H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable 3.4

Continuous testing tool I

Version: 1.0

Publication Date: 30-June-2020

Disclaimer:

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

Deliverable Card

Deliverable	D3.4
Title:	Continuous testing tool I
Editor(s):	André van Hoorn (UST) and Thomas F. Düllmann (UST)
Contributor(s):	André van Hoorn (UST), Thomas F. Düllmann (UST), Mainak Adhikari (UTR), and Pelle Jakovits (UTR)
Reviewers:	Giuliano Casale (IMP), Stefano Dalla Palma (TJD)
Type:	Report
Version:	1.0
Date:	30-June-2020
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) will provide 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 first version of the CTT, which supports the definition of an extensible set of tests and test infrastructures in TOSCA models via RADON's Graphical Modeling Tool (GMT), as well as the deployment and execution of the CTT-generated tests using the TOSCA orchestrator xOpera.

The first CTT version includes support for an initial set of load and deployment tests, as well as corresponding test infrastructures, which we have made available via the RADON Particles template library. CTT exposes a REST-based API packaged as Docker containers, and can currently be used as a standalone tool, including the integration in a CI/CD pipeline. The main part of this document (i) lists the requirements that drive the CTT software development, (ii) describes CTT's software architecture and its current implementation status, and (iii) provides a demonstration of use based on the SockShop application, which is one of the RADON examples. All CTT-related artifacts described in this document are publicly available on GitHub.

Glossary

AEML	Abstract Entities Modeling Layer
CI/CD	Continuous Integration/Delivery
CSAR	TOSCA Cloud Service Archive
CTT	Continuous Testing Tool
DEML	Deployable Entities Modeling Layer
GMT	Graphical Modelling Tool
KPI	Key Performance Indicator
QoS	Quality of Service
SUT	System Under Test
TI	Test Infrastructure
TOSCA	Topology and Orchestration Specification for Cloud Applications

Table of contents

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* [D2.3], 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 [D2.3].

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.

This deliverable reflects that with the current CTT release associated to this deliverable it is already possible to:

- Define test-related information in TOSCA models using the Graphical Modeling Tool (GMT). Therefore, we have defined a new (extensible) hierarchy of CTT-provided TOSCA policy types, node types, implementation artifacts, and blueprints available in the RADON Particles template library. The set of built-in test types supports a representative set of both functional and non-functional tests, focusing on deployment and load tests.
- Use the CTT server to execute a test workflow comprised of (i) importing the test-augmented TOSCA models (as CSAR files) from GMT, (ii) generating test artifacts (CSAR files for the system under test and the test infrastructure), (iii) deploying the CSAR files using RADON's TOSCA Orchestrator xOpera, (iv) executing the tests, and (v) accessing the test results.
- Use CTT's extension mechanisms to define custom test types and add custom test drivers (agents).

The remainder of the introduction summarizes the objectives of the deliverable, highlights the main achievements regarding CTT so far, and outlines the structure of this deliverable.

1.1 Deliverable objectives

The main objective of this deliverable is to document the first version of the CTT architecture and implementation, as well as to provide an outlook on the following steps that will be conducted in the remaining period of the RADON project. This objective can be broken down into the following parts that are reflected in the structure of this deliverable:

- An overview of the requirements that drive the development of CTT. The presented information is a self-contained compilation of CTT-related information published in the RADON deliverables [D2.1], [D2.2], [D2.3], and [D6.1].
- A description of the architecture and implementation of the current CTT version, covering the domain model, the workflow, the hierarchy of modeling types, the server and agent components, as well as the extension points. The presented information builds on and considerably extends the CTT-related high-level information on the CTT architecture and integration in the RADON deliverables [D2.3] and [D2.4].
- A step-by-step description of the CTT workflow for creating, deploying, and executing tests using one of the RADON demo applications. The presented scenario already includes the interaction with the RADON components GMT and xOpera.
- An overview of the future development of CTT to be presented in the final CTT deliverable [D3.5].

1.2 Overview of main achievements

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

- A concise specification of the requirements related to CTT.
- Design of the continuous testing workflow.
- Design and implementations of the (extensible) modeling approach using standard TOSCA concepts, available as part of the RADON Particles.
- A design of the initial tool architecture, comprising the CTT server and agents, as well as the module concept for custom extensions.
- An implementation of the first CTT version using state-of-the-art technologies such as Python, OpenAPI, and Docker. CTT can be easily deployed using publicly available Docker images and used via a REST API.
- The interplay of the current CTT version with the RADON components GMT, Particles, and xOpera.
- A demonstration of use based on an example RADON application.

1.3 Structure of the document

The remainder of this deliverable report is structured as follows.

- Section 2 provides an overview of the requirements that serve as the basis for the development of CTT.
- Section 3 describes the tool architecture and implementation.
- Section 4 demonstrates the use of CTT for testing one of the RADON demo applications.
- Section 5 concludes the document by summarizing its contents and giving an outlook on the future CTT developments that will be conducted after the release of this deliverable.
- The Appendix summarizes the level of compliance for the requirements at this stage.

Artifacts supplementing this deliverable are publicly available online [D3.4-Data].

2 Requirements

Requirement-related information for CTT has previously been documented in the deliverables [D2.1], [D2.2], [D2.3], and [D6.1]. In order to make this CTT deliverable self-contained, we briefly summarize the CTT-related requirements in this section.

2.1 Actors

Deliverable [D2.1] defines a set of actors intended to be supported by RADON. All actors for which testing-related activities are defined are relevant for CTT and RADON's continuous testing workflow [D2.3], namely (i) the Software Developer, (ii) the Release Manager, and (iii) the QoS Engineer.

2.2 Use cases and usage scenarios

Deliverable [D2.1] defines use cases for CTT, including the interaction with the RADON user and other RADON components. The use cases are grouped into three main usage scenarios, namely the definition, execution, and maintenance of tests (Table 1):

Table 1. CTT usage scenarios

Id	Title	Description
US 1	Define Test Cases	Using the RADON IDE, the RADON user defines the test cases (Define test cases in the diagram). Based on the definition, the Continuous Testing Tool generates the executable test artifacts (Generate test artifacts use case), e.g., when being triggered by the RADON IDE (Generate test artifacts use case).
US 2	Test Execution	Based on the test case definition, the artifacts can be executed (Execute test) after being triggered by either a RADON user request through the IDE (Run test), or by the CI/CD pipeline. To run the test, the required infrastructure is deployed by the RADON Runtime (Deploy test infrastructure). During test execution specific metrics are being collected and later used to generate the report (Generate test report), which can be shown in the IDE (Display test result).
US 3	Test Maintenance	Based on the data collected at runtime by the Monitoring Tool, input data for the defined test cases is updated to provide more realistic tests (Update test). This can be triggered through the IDE (Update tests).

2.3 Requirements table

Table 2 lists the CTT-related requirements as published in the deliverables [D2.1] [D2.2].

The requirements are separated into the requirements related to (i) the FaaS(/microservices) module and (ii) the data pipeline module, as well as the modeling and IDE integration. The development of the data pipeline module is in the responsibility of UTR.

Table 2. CTT requirements table

Id	Requirement Title	Priority
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
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
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
R-T.3.3-13	The FaaS testing module of the TESTING_TOOL could have a report feature	Could
R-T.3.3-14	The FaaS testing module of the TESTING_TOOL must have a graphic user interface.	Must
R-T.3.3-15	The FaaS testing module of the TESTING_TOOL should have a command line interface.	Should have
R-T.3.3-16	The FaaS testing module of the TESTING_TOOL must be integrated into DevOps practices	Must
Data pipeline module		
R-T3.3-2	The data pipeline testing module of the TESTING_TOOL should support user configurable data production profiles	Should
R-T3.3-3	The data pipeline testing module of the TESTING_TOOL must be able to inject additional pipeline components into the data pipeline for generating synthetic input data	Must
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
R-T3.3-5	*Removed as detailed in [D2.2]*	
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
R-T3.3-7	It could be useful for the data pipeline testing module of the	Could

	TESTING_TOOL to have a graphical user interface for configuring tests and displaying test results	
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
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
IDE		
R-T2.3-7	The IDE must provide support in launching the TESTING_TOOL and to trigger the execution of tests	Must
R-T2.3-8	The IDE must provide access to a report based on the results received from the TESTING_TOOL	Must

3 CTT architecture and implementation

In this section, we provide a detailed description of CTT's architecture and implementation, covering both aspects that apply to the final CTT tool at the end of the project duration and those aspects that apply to the current version.

This chapter is structured as follows. Section 3.1 provides a high-level overview of the architecture with the goal of introducing the core CTT components. The overview covers CTT's functionality until the end of the project period. Section 3.2 summarizes the core design decisions w.r.t. concepts, technologies, and development. Sections 3.3 to 3.7 detail these contributions, structured according to Kruchten's 4+1 architectural view model [Kru95] into a *logical* view, an *implementation* view, a *deployment* view, a *process* view, and a *scenarios* view. Finally, we elaborate briefly on the planned data pipeline module (Section 3.8).

3.1 Architecture overview including core design decisions

This section starts by providing a high-level overview of the workflow (Section 3.1.1), followed by a more detailed overview of the technical architecture and the interaction with the environment (Section 3.1.2).

3.1.1 Workflow

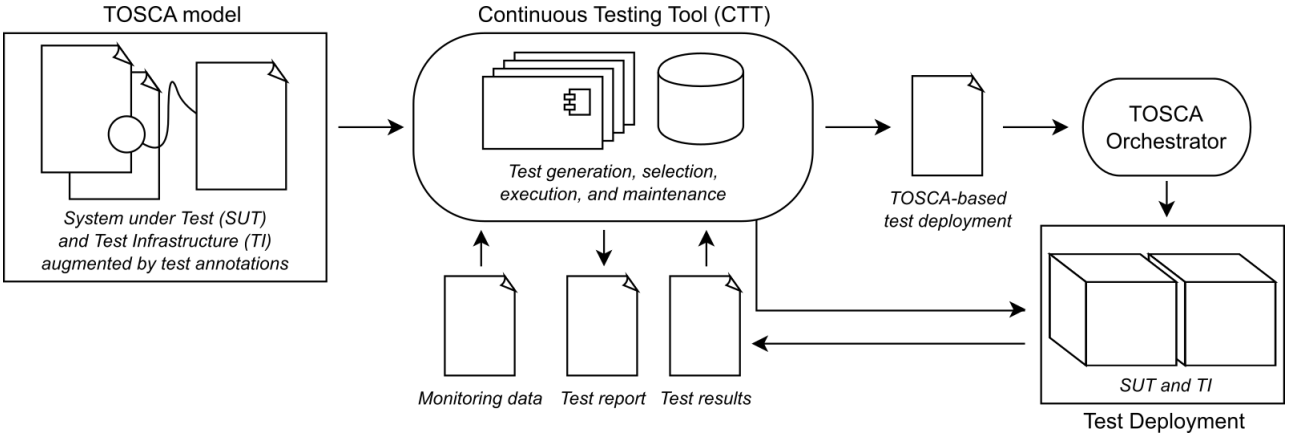
In order to test an application, the user is supposed to create a model of the system she intends to test (SUT) with CTT in GMT. This model is then annotated with details about the tests she wants CTT to execute. Based on the types of tests included in the annotations, the models for the underlying test infrastructure are also created or may already be available. The resulting models are TOSCA models that can be exported and are included in the Git repository of a system. With these artifacts ready, the workflow of CTT itself can be commenced.

In the first step, a CTT project is created using the Git repository information and a custom name for the project. The next step is the test artifact generation, which requires the selection of the previously created TOSCA models for the SUT and the test infrastructure (TI). Once these models have been processed, the following step is the deployment of both, the SUT and the TI in preparation of the actual tests. When the deployments are finished, the execution of the tests is triggered. The results of the tests can be collected by creating a result set in the last step. Finally, the result set can be downloaded for further processing and inspection.

3.1.2 Technical architecture and interaction with the environment

Figure 1 aims to provide a high-level overview of CTT and its interaction with its environment in terms of artifacts, tools, and infrastructures.

Figure 1. High-level architecture [GvHZ+20]



A user defines tests by adding them to a TOSCA service template for the system under test (SUT). CTT-specific TOSCA node types and policy types are defined in the RADON Particles template library [D4.3] [D4.4] for expressing different types of tests and test infrastructures (TIs). For instance, CTT allows the definition of a load test to be executed by a test infrastructure comprised of a load driver such as JMeter [JMeter20]. CTT imports the TOSCA model and provides possibly refined TOSCA models with the SUT and TI. The refinement includes novel approaches for testing in continuous software engineering and DevOps, e.g., the generation and selection of tests tailored to microservices, functions, and operational profiles [AFJ+20] [SAD+19]. After being deployed by a TOSCA orchestrator such as xOpera, the tests are executed and the test results are made available to the user in the form of the raw test results or a revised test report. Monitoring data provided by the monitoring tool is an input to CTT to update and refine tests. CTT is designed as an extensible framework that allows the definition of new test types, metrics, and tools.

CTT provides a REST-based interface. Via the interface users can execute the continuous testing on-demand, include it as a part of the CI/CD process, or via the RADON IDE.

3.2 Summary of Design Decisions

This first CTT deliverable concludes the tool's first development phase, which included the requirements engineering, the definition of the high-level architecture as presented in the previous section, as well as the development and release of the first version of the tool. During the course of these activities, we have made some design decisions that shape CTT's architecture and development.

The remainder of this section provides a concise summary of these decisions.

Conceptual decisions

- Project-based test management and user interaction with the CTT tool. This decision was driven by CTT's domain model and had a major influence on CTT's API, which will both be detailed in Section 3.3.
- Decomposition of CTT's implementation into the modeling types, the CTT server, and the CTT agents. This decomposition and the implementations of the components will be detailed in Section 3.4.
- Use of standard TOSCA modeling concepts such as policies, nodes, and service templates to define tests and test infrastructures. CTT's particular modeling contribution is the reusable and extensible hierarchy of CTT-related modeling types provided via RADON's particle template library. CTT's type modeling will be detailed in Section 3.4.1.
- Ability to extend CTT's feature set by custom modules via extension points. The concept is used to implement CTT's FaaS/microservices and data-pipeline modules, and incorporates an extensible modeling type hierarchy. CTT's module concept is described in Section 3.4.4.
- Ability to use CTT in three usage contexts, namely by direct REST API interactions, CI/CD, and the RADON IDE.
- High-level architecture, as presented in Section 3.1. The high-level architecture has been designed to implement the functionality desired for the final version of the CTT tool, i.e., covering both this deliverable and the final deliverable. However, we keep the freedom to revise minor parts of the architecture for the next deliverable.
- For this first deliverable, the goal was to establish the architecture and its implementation, as well as basic support for the FaaS/microservices module. The research contributions have so far been developed without tight technical coupling to CTT. The integration of the data pipeline module and the integration of the research approaches is subject to the next deliverable. Also, the remaining functionality is subject to future development toward the final CTT deliverable, as outlined in Section 5.

Technological and development decisions

- Python is used as the main programming language for the CTT server and the included CTT agents. As the integration with the agents is conducted via REST-based interfaces, developers of extensions are free to select a language for their CTT agent.
- GitHub is used as CTT's development platform for
    - Git repositories including the source code artifacts. The list of repositories is provided in Section 3.5.
    - issue management: https://github.com/radon-h2020/radon-ctt/issues
    - project status dashboard: https://github.com/orgs/radon-h2020/projects/2

- Docker is used for packaging CTT's server and agent components. The Docker images are available via DockerHub. An overview of CTT's components available on DockerHub is available in Section 3.5.
- All CTT components are publicly available under the Apache License 2.0 open-source license.

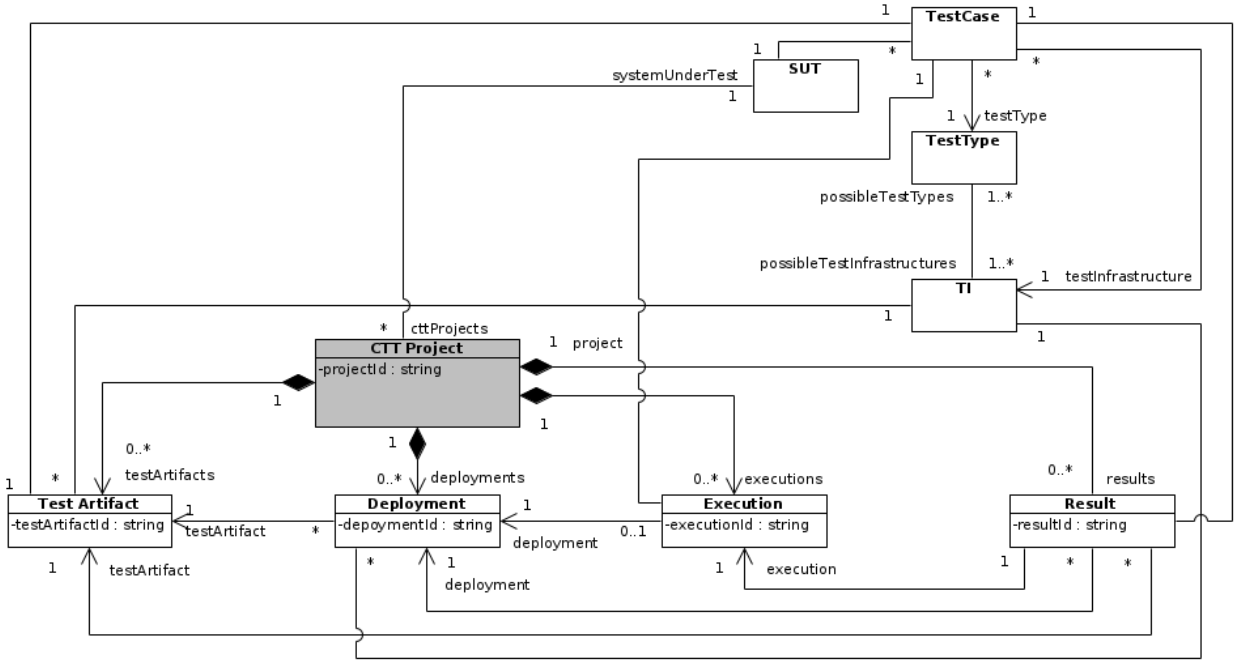
3.3 Logical view

This section describes CTT's domain model (Section 3.3.1), the Server API (Section 3.3.2), as well as the interaction with users and other tools (Section 3.3.3).

3.3.1 Domain model

The class diagram in Figure 2 depicts the entities and relationships in the CTT domain. A (CTT) project represents the highest level and relates to an SUT that CTT is used for. For the SUT, a set of test cases is defined — each being of a defined type (e.g., the SUT should be tested with two different load tests (test cases) of the type JMeter load test). While a test type can be executed by a set of TIs (e.g., master-only or master-worker JMeter setups), a test case is associated with a single explicitly defined TI (i.e., specific TOSCA blueprint). For a project, there can be multiple test artifacts, each representing a test and the deployment configurations that shall be used to execute the test. Based on one set of test artifacts, deployments can be made. With a deployment in place, the actual test can be executed, and the results of the test can be obtained.

Figure 2. CTT domain model



3.3.2 CTT's Server API

The server API is divided into five parts. Every part represents one step of the workflow and begins with the creation of an entity that is related to the previous step and the respective entity. Table 4 gives an overview of the entities and their endpoints.

Table 4. High-level endpoints in the Server API

Endpoint	Description
`/project`	A project based on a Git repository with a given name.
`/testartifact`	Selected test artifacts from a previously created project.
`/deployment`	Deployment of SUT and TI using the selected test artifacts.
`/execution`	Execution of tests by the TI targeting the SUT.
`/result`	Collection and inspection of the test results.

For each of the high-level endpoints there are several operations, listed in Table 5, that allow the manipulation of the entities. The detailed API can be found in the OpenAPI format in the CTT GitHub repository[1]. To view it in a user-friendly way, it can be loaded in the Swagger Editor[2].

Table 5. Endpoints in the Server API

REST Verb	Endpoint	Description
Project	**Operations for creating, deleting, querying projects**	
GET	`/project`	Get a list of all projects
POST	`/project`	Create a project
DELETE	`/project/{project_uuid}`	Delete a project
GET	`/project/{project_uuid}`	Retrieve a project
TestArtifact	**Operations for creating, deleting, querying test artifacts**	
GET	`/testartifact`	Get all test artifacts
POST	`/testartifact`	Create a test artifact
DELETE	`/testartifact/{testartifact_uuid}`	Delete a test artifact
GET	`/testartifact/{testartifact_uuid}`	Retrieve a test artifact
Deployment	**Operations for creating, deleting, querying deployments**	

[1] https://github.com/radon-h2020/radon-ctt/blob/master/ctt-server/openapi_server/openapi/openapi.yaml
[2] https://editor.swagger.io

GET	`/deployment`	Get all deployments
POST	`/deployment`	Create a deployment
DELETE	`/deployment/{deployment_uuid}`	Delete a deployment
GET	`/deployment/{deployment_uuid}`	Retrieve a deployment
Execution	**Operations for creating, deleting, querying executions**	
GET	`/execution`	Get all executions
POST	`/execution`	Create an execution
DELETE	`/execution/{execution_uuid}`	Delete an execution
GET	`/execution/{execution_uuid}`	Retrieve an execution
Result	**Operations for creating, deleting, querying results**	
GET	`/result`	Get all results
POST	`/result`	Create new result
DELETE	`/result/{result_uuid}`	Delete a result
GET	`/result/{result_uuid}`	Retrieve a result
GET	`/result/{result_uuid}/download`	Download the result

3.3.3 Interactions with users and other tools

Users interact with CTT as part of the required set of use cases and usage scenarios. To achieve the respective functionality, CTT interacts with other tools. The remainder of this section details the user and tool interactions.

Interaction with the users

The user interaction with CTT is based on the following use cases:

- Specification of Test-related Information
- Test Case Generation
- Test Deployment
- Test Execution
- Result Interpretation and Presentation
- Feedback from Testing

The use cases are implemented by CTT and provided via the previously described API. Users interact with CTT either directly via this API or other tools, particularly, RADON IDE, CI/CD.

The workflow is as follows:

1. Create a CTT project by providing a name and the URL to the repository that is supposed to be tested.
2. Create test artifacts by specifying the paths to the TOSCA artifacts used for the system under test (SUT) and the test infrastructure (TI). This step collects all information necessary for the following deployment and execution steps.
3. The create deployment step first deploys the SUT and TI using the xOpera orchestrator. When both have been deployed successfully, the endpoints of the deployed services are known and can be used to execute the actual tests in the following step.
4. Create execution provides the TI with the test configuration including dynamic properties required for successful execution (e.g., IP address of the SUT) and then triggers the start of the test execution.
5. Finally, after the test has finished, the results of the test execution can be obtained using the results section of the API.

The workflow is a consequence of the use case/usage scenarios defined in the requirements deliverables [D2.1] [D2.2] as well as the continuous testing workflow defined in ([D2.3]).

Interaction with other tools

CTT interacts with the RADON IDE/GMT, the Particles, the Orchestrator (xOpera), CI/CD, and the Monitoring system, as detailed in [D2.4].

- *GMT* is used to model the tests and the test infrastructures. CTT interacts with GMT via the exported CSAR files. The *RADON IDE* provides access to CTT's functionality via the defined REST-based API.[3]
- The *RADON Particles* include the CTT-related type hierarchy that also serves as the basis for custom extensions in the form of additional test types, tools, and blueprints. CTT interacts with RADON particles via the TOSCA-based models and implementation artifacts.
- CTT uses the *Orchestrator (xOpera)* for the deployment of the SUT and the test infrastructure. CTT interacts with xOpera via xOpera's API and the CSAR files to be deployed.
- CTT can be used for testing SUTs within their *CI/CD* pipelines. Therefore, the CI/CD pipeline invokes CTT via its API.
- CTT acquires RADON's Monitoring component to get access to runtime data for refining operational profiles.[4]

[3] The integration with the Monitoring system is subject to the next deliverable.
[4] The integration with the Monitoring system is subject to the next deliverable.

3.4 Implementation view

CTT's architectural structure is decomposed into the components modeling types, CTT server, and CTT agent(s). Orthogonal to their functionality, these components provide extension points to allow users to add functionality based on custom needs, e.g., additional test types, tools, or infrastructures.

Section 3.4.1 to 3.4.3 describes the three components, while Section 3.4.4 describes the extension points.

3.4.1 Modeling types

CTT relies on TOSCA modeling concepts to augment the TOSCA models of the SUT as well as to define the TI. Therefore, we have defined a CTT-specific modeling type hierarchy.

Test case types are modeled using TOSCA policy types. Reusable component types for the test infrastructures, e.g., test drivers, are modeled as TOSCA node types. Reusable test infrastructures are provided as blueprints.

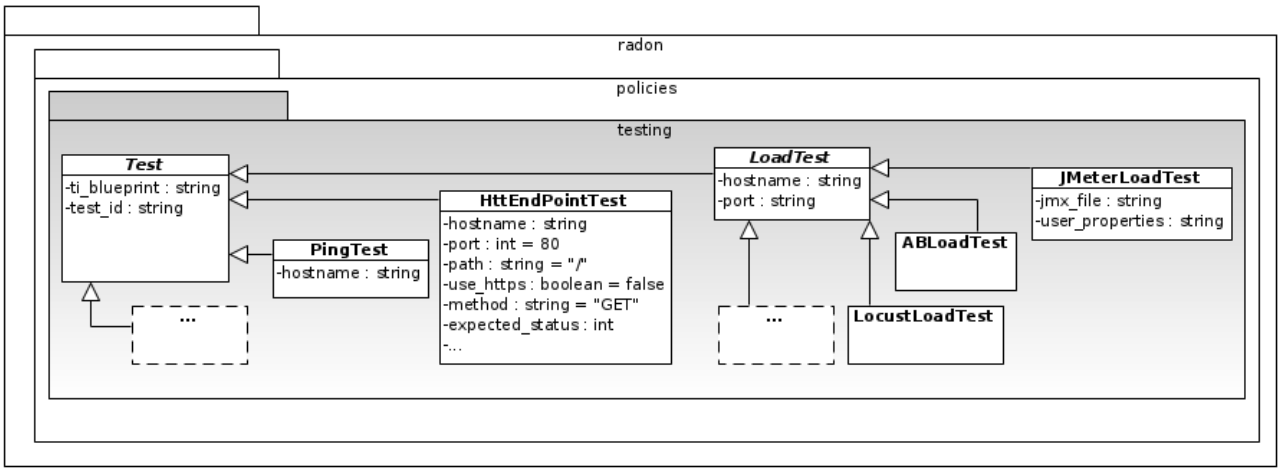
CTT's type hierarchy is integrated into RADON's modeling profile [D4.3], following the modeling type hierarchy and namespace: `radon.[entity-type].[purpose-identifier*].[entity]`. The CTT types are implemented in RADON Particles, which is the RADON template library containing reusable definitions and extensions, in this case, for testing. Following the RADON modeling approach [D4.3], we provide abstract (AEML) and deployable modeling entities (DEML).

The remainder of this section gives an overview of the CTT-related type hierarchy in the form of policy types, node types, and blueprints.

CTT Policy types

The (abstract) parent type of any CTT test type and test case is `Test`. The type includes a reference to a CTT TI blueprint and a test identifier as attributes. As detailed below, the blueprint is a TOSCA service template defining the infrastructure that executes the test. The class diagram in Figure 3 shows CTT's policy type hierarchy.

Figure 3. CTT policy types



The policy types `PingTest` and `HttpEndPointTest` are concrete policy types that can be used to define deployment test cases of an SUT to test whether they respond to requests on different protocol levels.

The type hierarchy includes another (abstract) policy type for load tests, namely `LoadTest`. Example concrete policy types for load tests include those for the JMeter (`JMeterLoadTest`), Apache Bench[5] (`ABLoadTest`), and Locust[6] (`LocustLoadTest`) tools. As detailed for `JMeterLoadTest`, the test cases include attributes such as a reference to a load test script and additional properties.

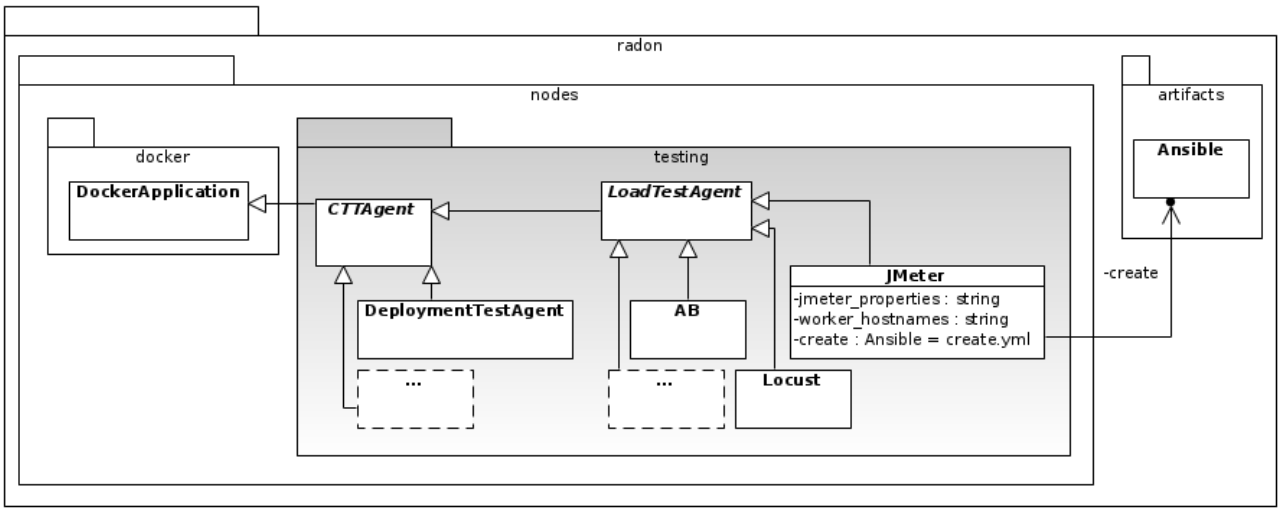
Additional types are available and can be added at the places indicated by "`...`".

CTT Node types

The (abstract) parent of any test infrastructure node type is `CTTAgent`, which derives from RADON's type for Docker applications (`DockerApplication`). The class diagram in Figure 4 shows CTT's node type hierarchy.

[5] https://httpd.apache.org/docs/2.4/programs/ab.html
[6] https://locust.io/

Figure 4. CTT node types



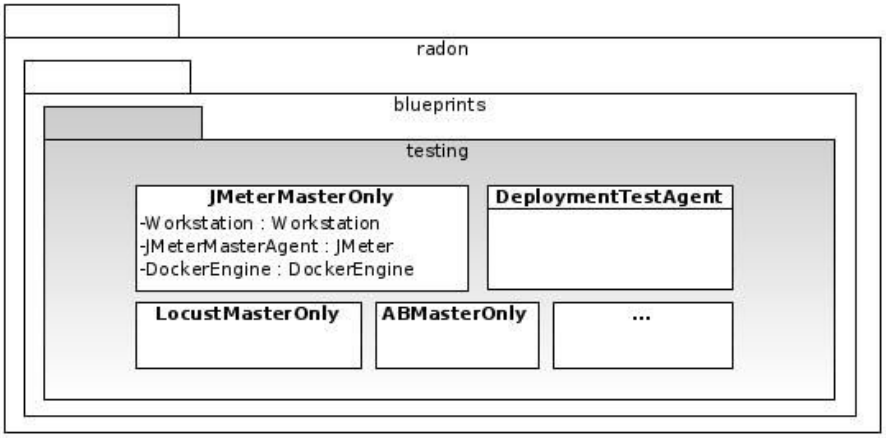
Currently, CTT's node type hierarchy includes node types for executing deployment tests and load tests. The concrete node type (`DeploymentTestAgent`) is able to execute deployment tests such as the previously introduced `PingTest` and `HttpEndPointTest`. The type LoadTestAgent is abstract and serves as the basis for concrete node types for representing agents for load testing tools such as JMeter (`JMeter`), Apache Bench (AB), and Locust (`Locust`), corresponding to the respective policy types introduced before. As exemplified for `JMeter`, the node type includes attributes such as additional configuration properties or the list of worker nodes (for high-scale load test experiments). Moreover, the node types require an implementation artifact so that CTT can trigger the deployment of the respective node in the test infrastructure via the Orchestrator.

Additional types are available and can be added at the places indicated by "`...`".

CTT Blueprints

CTT provides a set of TI blueprints, i.e., TOSCA service templates that represent the test infrastructure needed to execute the modeled tests. The package diagram in Figure 5 shows selected blueprints that correspond to the previously mentioned test types and test infrastructure nodes.

Figure 5. CTT policy types



For example, the blueprint `JMeterMasterOnly` is a TOSCA service template that comprises the `JMeter` node type deployed on a Docker engine (`DockerEngine`) hosted on a workstation (`Workstation`), where both `DockerEngine` and `Workstation` are node types already available in RADON Particles. For the JMeter case, various other blueprints are possible, particularly a JMeter master/slave setting with several agents enabling a load test with higher workload intensity. Figure 6 shows the `JMeterMasterOnly` blueprint in GMT:

Figure 6. `JMeterMasterOnly` blueprint as shown in GMT

3.4.2 CTT Server

The role of the CTT server is to provide a set of endpoints that allow controlling all steps related to continuous testing. This includes the creation of a test project based on an existing Git repository, the creation of test artifacts to be used for the subsequent deployment of the system under test (SUT) and the test infrastructure (TI). Following the deployment, the TI runs tests against the SUT, which can be later retrieved via the CTT server.

The CTT server is implemented with Python 3 and uses the Python-Flask framework[7]. To generate the framework, we used the Swagger server stub generation feature[8] based on an OpenAPI definition of the API.

[7] https://flask.palletsprojects.com/en/1.1.x/
[8] https://editor.swagger.io

For every endpoint mentioned in the Server API, there is one class representing an instance. These entities have already been introduced in the domain model.

To persist data, we use SQLite, which is easily integrated using the declarative notation variant of SQL Alchemy[9]. This allows to annotate Python classes and directly map them to database objects.

In order to deploy TOSCA models, we use the xOpera orchestrator, which is currently invoked as a command-line tool.

The CTT Server is published as a Docker Container to be easily deployable.

3.4.3 CTT Agent

The CTT Agent provides a uniform interface for interaction in the form of a REST-based API between the CTT Server and the testing tools. It is deployed together with one or multiple test tool modules inside a Docker container to act as test infrastructure (TI) and execute tests against the system under test (SUT).

Agent API

The agent itself does not expose any functional endpoints, as it is only the basic framework for the different testing tools.

The basic API concept for each testing tool (module), as sketched in Table 6, is that in a first step, all information and, if applicable, files are transmitted to create a test configuration. In a second step, the execution of the configured test can be triggered. Finally, the results can be obtained.

Table 6. Endpoints in the Agent API

Configuration	**Operations for creating, deleting, querying testing tool configurations**	
GET	`/{testingtool}/configuration/{config_uuid}`	Get configuration
POST	`/{testingtool}/configuration/`	Create configuration
DELETE	`/{testingtool}/configuration/{config_uuid}`	Delete a configuration
Execution	**Operations for triggering and querying testing tool executions**	
GET	`/{testingtool}/execution/{execution_uuid}`	Get execution results
POST	`/{testingtool}/execution`	Create (run) execution

Implementation

The basic API is realized with Python 3 and the Python Flask framework and is packaged inside a Docker container that allows deriving from it and packaging arbitrary testing tools inside a Docker

[9] https://www.sqlalchemy.org/

container. This container is also included in the respective node type in the Radon Particles repository.

Supported agents

Currently, we provide two pre-built agents: one for load tests using JMeter[10] and one deployment test agent that can check for a successful deployment of the target system by either a Ping test or an HTTP test, which connects to a given resource and checks the HTTP response code.

Additional testing tools can be added by using the extension points detailed in the following section.

3.4.4 Extension Framework

CTT is designed to be extensible in terms of possible integrations with testing tools. To achieve that, it provides capabilities to add custom modules in every component related to CTT. Namely, an extension needs to comprise three parts, which are detailed in the following sections.

TOSCA Types

As a first step for extending the testing capabilities of CTT with a custom tool, the Tosca types need to be created. These would be integrated similarly to the JMeter test types in the type hierarchy presented in Section 3.4.1. Together with the definition, also a deployment description in the form of an Ansible playbook is required.

CTT Integration

In order to be able to use the custom tool with CTT, small plugins in the server and client are needed as test tools, and their inputs are very diverse.

On both server and agent side, there are essentially two steps that need to be implemented in a tool-specific manner. First, after the agent has been successfully deployed, any test parameters are passed from the server to the agent in the *configuration* step. Once the execution is triggered on the server side, it notifies the agent to start the execution in the *execution* step.

For both the *configuration* and the *execution* step, a plugin needs to be added to the server and the agent to ensure the proper integration of the custom testing tool.

Plugins are implemented in Python 3 and can be easily integrated using the Python Flask framework. To illustrate the development of a plugin for this purpose, we provide an example in Section 4.

[10] https://jmeter.apache.org/

3.5 Deployment view

The CTT server is intended to be deployed as a Docker container and is already available as such. The system under test (SUT) and the test infrastructure (TI), namely the CTT agents, are deployed to the destination that is defined in the TOSCA models. The CTT agents are also packaged as Docker containers to be easily deployable in a wide variety of environments.

Table 7 to Table 9 detail the locations of the relevant resources for development and deployment.

Table 7. CTT Server source code and release URLs

CTT Server	
GitHub repository	https://github.com/radon-h2020/radon-ctt/
DockerHub	https://hub.docker.com/r/radonconsortium/radon-ctt
Docker Image	`radonconsortium/radon-ctt:latest`

As there are several testing tools that we support, there is a CTT agent variant for each testing tool (e.g., JMeter). The variants are reflected in different tags for the Docker image.

Table 8. CTT Agent source code and release URLs

CTT Agent	
GitHub repository	https://github.com/radon-h2020/radon-ctt-agent/
DockerHub	https://hub.docker.com/r/radonconsortium/radon-ctt-agent
Docker Images	`radonconsortium/radon-ctt-agent:{testingtool}`

The types are part of the publicly available Particles repository.

Table 9. CTT modeling types source code URLs

Modeling Types	
CTT policy types	https://github.com/radon-h2020/radon-particles/tree/master/policytypes/radon.policies.testing
CTT node types	https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.testing
CTT blueprints	https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.blueprints.testing

3.6 Process view

Each interaction with CTT via its REST-API results in the execution of internal actions inside the CTT tool, i.e., the CTT server and agents, as well as the external interaction as part of the integration with the related RADON tools. We refer to [D2.4] for views on the interaction using UML sequence diagrams.

3.7 Scenarios view

The current version of CTT supports the following contexts in which CTT can be used:

1. *Use of CTT as a standalone on-demand testing tool.*

   The user starts the CTT server and uses it via the provided REST-based API by executing the defined workflow. The CTT server may be deployed on the user's local machine or a remote machine. The user uses CTT with defined CSAR files for the SUT and TI, which usually have been created using GMT. As this usage context is the basis for the other contexts, it will be illustrated in detail in Section 4.

2. *Use of CTT as part of the CI/CD pipeline*

   This usage context is similar to the previous one. The main difference is that instead of manual calls to the CTT API, the respective interaction is conducted via the CI/CD pipeline scripts. A repository with an example CI/CD pipeline using CTT is provided online.[11]

The next context to be supported, which is subject to the final CTT deliverable, will be the use of CTT via the RADON IDE.

3.8 Continuous testing of data pipelines

When considering testing data pipeline applications, the current CTT features like defining, generating, executing tests, and reporting test results can be applied directly. 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 for designing and orchestrating data pipelines [D5.5], the whole pipeline is divided into several sub-tasks, where each sub-task acts as an independent pipeline block. This means that the whole application is composed of multiple independently deployable, scalable, and schedulable tasks, and it is important to be able to extract metrics for each of them while testing the

[11] https://github.com/radon-h2020/radon-ctt-integration

whole data pipeline to be able to accurately pinpoint where the performance or other issues are located in the pipeline.

For example, when load testing data pipelines, 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. Furthermore, the expected data-intensive nature of data pipelines means that generating data for lead testing of data pipelines should be efficient and scalable.

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. To enable extracting fine-grained run-time metrics, we plan to utilize 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.[12] The goal of the data pipeline module will also be to dynamically set up exporting its run-time monitoring data, configure Prometheus to start collecting such data, generate metrics, and to record them in the test report.

For generating data for load testing data pipelines, the current approach is to use JMeter and also the QT tool from the DICE project [DICE17]. In the initial lab validations, we used JMeter to push a set of images as input to the Thumbnail generation with the data pipelines demo application. For generating a heavy load of textual data (such as tweets in the form of JSON documents from Twitter datasets), QT is a good candidate as it is designed for generating data for data-intensive applications and is able to generate data which behaves similarly to the input dataset, and thus allows to test data pipelines under real-life like input loads.

[12] https://github.com/radon-h2020/demo-datapipeline-validation-and-load-testing

4 Demonstration of use

This section exemplifies the usage of CTT with the SockShop[13] application, which is one of the RADON demo applications. The SockShop is an open-source microservice example using Docker containers and a wide range of configurations for easy deployment (e.g., docker-compose and Kubernetes). We forked the project on GitHub and enriched it with artifacts for usage with the CTT tool as a way to showcase the process of using CTT with an existing project.[14]

We use the demonstration of use presented in this section also as a getting started guide for CTT users on the CTT website.[15]

4.1 Outline of the demonstration

In general, the steps in CTT (detailed below) are as follows:

1. Modeling test cases in the Graphical Modelling Tool (GMT)
2. Starting the CTT Server
3. Executing the CTT workflow
4. Adding custom extensions

4.2 Modeling

Modeling is done via RADON's Graphical Modelling Tool (GMT). We refer to the GMT documentation for further details on using the tool.[16]

A prerequisite for the CTT-related modeling is a TOSCA service template for the system under test. In this example, we use the SockShop. A service template (blueprint) for the SockShop is available in GMT as part of the Particles template library (displayed in Figure 7):

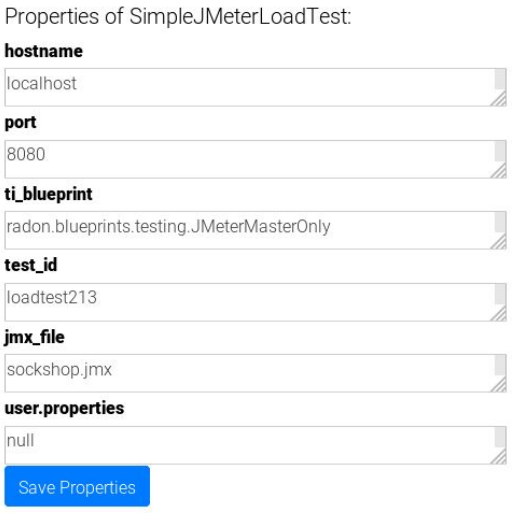
Figure 7. `SockShop` blueprint icon as shown in GMT

Open this blueprint and navigate to GMT's Topology Editor. The next step is to add a test case to the SockShop. To do so, select `Other > Manage Policies`. An editor for defining policies opens. In this example, we will define a JMeter load test. After clicking on the `Add` button, define a name for the test policy (e.g., "SimpleJMeterLoadTest") and select the type

[13] https://github.com/microservices-demo/microservices-demo
[14] https://github.com/radon-h2020/demo-ctt-sockshop
[15] https://continuous-testing-tool.readthedocs.io
[16] https://github.com/radon-h2020/radon-gmt/

`radon.policies.testing.JMeterLoadTest`. You can now fill the properties of the load test via the dialog window displayed in Figure 8:

Figure 8. Policy editor for defining a `JMeterLoadTest` as shown in GMT



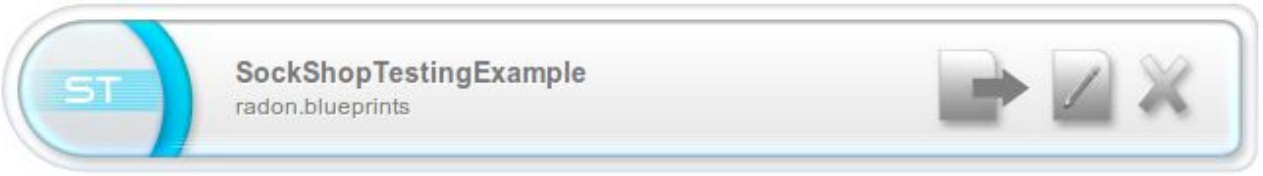
In this example, we define the load test to access the SockShop available on `localhost:8080`, using the load test script `sockshop.jmx`, and select the predefined test infrastructure template `radon.blueprints.testing.JMeterMasterOnly`, which is also available in GMT. Custom test infrastructure templates can be created and used following this pattern.

After saving, the test case policy can be assigned to the SockShop node. Therefore, enable the `Policies` option in GMT's top menu and set the policy's `isActive` property. The following screenshot shows the SockShop service template with two additional test cases (Figure 9):

Figure 9. CTT policies attached to the SockShop as shown in GMT

A prepared service template of the SockSop including CTT tests is already included in RADON's Particles library and available in GMT (Figure 10):

Figure 10. SockShop blueprint icon as shown in GMT



This service template will be used in the remainder of this section.

4.3 Starting the CTT server

The easiest way to start CTT is by invoking the publicly available Docker container:

```
docker run -t -i --name RadonCTT -p 18080:18080 \
-v /var/run/docker.sock:/var/run/docker.sock radonconsortium/radon-ctt:latest
```

To check whether the CTT server has started properly, you should be able to access the OpenAPI-based interface via a web browser: http://localhost:18080/RadonCTT/ui/

However, for the remaining step, we will interact with the CTT server via Curl.[17]

4.4 Executing the Workflow

4.4.1 Creating a Project

The first step is to create a CTT project by providing a project name and a Git repository URL. The repository contains TOSCA service templates.

The following Git repository already includes a ready-to-use example with the Weaveworks SockShop application: https://github.com/radon-h2020/demo-ctt-sockshop/.

To create a project based on the SockShop example repository, execute the following:

```
curl -X POST "http://localhost:18080/RadonCTT/project" \
 -H  "accept: */*" -H  "Content-Type: application/json" \
-d
```

[17] https://curl.haxx.se/

```
"{\"name\":\"SockShop\",\"repository_url\":\"https://github.com/radon-h2020/demo-ctt-sockshop.git\"}"
```

The response includes a UUID for the project that is required for the further interaction with the project. For example, the following output includes the UUID `d3cb9d75-73df-422e-9575-76e7a5775f8e` of the project that was just created:

```
{
  "name": "SockShop",
  "repository_url": "https://github.com/radon-h2020/demo-ctt-sockshop.git",
  "uuid": "d3cb9d75-73df-422e-9575-76e7a5775f8e"
}
```

4.4.2 Generating Artifacts

To create test artifacts for the project, the paths to the TOSCA CSAR files that include the system under test (SUT) and the test infrastructure (TI) must be passed along with the previously obtained project UUID to the next command. In the example repository, the CSAR files for the SUT and TI are stored in the `radon-ctt` folder, i.e., `radon-ctt/sut.csar` and `radon-ctt/ti.csar`.

```
curl -X POST "http://localhost:18080/RadonCTT/testartifact" \
-H  "accept: */*" -H  "Content-Type: application/json" \
-d
"{\"project_uuid\":\"d3cb9d75-73df-422e-9575-76e7a5775f8e\",\"sut_tosca_path\":\"radon-ctt/sut.csar\",\"ti_tosca_path\":\"radon-ctt/ti.csar\"}"
```

The response includes the UUID of the created test artifacts, i.e., the executable CSAR files of the SUT and the TI. For example, the following output includes the UUID `87a2d052-93ce-43d2-b765-74b0cef9df92` of the artifacts that were just created:

```
{
  "commit_hash": "4d1135e3d762dc81210d905932711a991b7b4373",
  "project_uuid": "d3cb9d75-73df-422e-9575-76e7a5775f8e",
  "sut_tosca_path": "radon-ctt/sut.csar",
  "ti_tosca_path": "radon-ctt/ti.csar",
  "uuid": "87a2d052-93ce-43d2-b765-74b0cef9df92"
}
```

4.4.3 Deploying the SUT and the TI

To deploy the test artifacts (SUT and TI) using the xOpera TOSCA orchestrator, the artifact UUDI needs to be provided to the next command:

```
curl -X POST "http://localhost:18080/RadonCTT/deployment" \
 -H  "accept: */*" -H  "Content-Type: application/json" \
 -d "{\"testartifact_uuid\":\"87a2d052-93ce-43d2-b765-74b0cef9df92\"}"
```

This step will take a while, depending on the system and type of systems. You can watch the progress by inspecting the CTT log and watching the Docker processes:

```
watch docker ps
```

The response includes the UUID of the deployment. For example, the following output includes the `UUID 5f435990-8a1a-4741-a040-6db2fe552603` of the deployment that was just created:

```
{
  "testartifact_uuid": "87a2d052-93ce-43d2-b765-74b0cef9df92",
  "uuid": "5f435990-8a1a-4741-a040-6db2fe552603"
}
```

The deployed SockShop application is now reachable in the web browser via: http://localhost/

The deployed JMeter agent is reachable via http://localhost:5000/. However, the REST-based interface is intended to be used by the CTT server and not by end-users.

4.4.4 Executing the Test

To execute the test, the deployment UUID needs to be provided in the next command:

```
curl -X POST "http://localhost:18080/RadonCTT/execution" \
 -H  "accept: */*" -H  "Content-Type: application/json" -d
"{\"deployment_uuid\":\"5f435990-8a1a-4741-a040-6db2fe552603\"}"
```

The test is now executing. You can watch the progress in the CTT server log and by attaching to the JMeter agent log (showing raw result statistics):

```
docker logs -f JMeterAgent
```

After the test has finished, the response includes the UUID of the text execution. For example, the following output includes the UUID `beead8ea-8e3e-42ec-ad1c-7e2b3e5b4492` of the most recent execution:

```
{
  "agent_configuration_uuid": "17321ea2-67c9-40fb-9e84-c761a39680a0",
  "agent_execution_uuid": "91905534-d423-4933-b363-d84a647ac619",
  "uuid": "beead8ea-8e3e-42ec-ad1c-7e2b3e5b4492"
}
```

4.4.5 Creating Test Results

To create the test results, the execution UUID needs to be provided in the next command:

```
curl -X POST "http://localhost:18080/RadonCTT/result" \
 -H  "accept: */*" -H  "Content-Type: application/json" \
 -d "{\"execution_uuid\":\"beead8ea-8e3e-42ec-ad1c-7e2b3e5b4492\"}"
```

After the creation has finished, the response includes the UUID of the test results. For example, the following output includes the UUID `a2c6bc9f-7c1f-4060-b80b-3c66e3487db9`:

```
{
  "execution_uuid": "beead8ea-8e3e-42ec-ad1c-7e2b3e5b4492",
  "results_file": "/tmp/RadonCTT/result/a2c6bc9f-7c1f-4060-b80b-3c66e3487db9",
  "uuid": "a2c6bc9f-7c1f-4060-b80b-3c66e3487db9"
}
```

4.4.6 Inspecting Test Results

To inspect the test results, the execution UUID needs to be provided in the next command:

```
curl -X GET
"http://localhost:18080/RadonCTT/result/a2c6bc9f-7c1f-4060-b80b-3c66e3487db9/download
" -H  "accept: application/json"
```

The response includes a `Results.zip` file with the test results.

For your convenience, feel free to download a sample `Results.zip`.[18] Among other contents, the file includes an HTML-based report (in the dashboard/ directory).

[18] https://continuous-testing-tool.readthedocs.io/en/latest/_static/Results.zip

4.5 Extension for custom needs

In order to extend the CTT with additional custom testing tools, the following steps are necessary. In terms of modeling, it is required to define custom test policies that provide details about the test parameters (e.g., test duration, concurrency, payload) your testing tool needs to configure the test execution. To use your testing tool as a TI, it is necessary to create an agent type so it can be deployed with the orchestrator xOpera. As a last step in the modeling context, the policies and types need to be integrated into the service templates and blueprints of the actual application that acts as the SUT.

In order to integrate a custom testing tool, there are two modules that need to be implemented. The CTT agent module controls the custom testing tool and acts as interface between the CTT server and the testing tool. The other module is the CTT server module that, after reading a policy for the custom testing tool transforms the information and controls the CTT agent with the custom agent module to act accordingly.

For both, the server and agent module, there is an existing plug-in framework with examples that allow easy extension.

We also provide example Dockerfiles for packaging the CTT agent together with the custom agent plugin. The resulting Docker image then can be referenced in the `create.yml` of the service template.

5 Conclusions

This deliverable presented the current version of the Continuous Testing Tool (CTT), giving a detailed overview of CTT's architecture and implementation, as well as a demonstration of use.

At this stage, 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 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.
- The use of CTT by the pre-compiled Docker images, and interacting directly via the REST-based API or via CI/CD scripts.

Artifacts supplementing this deliverable are publicly available online [D3.4-Data].

For the next (and final) CTT deliverable [D3.5] we plan the following works:

- Implementation of additional test types and tools based on the RADON use cases
- Development of the data pipeline plugin, including the integration of QT [DICE17]
- Support for operational feedback, including the integration with RADON's monitoring tool
- Integration of test case selection/optimization approaches, e.g., based on [AFJ+20] and [SAD+19]
- Integration into the RADON IDE via a respective CTT IDE plugin

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. 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 10. Achieved level of compliance to the CTT usage scenarios

Id	Requirement Title	Priority	Level of compliance
US 1	Define Test Cases	MUST_HAVE	✔✔✔
US 2	Test Execution	MUST_HAVE	✔✔
US 3	Test Maintenance	MUST_HAVE	✔

Requirements List

Table 11. 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	✔✔
R-T.3.3-11	The FaaS testing module of the TESTING_TOOL must support the execution of tests cases in the CD	Must	✔✔✔

	pipeline		
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	✔
R-T.3.3-13	The FaaS testing module of the TESTING_TOOL could have a report feature	Could	✔
R-T.3.3-14	The FaaS testing module of the TESTING_TOOL must have a graphic user interface.	Must	✔
R-T.3.3-15	The FaaS testing module of the TESTING_TOOL should have a command line interface.	Should have	✔✔✔
R-T.3.3-16	The FaaS testing module of the TESTING_TOOL must be integrated into DevOps practices	Must	✔
Data pipeline module			
R-T3.3-2	The data pipeline testing module of the TESTING_TOOL should support user configurable data production profiles	Should	✔
R-T3.3-3	The data pipeline testing module of the TESTING_TOOL must be able to inject additional pipeline components into the data pipeline for generating synthetic input data	Must	✔
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	✔
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**
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	✔

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**
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	✔✔✔
IDE			
R-T2.3-7	The IDE must provide support in launching the TESTING_TOOL and to trigger the execution of tests	Must	**M24**
R-T2.3-8	The IDE must provide access to a report based on the results received from the TESTING_TOOL	Must	**M24**

References

[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)

[D2.1] RADON Consortium, Deliverable D2.1: Initial requirements and baselines, 2019.

[D2.2] RADON Consortium, Deliverable D2.2: Final requirements, 2020.

[D2.3] RADON Consortium, Deliverable D2.3: Architecture and integration plan I, 2019.

[D2.4] RADON Consortium, Deliverable D2.4: Architecture and integration plan II, 2020.

[D3.4-Data] RADON Consortium, Artifacts for Deliverable D3.4: Continuous testing tool I, 2020. https://doi.org/10.5281/zenodo.3895701

[D3.5] RADON Consortium, Deliverable D3.5: Continuous testing tool II, 2021

[D4.3] RADON Consortium, Deliverable D4.3: RADON Models I, 2019.

[D4.4] RADON Consortium, Deliverable D4.4: RADON Models II, 2020.

[D5.5] RADON Consortium, Deliverable D5.5: Data pipeline orchestration I

[D6.1] RADON Consortium, Deliverable D6.1: Validation Plan, 2019/2020.

[D6.2] RADON Consortium, Deliverable D6.2: Initial Validation Results, 2020.

[DICE17] DICE project, Deliverable 5.5. DICE testing tools –Final version, 2017.

[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

[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

[JMeter20] Apache Software Foundation: Apache JMeter, https://jmeter.apache.org/, 2020.

[Kru95] Philippe Kruchten: The 4+1 View Model of Architecture. IEEE Software 12(6): 42-50, 1995.