D5.2	
Version	1.0
Author	ZHAW
Dissemination	PU
Date	31-12-2019
Status	FINAL

D5.2 ElasTest Test Support Services v2

Project acronym	ELASTEST
Project title	ElasTest: an elastic platform for testing complex distributed large software systems
Project duration	01-01-2017 to 31-12-2019
Project type	H2020-ICT-2016-1. Software Technologies
Project reference	731535
Project website	http://elastest.eu/
Work package	WP5
WP leader	Andy Edmonds
Deliverable nature	Public
Lead editor	Andy Edmonds
Planned delivery date	31-12-2019
Actual delivery date	20-12-2019
Keywords	Open source software, cloud computing, software engineering, operating systems, computer languages, software design & development, service delivery.

Funded by the European Union

License

Contributors

Name	Affiliation
Juan Navarro	URJC
Mica Gallego	URJC
Varun Gowtham	TUB
Cesar Sanchez	IMDEA
Felipe Gorostiaga	IMDEA
Pablo Chico de Guzman	IMDEA
Luis Miguel Danielsson	IMDEA
Kimon Moschandreou	REL
Avinash Sudhodanan	IMDEA
Juan Caballero	IMDEA
Andy Edmonds	ZHAW

Version history

Version	Date	Author(s)	Description of changes
0.1	29.03.2018	Andy Edmonds	Initial draft and outline.
0.2	04.11.2019	Francisco Diaz, Avinash Sudhodanan	Initial contributions on EUS and ESS.
0.3	04.11.2019	Andy Edmonds	Editing and formatting.
0.4	06.11.2019	Varun Gowtham, Cesar Sanchez, Felipe Gorostiaga	Initial contributions on EDS and EMS.
0.5	12.11.2019	Kimon Moschandreou	Initial contribution on EBS.
0.6	13.11.2019	Andy Edmonds	Integration of initial contributions complete.
0.7	15.11.2019	Francisco Gortázar Bellas	Updates on EUS (contributions, SotA).
0.8	21.11.2019	All	Updates and edits to sections 4 and 5.
0.9	25.11.2019	All	Final contributions before peer review.
1.0	20.12.2019	Magda Kacmajor, Andy Edmonds	Finalisation of deliverable after peer-review comments.

Table of Contents

Index of Figures

Index of Tables

Glossary of Acronyms

Acronym	Definition
CI (Continuous Integration)	This refers to the software development practice with that name.
Cross-Origin State Inference (COSI) attack	A web attack where a malicious web site infers the state of the victim at another web site
EK	ElasTest Kubernetes
ElasTest Big data analysis Service (EBS)	One of ElasTest's Test Support Services. See section 4.4.
ElasTest Monitoring Service (EMS)	One of ElasTest's Test Support Services. See section 4.3.
ElasTest Security check Service (ESS)	One of ElasTest's Test Support Services. See section 4.5.
ElasTest sensor, actuator and Device emulator Service (EDS)	One of ElasTest's Test Support Services. See section 4.2.
ElasTest Test and Orchestration Manager (ETM)	Main ElasTest coordinating entity of T-Jobs
ElasTest User impersonation Service (EUS)	One of ElasTest's Test Support Services. See section 4.1.
FOSS (Free Open Source Software)	This refers to software released under open source licenses.
Fundamental Modelling Concepts (FMC)	A modelling framework for the description of software-systems.
HEK	Highly Scalable ElasTest Kubernetes
IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service)	This refers to different models of exposing cloud capabilities and services to third parties.
IIoT	Industrial Internet of Things
Instrumentation	This refers to extending the interface exposed by a software system for achieving enhanced controllability (i.e. the ability to modify behaviour and runtime status) and observability (i.e. the ability to infer information about the runtime internal state of the system).
Man-in-the-Middle (MitM) attack	A type of security exploit where an interloper is transparently inserted between what should be a one-to-one communication.

MoM	Monitoring Machines
QoS (Quality of Service) and QoE (Quality of Experience)	QoS and QoE refer to non-functional attributes of systems. QoS is related to objective quality metrics such as latency or packet loss. QoE is related to the subjective quality perception of users. In ElasTest, QoS and QoE are particularly important for the characterization of multimedia systems and applications through custom metrics.
Service Oriented Architecture (SOA)	An architectural style used to design distributed service-based applications.
SiL (Systems in the Large)	A SiL is a large distributed system exposing applications and services involving complex architectures on highly interconnected and heterogeneous environments. SiLs are typically created interconnecting, scaling and orchestrating different SiS. For example, a complex microservice-architected system deployed in a cloud environment and providing a service with elastic scalability is considered a SiL.
SiS (Systems in the Small)	SiS are systems basing on monolithic (i.e. non-distributed) architectures. For us, a SiS can be seen as a component that provides a specific functional capability to a larger system.
SuT (Software under Test)	This refers to the software that a test is validating. In this project, SuT typically refers to a SiL that is under validation.
T-Job (Testing Job)	We define a T-Job as a monolithic (i.e. single process) program devoted to validating some specific attribute of a system. Current Continuous Integration tools are designed for automating the execution of T-Jobs. T-Jobs may have different flavours such as unit tests, which validate a specific function of a SiS, or integration and system tests, which may validate properties on a SiL as a whole.
Test Support Service (TSS)	A service acquired on-demand and use in support of a T-Job.
TiL (Test in the Large)	A TiL refers to a set of tests that execute in coordination and that are suitable for validating complex functional and/or non-functional properties of a SiL on realistic operational conditions. We understand that a TiL can be created by orchestrating the execution of several T-Job.
TO (Test Orchestration)	The term orchestration typically refers to test orchestration understood as a technique for executing tests in coordination. This should not be confused with cloud orchestration, which is a completely different

	concept related to the orchestration of systems in a cloud environment.
TORM (Test Orchestration and Recommendation Manager)	Is an ElasTest functional component that abstracts and exposes to testers the capabilities of the ElasTest orchestration and recommendation engines.
Web Real-Time Communication (WebRTC)	Enables audio and video streams to work in web pages using a direct peer-to-peer paradigm.

1 Executive Summary

Developers and therefore end-users receive productivity gains from the use of supporting services to deliver and use their own servce. The same should also be provided to testers of software and services. In this deliverable we report updated results of designing and implementing Test Support Services that are used within the ElasTest platform. Through the use of orchestration in WP4, these services can also be presented to the T-Job owner in a uniform fashion.

In this deliverable, after the introduction, we detail the updated (from [D5.1]) common elements that all services are required to have in order to be used as ElasTest Test Support Services. For each of the five services in Work Package 5 (WP5), we describe the updates related to each including aspects related to design and implementation. Those Test Support Services are:

- **ElasTest User impersonation Service (EUS)**: This service enables the impersonation of end-users in their tests through GUI (Graphical User Interface) instrumentation and through mechanisms for QoS (Quality of Service) and QoE (Quality of Experience) evaluation.
- **ElasTest sensor, actuator and Device emulator Service (EDS)**: This service is useful for enabling tests to emulate customized device behaviour at the time of testing IoT (Internet of Things) applications.
- **ElasTest Monitoring Service (EMS)**: This service leverages runtime verification ideas (in turn inspired by formal verification) to represent the system behaviour as sequences of events that can be monitored in universal ways.
- **ElasTest Big data analysis Service (EBS)**: Enables the collection, analysis and visualization of large volumes of logs.
- **ElasTest Security check Service (ESS)**: For security vulnerability checking targeting specifically the problems of the main large-scale deployed system.

Ultimately, the main output of this deliverable is the software, which is delivered from this technical work package, including common guidelines on how to create a Test Support Service and the architectures and implementations of each of the five Test Support Services. All of the implementations are available on GitHub[1]. Nonetheless, from the first 18 months of the ElasTest project not only has there been advancements n the software but also a shift in reporting scientific progress in the form of presentations and papers. Each service also details its contributions in this respect.

[1] https://github.com/elastest/

2 Introduction

In this deliverable we describe the work of WP5 from month 19 up to month 36 of the project ElasTest.

The work package five's goal has been to provide (design, implement) and deliver Test Support Services (TSS) that support and provide additional functionality in the creation of T-Jobs. These services are provided as on-demand services when needed by T-Jobs. This means that only when the functionality is declared, the service/services are instantiated. These services are defined according to the general architecture principles and style specified in D2.3, namely as cloud-native, microservice-based services [TSS1]. They also satisfy the requirements and validation specifications described in D2.5. Using the functionality of these services is through well-defined API and UI interfaces. How the API is defined, again follows the D2.3 mandated adoption of OpenAPI[2], however user interfaces are considered to be optional.

These services are to be ultimately used and consumed by the ElasTest Test and Orchestration Manager (ETM), however they can be used independently of this through the API or UI of the ElasTest Service Manager (ESM).

This deliverable is aimed at those wishing to understand how the updated Test Support Services are designed and implemented through:

1. Understanding the common guidelines required to be adhered by each service.
2. Understanding how services can be design and implemented and integrated into the ElasTest service. In this case, the "how" is given by example of the five different services within WP5 and their related updates.
3. Providing information about each service's beyond the state of the art capabilities and specific details through various service-specific publications made during the second period (M19-M36).

This deliverable is structured in the following way. In section 3, we present updates to the common guidelines that each service should follow from design, through implementation to deployment and delivery. In section 4, we provide the detailed information on updates to the features, architecture and other outputs of each Test Support Service. Finally, in section 5 we conclude the deliverable.

Reading note: When reading this deliverable, it is useful to have [D5.1] available to easily refer to the section that is cross-referenced. This has been done to reduce unnecessary text reproduction.

[2] https://www.openapis.org

3 Test Support Service Management

This section contains the updates and necessary changes to support the service management based on the work reported in [D5.1].

Below is a table that lists where to read and understand on a service management topic and lists what sections are updated and dealt with in this deliverable. Topics that do not have an update are considered as stable and already dealt with sufficiently in [D5.1].

Topic	Updated	Description
Definitions	No	See [D5.1]
TSS Lifecycle	No	See [D5.1]
TSS interactions with ET	No	See [D5.1]
TSS Description	**Yes**	**See D5.2, Section 3.1.4**
TSS Monitoring	No	See [D5.1]
TSS Health Checks	No	See [D5.1]
TSS & Creating Computational Resources	**Yes**	**See D5.2, Section 3.1.7**
TSS Costing	No	See [D5.1]
TSS Testing	**Yes**	**See D5.2, Section 3.1.9**
TSS Documentation	No	See [D5.1]
TSS Creation	No	See [D5.1]

Table 1 TSS Management Changes

Test Support Services are services used by tests via T-Jobs. They provide additional functionality that the application/service developer can use but not be responsible for. Adding support services allows for rapid additional specific functionality. Consider the EUS; it provides to tests the ability to control web browsers and emulate a real user using it. All services should provide some way of programmatic usage of the service, so that it can be configured/used by the test code. Naturally many services provide an additional graphical interface to be used directly by the tester. Again, if we consider the EUS; it provides the web browser inside ElasTest main interface to be managed by the tester if he wants.

The technologies used in WP5 are directly influenced by the technology decisions made in WP3. As such, all services are presented for deployment as Docker[3] container images that are composed and deployed using both Docker's docker-compose[4] technology and Kubernetes. With new updates to the ESM, Kubernetes[5] can now be used as an alternative technology to carry out the same composition as docker-compose, but with the added benefits of dynamic scaling and additional reliability.

.

3.1.1 Definitions

Please see [D5.1].

[3] https://www.docker.com

[4] https://docs.docker.com/compose/

[5] https://kubernetes.io

3.1.2 TSS Life Cycle

Please see [D5.1].

3.1.3 TSS Interaction with ElasTest

Please see [D5.1].

3.1.4 TSS Description

The definition of a TSS is contained in a file named `elastestservice.json`. In ElasTest, this description file is hosted within the service implementation repository, however, it can be located anywhere the ETM has access to. New TSS's can now also be imported to ElasTest via the ESM's import functionality. The only change of note here is that the `Manifest's manifest_content` field can now contain a docker-compose description or a Kubernetes manifest that describes the service's deployment implementation. From [D5.1], the `elastestservice.json` is a JSON document and has the following structure:

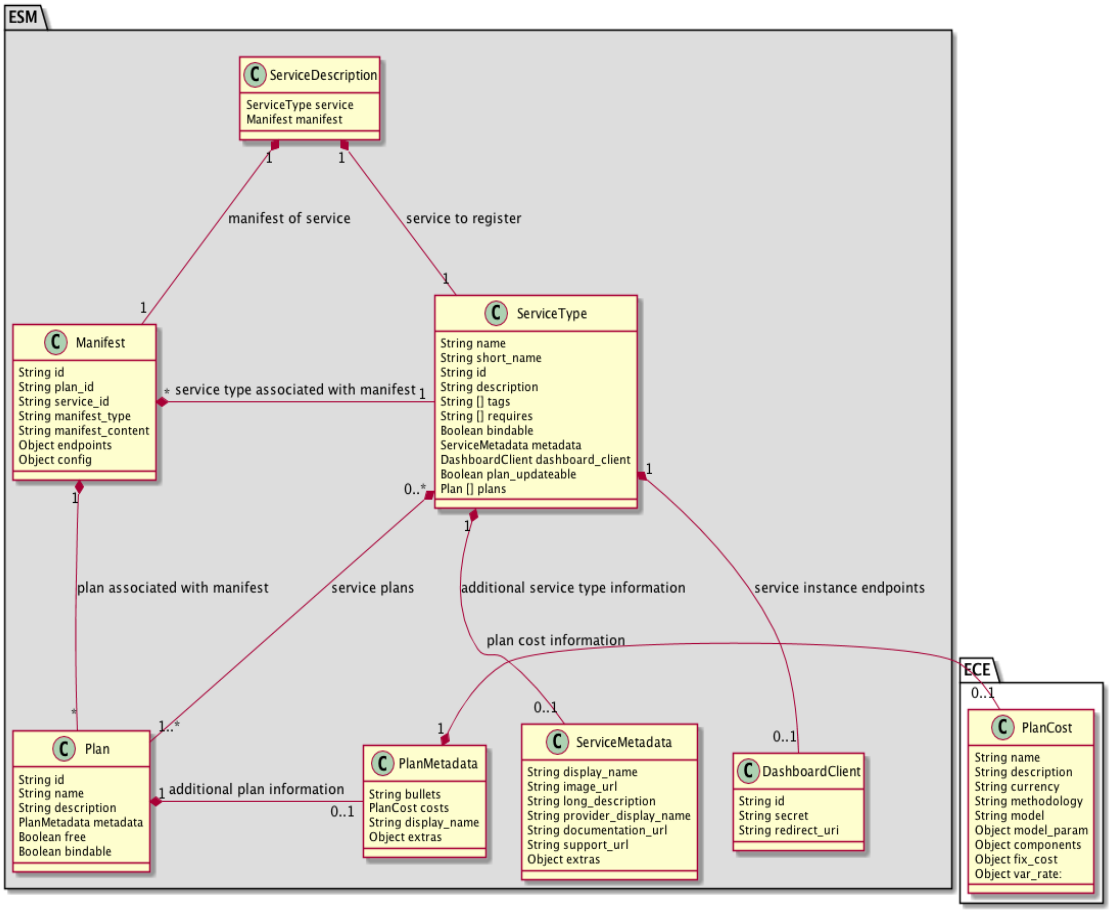

Figure 1 TSS Descriptor File's Document Model.

3.1.5 TSS Instance Monitoring for T-Jobs

Please see [D5.1].

3.1.6 TSS Health Check

Please see [D5.1].

3.1.7 TSS & Creating New Computational Resources

All services can create new containers during their execution. This allows services to request further resources, allowing them to scale themselves.

With using docker-compose-based manifests, each service implementation manages the scaling of itself. Services have a fixed set of resources depending on the selected plan and so the goal of a service implementation is more to maintain the service level delivered to the user of the service.

With using Kubernetes-based manifests, dynamic scaling is now supported by the support of Kubernetes in the ESM. This dynamic scaling provided for service implementors and providers in order to provide reliability to their service consumers. Ultimately, how many replicas of a service component instance (e.g. a container) is managed by declaring how many replicas are required in the service's manifest. An example of setting the number of replicas can be seen within the Kubernetes documentation[6].

3.1.8 TSS Costing

Please see [D5.1].

3.1.9 TSS Testing

It is important to implement tests for TSSs. All types of tests are important and should take into account the test pyramid[7]. There is direction on what's needed to test the implementation of a service in [D5.1]. Also, the work that happens in WP6 is important and related.

Nonetheless, during the course of WP5 work it was seen that a useful contribution could be made that validated the service manifest. This has the advantage of avoiding errors that arose from manifests that did not comply correctly to the docker-compose model.

TSS Manifest Validation

Despite an active research field around cloud-native architectures, the dominant industry trends are around container-native architectures. These are centred around Docker images and declarative orchestration (composition) files referencing the images. A major issue in the context of end to end testing is the drill-down into these files. While existing scanning tools cover the quality of Docker container images well (DIVA [TSS2], Clair [TSS3], Hadolint [TSS4]), a research and innovation gap was identified for the orchestration files. We have therefore introduced Docker Compose Validator, with a

[6] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

[7] https://martinfowler.com/bliki/TestPyramid.html

primary focus on Docker Compose files but also capable of addressing quality issues in Kubernetes deployments and OpenShift templates. The validator is able to check for invalid syntax such as duplicate names which are not checked by much of the Docker-based tools in regular operation. This concerns container names, port numbers and labels, among other syntactic elements. A heuristic validation is able to reveal typical typos through string similarity. Moreover, it can discover inconsistencies in labels to enforce labelling policies. All validation results can be output synchronously to the terminal to aid developers or be asynchronously submitted to a Kafka pub/sub system for further processing. An integration with the ElasTest framework itself, to run the validator as a service, has been conceptually designed and can be integrated into build pipelines but as of yet has not been deployed onto the CI/CD infrastructure[8] of ElasTest.

3.1.10 TSS Documentation

Please see [D5.1].

3.1.11 TSS Creation

Please see [D5.1].

[8] http://ci.elastest.io/jenkins/

4 ElasTest Test Support Services

In this section each of the five core ElasTest Test Support Service (TSS) is described. All of these services are available from the main ElasTest repository[9] and are available under the open source Apache 2.0 license[10], unless otherwise specified.

4.1 ElasTest User Impersonation Service

4.1.1 Introduction

The ElasTest User Impersonation Service (EUS) TSS is devoted to providing the appropriate technologies for impersonating users in end-to-end tests. This is achieved by handling GUIs (Graphical User Interfaces) using automation techniques.

In the second period of the project we focused into expand other functionalities more related with elasticity and Quality of Experience, as specified in the DoA. However, due to demands of the verticals, we also included specific features to enable a boost in productivity for them. Currently, EUS provides the ability to impersonate users manipulating web applications with Chrome and Firefox, and support can be expanded to other browsers, e.g. Opera or Edge, and to allow the impersonation of mobile applications too.

On the one hand, the EUS can now deploy browsers on an infrastructure provided by AWS or on a cluster of Kubernetes when ElasTest is running in EK (ElasTest Kubernetes) or HEK (Highly Scalable ElasTest Kubernetes) modes, so that the resources used by ElasTest can be increased when the workload is high.

On the other hand, the EUS now provides a measure that indicates the quality of a video stream with respect to the user experience, for example, by comparing the video streamed and received between two participants in a call.

Finally, we included specific use-case-driven features like cross browsing, which allows testers to perform a manual testing session on a browser and reproduce automatically and in parallel the human interactions on another, different, browser. This is especially useful when the same tests need to be exercised using two different browsers, for instance, Chrome and Firefox.

4.1.2 Features

The list of high-level capabilities provided by EUS at the moment of this writing is the following:

1. Use browsers manually.
2. Drive browsers GUIs in an automated way.
3. Automate and assess WebRTC applications.

[9] https://github.com/elastest

[10] https://opensource.org/licenses/Apache-2.0

4. **New**: Measure the end-user's perceived quality by means of QoE and QoS indicators.
5. Record of browser in automated and manual sessions.
6. **New**: Multimedia QoE video can be analysed using different full-reference algorithms:
    a. Video Multi-Method Assessment Fusion (VMAF) [11].
    b. Structural Similarity Index (SSIM) [12].
    c. Peak Signal-to-Noise Ratio (PSNR) [13].
    d. Visual Information Fidelity in Pixel Domain (VIFp)
    e. Multi-Scale SSIM (MS-SSIM) [14].
    f. Human Visual System-based PSNR (PSNR-HVS and PSNR-HVS-M) [15].
7. **New:** Start multiple browsers at once (Cross Browsing)

These capabilities are exposed by EUS by means of a REST API. The definition to this REST API has been defined using Open API notation. This specification is available on the EUS GitHub repository[16]. Moreover, this API can be reviewed in a web friendly format in the official ElasTest documentation[17].

4.1.3 Baseline Concepts and Technologies

The concepts and technologies for the EUS are the same as in [D5.1]:

- EUS provides a *Browser as a Service* (BaaS) capability
- It is built on top of popular technologies such as Selenium[18] , an extension of the W3C WebDriver recommendation[19].
- To manage the browser instances controlled by Selenium, EUS uses Docker containers.
- These containers can also be initiated with Kubernetes and in an AWS instance, making use of a public AMI image created by the ElasTest project.
- For managing the life cycle of containers, EUS is using the service of ElasTest Platform Manager (EPM) core component.

4.1.4 Component Architecture

The following figure shows how the EUS class diagram looks like after implementing the latest functionalities and modifications.

[11] https://github.com/Netflix/vmaf
[12] http://www.imatest.com/docs/ssim/
[13] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio
[14] https://en.wikipedia.org/wiki/Structural_similarity#Multi-Scale_SSIM
[15] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio#Variants
[16] https://github.com/elastest/elastest-user-emulator-service/blob/master/api.yaml
[17] https://elastest.io/docs/api/eus/
[18] https://www.seleniumhq.org/
[19] https://www.w3.org/TR/webdriver/

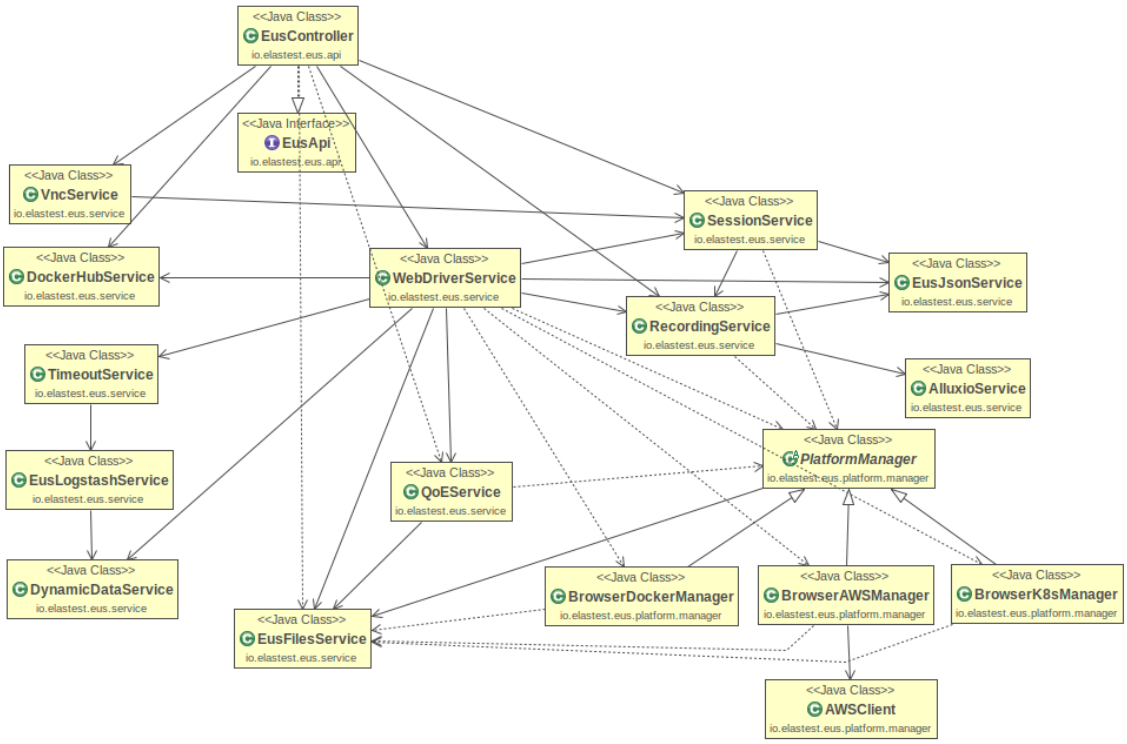

Figure 2 EUS Class diagram

4.1.4.1 Sequence Diagrams

To explain how the new functionalities added to the EUS work, two sequence diagrams are described below (Figure 3).

In the class diagram above, you can see several classes that did not exist in the previous version of this document. Some of these classes included in the hierarchy under the class `PlatformManager` (`BrowserDockerManager`, `BrowserAWSManager` and `Browserk8sManager`) are responsible for provisioning the browsers in the different platforms with which ElasTest works (Docker, AWS and K8s).

The following sequence diagram imported from the previous version of this document illustrates how to create and delete a browser within the T-Job life cycle, but in this case the EPM will be able to provision the browser on one of the platforms mentioned above.

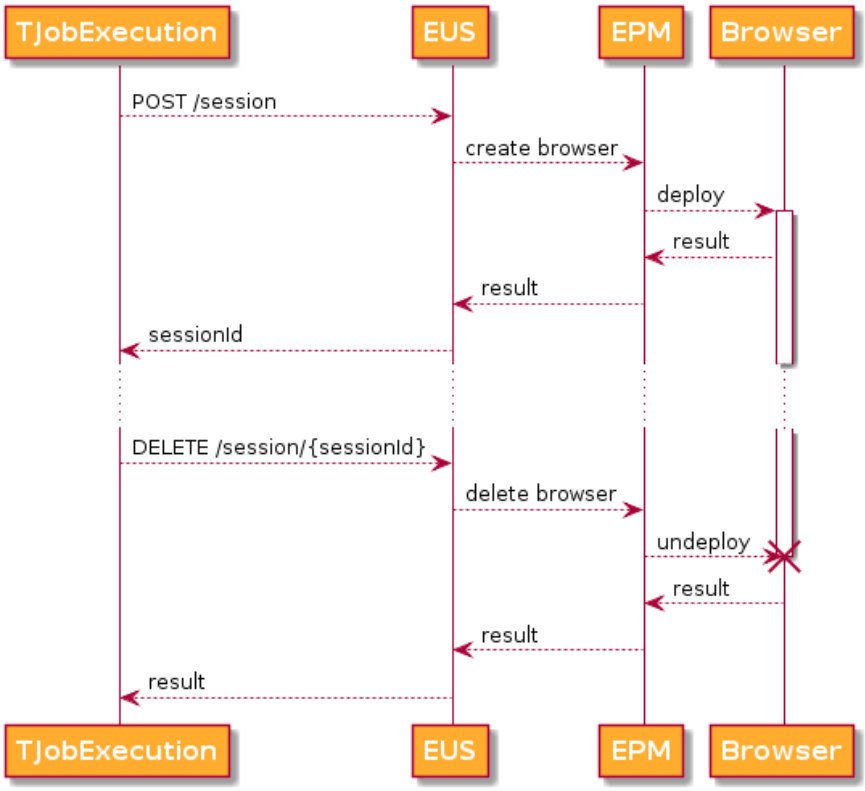

Figure 3 EUS Sequence Diagram of Creation

The following picture (Figure 4) shows the sequence diagram the process of calculating QoE metrics for video, which is calculated from the original video emitted by one end and the video received by another end.

It is assumed that two browsers were previously created: `BrowserPresenter` (the one that sends the video) and `BrowserViewer` (the one that receives it).

The EUS API will be called by a GET request whose path is "`/session/{sessionIdPresenter}/webrtc/qoe/meter/start`" and will be sent as parameters: (1) the `sessionId` of the `BrowserPresenter` and the path of the original video inside it; and (2) the `sessionId` of the `BrowserViewer` and the path of the video received inside it.

The EUS will start the `QoEMeterService`, get the two videos from the browsers to send them to the service and call you asynchronously to calculate the metrics. The REST request is then answered. When the asynchronous process finishes, the EUS will recover the Comma Separated Value (CSV) files generated in the service, it will calculate the average of the metrics for each one of the CSVs and it will save everything in the execution folder, so that it is attached in the execution. Finally, the true value will be assigned to a variable that designates the state of end of the process and meanwhile the test will be in charge of polling the EUS to ask for the state.

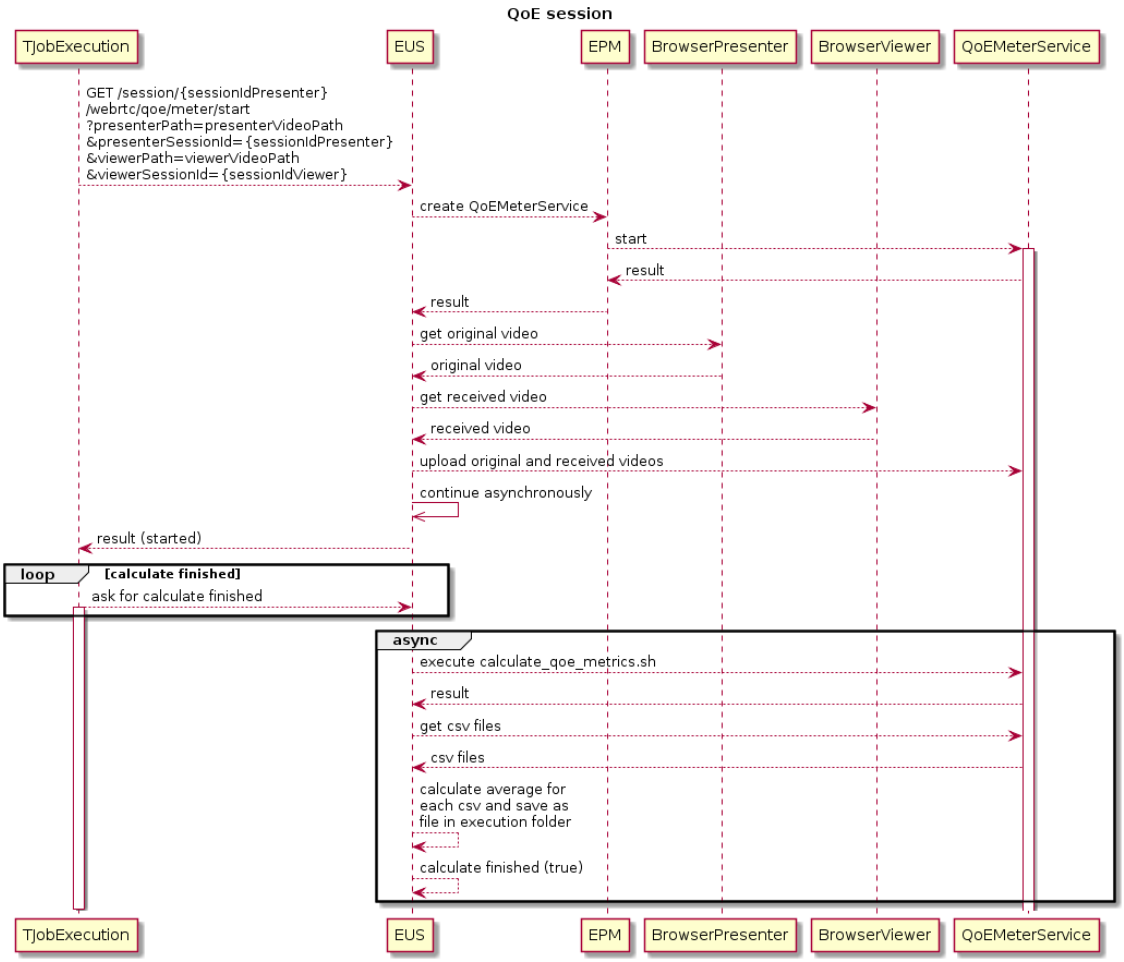

Figure 4 EUS Sequence Diagram of the Process of Calculating QoE

4.1.5 Implementation Details

- The main repository of the EUS is here[20].
- The API of the EUS is available here[21].
- Other than the reporting of testing from WP6 in D6.3, further code coverage can be viewed at codecov[22].

4.1.6 Contributions

As a result of the efforts on task 5.1 (EUS), we have published a paper on the Journal of Computing on understanding and estimating quality of experience in the context of WebRTC applications [EUS1]. The findings of this study enabled us to write another paper on the Journal of Electronics, about practical evaluation of video quality for WebRTC applications using the VMAF algorithm [EUS2]. The results from this paper were implemented as part of the EUS, along with some of the state-of-the-art algorithms, and

[20] https://github.com/elastest/elastest-user-emulator-service
[21] https://elastest.io/docs/api/eus/
[22] https://codecov.io/gh/elastest/elastest-user-emulator-service

can be used seamlessly and transparently to assess the quality of video-based applications that use this WebRTC W3C standard.

Current efforts are on providing an integral way of assessing audio and video quality. This feature is still to be integrated in the ElasTest platform, but URJC is already writing a paper that will be submitted by end December 2019. URJC, the leaders of the EUS, attended and presented at SeleniumConf 2019 based on their experiences of Selenium.

4.1.7 Progress Beyond the State of the Art

The EUS component pushes forward the state-of-the-art on web testing by providing specific support for assessing the quality of experience of WebRTC applications automatically during a test run. To the best of our knowledge this has not been provided by any other testing tool.

Furthermore, we have set ground on evaluating and managing QoE in the context of the WebRTC standard and adapted streaming QoE algorithms to this real-time communication standard.

4.2 ElasTest Device Emulator Service

4.2.1 Introduction

ElasTest Device Emulator Service (EDS) is a test support service available in ElasTest which provides the functionality of emulating the behaviour of sensors and actuators. It is possible for SuT and/or T-Jobs to access facilities of EDS through a REST API for the purpose of requesting a specific type of emulated sensor or actuator, and further wire the requested emulated devices together to form an application. This application depends on the cause and effect relationship provided by the sensor and actuator respectively. For a more detailed understanding refer to [D5.1].

4.2.2 Features

The features provided by EDS are the following:

1. Dedicated gateway that can handle all the REST requests. The gateway is also responsible in directing incoming requests to subscribed applications. It also acts as a placeholder for values pushed by applications.
2. Dedicated backend application that supports dynamic initialization of sensors.
3. Method to wire the emulated devices together to form IoT applications.
4. Method to change the behaviour of the emulated devices during run-time and in turn affect the behaviour of the application.

More details can be found in the previous [D5.1].

4.2.3 Baseline Concepts and Technologies

EDS relies on OpenMTC, a reference implementation of the oneM2M machine to machine (M2M) communication standard. EDS implements the functionality of device emulation using OpenMTC and opens up through the REST API so that external applications can make use of EDS.

More details can be found in the previous [D5.1].

4.2.4 Component Architecture

Although, the component architecture, the API and sequence diagrams have not changed since [D5.1], efforts have gone into making sure that EDS is suitable for using the vertical demonstrator use cases. Apart from the already mentioned features in [D5.1], improvement was made such that it was possible to modify the behaviour of the emulated device during run time. This feature was used in the validation experiments of WP7.

The improvements made can be accessed using the following request on the Device Emulator (DE).

Method	URL	Description
POST	`/onem2m/{DE name}/request`	Post a request to DE

The JSON object in the POST request can contain a “modify” field containing the following sub-fields used to change the configuration of the DE during run time:

Sub-field	Type	Value	Action
`onoff`	`String`	`ON`	Switch on the DE, either sensor or actuator.
`onoff`	`String`	`OFF`	Switch off the DE, either sensor or actuator.
`period`	`Integer`		Increase or decrease the data generation period of the sensor.
`delay`	`Integer`		Increase or decrease the actuation delay of the actuator.
`min`	`Integer`		Minimum value of uniform distribution using which sensor samples a data point.
`max`	`Integer`		Maximum value of uniform distribution using which sensor samples a data point.

4.2.5 Implementation Details

- The main repository of EDS is available on GitHub[23].
- The API documentation is available here[24].
- Other than the reporting of testing from WP6 in D6.3, further code coverage can be viewed at codecov[25].

4.2.6 Contributions

The EDS provides an extendable framework for the purpose of device emulation. Together with ElasTest, EDS brings together testing IIoT applications with CI. EDS is developed around OpenMTC which is also a FIWARE enabler. Specific contributions include:

1. Identification of basic devices used to compose an IIoT application. This enables user construct complex applications using basic devices as building blocks.
2. Implementation of a generic device emulator, able to inherit a device model and furthermore use the device model to emulate a specific device.

[23] https://github.com/elastest/elastest-device-emulator-service
[24] http://elastest.io/docs/api/eds/
[25] https://codecov.io/gh/elastest/elastest-device-emulator-service

3. The user can now register an application with the framework and request specific devices and wire them together in any fashion. The oneM2M gateway in EDS enables this functionality, where a user can publish or subscribe to oneM2M container. Such a container can act as placeholders for either sensor data or actuator signals or act as a data compartment for isolating application activities.
4. Ability to change the behaviour of a specific device during run time, such that it affects a required aspect of the application.
5. Not restricted to the given set of devices, a user can easily extend the device emulators to further emulate other devices. The device emulators can also be extended as function emulators, which can emulate a functionality of an IIoT application.

4.2.7 Progress Beyond the State of the Art

The component progresses the state of the art in the following ways:

- Provides a component as a service for device impersonation compliant with the oneM2M standard, focusing on the industrial environment.
- Not only does the component act as a standalone service, but when paired with other components and facilities offered by ElasTest, it is possible to achieve higher testing efficiency, specifically for testing IIoT/IoT applications.

These progresses are validated by the State of the Art report, as presented in section 4.16 "Device impersonation" of D2.4: SotA Revision v2. The EDS component can be easily integrated into existing oneM2M compliant domains so that the penetration effort for using ElasTest for the purpose of testing, is reduced.

4.3 ElasTest Monitoring Service

4.3.1 Introduction

The ElasTest Monitoring Service (EMS) eases the development of T-Jobs by correlating events received from the SuT, and optionally from the T-Jobs and other services. Tests are simpler and more reusable because the EMS is programmable using a domain-specific language (DSL) specifically tailored for testing. In a nutshell, the EMS receives "events" from the SuT that are correlated, and trigger responses to the T-Job (depending on the condition described by the T-Jobs in the subscription to the EMS). Both input and output events can contain rich information.

The Monitoring Machines (MoM) and the Stampers are the core elements of the EMS. They provide the DSL that simplifies the description of whether a test passes or fails.

The Stampers remain the same as in the previous version of this deliverable.

The expressivity of the MoMs language has been augmented to describe new properties based on the feedback of industrial partners.

4.3.2 Features

In the new version, we have included the following features:

- Support for "nested" **structured input events** encoded as JSON. Before, the payload of a message was a flat string, that was opaque to the EMS, and could only be sent to the T-Job without further analysis. Consequently, only meta-data in the event was available for the analysis. Now, the DSL allows to interpret a string field (e.g. the payload) in a JSON object (the input event), allowing to directly decode the string to a JSON object itself, provided that it has the correct structure of a serialized JSON.
  Consider for example the following event, in JSON notation:

  ```
  {"timestamp":"2019-11-06T15:47:32Z","message":"{"value": 5}"}
  ```

  We can use the new `getJSON` feature of the EMS to unmarshal the value of the field "message" and access its numeric field "value" in the following way:

  ```
  stream num cpuload := e.getJSON(message).getnum(value)
  ```

  This has proven very useful in simplifying tests for the IoT vertical.

- Support for the emission of **structured output** as JSON. Before, the payload of the output events could only be a value from the set of datatypes supported by the EMS. Now, the language permits to define templates of output events that get filled dynamically with the values computed as observations by the EMS. In this manner, the T-Job can receive rich and structured information and not only pass/fail. This greatly simplifies the reporting and investigation of the causes of a test failure. This feature was also requested by the IoT vertical. For example,

we can define a template for a trigger directive that includes (a) the value of the numeric stream `cpuload` in a field with the same name and (b) the id of the CPU in a string field, over a channel `#cpuload` whenever the boolean stream or predicate `highload` becomes true using the following line:

```
trigger highload do emit `{“cpuload”: “%cpuload”, “cpuname”: “%name”}` on #cpuload
```

- The Monitoring Machines can interpret a **`timestamp`** in a string field as a numeric value, which allows to calculate durations and delays. This can be used to define tests in terms of the absence or presence of events within an interval of time. Take for example the following two events:

  ```
  {"timestamp":"2019-11-06T15:47:32Z","message":"{"value": 35 }"}
  ```

  We can get the timestamp of this event as the seconds elapsed since the Unix epoch using:

  ```
  stream num epoch := e.getnum(timestamp)
  ```

- The Monitoring Machine language now allows accessing the value of streams at **previous instants**. This feature allows to perform a computation between successive values of the same stream. In particular, in combination with the support for timestamps, this feature allows to compute the delay between two successive events. Following the previous example, we can define a stream to calculate the delay between every two successive events:

  ```
  stream num epochdiff = epoch - last epoch
  ```

- An important new feature is **parameterization**. The new Monitoring Machines language allows defining many streams at once using a vector notation, which makes it easier to scale tests to many components, by reducing duplications of stream definitions. For example, if the input events contain a field with the CPU id like this:

  ```
  {“cpuid”: 2, “cpuload”: 85}
  ```

  we can get the load of every CPU in a different stream using the following line:

  ```
  stream num cpuload [i:0..7] := if e.getnum(cpuid) = i
         then e.getnum(cpuload)
  ```

  With this feature the IoT vertical can uniformly define tests that check many different sensors at once, without redefining the test.

- **Else values**. The EMS language now supports assigning a value to a stream when a condition is not met, via a newly added `If-then-else` operator. For example, we can now define a default value for a stream if it is missing a field:

```
stream num cpuload := if e.path(cpuload)
                          then e.getnum(cpuload)
                          else 0
```

Before, the else value could not be assigned a value and the stream would be undefined.

- The event subscribers can connect to a websocket server at port 8181 in the EMS to receive events sent to the `#websocket` channel. This simplifies the development of T-Jobs as the websocket interface is simpler than the other interfaces previously offered (RabbitMQ and Logstash).

The EMS can now be used to perform **offline monitoring**, which allows to rerun a test over the dump of an ElasTest T-Job execution. This can be used to re-evaluate a passed test or to evaluate a new test on a finished execution, as long as the action performed upon the SuT are compatible between the executed test and the new test. This compatibility can also be easily monitored offline. For example, a test that exercises a web service can be used to check the property that every request is answered timely, even though the original test was not about checking response times. With the offline testing feature, the new test will check that every request was responded timely using the dumped trace, without the need of rerunning the System Under Test. This feature has the potential to define few executions and check them post-mortem simplifying test suites and reducing costs.

4.3.3 Baseline Concepts and Technologies

The concepts and technologies are the same as in [D5.1]:

- The EMS defines a REST API to describe the methods in the configuration endpoint. The backend that serves the API was developed using Go-swagger[26].
- The EMS uses Logstash[27] as the input layer to receive events from different sources, and also as the output layer to deliver events to different subscribers.
- The part of the engine that performs the computation according to the definition of the Monitoring Machines and the Stampers was written in Go.
- We use Pigeon[28] to generate the parser for the MoMs and the Stampers.
- For interprocess communication, we use Protocol Buffers[29].

[26] https://github.com/go-swagger/go-swagger

[27] https://www.elastic.co/products/logstash

[28] https://github.com/mna/pigeon

[29] https://developers.google.com/protocol-buffers

4.3.4 Component Architecture

The design and architecture have remained the same as reported in [D5.1], even though the functionality of the subcomponents is more sophisticated.

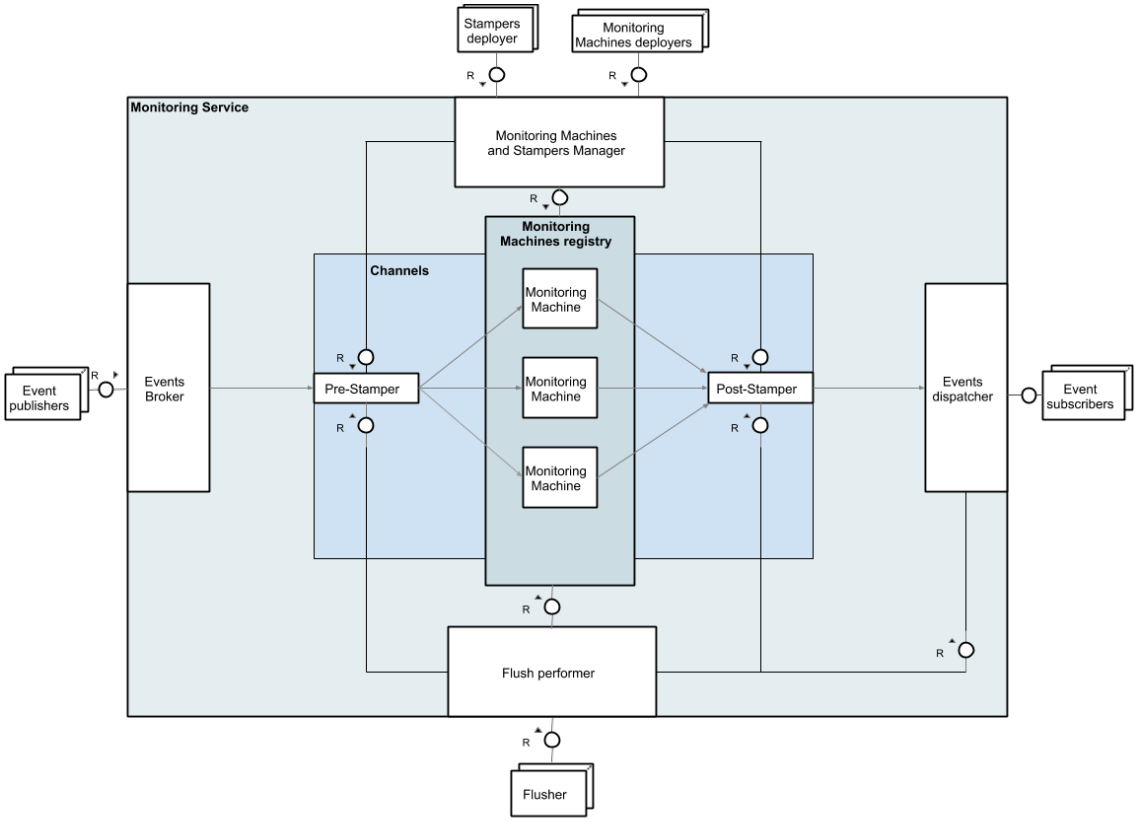

Figure 5 EMS Architecture

The EMS can be configured by the Stampers/MoMs deployers via the Monitoring Machines and Stampers Manager to correlate events that are sent by the Event Publishers. The input events are handled by an Events Broker in the Monitoring Service and sent to the engine. The Pre-Stamper in the engine will assign a channel to every input event, and the deployed Monitoring Machines will compute and possibly generate output events. These newly created output events are passed through the Post-Stamper, then routed to the Events Dispatcher and finally output to the registered Event subscribers. The state of the EMS can be flushed at any time to the initial state by an external Flusher exercising the Flush performer.

A more detailed explanation of every architectural component and their interactions can be found in [D5.1].

Use Case Diagrams

The use cases of the Monitoring Service remain unchanged: the external components can publish events, manage the Monitoring Machines and the Stampers, subscribe to receive events, or reset the service to its initial state.

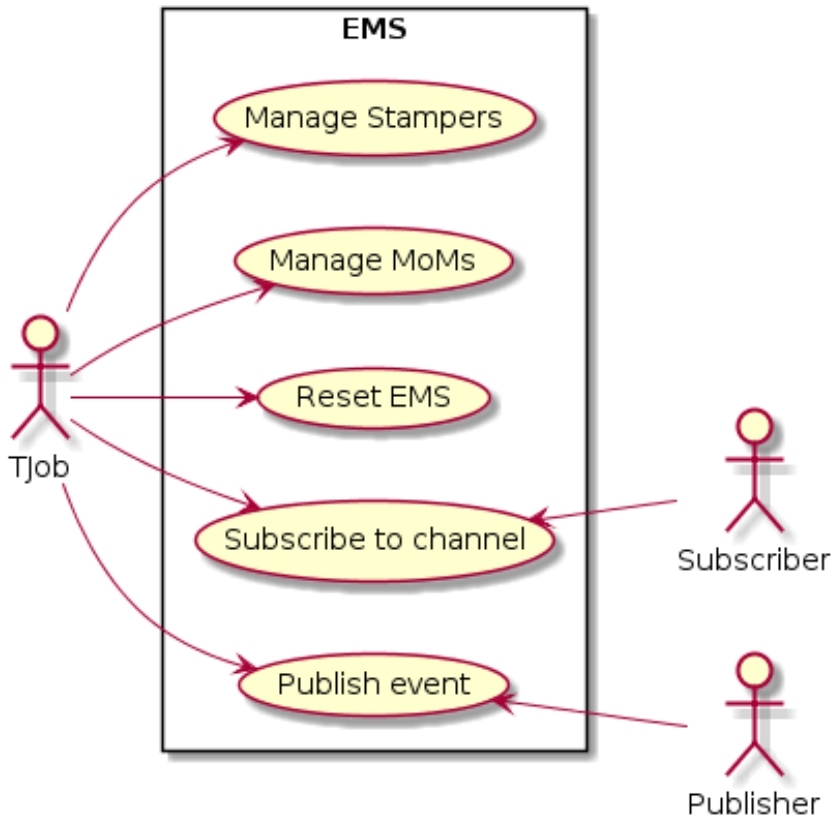

Figure 6 EMS Use Cases

More in-depth explanation of the use cases, along with their sequence diagrams can be found in [D5.1].

Apart from the endpoints for event publication listed in [D5.1], the EMS now offers a websocket endpoint for event subscription at port 3232.

4.3.5 Implementation Details

- The main repository of the EMS is here[30].
- The API of the EMS is available here[31].
- Other than the reporting of testing from WP6 in D6.3, further code coverage can be viewed at codecov[32].

4.3.6 Contributions

In a nutshell, the main contributions of the Monitoring Service are:

[30] https://github.com/elastest/elastest-monitoring-service/
[31] http://elastest.io/docs/api/ems/
[32] https://codecov.io/gh/elastest/elastest-monitoring-service

- Striver, a language for defining and monitoring properties over timed event streams online[33].
- dLola, a framework to monitor properties online in a distributed manner[34].
- i2kit, an orchestration tool that creates virtual machines using Linuxkit, reducing their footprint and keeping the advantages of this method[35].
- "Tests orchestrator" is a tool to define a TiL based on the composition of several ElasTest T-Jobs. This is still a work in progress and not specifically a KPI of the EMS itself. The EMS was identified as a component that could boost the potential of test orchestrations to support data transfer between related tests.
- A framework to analyse the dumped traces of an execution offline.

The influence of each contribution in the Monitoring Service is explained in the following section.

4.3.7 Progress Beyond the State of the Art

The EMS component consists of a very efficient monitoring service that is at the same time descriptive for easing the description of tests. Particularly, this component goes beyond the state of the art in:

- The design of novel algorithms from runtime verification which guarantees theoretical resource bounds, implemented in the core evaluation engine.
- The design of a DSL, called monitoring machines, that offers the test designer the possibility to simplify the description of tests.
- An offline prototype of the monitoring service that allows to evaluate queries offline, enabling the check of tests from the dumps of already executed tests.
- The implementation using development technologies that are very efficient in the use of resources, including memory, like the Go programming language.
- The design and correct implementation of a decentralized evaluation algorithms for the core language.

During the development of the project, it was identified that the resources required by the platform were sometimes prohibitive, which was a risk of the adoption of the technology. Even though the main direction followed by the project was to port the platform to Kubernetes, which allows elasticity and scalability, we also designed in this component an alternative solution that goes beyond the state of the art:

---

33 https://www.researchgate.net/publication/328797226_Striver_Stream_Runtime_Verification_for_Real-Time_Event-Streams_18th_International_Conference_RV_2018_Limassol_Cyprus_November_10-13_2018_Proceedings

34 https://www.researchgate.net/publication/336247644_Decentralized_Stream_Runtime_Verification

35 https://www.researchgate.net/publication/323471324_i2kit_A_Tool_for_Immutable_Infrastructure_Deployments_based_on_Lightweight_Virtual_Machines_specialized_to_run_Containers

i2kit: a tool that allows to convert docker Pods into VMs, which immediately enables scalable platforms by elasticity.

4.4 ElasTest Big Data Service

4.4.1 Introduction

The ElasTest Big-Data Service (EBS) is a Test Support Service that provides an on-demand computing engine based on Apache Spark[36] to be utilized by tests inside ElasTest. The purpose of EBS is to allow tests (T-Jobs) or other components to define their computation requirements using Spark API and use it to perform complex distributed calculations on top of the Spark engine.

4.4.2 Features

There were no new features added in EBS, but in the service was updated in order to deploy it on Kubernetes. The objectives of auto-scaling and Spark optimization are addressed by the new version using Kubernetes tools and techniques Updates are required, however, to allow for complete communication with the Spark cluster.

4.4.3 Baseline Concepts and Technologies

The main purpose of EBS is to provide a scalable and disposable parallelised computing engine to any tests or other components that require it. This computing engine is based on Apache Spark, which is the most widely adopted distributed processing engine currently available. Spark does not only provide a very fast compute engine, but it can also integrate with a wide variety of data sources, allowing for easier future extensions.

The current version of EBS was designed and provided as a computing engine separate from all data persistence services. This approach allows spark clusters to be commissioned and decommissioned on demand, but there is not data-locality awareness on Spark jobs since all data are stored outside EBS docker. However, deployment on Kubernetes splits the EBS in several independent services and establish communication protocols and policies for providing a valid service to the EBS user, who uses an expandable Spark cluster and stores data on HDFS.

4.4.4 Component Architecture

The architecture of the EBS has not changed since D5.1. The current architecture is shown in the below image (Figure 7).

[36] https://spark.apache.org

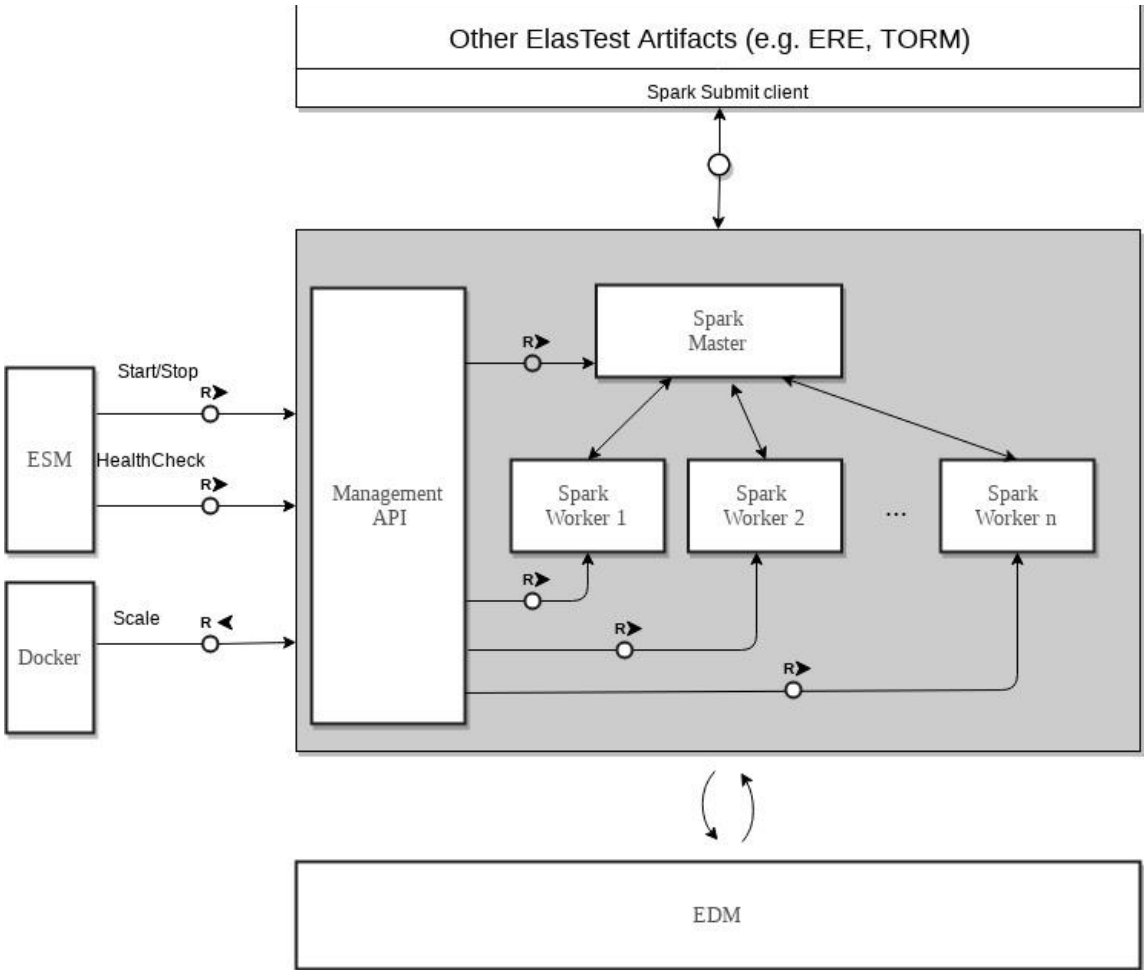

Figure 7 EBS FMC Diagram

For further details on this, please see [D5.1].

4.4.5 Implementation Details

- The EBS code repository can be found on GitHub[37] and is licensed using Apache 2.0.
- Within that repository, there is documentation[38] detailing how to run, use and extend the EBS.
- The API of the EBS can be viewed online here[39].
- Other than the reporting of testing from WP6 in D6.3, further code coverage can be viewed at codecov[40].

[37] https://github.com/elastest/elastest-bigdata-service

[38] https://github.com/elastest/elastest-bigdata-service/tree/master/docs

[39] https://elastest.io/docs/api/ebs/

[40] https://codecov.io/gh/elastest/elastest-bigdata-service

4.4.6 Contributions

Since EBS is mostly implementation work, its research is targeted in exploiting containerized scalable computing architectures for commercial purposes.

More specifically, demand for deploying applications in external customer data centres introduces increased complexity in describing distributed application prerequisites to infrastructure teams. This complexity is extended by the huge diversity of infrastructure architectures due to the multitude of options in that sector (i.e. public/private clouds, bare metal systems, distributed operating systems and container orchestration systems) are all options that may be used alone or in different combinations in order to create the best approach for each case.

In this ecosystem, having a scalable distributed computing engine in the form of a single component, able to be deployed in the majority – if not all – of the aforementioned solutions, without maintaining different configurations, is a huge key to success for software vendors and integrators. It is therefore our main target to bullet-proof, extend and productize the usage of EBS as a reusable, portable and scalable computing engine as a standalone system or a software component of a larger solution. Deploying on Kubernetes and a new API for Spark are steps towards easier encapsulation inside an existing complex infrastructure.

4.4.7 Progress Beyond the State of the Art

There were not any plans for developing beyond the state of the art for EBS.

4.5 ElasTest Security Service

4.5.1 Introduction

The ElasTest Security Service (ESS) enables testing the security of cloud-based web applications. ESS can be leveraged to detect common web application weaknesses and sophisticated privacy attacks.

4.5.2 Features

In the first half of the project we implemented support for testing 40 common web application weaknesses (detailed in [D5.1]).

In the second half of the project we have developed support for the automatic detection of Cross-Origin State Inference (COSI) attacks, a class of browser side-channel privacy leaks[41] that enable inferring the state of a user at a target site. Such leaks can be used to launch different attacks against a user logged into a remote target website (e.g., Amazon, LinkedIn) such as login detection, user deanonymization, account type detection, and access detection. Automatic detection of COSI attacks is a novel feature of ESS, not available in prior web testing tools.

In addition to COSI detection, the final ESS release has also integrated support for TestLink[42], which complements the support for tests written programmatically using TJobs that was implemented in the first half of the project.

4.5.3 Baseline Concepts and Technologies

*What are COSI attacks? C*OSI stands for Cross-Origin State Inference. In a COSI attack, the attacker convinces the user to visit a malicious web page, e.g., by sending the victim an email with the malicious URL. The malicious web page infers the state of the visiting victim user at a target web site (e.g., Amazon, LinkedIn). COSI attacks can have serious consequences such as user deanonymization, login detection, account detection, and access detection. For example, the attacker can infer that the victim is logged into the target site (and thus owns an account at the site), which is dangerous for privacy-sensitive sites. The attacker may also be able to deanonymize the user, e.g., determining that the victim is the author of a blog that criticizes the board of directors of a company.

COSI attacks are a type of browser side-channel attacks, where a malicious web page leverages cross-origin interaction features available in web browsers as side-channels to infer sensitive information about the state of a user in a target site. Since the target site has a different origin, COSI attacks allow violating the same-origin policy implemented by the browser[43].

How does the COSI attack detection work? As input to the COSI attack detection functionality, the tester provides a TJob that contains different state scripts. A state scripts automatically logs into the target site using a configurable browser and the

[41] https://github.com/xsleaks/xsleaks
[42] http://testlink.org
[43] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

credentials of an account with a specific configuration. For example, we may create multiple user accounts with different configurations, e.g., premium and free accounts, two users that own different blogs, or authors that have submitted different papers to a conference management system. During the state script execution, the ESS identifies the state-dependent URLs (SD-URLs) associated to the SuT. SD-URLs are those SuT URLs that respond differently depending on the state of the requester. Once the SD-URLs have been identified, they are matched with an attack class database to identify whether any COSI attacks are possible. The identified attacks are combined to create attack pages and they are reported to the tester. The tester can verify the attacks using the generated attack pages.

4.5.4 Component Architecture

The following figure (Figure 8) details the architecture of the COSI attack detection functionality implemented by the ESS.

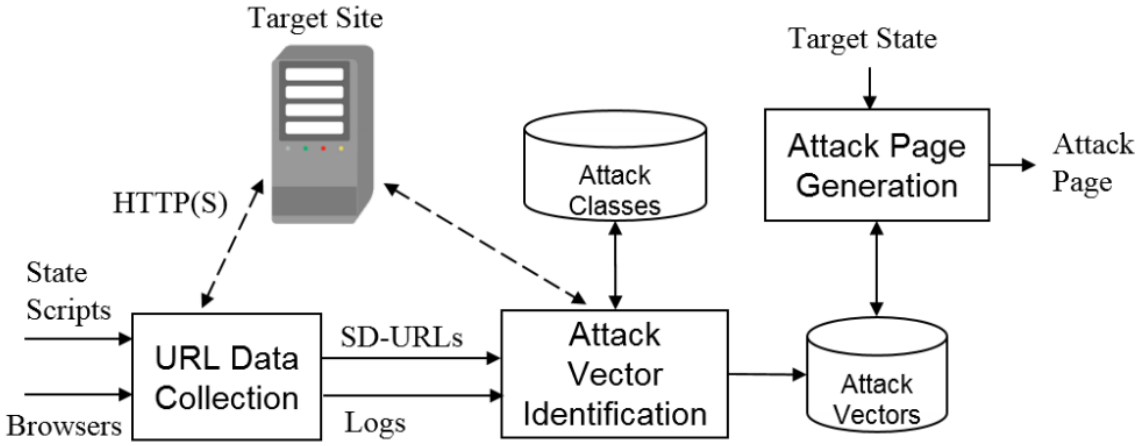

Figure 8 ESS Architecture Update

4.5.5 Implementation Details

- The link to the ESS code is repository is here[44].
- The API documentation is available here[45].
- Other than the reporting of testing from WP6 in D6.3, further code coverage can be viewed at codecov[46].

4.5.6 Contributions

The following are the main contributions achieved during the development of the ESS. These contributions are detailed in our paper [ESS1] and as accepted at NDSS19 [ESS2]:

- We developed a novel approach to identify and build complex COSI attacks that differentiate more than two states and support multiple browsers.

[44] https://github.com/elastest/elastest-security-service
[45] https://elastest.io/docs/api/ess/
[46] https://codecov.io/gh/elastest/elastest-security-service

- We discovered a novel browser side-channel based on the `window.postMessage` feature that affects the three major browsers (Chrome, Firefox, Edge) and can be leveraged to attack popular web sites.
- We performed the first systematic study of COSI attacks, identifying 40 attack classes, of which 19 generalize prior attacks and 21 are new variations.
- We applied the COSI attack detection functionality to 62 targets, including four stand-alone web applications (GitLab[47], GitHub[48], HotCRP[49], OpenCart[50]) and 58 popular web sites from the Alexa top 500. We found COSI attacks against all of them: account deanonymization attacks in 36, account type detection attacks in 5, SSO status attacks in 12, and access detection attacks in 5. The detected attacks include, among others, deanonymization attacks for determining if the victim is:
- the reviewer of a paper in the HotCRP conference management system,
- the owner of a blog in blogger.com,
- the owner a GitLab/GitHub repository.
- We have responsibly disclosed the identified attacks to the affected vendors. Many of them have already fixed the issues (e.g., linkedin.com, amazon.com, and HotCRP) or plan to fix them soon (e.g., Google).

4.5.7 Progress Beyond the State of the Art

As part of the research we did for the privacy checking functionality, we make the following advancements with respect to the state of the art.

- We have presented COSI attacks as a comprehensive category that covers attacks previously considered separate under different names such as login detection attacks, login oracle attacks, cross-site search attacks, URL status identification attacks, and cross-site frame leakage attacks.
- We have developed a novel approach to identify and build complex COSI attacks that differentiate more than two states and support multiple browsers.
- We have implemented the first automated tool for detecting COSI attacks. Previous web testing tools do not provide this functionality.

[47] https://about.gitlab.com/
[48] https://github.com/
[49] https://github.com/kohler/hotcrp
[50] https://www.opencart.com/

5 Conclusions

The work of WP5 has resulted in an integrated platform including the five Test Support Services that adhere to a common design, implementation and deployment approach. Along with this, research activities have taken place and have been published. Specifically, for each TSS we conclude:

- **EUS**: it enables the impersonation of end-users in tests through GUI instrumentation. During the second half of the project, this service has evolved into a fully capable QoE manager, by providing several algorithms that enable end users to assess real-time communication applications in an effortless way. The research carried out in the EUS has led to the publication of two papers on QoE [EUS1, EUS2], and a third one including audio and video assessment is being written at the moment of the submission of this deliverable.

- **EDS**: it enables impersonating/emulating behaviour of sensors and actuators towards building IIoT applications. EDS uses OpenMTC, a reference implementation of M2M communication standard oneM2M. The service can be integrated into existing IIoT application infrastructure with minimal effort. Combined with the facilities offered by ElasTest, a tester can leverage EDS for not only rapid prototyping of IIoT applications, but also can construct test cases and execute such tests to check the correct operation of the application. In addition, it is possible to change the behaviour of the emulated device at runtime, which opens the possibility for testing robustness. The ultimate goal of providing EDS is to help the tester realize large and complex IIoT applications and test them with relative ease. The advantages are the following:

  1. Rapid prototyping: To realize a full scale IIoT application, the corresponding device modules are first procured and then interconnected. This process is time consuming and expensive, in addition, there is no guarantee that at the end the application is worth the investment. Using device impersonation, it is possible to economically assess the feasibility.
  2. Ease of testing with ElasTest: With device impersonation, rapid prototyping brings testing processes which also can vary in size and complexity depending on the scale of testing. EDS with ElasTest, is able to address the issue of simplifying testing activity for IIoT applications

  To conclude, EDS was able to provide the facilities required for IIoT applications for rapid prototyping and simplifying testing processes. It has been primarily used in the IIoT vertical demonstrator of WP7.

- **EMS:** it eases the development of TJobs by correlating events received from the SuT, and optionally from the TJobs and other services. In the second half of the project, we have extended it with features to facilitate the development of tests,

inspired by verticals, particularly by the IoT verticals. Most of the functionality could be provided by the core engine of the EMS without deep changes. This engine of the service is a novel real-time stream runtime verification engine, known as Striver and published in the main conference on runtime verification - which is a very powerful illustration of how a good theory can be very practical. The main new developments for the user have been with respect to the DSL language provided, known here as monitoring machines, which essentially is pre-processed and evaluated by the core engine. The main additions were the facilities incorporated for structured input and output events (as JSONs), the ability to encode and reason about time and time passage, and parameterized streams specifications. These have been empirically proven to be very valuable to simplify the development and reuse of tests. One final aspect was the offline mode of the EMS that allows to check test specifications against dumps of traces, as long as the actions upon the SuT are compatible (between the actual test run previously and the new test), which can also be monitored. Finally, one of the most interesting lessons is that the EMS enables new possibilities beyond the goals of the project, for example to increase the expressivity of test orchestrations allowing data communication between different but related tests within a test suite. This is collaborative work between IMDEA, URJC and CNR that will continue beyond the end of the project.

- **EBS:** provides an ElasTest user a Spark-based computation engine for handling big data sets, performing calculations and manipulating data. The results can be stored in Hadoop Filesystem (HDFS) within ElasTest. The service can be used for parsing and analysing logs, statistical analysis of any kind of measurement and for manipulating any result set no matter how big it is. A REST API for interacting with EBS is also provided to simplify access on the service, send commands and receive the results. EBS is designed as a self-contained service and can be deployed independently and add the power of Spark engine in any environment. The deployment on Kubernetes will make EBS service more flexible and scalable either within ElasTest or as an independent service.

- **ESS:** it enables the security testing of cloud-based Web applications. The main benefit of using ESS over other web security testing tools is that it is the first tool to enable automatic detection of Cross-Origin State Inference (COSI) attacks. In addition, it supports the detection of 40 common Web application security weaknesses[51]. As part of the research we did for ESS, we advanced the state-of-the art by presenting COSI attacks as a comprehensive category, introducing a novel approach to identify and build complex COSI attacks, and discovering a novel browser leak based on `window.postMessage` that affects Chrome, Firefox, and Edge. To measure the effectiveness of the COSI attack detection we tested four stand-alone web applications and 58 popular web sites, finding previously unknown COSI attacks against each of them. We responsibly disclosed

[51] https://www.owasp.org/images/b/b0/OWASP_Top_10_2017_RC2_ Final.pdf

the identified attacks to the affected vendors and many of the vulnerabilities have already been patched.

6 Appendix

6.1 References

[D5.1] D5.1 ElasTest Test Support Services v1 (06/30/18). Last accessed 18.11.2019. https://elastest.eu/resources/deliverables/D5.1_ElasTest_Test_Support_Services_v1_FINAL.pdf

[TSS1] J. Gilbert, Cloud Native Development Patterns and Best Practices. Packt Publishers, 2018.

[TSS2] DIVA: Docker image vulnerability analysis, appearing in: Rui Shu, Xiaohui Gu, William Enck: A Study of Security Vulnerabilities on Docker Hub. CODASPY 2017: 269-280, https://doi.org/10.1145/3029806.3029832

[TSS3] Clair: Vulnerability Static Analysis for Containers. Last updated 2019-11-12. https://github.com/quay/clair

[TSS4] Hadolint: Dockerfile linter, validate inline bash, written in Haskell. Last updated 2019-11-07. https://github.com/hadolint/hadolint

[EUS1] García, B., Gallego, M., Gortázar, F. et al. Computing (2019). Understanding and estimating quality of experience in WebRTC applications. Journal of Computing, 101: 1585. https://doi.org/10.1007/s00607-018-0669-7

[EUS2] Boni García, Luis López-Fernández, Francisco Gortázar, & Micael Gallego. (2019). Practical Evaluation of VMAF Perceptual Video Quality for WebRTC Applications. Journal of Electronics. http://doi.org/10.3390/electronics8080854

[ESS1] A. Sudhodanan, S. Khodayari and J. Caballero, “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks,” arXiv, 2019.

[ESS2] Avinash Sudhodanan, Soheil Khodayari and Juan Caballero, “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks,” Network and Distributed System Security Symposium (NDSS), 2020.