D4.3	
Version	1.0
Author	URJC
Dissemination	PU
Date	30-12-2019
Status	FINAL

D4.3 Test Orchestration basic toolbox 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	WP4
WP leader	URJC
Deliverable nature	Other
Lead editor	URJC
Planned delivery date	31-12-2019
Actual delivery date	31-12-2019
Keywords	Open source software, cloud computing, software engineering, operating systems, computer languages, software design & development

Funded by the European Union

License

Contributors

Name	Affiliation
Piyush Harsh	ZHAW
Eduardo Jiménez	URJC
Francisco Gortázar	URJC
Micael Gallego	URJC
Francisco Díaz	URJC

Version history

Version	Date	Author(s)	Description of changes
0.1	30/10/2019	URJC	Structure of document and first contents
0.2	02/12/2019	URJC	Introduction, Section 2, Section 3 and Section 5
0.3	03/12/2019	URJC	Section 4
1.0	30/12/2019	URJC	Final version

Table of contents

List of figures

List of tables

Glossary of acronyms

Acronym	Definition
API	Application Programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CRUD	Create, Read, Update and Delete
CUT	Cloud Unit Testing
CWL	Common Workflow Language
DoA	Description of Action
DSL	Domain-Specific Language
EBS	ElasTest Big data Service
ECE	ElasTest Cost Engine
EDM	ElasTest Data Manager
EOE	ElasTest Orchestration Engine
ERE	ElasTest Recommendation Engine
ESM	ElasTest Service Manager
ESS	ElasTest Security Service
ETM	ElasTest Tests Manager
EUS	ElasTest User Impersonation Service
FMC	Fundamental Modeling Concepts
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
OASIS	Organization for the Advancement of Structured Information Standards
REST	REpresentational State Transfer
SiL	System in the Large
SPA	Single Page Application architecture
SUT	System Under Test
SWOT	Strengths, Weaknesses, Opportunities, Threats
TE	Test Engine
TiL	Test in the Large
TJob	Testing job
TOSCA	Topology and Orchestration Specification for Cloud Applications

TSS	Test Support Service
UML	Unified Modeling Language
WP	Work Package
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

1. Executive summary

ElasTest is an open source platform aimed to ease the testing process of large distributed and heterogeneous software systems. This deliverable is focused on the technical details of the following core components of ElasTest, namely:

- ElasTest Tests Manager (ETM), which is the brain of ElasTest and the main entry point for developers.
- ElasTest Orchestration Engine (EOE), which is responsible of selecting, ordering, and executing a group of test (called TJobs).
- ElasTest Cost Engine (ECE), which is responsible of managing the cost of TJob executions.

Regarding ETM, we have defined a REST API and a web user interface around the concepts of testing jobs (TJobs) and System Under Test (SUT). Concretely, the initial version of the ETM allows end users to define their system under test, define their testing jobs and run them. The ETM takes care of starting the SUT, running the tests defined in the TJob and stopping the SUT afterwards. The ETM keeps a log of all TJobs executing during the history, along with all their related information: logs and metrics. of the project, improved visualization tools focused on troubleshooting those tests in error will be designed and developed.

Regarding EOE, we hypothesize that the concept of orchestration, understood as a novel way to select and execute a group of TJobs within ElasTest, can be a relevant way to improve the testing process. To that aim, two different notions are considered: i) Topology generation, that is, to define a graph of TJobs (edges) and checkpoints (vertices). ii) Test augmentation, that is, to reproduce custom operational conditions of the SUT reusing the orchestration capabilities. During the first review period we implemented the first one. During this second period, we implemented the test augmentation part. In the future we plan to release a reference implementation of the data-driven approach for tests.

Regarding the ECE, we have implemented real cost calculation based on actual cost of execution of a TJob, and retrieval of monitoring information from ElasTest.

2. Introduction

Testing large distributed and heterogeneous software systems on cloud-based platforms is increasingly complex. This kind of software systems aggregates different distributed components, which are typically built and run based on Infrastructure as a Service (IaaS) combined with operation tools and services such as Continuous Integration (CI), container engines, or service orchestrators. The complete assessment of these systems is challenging since developers face with many different problems, including the difficulty to test the system as a whole due to diversity of individual

components, or the coordination of these components due to the distributed nature of the system [1]. Recent surveys confirm the existence of a significant gap between the current and the desired status of test automation for distributed heterogenous system, prioritizing the relevance of test automation features for these systems [2].

To contribute in the solution of this problem, the ElasTest platform provides an integrated toolbox for end-to-end test automation along the development life cycle, including test case management, System Under Test (SUT) deployment, instrumentation, and monitoring for large distributed and heterogeneous software, including web and mobile among others.

The core functionality of ElasTest is provided by the ElasTest Tests Manager (ETM), which is the brain of ElasTest and the main entry point for developers. The core functionality provided by ETM is augmented by means of so called Test Engines (TE). A Test Engine is a component that provides complementary features in the ElasTest platform. ElasTest offers several TEs at the time of this writing, namely:

- ElasTest Recommendation Engine (ERE). This engine provides recommendations about tests to the user. This engine is described in private deliverable D4.2 titled “Test Recommendation Engines v1”.
- ElasTest Question & Answer Engine (EQE). This engine provides a question & answer (Q&A) interface. End-users can, by means of this interface, ask questions about their test suites, and how to improve it. This engine is described in the private deliverable D4.4 “Test Recommendation Engines v2”.
- ElasTest Orchestration Engine (EOE). This engine is responsible of providing capabilities for selecting, ordering, and executing a group of TJobs in ElasTest. Recall, that TJobs are technologically neutral. In other words, ElasTest supports tests coded in any language and using any testing framework.
- ElasTest Cost Engine (ECE). This engine is responsible of managing the cost of TJob executions.

The complete description of the ElasTest architecture at the end of the project is described in deliverable D2.5, titled “ElasTest requirements use-cases and architecture v2”. This deliverable (D4.3) is focused on the technical description of the above-mentioned components. First, we present the features, baseline concepts and design/implementation details of ETM in Section 3. Then we present the EOE and ECE in Section 4 and 5 respectively. To conclude the deliverable, some conclusions and future work are discussed in section 6.

3. ElasTest tests manager

3.1. Introduction

As described in deliverables D2.3 (“ElasTest requirements, use-cases and architecture v1”), and D4.1 ("Test orchestration basic toolbox v1”), the ElasTest Tests Manager (ETM) is the main controller of ElasTest. It is the entry point used by users through its web interface and REST API. The main feature of this component is the coordination of the rest of the platform components to work together to give users the ability to manage the execution of end to end tests to verify complex distributed applications. This component has been extensively modified during the second period of the project in order to enable several ElasTest deployment (installation) modes: mini, ElasTest on Kubernetes (EK), and High-availability ElasTest on Kubernetes (HEK).

ElasTest as its own name indicates must be elastic, in order to support many integration and end-to-end tests running in parallel, each one with its own TSSs. In order to achieve elasticity, the ETM can now deploy the components needed by a TJob on K8s[1] (Kubernetes). When ElasTest is deployed on a K8s cluster, the ETM will deploy TJobs, TSSs, TE and integrated external tools on K8s too. In this way, if ElasTest is low on resources, new VMs are started within the K8s cluster providing the necessary additional resources to deploy the requested ElasTest components. However, sometimes a smaller deployment is required (for instance, for evaluation purposes), and ElasTest can also be deployed in an environment with fewer resources. This is why the *Mini* mode has been created. In this mode the functionalities of several modules (Logstash, RabbitMq, ESM, and EUS) of ElasTest have been included in the ETM component, in order to reduce the resources required for ElasTest to work.

This section is devoted to describing the new functionalities added to the ETM, and the changes made to its components as a result of these functionalities, since milestone M18 (June 2018). The rest of this section is structured as follows. Section 3.2 presents the main features of ETM. Section 3.3 presents a detailed description of the technologies used in the implementation of the component. Section 3.4 describes the internal architecture of the component and how it is implemented. Finally, section 3.5 details the main aspects related to source code that implements the ETM.

3.2. Features

The list of new features implemented in the ElasTest Tests Manager (ETM) component since M19 is summarized in the following table.

[1] https://kubernetes.io/

Feature	Description
Compare TJob executions	As an ElasTest user, I want to compare the results of several executions of a TJob toeasily identify the differences.
Cross-browsing	As an ElasTest user, I want to be able to execute a TestLink Test Plan in a browser and simultaneously reproduce all my actions in a different browser in order to save some effort during manual testing when several different browsers are to be tested. This functionality is provided in cooperation with the EUS (the browsers-as-a-service component).
Upload and download files from a browser	As an ElasTest user, I want to be able to upload or download a file from or to the browser that I'm using during a manual test. This functionality is provided in cooperation with the EUS.
Select Browser in manual version	As an ElasTest user, I want to choose the browser version with which to perform the manual test. This functionality is provided in cooperation with the EUS
Multi Axis TJob	As an ElasTest user, I want to execute a TJob with several configurations with a single click one on the execution button.
Add attachments to a TJob execution	As an ElasTest user, I want to be able to attach files to a TJob execution using the API rest of ElasTest.
Deploy a TJob components on K8s	As a Software Architect, I want ElasTest to be able to deploy its test components in K8s in order to get the elasticity my testing process needs.
Get Metrics from an external source when ElasTest is deployed on K8s	As an ElasTest user, I want to be able to obtain the metrics of an external SuT from an external source. In this case Prometheus when ElasTest is deployed on K8s.

Table 1. ETM new features from M19

3.3. Baseline concepts and technologies

The ETM is composed internally by several sub-components. The main sub-component is the ETM Core, which provides a REST API and a Web Socket interface. This backend service is used by the ETM Graphical User Interface (GUI) implemented as a Single Page Application architecture[2] (SPA). The ETM Core is responsible to coordinate the rest of the ElasTest components and other internal sub-components.

[2] https://en.wikipedia.org/wiki/Single-page_application

The ETM Core is implemented in the Java language using the Spring Boot framework[3]. The ETM GUI is implemented in the TypeScript language using the Angular framework[4]. Other sub-components used in the ETM are:

- Logstash[5]: Used to retrieve and process logs and metrics during TJob executions. This information is then stored in an ElasticSearch[6] provided by the ElasTest Data Manager component (EDM). Logstash and ElasticSearch are part of elastic stack[7], the leading open source stack used to gather, process, register and analyze logs, metrics and any kind of KPI of Internet applications.
- RabbitMQ[8]: Used to send real time information from ETM-core to the frontend my means of WebSockets. RabbitMQ is a leading message queuing software well integrated with the Spring technologies used in the ETM core.

In the rest of the section, we describe the interactions between the ETM Core and the rest of the subcomponents of the ETM.

3.4. Component architecture

In the following subsections, we outline a general overview of the internal structure of the ETM using several class and component diagrams. The interaction of the different modules is described with UML sequence diagrams. Finally, a detailed data model is presented.

3.4.1. Component diagram

The ETM is composed of the following sub-components:

- **ETM Core:** Service backend that coordinates all other internal sub-components and interacts with the rest of ElasTest components.
- **ETM GUI:** The graphical user interface with which the user interacts. This component is also called 'Angular GUI' to emphasize the technology used to implement it.
- **RabbitMQ:** A messaging broker used to send logs and metrics in real time to the user interface.
- **Logstash:** A server-side data processing pipeline that ingests data, transforms it, and then sends it to RabbitMQ and ElasticSearch. ElasticSearch is a sub-component of the ElasTest Data Manager (EDM) component used to register all information gathered during test execution.

[3] https://spring.io/projects/spring-boot

[4] https://angular.io/

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

[6] https://www.elastic.co/products/elasticsearch

[7] https://www.elastic.co/

[8] https://www.rabbitmq.com/

- **Dockbeat** and **Filebeat:** These services are used for log retrieval and monitoring of TJobs executed as docker containers. These services are well integrated with the elastic stack.
- **TestLink:** The ETM includes an instance of this project to manage manual tests.
- **Jenkins:** The ETM includes an instance of this project to manage tests.

These sub-components are illustrated in Figure 1.

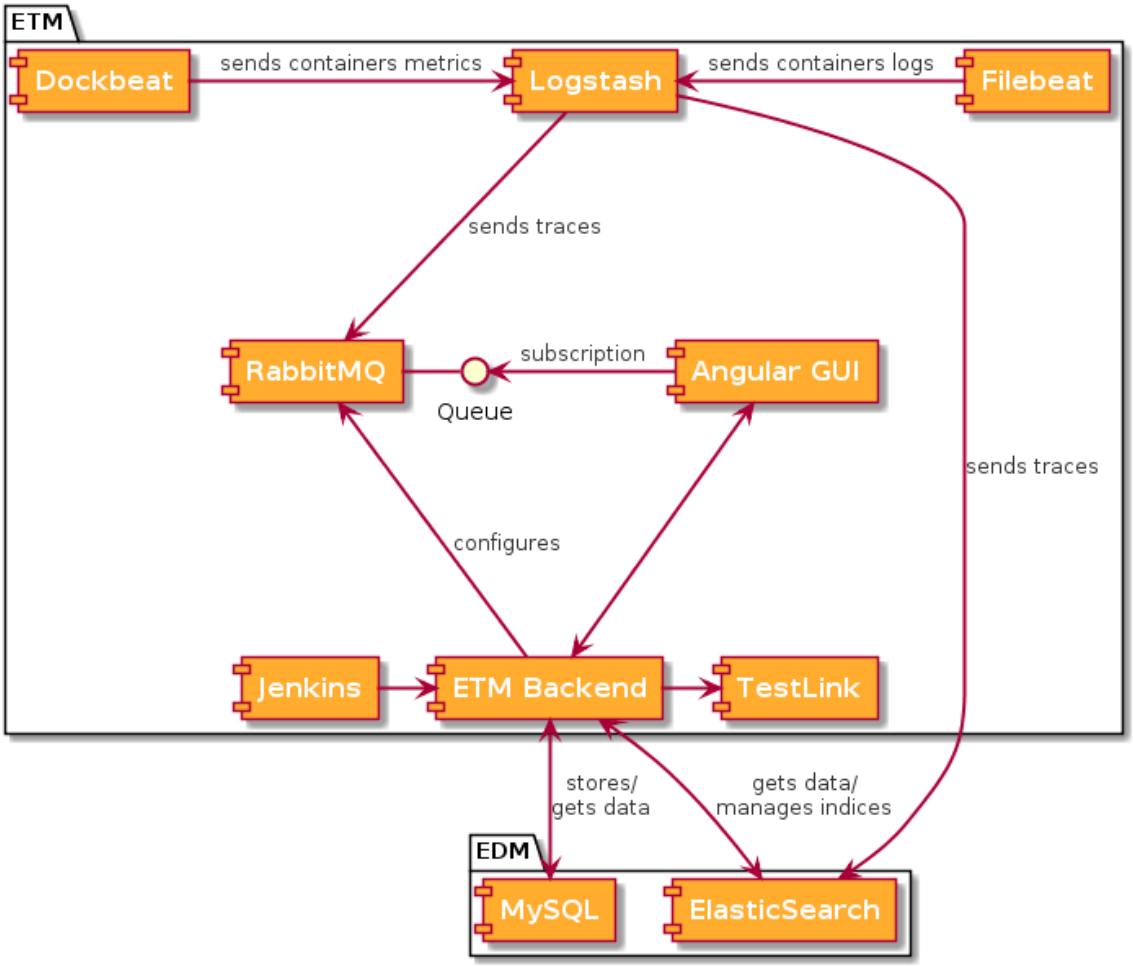

Figure 1. ETM sub-components

ElasTest offers a lightweight mode called *ElasTest Mini*, which integrates the Logstash and RabbitMQ subservices into the backend code, among other components like ESM and EUS (see Figure 2). In order to reduce the use of resources, in ElasTest Mini MySQL is used to store the monitoring data, instead of ElasticSearch.

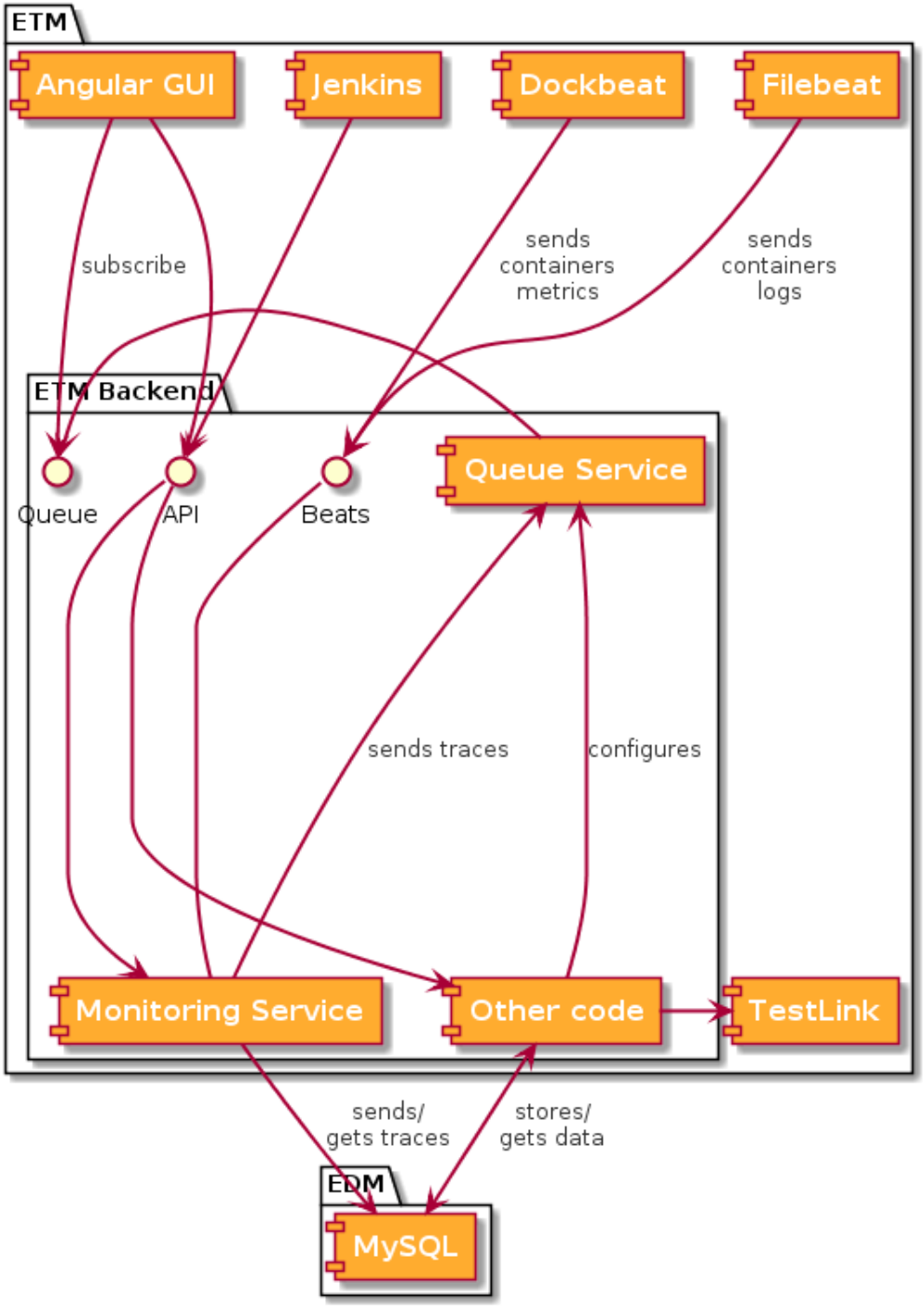

Figure 2. ETM Mini sub-components

3.4.1.1. Modules used for TJob execution

Several ETM modules interact to execute a TJob in the ETM. These modules and their relations are shown in Figure 3.

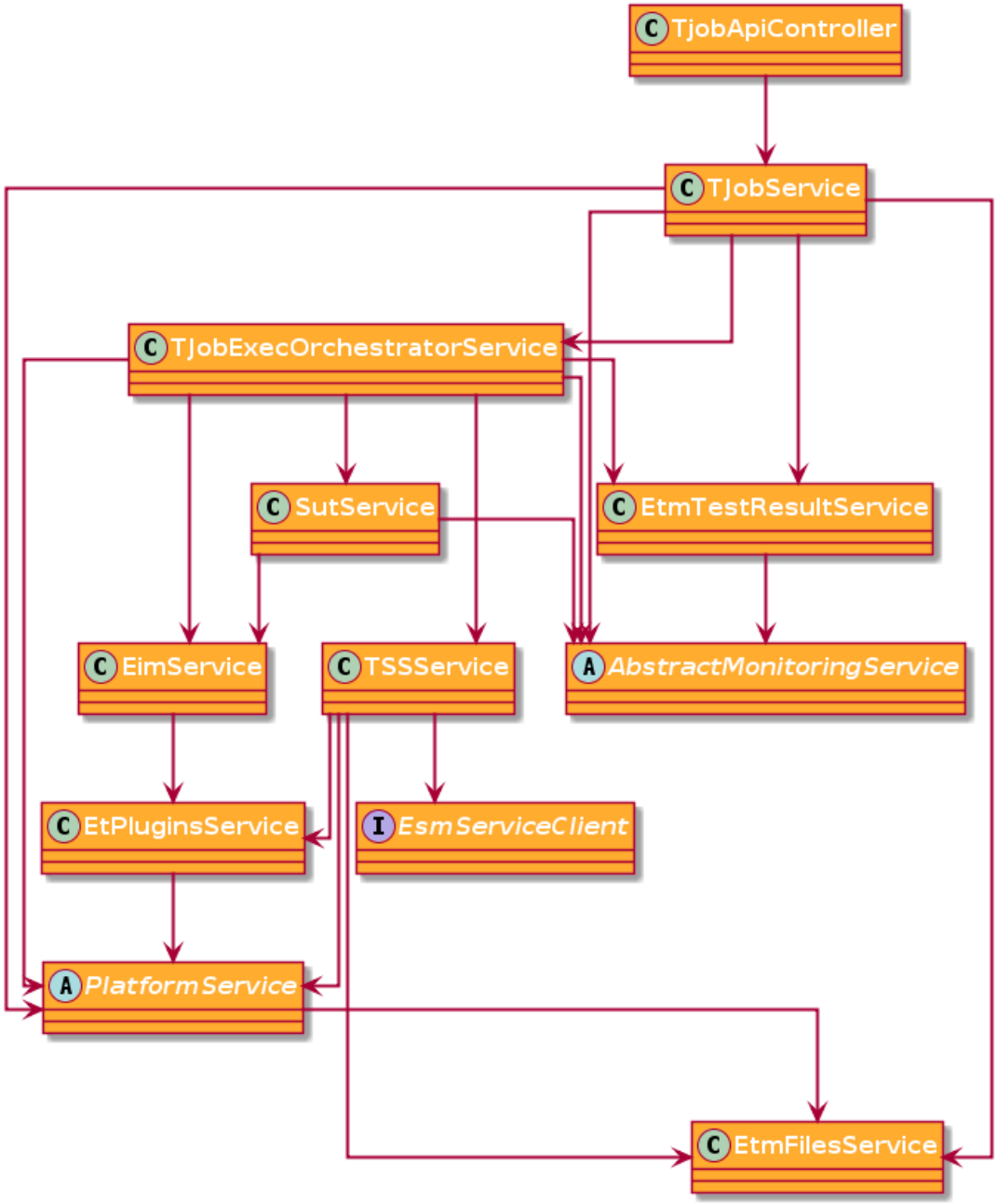

Figure 3. ETM Core modules used to execute TJobs

When a TJob is executed using the graphical interface, the ETM GUI interacts with the **TJobApiController** to manage TJobs. This controller makes use of **TJobService** to process the requests received. Later on, **TJobExecOrchestratorService** takes control of the execution using the following services:

- **TSSService:** interacts with ElasTest Service Manager (ESM) component to manage the Test Support Services (TSSs) associated to the TJob. It is performed

using the **EsmServiceClient** interface thought the ESM API. This interface has two implementations:

  - **EsmServiceClientImpl:** In charge of interacting with the ESM.
  - **MiniEsmServiceClient:** This implementation is used only in the Mini mode of ElasTest, and acts as the ESM, since in this mode the full fledged ESM is not included.
- **SutService:** used in case the TJob has an associated SuT.
- **PlatformService:** this class abstracts the platform over which ElasTest is running (Docker or K8s). The **PlatformService** implements the behavior shared between different platforms and defines the rest that will be implemented by the following classes that inherit from this abstract class:
  - **DockerServiceImpl:** makes use of Docker to manage services.
  - **K8ServiceImpl:** makes use of Kubernetes to manage services.
- **EimService:** this service is only used if the TJob has an associated SuT "deployed outside" ElasTest and "Instrumented by ElasTest". In this case, the sevice, will be in charge of interacting with the EIM, which will connect to the SuT via SSH and install Beats agents to obtain monitoring information.
- **AbstractMonitoringService:** this abstract service is in charge of monitoring data management. Has two childs:
  - **ElasticsearchService:** communicates with ElasticSearch to manage monitoring data.
  - **TracesSearchService:** used in ElasTest Mini. Communicates with MySQL to manage monitoring data.

Main modules of ETM GUI are shown in Figure 4.

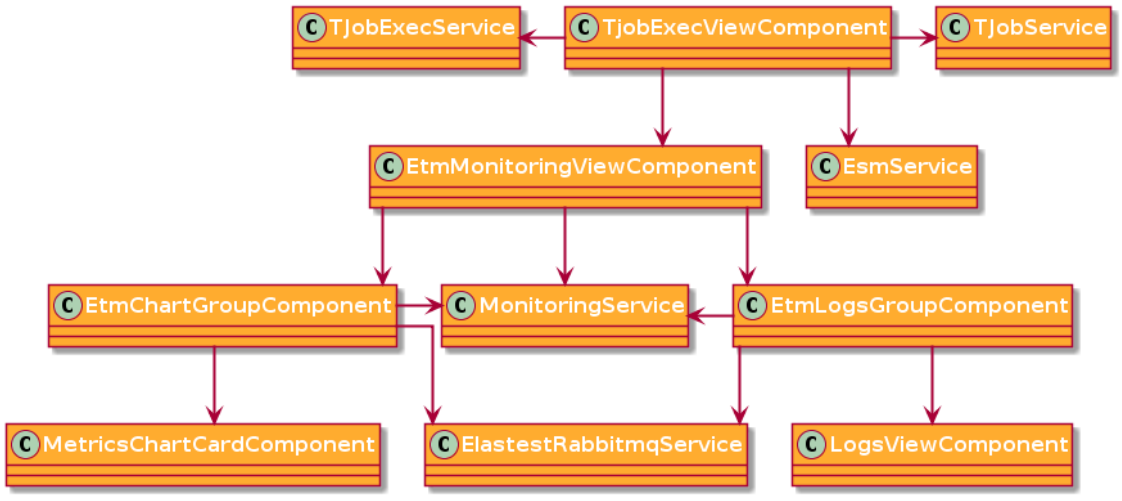

Figure 4. ETM GUI components

These modules have the following responsibilities:

- **TjobExecViewComponent**: this is the main component, which contains all the logic of the Executing TJob page.
- **EsmService**: this service is in charge to manage information related to TSSs coming indirectly from ESM.
- **TJobService**: This service manage all the information related to TJobs. It is mainly used to populate the GUI with information related to them.
- **TJobExecService**: It updates the interface in real time during the execution of TJobs. This is done implementing a polling strategy.
- **ElastestRabbitmqService:** creates the necessary connections with RabbitMQ to obtain logs and metrics in real time during the TJob execution.
- **EtmMonitoringViewComponent:** is responsible for managing everything related to metrics and logs, making use of the information obtained from RabbitMQ and, occasionally, from Elasticsearch.

3.4.1.2. Modules used in LogAnalyzer

The LogAnalyzer is the part of the ETM that allows the user to analyze logs retrieved during tests executions. The user can mark and filter log entries with certain patterns or contents. Figure 5 shows the main GUI modules of the LogAnalyzer, which consists on the following modules:

- **LogAnalyzerComponent:** is the high-level component for LogAnalyzer. It uses **LogAnalyzerService**, that contains all the logic of the tool.
- **GetIndexModal:** This module is responsible for obtaining the available executions (through **ProjectService, TJobExecService and ExternalService)** so that the user can select the ones he wants and then process and pass them to LogAnalyzerComponent to perform the search for logs.
- **MonitoringService:** this service is used to make queries to get the logs through the backend.

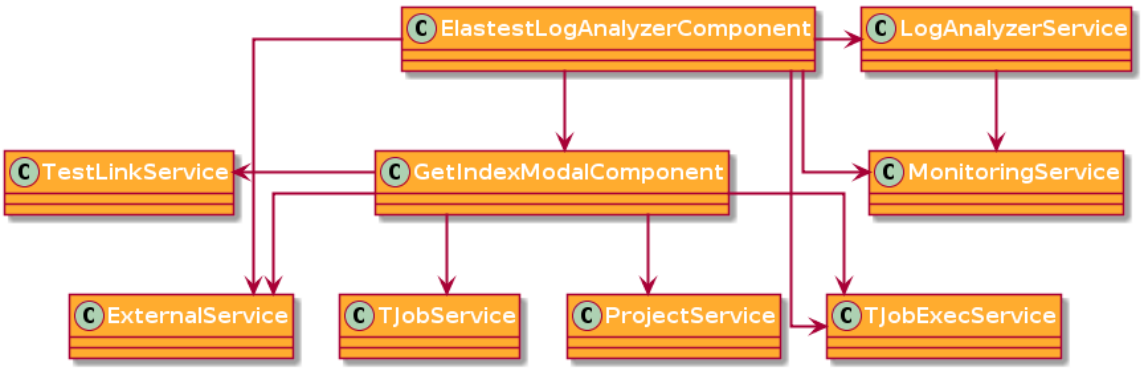

Figure 5. LogAnalyzer modules

3.4.1.3. Modules used in TestLink integration

Several improvements and functionalities have been introduced to facilitate the execution of manual tests with TestLink:

- **Upload/Download** files to/from browser. In that way, if a manual tester needs to upload a file to a web application he is testing, ElasTest provided browsers can access to this file. In the opposite way, if a file is generated in the web page being tested, this file can be downloaded by ElasTest user.
- **Crossbrowser:** it's possible to execute manual testing with multiple browsers. The actions carried out in any of the browsers will also be carried out in the rest of the browsers. This module implements all the logic needed to simulate interaction with several browsers at the same time.
- **Pause execution** of TestLink test cases and **resume** it later.

3.4.2. Metrics and logs

An important feature added to the log and metrics section is the LogComparator, which allows to compare the logs from two executions, highlighting with colors the lines where there have been changes. With this tool the user can easily see what has changed and find errors revealed in the two executions. The Log Comparator can be used in two different sections:

- In the *Execution Comparator* section
- When the last execution of a TJob was satisfactory but a *new execution fails*. It will be shown automatically in the display of the failed execution and will show the changes between the current (failing) execution and the last successful one.

There are several filters that that can be applied when using the Log Comparator, divided into two categories:

- View:
  - Complete logs: compare all available logs. (Default)
  - Test Logs: compare only the logs of the test cases.
  - Failed Tests: compare only the logs of the failed test cases.
- Comparison:
  - Complete: displays the complete message as it is saved.
  - No timestamp: if the message contains timestamp, it is removed. (Default)
  - Time diff: includes the time difference between traces in the logs comparison.

Figure 6 shows the Log Comparator sequence diagram highlighting how this component, upon activation by the user, gathers the necessary data to perform the comparison:

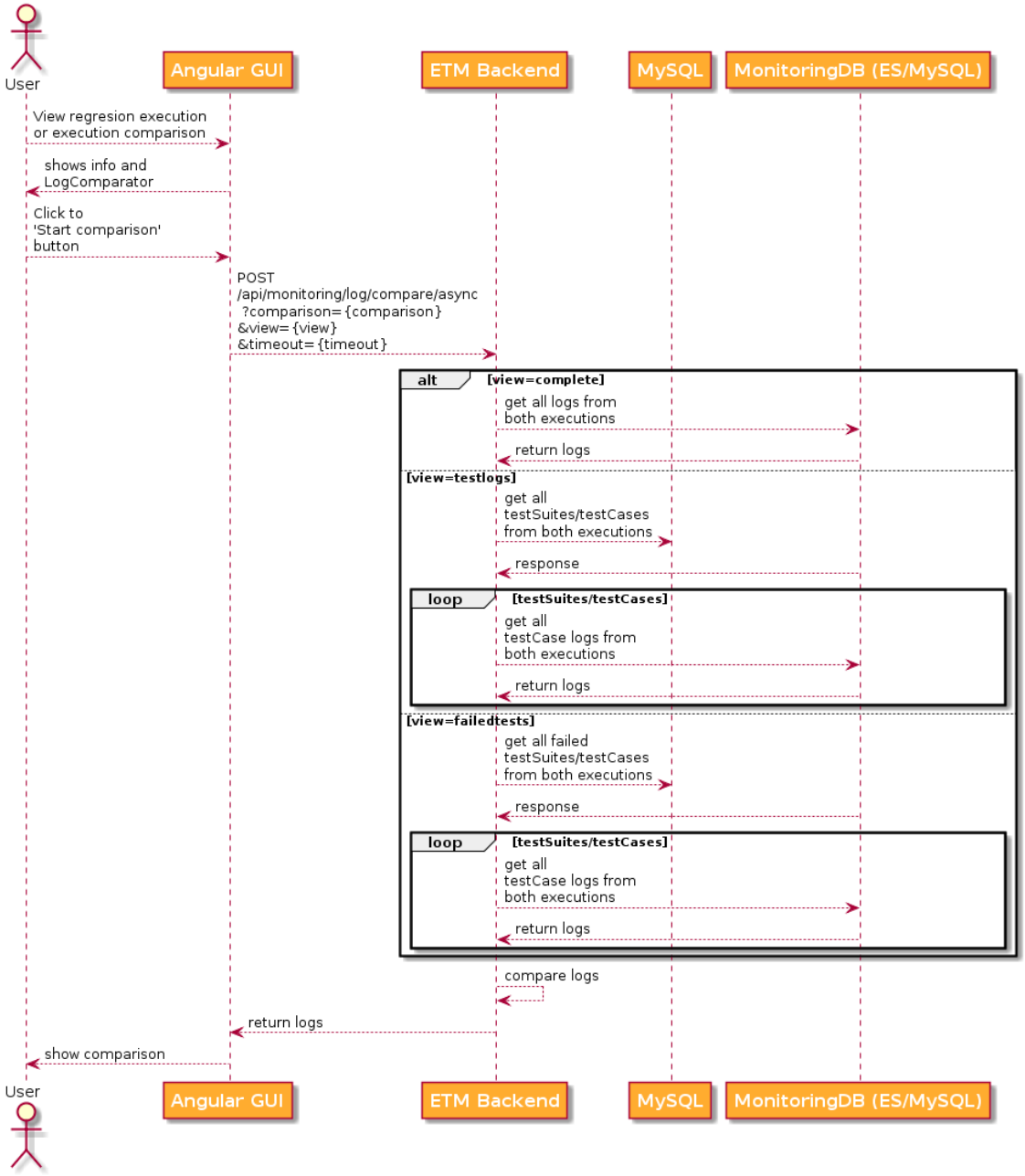

Figure 6. LogComparator sequence

3.4.3. TJobs Execution elasticity

When ElasTest is running over K8s, the ETM will deploy the resources to execute a TJob (TJob, Sut, TSSs), TEs (Test Engines) and other integrated services (like Jenkins on K8s), making use of the Java client provided by fabric8[9]. To see this with a simple example, Figure 7 describes how a TJob is executed from the ETM.

[9] https://github.com/fabric8io/kubernetes-client

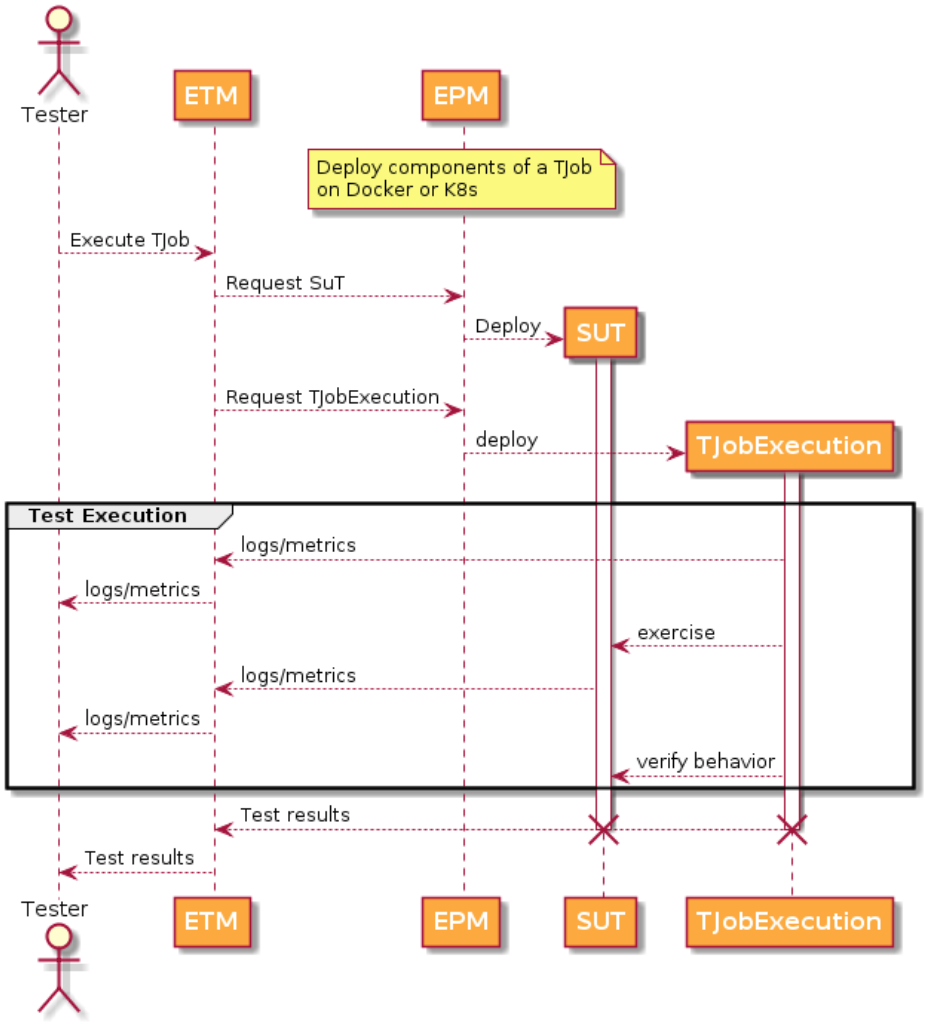

Figure 7. Sequence diagram of a TJob execution

In the diagram, the EPM is the component responsible of deploying the components associated with a TJob (in this case, the SUT and TJobExecution) on Docker or K8s, depending on where ElasTest was deployed.

Each resource that the ETM deploys on K8s is associated with a K8s component as described in the following table.

ETM Resource	K8s Resource	K8s Resource description
TJob	Job	Controller that deploy one or more pods and ensures that some of them finalice with success.
Sut	Pod	Minimal deployment unit on K8s

TSS	Deployment	Allow easy updating and define the number of replicas of a POD (when ElasTest is running in Mini mode)
TE	Deployment	Allow easy updating and define the number of replicas of a POD
Jenkins	Deployment	Allow easy updating and define the number of replicas of a POD
Dockbeat	Daemonset	Deploys an instance of a pod in each kubernetes node.

Table 2. Kubernetes components used by ETM

3.5. Code links

The ETM is composed of several sub-components: the ETM Core, the ETM GUI, Logstash, RabbitMQ, Filebeat, Dockbeat, Jenkins and TestLink. From them, the ETM Core and the ETM GUI have been developed entirely in the context of the ElasTest project. The other components are available with open source licenses. All these components are executed in different docker containers except for the ETM GUI that is executed entirely in the web browser.

The development of the ETM has used the following GitHub repository:

https://github.com/elastest/elastest-torm

3.5.1. Validation

The ETM have been extensively validated in several ways:

- The experiments conducted in the context of project's WP7 evidence that ETM accomplish its main objective to enable TJobs execution coordinating the rest of ElasTest components accordingly.
- An extensive number of unit, integration and end to end tests have been implemented and are executed automatically in the continuous integration system. These tests evidence the features implemented in the ETM are behaving as expected and regressions are detected quickly.
- The ElasTest platform (coordinated by the ETM) is being used to implement and execute end to end tests of Kurento[10], an open source WebRTC platform used to implement videoconference web applications.

[10] http://www.kurento.org/

- The ElasTest platform is being used to implement and execute end to end tests of OpenVidu[11], a family of open source libraries for different frameworks (web and mobile) to include videconference with ease into any application.

3.5.2. Discussion

The ETM is now mature enough to be used in real world projects. It provides a broad set of features to test complex distributed applications like browsers as a service, observability during testing, log comparison, etc. Also, it provides features to perform manual testing more productive (thanks to simultaneous cross browser testing) and bug fixing more easily (thanks to observability during testing). ElasTest has shown growing adoption in the industry as shown in the download statistics of DockerHub statistics (10K+ at the time of writing).

ElasTest has been evaluated for software development teams in companies like Panel Sistemas, Zooplus, Idealista and Ericsson. As a result of these evaluations, bugs were reported, and new features requested. ElasTest as a whole and ETM in particular have been enhanced with feedback from community.

3.6. Research results and plans

As stated in the previous version of this deliverable (D4.1), a main research direction for the ETM has been the automatic analysis of the information gathered during the execution of a test. For example, in case of regression, it is very useful to compare logs and metrics obtained from failed tests with the information of equivalent succeeding tests. Another research line has been the comparison of the information gathered executing the same tests against different configurations of the same SUT, detecting the best configuration attending to different aspects like CPU consumption, bandwidth usage, latency, requests per second, etc. These features have been used by the verticals in the context of WP7 to validate the different objectives of the project.

4. ElasTest orchestration engine

4.1. Introduction

The concept of test orchestration is one of the three main principles of the project as specified in the ElasTest Description of Action (DoA) document [3]:

> *ElasTest is a cloud platform designed for helping developers to test and validate SiL (see definitions above), while maintaining compatibility with current CI practices and tools. For this, ElasTest bases on three principles: (1) instrumentation (i.e. customization of the SUT infrastructure so that it reproduces real-world operational behavior);* ***(2) test orchestration (i.e. to combine***

[11] https://openvidu.io/

***intelligently testing units for creating a more complete test suite following the "divide and conquer" principle)**; and (3) test recommendation (i.e. to use machine learning and cognitive computing for recommending testing actions and providing testers with friendly interactive facilities for decision taking). Hence, ElasTest main objectives relate to improving the testing of SiL.*

This orchestration mechanism is one of the main novelties of the ElasTest project and its precise conception, formalization and consolidation is one of our main research objectives. For a state-of-the-art on test orchestration see D4.1 ("Test orchestration basic toolbox v1").

Two main mechanisms are proposed in the ElasTest DoA to implement test orchestration:

1. Topology generation. This concept allows the actual implementation of test orchestration, including a notation to define test orchestrations. The idea is that testers define the different TJobs (edges) and checkpoints (vertices).
2. Test augmentation. This concept consists on introducing new TJobs an original test to reproduce custom operational conditions of the SUT. This way, in addition to test functional features of the SUT, other non-functional attributes (such as performance, scalability or reliability) can be assessed.

During the first period of the project, we devoted efforts to the first mechanism above. We developed a Jenkins extension that allows end-users to define their own testing topology, and ElasTest takes care of executing such topology applying the necessary checkpoints, and providing at the very end of the process a report with the results.

During this second period, we focused on the second mechanism, namely test augmentation. The test augmentation approach requires the combination efforts of the programmable test orchestrator (Task 4.2) and the instrumentation manager (Task 3.4) and agents (Task 3.3). By leveraging the instrumentation capabilities of ElasTest, the test orchestrator can, giving a TJob, run additional TJobs reproducing custom operational conditions of the SUT.

The rest of this section is structured as follows. Section 4.2 presents the set of features initially planned for Task 4.2. Section 4.3 presents a detailed description of the test augmentation approach in ElasTest. Section 4.4 describes the publications so far. Finally, Section 5.5 concludes.

4.2. Features

The list of requirements for the ElasTest Orchestration Engine (EOE) component is summarized in the following table.

Requirement	Description
Topology generation	Define a test orchestration notation for users to define TiL (Test in the Large) by aggregating different TJobs
Jenkins DSL notation	Leverage Jenkins shared library technology to allow the definition of orchestration topologies so that users can define a TiL by aggregating different TJobs
EOE DSL parser	To enable the EOE to parse Jenkins notation
EOE communication manager	To enable the EOE to support data-driven orchestration
EOE proxy	To intercept requests from the ETM to TSSs to share sessions among different tests
Reference implementation	Create some reference implementation of the data-driven approach, for example using the JUnit 5 extension model
Test augmentation	New TJobs can be added to the orchestration in order to reproduce custom operational conditions of the SUT or non-functional attributes (such as performance, scalability or reliability)
Include extra checkpoints	Integrate techniques (new or existing) to include automated assertions in existing orchestrations to improve test coverage of orchestrated TJobs by adding extra checkpoints (especially in data-driven approach)
Test prioritization	Prioritize TJob executions using on a multi criteria approach based on dissimilarity and resource availability

Table 3. Orchestrator requirements

4.3. Component architecture

In ElasTest, test orchestration is understood as the interconnection of different TJobs expressed as a graph. The precise form of the graph which describes the order in which tests get executed is specified by the tester. During the second period, two features were implemented: the test augmentation approach, and the test prioritization support.

For the **test augmentation mechanism**, we introduced in the test orchestrator the concept of custom operational conditions. G iven a TJob, and a set of operational conditions, the EOE will execute as many instances of the TJob as needed in order to exercise the SUT with all the possible combinations of operational conditions. At the end, a checkpoint is created in order to decide if the augmented TJob passes the test or fails.

The current implementation of the EOE support the operational conditions currently defined in the EIM (ElasTest Instrumentation Manager):

- Packet loss: a ratio of network packets to be discarded and not delivered.
- CPU burst: a ratio of CPU load to be forced.
- Teardown nodes: a ration of nodes (i.e., application instances) to teardown.

The test augmentation feature allows end-users to specify different values for each of these operational conditions, and the EOE will force all the possible combinations of packet loss, CPU burst and teardown nodes. This approach simplifies enormously the definition and execution of this kind of tests, enabling chaos testing within organizations.

Snippet 1 below shows a possible test augmentation definition. The exit condition is set to exit the orchestration as soon as any of the TJob executions fail. We specify 2 different packet loss values and 3 different CPU burst values, which result in a total of 6 possible combinations: (0.1, 0.2), (0.1, 0.5), (0.1, 0.8), (0.5, 0.2), (0.5, 0.5), (0.5, 0.8). Then we specify that we want to test all those combinations using TJob "myjob1", so the EOE will run this TJob 6 times, each time using a different operational condition.

```
@Library('OrchestrationLib') _

// Configuration
orchestrator.setContext(this)
orchestrator.setExitCondition(OrchestrationExitCondition.EXIT_ON_FAIL)
orchestrator.setPacketLoss(0.1, 0.5)
orchestrator.setCpuBurst(0.2, 0.5, 0.8)

// Graph
def result = orchestrator.runJob('myjob1')
```

Snippet 1.Test augmentation definition

We can as well be more specific about when the test execution fails or passes by providing thresholds on different metrics that the test orchestrator can capture using the ElasTest Monitoring Service (EMS). Consequently, three different ElasTest components collaborate to ease the testing process. For instance, Snippet 2 shows a definition that uses the same operational conditions but specifies thresholds for different properties. If these thresholds do not hold, the test fails, otherwise it passes. Specifically, the EOE will check, with the aim of the EMS, that the CPU load will be below 50% for all the 6 TJob executions, and that the running time of each execution will be 5 seconds at most.

```
@Library('OrchestrationLib') _

// Config
orchestrator.setContext(this)
orchestrator.setExitCondition(OrchestrationExitCondition.EXIT_ON_FAIL)
orchestrator.setPacketLoss(0.1, 0.5)
```

```
orchestrator.setCpuBurst(0.2, 0.5, 0.8)
orchestrator.checkTime(LessThan(5, Time.SECONDS))
orchestrator.checkCpuLoad(LessThan(50, Unit.PERCENTAGE)

// Graph
def result = orchestrator.runJob('myjob1')
```

Snippet 2. Automatic test oracle generation to assess outcome of augmented test

Regarding the test prioritization support, in the second part of the project we have been researching the use of context-aware prioritization to decide in which order to execute the TJobs in the orchestration graph. This is useful when the time available for testing is limited and we want to run first those tests that are more likely to reveal a bug. Test prioritization is not usually driven by the context, i.e., the available resource. In our research work, we have studied the hybridization between a previous work on prioritization based on dissimilarity of so-far prioritized test cases [6], and the resource awareness that ElasTest can provide. We have developed a multi-criteria algorithm that prioritizes both dissimilarity and resources available (memory). Our preliminary results are very promising, and the algorithm outperforms the state-of-the-art alternatives.

At this moment we're preparing a paper to be submitted to ICST or AST (both deadlines in January) with the results of our research. Once the paper is submitted, we will integrate the algorithm into the EOE, thus allowing the orchestrator to use this multi-criteria prioritization algorithm to decide which TJob to execute next out of the graph of all possible TJobs.

4.4. Research results and plans

At the time of this writing, a publication about the EOE has been accepted in the following international conference:

- *A Proposal to Orchestrate Test Cases*. Boni García, Francesca Lonetti, Micael Gallego, Breno Miranda, Eduardo Jiménez, Guglielmo De Angelis, Carlos Santos, and Eda Marchetti. 11th International Conference on the Quality of Information and Communications Technology. Coimbra, Portugal, September 4-7, 2018.

The CodeURJC research group expects to submit the augmentation approach in a testing conference (probably AST or ICST workshops) by January. In parallel, the URJC team is working jointly with CNR on submitting a paper on test prioritization in the context of the test orchestration feature. Also, a joint research is been carried with IMDEA project partner to submit a paper focused on data-driven orchestration using EMS.

Finally, ElasTest is part of a PhD currently in development at the URJC research group. The thesis focuses on running tests in the past to find the origin of bugs. In this context the research group got this year a grant from the National Research Program that guarantees the sustainability of the PhD student, and therefore ElasTest as a research tool.

5. ElasTest cost engine

5.1. Introduction

The ElasTest cost engine (ECE) is the principal module that deals with estimating the cost of executing a test based on specified resources, and also with collecting resource consumption data post execution for calculating the true cost of execution. This section captures the architectural changes, enhanced requirements listing, updates to core features in ECE since D4.1.

5.2. Features

The list of requirements for the ElasTest Cost Engine (ECE) component is summarized in the following table.

#	Requirement	Description
1	Receive TJob information from ETM	The ECE should be able to get the list of TJobs from the ETM
2	Receive TJob information from ESM	The ECE should be able to get the service type cost definitions from ESM
3	Static Estimation of a TJob cost	The ECE should be able to estimate the cost of execution of a TJob statically using the cost model definitions received from the ESM
4	Retrieve monitoring information	The ECE should be able to query and get the actual monitored data capturing the events and resource consumption for a TJob execution
5	Actual calculation of the cost of execution of a TJob	The ECE should be able to calculate the real cost of an execution of a TJob based on the cost models and the monitored data values.
6	Extend cost model to support all ElasTest support service	The ECE task-force should define the cost models for relevant ElasTest services using meaningful metrics.
7	Make rest cost feature optional depending on the availability of EMP service	To know the real cost feature is not supported in a release outright and not see a broken feature error

Table 4: Cost Engine Requirements

Requirements 4, 5, and 7 were implemented since D4.1 and are reported as part of this deliverable.

5.3. Baseline concepts

The initial design of ECE was designed with the open source billing framework Cyclops in perspective. Since the implementation of the first prototype of this service (as reported in D4.1), it was realized that full capabilities as offered by Cyclops framework was not needed for ElasTest. The fact that the ElasTest service manager (ESM) (see D3.1 and D3.2) has the responsibility of maintaining the cost models as part of the OSBA specification, and availability of enhanced capability in the ElasTest monitoring platform (EMP) (see D3.1, and D3.2) for reporting of resource consumption data for various TJob related container processes, as well as support services, doing a simpler, alternative cost calculation workflow was more sensible.

5.4. Component architecture

No significant architectural updates were needed since D4.1. The cost models defined in D4.1 have been used in the implementation of the true cost of TJob post its execution. The cost processing flow chart and modulus interaction diagrams are already presented in D4.1 and thus has not been repeated here.

5.5. Implementation and code links

The UI underwent a refresh since D4.1. The ECE depends on the ETM, the ESM and the EMP for cost estimation and true cost calculations.

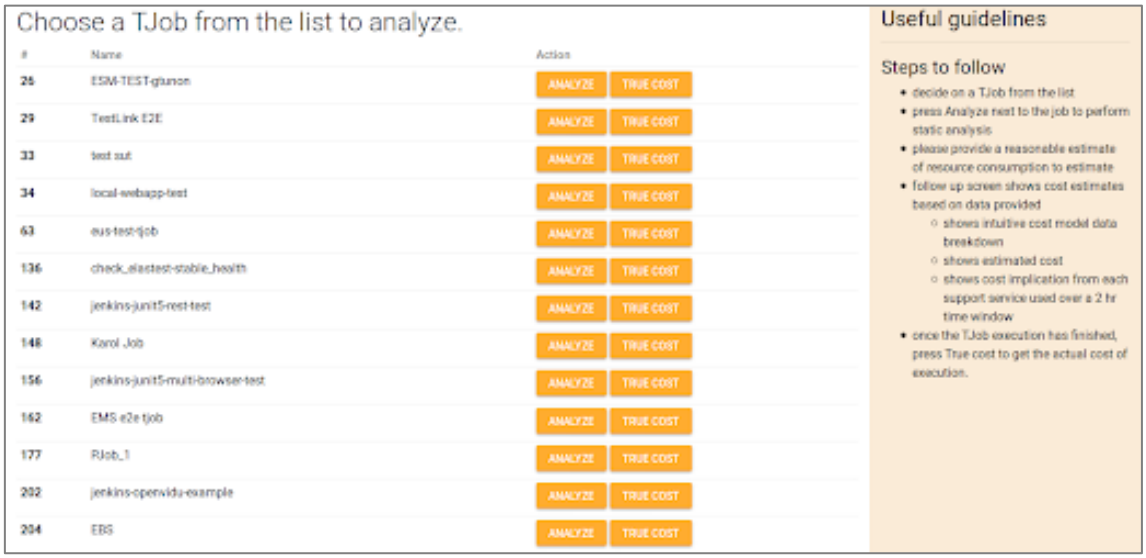

Figure 8. TJob list

Using the ETM's REST APIs, the ECE fetches the list of registered TJobs, and provides the users with an option to analyze the cost of execution. Furthermore, if tests have already been executed, the ECE allows the tester to see the true cost of historical test runs. Figure 8 above shows the list of all TJobs as retrieved from ETM.

Figure 9 shows the form a user has to fill in before a static cost analysis can be performed. The elements of the form depends on how the TJob has been defined at the time of registration.

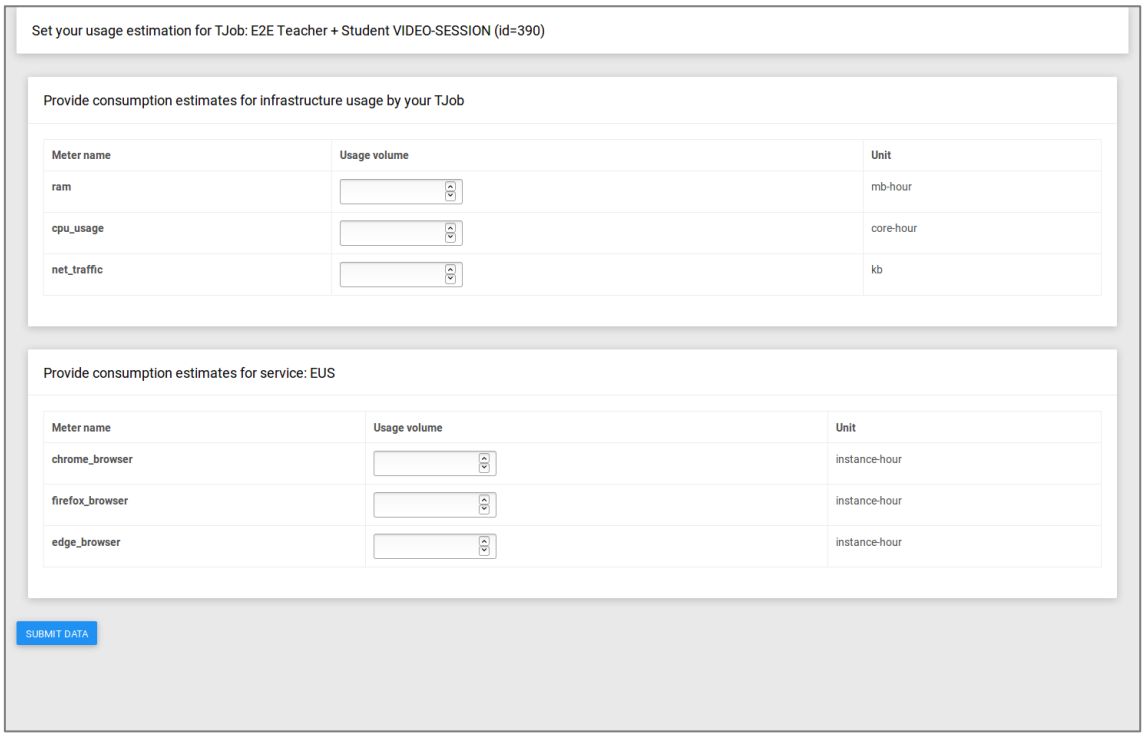

Figure 9. Static cost estimation form query

Figure 10 depicts the post analysis screen showing the cost model associated with any support service used within the TJob.

If alternatively, the user selects the 'true cost' option, the ECE will contact the EMP to fetch the resources consumed metrics from all historical run of this particular Job, and

using a cost model from the ESM, computes the actual cost of running those past execution within ElasTest. This is shown in Figure 11 below.

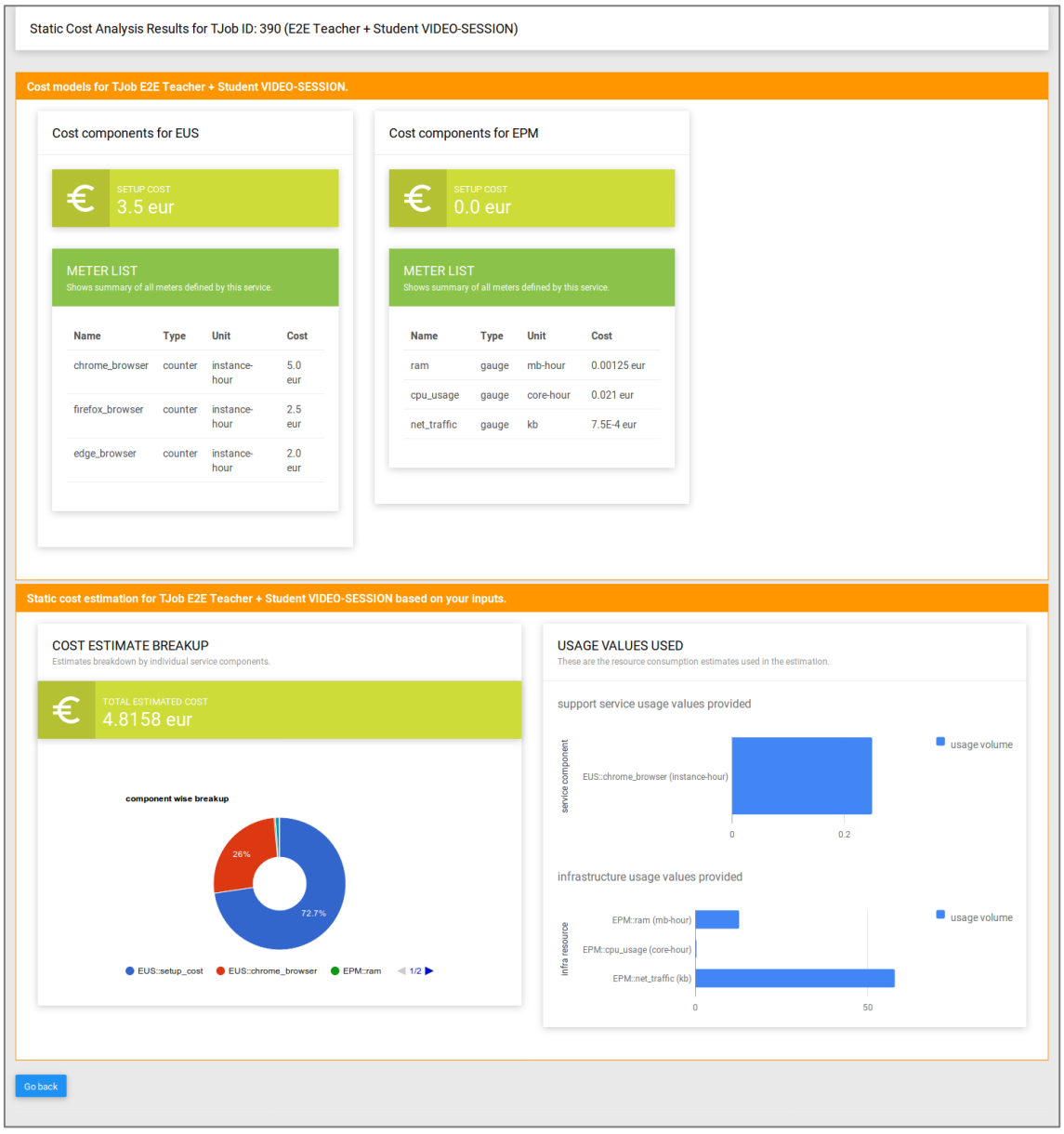

Figure 10. Static cost estimated output

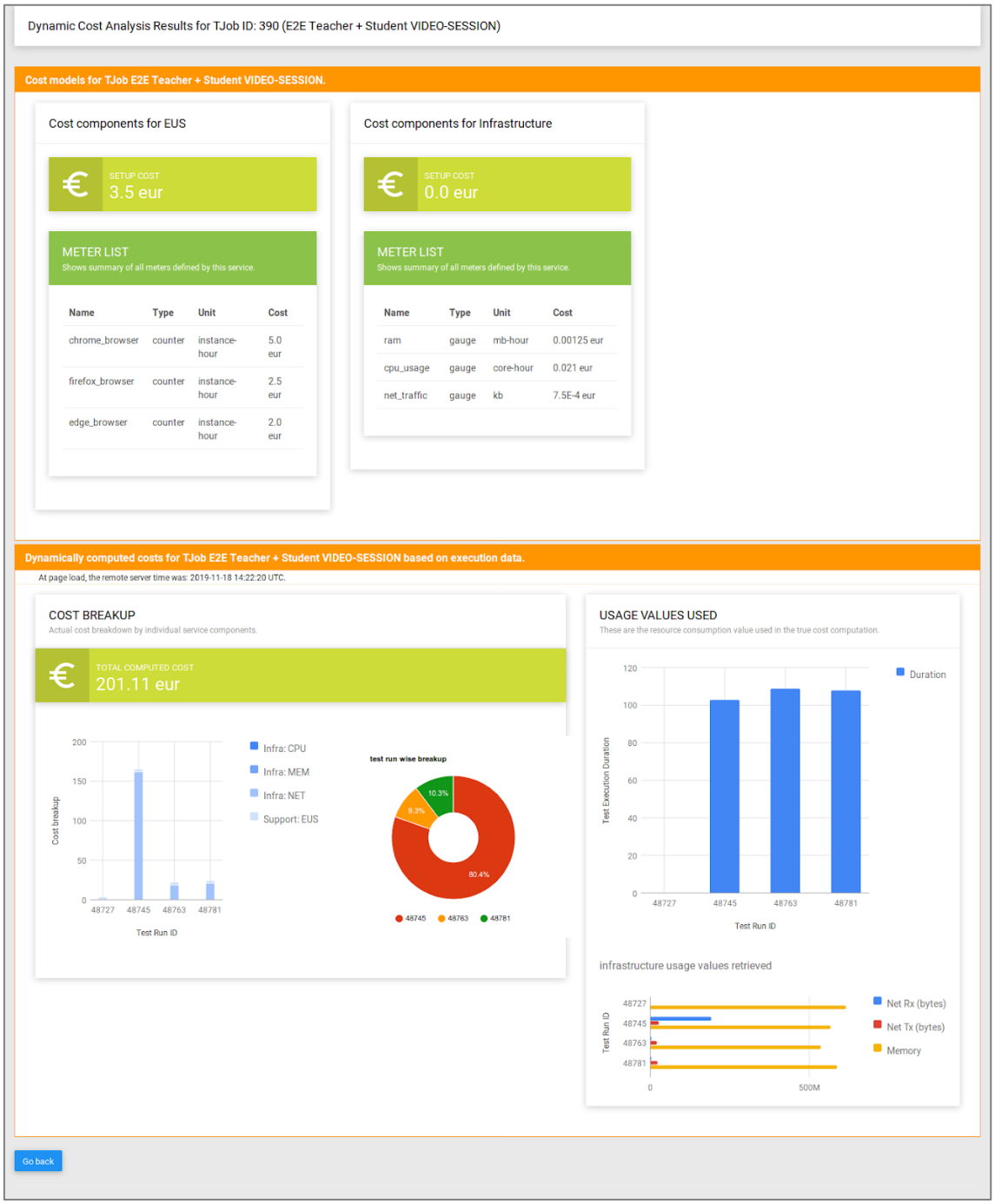

Figure 11. True computed costs of all executions for a selected TJob

The key technology parameters that characterizes the ECE are:

- Programming language: Java 8
- Framework: Spring framework
- Templating framework: Thymeleaf

The implementation exposes the RESTful interface through a class Controller.java that allows the ElasTest GUI ask ECE for cost estimation for a given TJob ID.

5.6. Limitations of current approach

The current true cost computation is based on the data received from the EMP, and thus the fidelity of the results are heavily based on the faithfulness of data received from REST calls to the EMP. The EMP collects resource consumed data from various agents which report periodically to it. In the currently integrated nightly deployment, the docker agents of the EMP are configured to report every 15 minutes. Thus it is possible that a TJob execution which gets started and finished in between the two collection epochs, will not be assessed for cost, simply due to the fact that the EMP reports no data on such executions.

One way to overcome this limitation is to add a lifecycle event processing engine which gathers all TJob execution states such as started, finished, etc. and compute the true cost calculation based on the duration of execution. Such an alternative is possible, but will require the adaptation of the cost models and will not be truly representative of actual resources consumed. It is for this reason, such an alternative was not pursued in this task.

6. Conclusions and future work

This deliverable provides a summary of the technical aspects of the following components of the ElasTest toolbox: i) the ElasTest Tests Manager (ETM), ii) the ElasTest Orchestration Engine (EOE), and iii) the ElasTest Cost Engine (ECE).

Regarding the ETM, its main objectives were: i) to allow the execution of end to end tests against complex distributed applications coordinating the rest of the ElasTest components and ii) to gather, register and analyze the information generated during test execution. These two objectives have been accomplished. The ETM have been implemented using several open source technologies (Docker, Elastic stack) and frameworks (Spring Boot, Angular). This component provides extensibility mechanisms to allow third party modules to be included in ElasTest. This mechanisms have been used to include all TSS and TE. Extensive validation have been performed to evidence that features provided by ETM are useful for testers in real projects and automated tests are executed to verify the expected behavior and detect regressions. The future work of ETM will be focused on the automatic analysis of the information gathered during test execution in several uses cases like regressions and comparisons of several SUT configurations.

Regarding the EOE, we conceive test orchestration as a novel way to select, order, and execute a group of TJobs. We distinguish two types of orchestration techniques. The first one is called *verdict-driven* orchestration, and it allows to create TJobs workflows by modeling TJobs as black-boxes, meaning that we only known its final verdict (i.e., passed or failed) after the execution. Each TJob verdict value can be used to create conditional paths within the orchestration workflow. The second approach presented in

this deliverable is called *data-driven*. This second approach is more complex due to the fact that tests within TJobs are supposed be composable, meaning that the test data (input) and test outcomes (output) are imported and exported by tests. The inconvenience of this approach is that new tests following these guidelines need to be created. On the other side, we can create richer test suites using the "divide and conquer" principle applied to testing, as hypothesized in the ElasTest DoA.

These orchestration approaches are being implemented in the ElasTest platform. Internally, ElasTest has been implemented following a microservices architecture based on Docker containers. The ElasTest component in charge of implementing the orchestration approaches is called ElasTest Orchestration Engine (EOE). This component is able to parse an orchestration workflow based on the DSL Jenkins Pipeline, sequencing, and executing in parallel tests according to the DSL (provided by testers). In order to ease the development of composable test as required in the data-driven approach, the ElasTest project is going to provide a reference implementation as a JUnit 5 extension [4]. This extension is not released at the time of this writing, although we can anticipate how the final JUnit 5 will look like. The following listing shows an example, in which input and output data are specified using Java annotations. Notice that the input data can declare some default value in order to be executed as single instances (i.e., outside the orchestration workflow). These data are later overridden by the EOE in the actual orchestration execution.

```
@ExtendsWith(ElasTestExtension.class)
class TJob1Test {

   @InputData
   String in1 = "default-value1";

   @InputData
   int in2 = 20;

   @InputData
   boolean in3 = false;

   @OutputData
   String out1

   @OutputData
   int out2

   @Test
   void myTest() {
       // my test logic
   }

}
```

Snippet 3. Data-driven JUnit 5 test case design

This work is the first step in our vision to create a novel testing theory for sequencing, ordering, and parallelization applied to software testing. This is an ambitious goal, and so, there is still a long path ahead. So far, we have focused in the first part of the problem, i.e. the definition of a topology generation to orchestrate tests. The next steps include actions to enhance the current model using test augmentation, i.e. introducing new TJobs to reproduce custom operational conditions of the SUT. Moreover, we plan to investigate additional techniques (new or existing) to include automated assertions (i.e., the oracle problem [5]) applied to the output data in the data-driven orchestration approach.

Regarding the ECE, cost estimation brings much needed financial transparency in any testing infrastructure. The process of accounting, rating, and charging and billing is a complicated process. Although in ElasTest we do not do billing, the complexity and challenges of accounting remains. As a first step, we have defined a reasonably flexible cost model and have prototyped the initial version of cost engine that performs static cost estimation based on defined cost plans of supporting services. In the immediate future, we begin implementing real cost calculation capability based on observed metrics and utilizing planned metering module. We will also enhance TJob registration process with usage model inclusion which will add more teeth to the cost estimation for the TJob.

7. References

[1] Mili, A. and Tchier, F., 2015. *Software testing: Concepts and operations*. John Wiley & Sons.

[2] Lima, B. and Faria, J.P., 2016, July. A Survey on Testing Distributed and Heterogeneous Systems: The State of the Practice. In *International Conference on Software Technologies* (pp. 88-107). Springer, Cham.

[3] ElasTest project Description of Action (DoA) – part B. Amendment 1. Reference Ares(2017) 343382. 23 January 2017.

[4] B. García, *Mastering Software Testing with JUnit 5*. Packt Publishing, 2017.

[5] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S., 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, *41*(5), pp.507-525.

[6] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 222-232. DOI: https://doi.org/10.1145/3180155.3180210