifacts

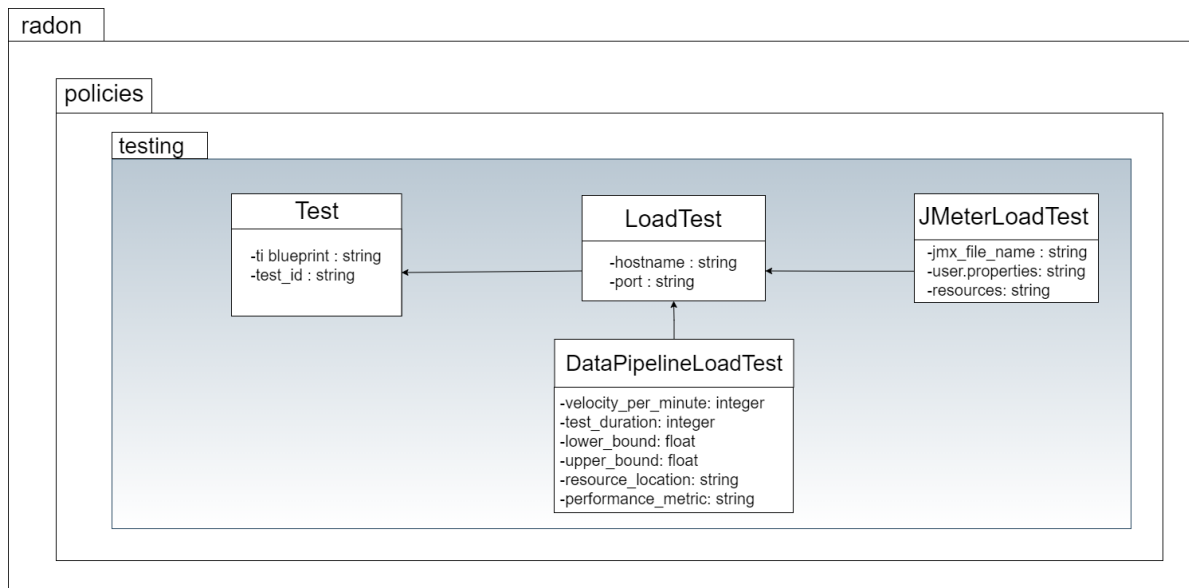


Figure 21. CTT-data pipeline policy types

Node types of the CTT data pipeline testing agents

CTTAgent is the (abstract) parent of any test infrastructure node type which is derived from RADON's type for Docker applications (DockerApplication). The class diagram of the CTT data-pipeline node type hierarchy is shown in Figure 22.

The CTT data-pipeline node type hierarchy includes node types for executing load testing of the data pipeline components. The LoadTestAgent type is abstract and serves as the basis for concrete node types for representing agents for load testing tools such as JMeter (JMeter). The JMeter node type includes attributes such as jmx file additional configuration properties.

The testing tool Data Pipeline Agent node includes defines how to deploy the Agent as a docker container and has no properties. The behaviour of the agent is configured through the testing Policy it receives from the CTT server.

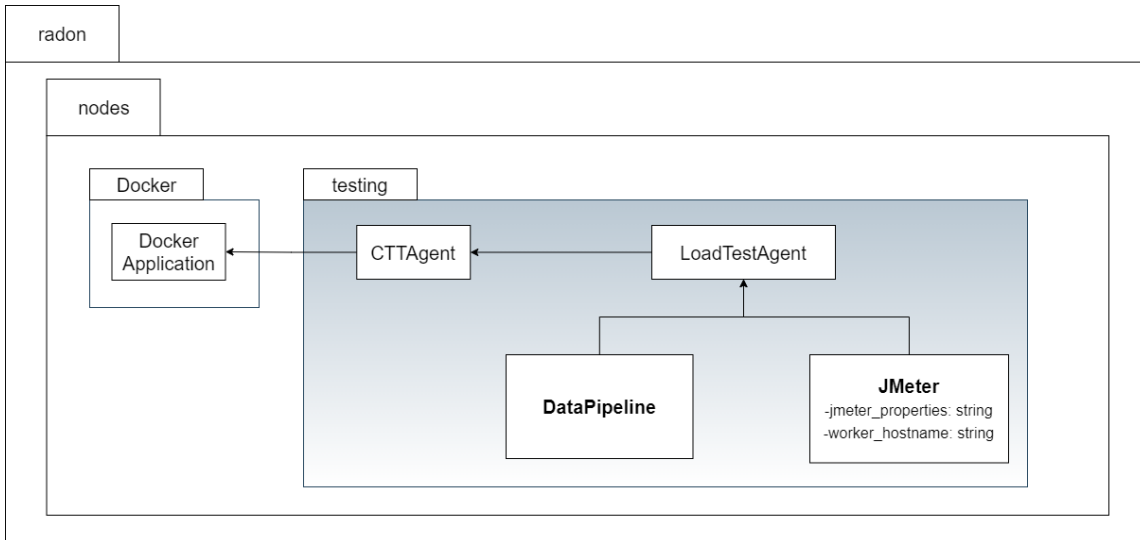


Figure 22. CTT-data pipeline agents node types

CTT data pipeline blueprints

Radon-particles repository contains a set of TI blueprints, which are TOSCA service templates to represent the test infrastructure required for executing the modeled tests which can be used by the CTT tool.

In the case of testing using JMeter, the blueprint `JMeterMasterOnly` is the TOSCA template that comprises a `JMeter` node type, deployed on a Docker Engine (`DockerEngine`) hosted on a Workstation (`Workstation`), where `DockerEngine` and `Workstation` are node types are available in RADON Particles [\[D3.4\]](#). Figure 23 shows the `JMeterMasterOnly` blueprint in GMT:

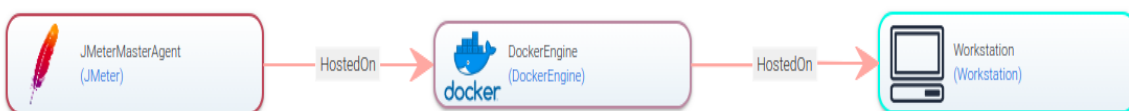


Figure 23. JMeterMasterOnly blueprint as shown in GMT

On the other hand, while testing the SUT using an Apache Nifi data pipeline, a TOSCA service template `NiFiTIDocker` is used, which consists of `ConsumeLocal` and `PubS3Bucket` data pipeline node types hosted on NiFi deployed as a Docker container inside an EC2 instance. It also includes the `DataPipelineAgent` as a Docker container, which is the CTT agent responsible to communicate with CTT server. Figure 24 shows the `NiFiTIDocker` blueprint in GMT.

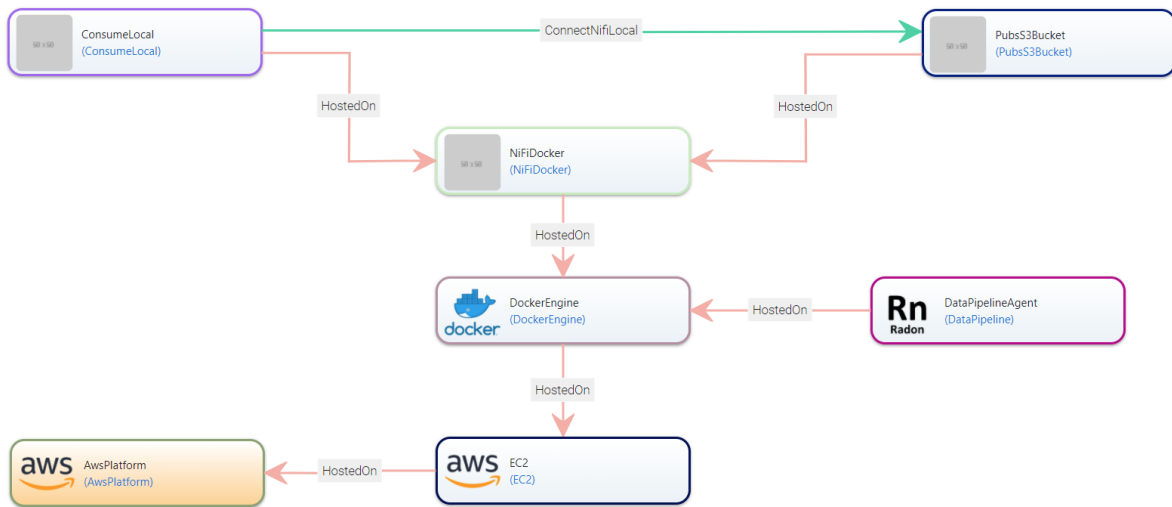


Figure 24. NiFiTIDocker blueprint as shown in GMT

5.3.2 CTT data pipeline agents

The existing features and extensibility of the CTT server makes it relatively straightforward to extend its capabilities with new CTT Agents that are deployed as completely separate software services. To implement how the testing is performed in the case of data pipelines, modifications to an existing CTT Agent and the creation of a new Agent was required.

In the case of `JMeterMasterOnlyblueprint`, we can reuse the existing JMeter CTT testing agent, which was not specifically designed for data pipeline testing. To modify the behaviour of the Agent, a custom JMeter test-plan implementation is created specifically for generating load data for the data pipeline under test (SUT), uploading the data to the data source used by the SUT and capturing performance metrics while the pipeline is being tested. However, this approach has the limitation that different data pipelines have different data sources and each new type of data source (e.g., S3 or MQTT) requires a custom JMeter test-plan implementation which is able to move generated data into the data source.

To avoid having to create custom JMeter test-plan implementations for every possible data source type, a new CTT Agent, `DataPipelineAgent`, was designed which utilizes the capabilities of the data pipeline node types to transfer generated load data to any of the data sources already supported by RADON data pipeline models. The `DataPipelineAgent` uses NiFi as the test infrastructure and two types of data pipeline blocks:

1. For generating data (e.g., `ConsumeLocal` for reading local files)
2. Transferring data to the required data sources. (e.g., `PubsS3` for sending data to an S3 bucket)

The data pipeline nodetypes are deployed automatically together with the agent by the RADON orchestrator. `DataPipelineAgent` initiates the data generation and is responsible for gathering test metrics, communicating with the CTT server and returning the test execution output to the CTT server once the test has finished.

5.3.3 Extension framework including load injector

To verify the performance of the data pipeline components in terms of multiple performance metrics including response time, resource utilization, etc, the CTT data-pipeline module is designed to be extensible in terms of possible integration of load injector such as JMeter based load injector and Apache NiFi-based load injector. The current TOSCA pipeline models are based on Apache NiFi. So, to validate the performance of the pipeline components, a load test is conducted by pushing multiple loads as a form of images or text data through the load injectors and measure the performance of the pipeline components. The technical details of two load injectors with a demo lab application are described in the following subsections.

JMeter-based load injector

For validating and testing the RADON Data Pipeline module, a demo application was considered, i.e. Thumbnail generation with data pipelines demo lab application, which divided the whole application into three pipelines: (i) reading images from an Amazon S3 bucket, (ii) invoking a lambda function with an image and receiving the corresponding thumbnail, and (iii) pushing the thumbnail to the same or different Amazon S3 bucket, as described in the deliverable [\[D6.2\]](#).

For load testing the demo lab application, the JMeter-based load injector is used to upload the images of a dataset stored in a folder to the AWS S3 bucket. The testing infrastructure comprises the node type `JMeter`, hosted on Docker and deployed on a Workstation. The JMeter node type requires certain configurations, which include the `.jmx` script as well as parameters like sample frequency, hostname, and port name.

A `JSR233` Sampler is used in the `.jmx` file, which contains a Java script, as shown in Figure [26](#). The script requires AWS credentials (Access Key and Secret Key) and the name and region of the AWS S3 bucket. The class `BasicAWSCredentials` is used for a basic implementation of the `AWSCredentials` interface that allows callers to pass in the AWS access key id and secret access key in the constructor. Also, the script uses `TransferManager`, which provides a simple API for uploading content to Amazon S3. The script reads from a folder that contains the path of the files or images to be uploaded, parses them by creating a new file name, and finally uploads them to the S3 bucket.

```

BasicAWSCredentials sessionCredentials = new BasicAWSCredentials(accessKey, secretKey);
AmazonS3 s3 = AmazonS3ClientBuilder.standard()
    .withRegion(region)
    .withCredentials(new AWSStaticCredentialsProvider(sessionCredentials))
    .build();
TransferManager xfer_mgr = TransferManagerBuilder.standard()
    .withS3Client(s3)
    .withDisableParallelDownloads(false)
    .build();

File f = new File(args[0]);
int index = args[0].lastIndexOf("/");
String fileName = args[0].substring(index + 1);
String obj = objectName + fileName;
Upload xfer = xfer_mgr.upload(bucketName, obj, f);
xfer.waitForCompletion();
xfer_mgr.shutdownNow();

```

Figure 25. JMeter Script for uploading files or images to S3 Bucket for load testing

Apache NiFi-based load injector

The second approach for testing the SUT involves another testing infrastructure, which contains a NiFi platform and related data pipelines hosted on an AWS (or OpenStack) instance. The major components of the pipeline are the node types `ConsumeLocal` and `PubS3Bucket`, which are used for injecting data into the SUT data pipeline. The `ConsumeLocal` node type reads files (e.g., images or JSON files) from a certain folder and hence it requires the path of the file as its configuration. On the other hand, files read by `ConsumeLocal` are sent to `PubS3Bucket` in order to upload them to the AWS S3 bucket. In this case, the bucket name, AWS region and AWS credentials (Access Key and Secret Key) file path need to be configured. Both of these nodes are deployed inside a NiFi instance and hence, the NiFi node type is used, where the version of Apache NiFi is specified. The NiFi node type is hosted on an EC2 node type, where the SSH key name and user is specified along with some other parameters like instance name, image, instance type and network. But the NiFi node type can be installed on any other platform that supports deploying Virtual Machines (e.g., OpenStack) or even as a Docker container. Also, the testing infrastructure can be easily modified using RADON GMT to switch `PubS3Bucket` out for any of the other RADON supported data pipeline nodes that send data to external systems to test data pipelines which use data sources such as MQTT, GridFTP, Google Cloud Storage, Azure bucket or SFTP.

5.4 Volume/velocity testing of CTT data pipeline module

To verify the performance of the data pipeline components in terms of multiple performance metrics including response time, resource utilization, etc., after pushing the high amount of requests through some load generation tools including JMeter and NiFi, we conduct a volume/velocity testing of the CTT data-pipeline module. For volume/velocity testing, a large volume of data needs

to be generated in various time frames using the above-mentioned two types of load injectors. In an initial lab validation, we have used JMeter to push a set of images as input to the Thumbnail generation with the data pipelines demo application, as depicted in deliverable [\[D6.2\]](#). Besides that, for generating a heavy load of image or files with different volume and velocity, we have designed an Apache NiFi load injector for load testing of the data pipeline. During load testing, it is not sufficient to generate load to the input data source and measure response time as this will measure the performance of the data source or only the start of the pipeline. It is important to gather run-time metrics for all the data pipeline components.

During volume/velocity testing, the main purpose of the CTT-data pipeline module is to provide support for exposing monitoring data and generating metrics at the level of individual data pipeline blocks, efficiently generating data for load testing data pipelines, and defining and setting up corresponding test infrastructures. The current TOSCA pipeline models are based on Apache NiFi and we have extracted the fine-grained run-time metrics (e.g., amount of bytes written, amount of byte read, amount of flowfile sent, amount of flowfile received) of the individual data pipeline component using `PrometheusReportingTask`.

In order to test the performance of the demo lab application, thumbnail generation data pipeline, initially, the CSAR files for SUT and TI are generated using the Graphical Modelling Tool (GMT), which are later deployed through the xOpera orchestrator. After the deployment of the SUT and TI, images from a dataset are sent to the source S3Bucket through JMeter or Nifi data pipeline in the TI. It is necessary to evaluate the contents of both the S3Buckets through volume testing. In such scenario, a Python script has been used to count the contents of both the S3Buckets as well as to compare the number of images uploaded in the source S3Bucket through JMeter/Nifi and the number of images uploaded to the destination S3Bucket after thumbnail generation using an AWS Lambda function. Besides that, other performance metrics including time of upload images into S3Buckets and execution time, involved in generating and uploading of thumbnails into the destination S3Bucket can also be retrieved. Velocity testing is also performed adjusting the sample frequency i.e., the number of images injected per minute to the S3Bucket by the JMeter or Nifi pipeline. Through a Python script the upload timestamps of the images in the source S3Bucket are checked and the number of images uploaded per minute is calculated. The results of all these tests are generated and stored in the form of .csv files for visualization. The overview of the methodology and framework for performance testing of the thumbnail generation data pipeline demo lab application is described in Figure [26](#). The detailed validation results of volume/velocity testing of the CTT-data pipeline module with different demo applications with additional datasets will be described in the last deliverable of WP6 in *M30*, i.e., “D6.5 Final Assessment Report”.

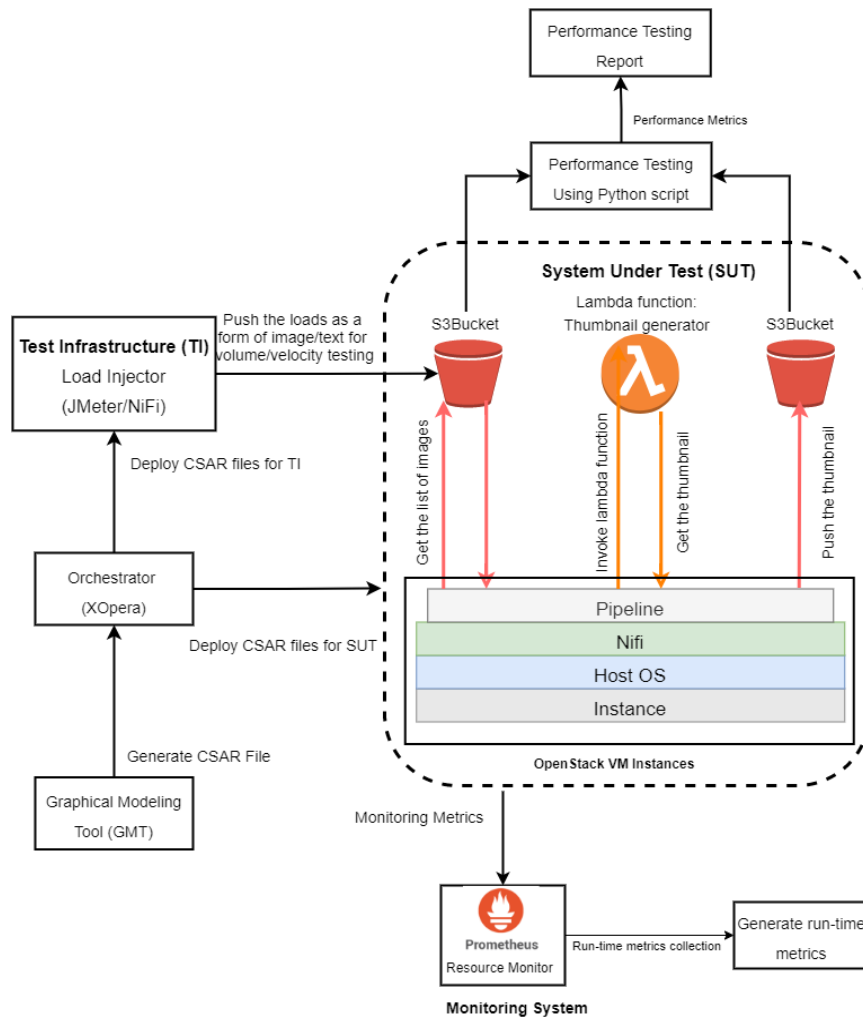


Figure 26. Volume/Velocity testing of of data pipeline module

6 Conclusions

This deliverable concludes the development of the Continuous Testing Tool (CTT).

At this stage, together with the developments reported in the initial deliverable [\[D3.4\]](#), CTT supports:

- The definition of tests and test infrastructures using a CTT-defined hierarchy of TOSCA policy types, node types, and blueprints, which is available in RADON's particles template library.
- The workflow of importing CTT-annotated CSAR files, as well as deploying (via the xOpera orchestrator) and executing the tests, and accessing the raw test results.
- A set of included deployment and load tests using well-known tools such as JMeter.
- The extension of functionality, e.g., new test types and tools, via CTT's extension points.
- A module dedicated to testing of data pipelines including new modeling types, a modified data pipeline agent, a data injection framework with components based on Apache JMeter and NiFi, and overview of velocity/volume testing.
- The use of CTT by the pre-compiled Docker images, and interacting directly via the REST-based API, via a CLI tool (e.g., for using it in CI/CD scripts), or the RADON IDE via a CTT plugin.
- The integration of selected approaches for DevOps-oriented load testing of microservices and FaaS, including the use of operational data, e.g., provided by RADON's Monitoring tool.

Appendix: Compliance with requirements

The following tables summarize the level of CTT’s compliance with the requirements at this stage, following the categories introduced in Section 2. In parentheses in gray, we list the level of compliance as reported in the initial deliverable [D3.4]. Moreover, in each case, we provide a brief statement explaining the (changed) level of compliance.

The labels specifying the “Level of compliance” are defined as follows:

- **Mnn** (scheduled): the requirement is not achieved by the current version; a level of ✓✓ is planned for month *nn*,
- ✓ (partially-low achieved): the requirement is partially-low achieved by the current version,
- ✓✓ (partially-high achieved): the requirement is partially-high achieved by the current version,
- ✓✓✓ (fully achieved): the requirement is fully achieved by the current version.

Usage Scenarios

Table 2. Achieved level of compliance to the CTT usage scenarios

Id	Requirement Title	Priority	Level of compliance
US 1	Define Test Cases	MUST_HAVE	✓✓✓ (✓✓✓)
The usage scenario was fully achieved with the previous deliverable.			
US 2	Test Execution	MUST_HAVE	✓✓✓ (✓✓)
The usage scenario is now fully achieved by the extensions presented in this deliverable that improve test execution.			
US 3	Test Maintenance	MUST_HAVE	✓✓✓ (✓)
The usage scenario is now fully achieved by the extensions presented in this deliverable, particularly the integration with monitoring and the presented approaches for microservices/FaaS and data pipelines.			

Requirements List

Table 3. Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Level of compliance
FaaS/microservices module			

R-T.3.3-10	The FaaS testing module of the TESTING_TOOL must support the generation of test cases from RADON models that are augmented by the test-related annotations.	Must	✓✓✓ (✓✓)
The requirement is now fully achieved by the extensions presented in this deliverable.			
R-T.3.3-11	The FaaS testing module of the TESTING_TOOL must support the execution of tests cases in the CD pipeline	Must	✓✓✓ (✓✓✓)
The requirement was fully achieved with the previous deliverable.			
R-T.3.3-12	The FaaS testing module of the TESTING_TOOL must be able to analyze monitoring data from production and update the annotations in the RADON model	Must	✓✓✓ (✓)
The requirement is now fully achieved by the extensions presented in this deliverable (Section 4)			
R-T.3.3-13	The FaaS testing module of the TESTING_TOOL could have a report feature	Could	✓✓✓ (✓✓)
We consider this requirement as fully achieved in the scope of the project. Compared to the previous deliverable, the contributions are in the RADON IDE plugin and the domain-based dashboard report.			
R-T.3.3-14	The FaaS testing module of the TESTING_TOOL must have a graphic user interface.	Must	✓✓✓ (✓)
The requirement is now fully achieved with the CTT IDE plugin.			
R-T.3.3-15	The FaaS testing module of the TESTING_TOOL should have a command line interface.	Should have	✓✓✓ (✓✓✓)
The requirement was fully achieved with the previous deliverable.			
R-T.3.3-16	The FaaS testing module of the TESTING_TOOL must be integrated into DevOps practices	Must	✓✓✓ (✓)
The requirement is now fully achieved by the extensions regarding the research approaches for DevOps-oriented load testing.			
Data pipeline module			
R-T.3.3-2	The data pipeline testing module of the TESTING_TOOL should support user configurable data production profiles	Should	✓✓✓ (✓)

The requirement is now fully achieved by incorporating data pipeline agents for load/performance testing. Users can modify the velocity of data and the type of data objects that are injected.			
R-T3.3-3	The data pipeline testing module of the TESTING_TOOL must be able to set up synthetic stream data generation for data pipelines under test	Must	✓✓✓ (✓)
The requirement is now fully achieved with the modified data pipeline agent, which injects data into the data source of the SUT data pipeline based on user configured velocity and type to data to be generated.			
R-T3.3-4	The data pipeline testing module of the TESTING_TOOL should be able to analyze log data of the data pipeline under test to generate performance metrics	Should	✓✓✓ (✓)
The requirement is achieved by Prometheus (NiFi PrometheusReportingTask) and Python Scripts			
R-T3.3-6	It could be useful if users can configure upper and lower bounds in the data pipeline testing module of the TESTING_TOOL for the performance metrics that are computed	Could	✓✓✓ (M24)
The requirement is achieved, User is able to define which performance_metric is to be tracked inside the radon.policies.testing.DataPipelineLoadTest policy and also define lower_bound and upper_bound values for it inside the policy.			
R-T3.3-7	It could be useful for the data pipeline testing module of the TESTING_TOOL to have a graphical user interface for configuring tests and displaying test results	Could	✓✓✓ (✓)
The requirement is now achieved by allowing users to define test parameters in the RADON Graphical Modelling Tool and displaying the performance metrics through a graphical user interface like Grafana.			
R-T3.3-8	It could be useful for the data pipeline testing module of the TESTING_TOOL to support running multiple different tests in a sequence on the same data pipeline.	Could	✓✓✓ (M24)
The requirement is now fully achieved by the new CTT <i>Execution of multiple tests</i> feature and injecting loads through JMeter and Nifi pipeline with different datasets			
Modeling			
R-T4.2-7	The models must be able to include the description of test cases for certain components (annotate test-related information).	Must	✓✓✓ (✓✓✓)

The requirement was fully achieved with the previous deliverable.			
IDE			
R-T2.3-7	The IDE must provide support in launching the TESTING_TOOL and to trigger the execution of tests	Must	✓✓✓ (M24)
The requirement is now fully achieved with the CTT IDE plugin.			
R-T2.3-8	The IDE must provide access to a report based on the results received from the TESTING_TOOL	Must	✓✓✓ (M24)
The requirement is now fully achieved with the CTT IDE plugin.			

References

- [A21] Ahmad Alnafessah: Artificial Intelligence Approach for Batch Completion Time Prediction. ICCAIS 2021. Accepted for publication.
- [ABT+21] Alberto Avritzer, Ricardo Britto, Catia Trubiani, Barbara Russo, Andrea Janes, Matteo Camilli, André van Hoorn, Robert Heinrich, Martina Rapp, and Jörg Henß: *A multivariate characterization and detection of software performance antipatterns*. ICPE 2021. To appear
- [ACJ+21] Alberto Avritzer, Matteo Camilli, Andrea Janes, Barbara Russo, Jasmin Jahic, André van Hoorn, Ricardo Britto, and Catia Trubiani. PPTAM^λ: What, where, and how of cross-domain scalability assessment. ICSA 2021. To appear
- [AFJ+20] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, André van Hoorn, Henning Schulz, Daniel S. Menasché, Vilc Queupe Rufino: *Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests*. J. Syst. Softw. 165: 110564 (2020)
- [BEF+19] Cor-Paul Bezemer, Simon Eismann, Vincenzo Ferme, Johannes Grohmann, Robert Heinrich, Pooyan Jamshidi, Weiyi Shang, André van Hoorn, Mónica Villavicencio, Jürgen Walter, Felix Willnecker: *How is Performance Addressed in DevOps?* ICPE 2019: 45-50
- [D2.7] RADON Consortium, Deliverable D2.7: RADON integrated framework II, 2021
- [D3.1] RADON Consortium, Deliverable D3.1: RADON DevOps methodology, 2021
- [D3.4] RADON Consortium, Deliverable D3.4: Continuous testing tool I, 2020
- [D5.5] RADON Consortium, Deliverable D5.5: Data Pipeline Orchestration I, 2019
- [D5.6] RADON Consortium, Deliverable D5.6: Data Pipeline Orchestration II, 2020
- [D6.2] RADON Consortium, Deliverable D6.2: Initial Validation Results, 2020
- [EBS+20] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dusan Okanovic, André van Hoorn: *Microservices: A Performance Tester's Dream or Nightmare?* ICPE 2020: 138-149
- [FP18] Vincenzo Ferme, Cesare Pautasso: *A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments*. ICPE 2018: 261-272
- [GvHZ+20] Alim Ul Gias, André van Hoorn, Lulai Zhu, Giuliano Casale, Thomas F. Düllmann, Michael Wurster: *Performance Engineering for Microservices and Serverless Applications: The RADON Approach*. ICPE Companion 2020: 46-49

- [HHM+17] Christoph Heger, André van Hoorn, Mario Mann, Dusan Okanovic: Application Performance Management: State of the Art and Challenges for the Future. ICPE 2017: 429-432
- [JMeter21] Apache Software Foundation: *Apache JMeter*, <https://jmeter.apache.org/>, 2021.
- [OBM+20] Dusan Okanovic, Samuel Beck, Lasse Merz, Christoph Zorn, Leonel Merino, André van Hoorn, Fabian Beck: *Can a Chatbot Support Software Engineers with Load Testing? Approach and Experiences*. ICPE 2020: 120-129
- [SAD+19] Henning Schulz, Tobias Angerstein, Dusan Okanovic, André van Hoorn: *Microservice-Tailored Generation of Session-Based Workload Models for Representative Load Testing*. MASCOTS 2019: 323-335
- [SAH+18] Henning Schulz, Tobias Angerstein, André van Hoorn: *Towards Automating Representative Load Testing in Continuous Software Engineering*. ICPE Companion 2018: 123-126
- [SHW20] Henning Schulz, André van Hoorn, Alexander Wert: *Reducing the maintenance effort for parameterization of representative load tests using annotations*. *Softw. Test. Verification Reliab.* 30(1) (2020)
- [SOH+19] Henning Schulz, Dusan Okanovic, André van Hoorn, Vincenzo Ferme, Cesare Pautasso: *Behavior-driven Load Testing Using Contextual Knowledge: Approach and Experiences*. ICPE 2019: 265-272
- [SOH+21] Henning Schulz, Dusan Okanovic, André van Hoorn, and Petr Tuma: *Context-tailored workload model generation for continuous representative load testing*. ICPE 2021. To appear
- [VHS+18] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, Helmut Kremer: *WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction — a model-driven approach for session-based application systems*. *Softw. Syst. Model.* 17(2): 443-477 (2018)