

D4.1

Version	1.0
Author	URJC
Dissemination	PU
Date	27-06-2018
Status	FINAL



D4.1 Test Orchestration basic toolbox v1

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	30-06-2018
Actual delivery date	30-06-2018
Keywords	Open source software, cloud computing, software engineering, operating systems, computer languages, software design & development



Funded by the European Union

License

This is a public deliverable that is provided to the community under a **Creative Commons Attribution-ShareAlike 4.0 International** License:

<http://creativecommons.org/licenses/by-sa/4.0/>

You are free to:

Share — copy and redistribute the material in any medium or format.

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

For a full description of the license legal terms, please refer to:

<http://creativecommons.org/licenses/by-sa/4.0/legalcode>



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	07/05/2018	URJC	Structure of document and first contents
0.2	07/05/2018	URJC	Orchestration engine section
0.3	18/05/2018	ZHAW	Cost engine section
0.4	18/05/2018	ZHAW	Table formatting fix
0.5	30/05/2018	URJC	Restructuration of orchestration engine
0.6	31/05/2018	ZHAW	Restructured cost engine section
0.7	31/05/2018	URJC	Test Orchestration and Recommendation Manager section
0.8	31/05/2018	URJC	ETM - Logs and Metrics section
0.9	31/05/2018	URJC	ETM - GUI diagrams
0.10	04/06/2018	URJC	Integrated version
0.11	06/06/2018	URJC	ETM description in Section 1
0.12	19/06/2018	URJC	Corrections after internal review
0.13	25/06/2018	URJC	ETM section improvements
1.0	27/06/2018	URJC	Final version

Table of contents

1. Executive summary	8
2. Introduction.....	8
3. ElasTest tests manager.....	10
3.1. Introduction	10
3.2. Features.....	11
3.3. Baseline concepts and technologies	12
3.4. Component architecture	13
3.4.1. Component diagram.....	13
3.4.2. Metrics and logs	18
3.4.3. Data model	19
3.4.4. Use cases	19
3.5. Code links	28
3.5.1. Validation	28
3.5.2. Discussion	29
3.6. Research results and plans.....	29
4. ElasTest orchestration engine	30
4.1. Introduction	30
4.2. Features.....	31
4.3. Baseline concepts and technologies	31
4.3.1. Test composition.....	32
4.3.2. Test parallelization	32
4.3.3. Orchestration languages	33
4.4. Component architecture	34
4.5. Code links	38
4.5.1. Validation	40
4.5.2. Discussion	42
4.6. Research results and plans.....	42
5. ElasTest cost engine	43
5.1. Introduction	43
5.2. Features.....	43
5.3. Baseline concepts.....	44
5.4. Component architecture	44
5.4.1. Architecture and workflows	45
5.4.2. Cost model elements	49
5.5. Implementation and code links.....	51
5.5.1. Validation	52
5.5.2. Discussion	54
5.6. Research results and upcoming plans.....	54
6. Conclusions and future work.....	54
7. References.....	56

List of figures

Figure 1. ETM sub-components	14
Figure 2. ETM Core modules used to execute TJobs.....	15
Figure 3. ETM GUI components	16
Figure 4. LogAnalyzer modules	16
Figure 5. ETM Core modules for TestLink integration	17
Figure 6. Metrics and logs in ElasTest	18
Figure 7. ETM Core Data Model	19
Figure 8. Define a TJob and execute it	20
Figure 9. Define a TJob with TSS and execute it.....	21
Figure 10. Create a TJob	22
Figure 11. TJob execution.....	23
Figure 12. Search logs with LogAnalyzer	24
Figure 13. Filter logs in LogAnalyzer.....	26
Figure 14. Mark logs in LogAnalyzer	27
Figure 15. Test Plan Execution	28
Figure 16. Verdict-driven test orchestration.....	35
Figure 17. Data-driven test orchestration.....	36
Figure 18. EOE schema	37
Figure 19. Using ElasTest orchestration Jenkins library.....	41
Figure 20. Execution of the orchestration in Full Teaching application.....	41
Figure 21. ECE FMC Architecture	45
Figure 22. Schematic showing execution events and resource usage metrics flow enabling real cost estimation.....	47
Figure 23. Flowchart showing steps in true cost computation.....	48
Figure 24. ECE landing page and selection of ece-test TJob as part of validation	52
Figure 25: ECE log sample	52
Figure 26. Cost analysis result page	53
Figure 27. ElasTest Jenkins CI server stages for ECE end-to-end integration test pipeline	53

List of tables

Table 1. ElasTest tests manager features.....	12
Table 2. Orchestrator requirements	31

Table 2. Orchestrator API	38
Table 3. Orchestrator exit condition alternatives	39
Table 4. Orchestrator parallel jobs verdict conditions.....	39
Table 5: Cost Engine Requirements	44
Table 6: Illustrative examples of cost models	50
Table 7 ECE REST interface methods supporting key GUI functions.....	51
Table 8 ECE container necessary environment parameters	51

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 several of the 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 tests in ElasTest (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. It keeps a log of all TJobs executing during the history, along with all their related information: logs and metrics. In the next stage 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 within ElasTest. To that aim, two different actions 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. At the time of this writing the topology part has already been implemented, leveraging the Jenkins pipeline DSL notation to create two different orchestration approaches: i) *verdict-driven* orchestration, i.e. connecting TJobs using its verdict (i.e., passed or failed) as Boolean condition; ii) *data-driven*, i.e. connecting TJobs using the test data (input) and test outcomes (output) handled internally by tests. In the future we plan to release a reference implementation of the data-driven approach for tests and also contribute in the test augmentation part, missing so far.

Regarding ECE, we have defined a flexible cost model and prototyped the initial version of cost engine that performs static cost estimation based on defined cost plans of supporting services. In the next part of the project, real cost calculation based on ElasTest metrics is planned.

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 heterogeneous 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.

ElaSTest core functionality 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 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 entitled “Test Orchestration basic toolbox v1”.
- ElaSTest Orchestration Engine (EOE). This engine is responsible of providing capabilities for selecting, ordering, and executing a group of TJobs in ElaSTest. TJobs is the name given in the ElaSTest jargon to the test entities to be executed in ElaSTest. 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 is described in deliverable D2.3, entitled “ElaSTest requirements use-cases and architecture v1”. This deliverable is focused in the technical description of several of the above-mentioned components. On the one hand, first we present the features, baseline concepts and design/implementation details of ETM in section 3. On the other hand, EOE and ECE are presented 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 deliverable D2.3 entitled “ElaSTest requirements, use-cases and architecture v1”, 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 consist in coordinate 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 (System in the Large, SiL).

ETM allows the users to define what tests are going to be executed against what SUT with what support services. All this information is modeled as a test Job (TJob) that can be executed. During the execution of TJobs, logs and metrics generated by tests and SUT components are registered. Also, all relevant information generated by Test Support Services (TSSs) are also registered. For example, if a TJob uses ElaSTest User Impersonation Service (EUS) to use browsers, during TJob execution, the console of the browser is registered with the rest of the information. All the information registered during TJob executions can be visualized in real time in the ElaSTest graphical user interface. It also can be analyzed after the execution with LogAnalyzer, a powerful tool to analyze and compare logs. In the future, more advanced analysis tools are planned.

ElaSTest is an extensible platform allowing third parties to augment the functionality provided by it. Some of the features provided by ElaSTest are already defined as plugins or external modules, showing the powerful of the platform in this aspect. ElaSTest has two types of external modules: Test Support Services (TSS) and Test Engines (TE). TSSs are modules used directly from the test code to provide them high level features to exercising the SUT or assert the expected results. For example, EUS is a TSS providing browsers as a service to test web pages. ElaSTest Security Service (ESS) provides security testing services to analyze web applications looking for vulnerabilities. Regarding to TE, they are modules used interactively by ElaSTest user. For example, ElaSTest Cost Engine (ECE) provides users cost information about TJob executions. ElaSTest is the host of this two types of third party modules. At the moment of writing this deliverable, only modules implemented by the project members are available in the platform, but it is planned to allow users to implement their own modules and install them in an ElaSTest instance.

Finally, some of the integrations of ElaSTest with external tools are implemented in the ETM. Currently, ElaSTest is integrated with the most used open source tools in the areas of continuous integration (Jenkins¹) and test management (TestLink²). These integrations requires changes in the data models used to manage TJob executions. For that reason, they are included in the ETM itself instead of as third party modules.

¹ <https://jenkins.io/>

² <http://testlink.org/>

This section is devoted to describe how ETM is implemented at milestone M18 of the project lifecycle (i.e. June 2018). The rest of this section is structured as follows. Section 3.2 presents the main features of ETM. Next section 3.3 presents a detailed description of the technologies used in the implementation of the component. Afterwards, section 3.4 describes the internal architecture of the component and how it is implemented. Finally, section 3.5 describe the main aspects related to source code.

3.2. Features

The list of features implemented in ElaSTest Tests Manager (ETM) component is summarized in the following table.

Feature	Description
Manage projects	As ElaSTest user, I want CRUD operations on projects to create, edit, remove and update test projects to group TJobs and SUTs
Create SUTs	As ElaSTest user, I want to create SUTs so I can specify how to start a SUT with the following options: Deployed by ElaSTest (Docker, Docker-compose, commands) or Deployed Elsewhere.
Manage SUTs	As ElaSTest user, I want CRUD operations on SUTs to create, edit, remove and update SUTs
Create TJobs	As ElaSTest user, I want to create TJobs so I can specify what SUT should be tested and how to execute tests against it
Manage TJobs	As ElaSTest user, I want CRUD operations on TJobs to create, edit, remove and update TJobs
Execute TJobs	As ElaSTest user, I want execute a TJob so logs, metrics and tests results can be recorder for further inspection
Dashboard	As ElaSTest user, I want to see projects and last TJob executions in a single screen so I can have an overview of the status of the platform
Review TJob executions	As ElaSTest user, I want to review finished TJob executions I can see what happened, especially in executions with failed tests
Test Support Services	As ElaSTest user, I want to specify what TSSs must be ready to use when a TJob is executed so Tests in TJob can use selected TSS when testing the SUT
Log analyzer	As ElaSTest user, I want to analyze, filter and mark logs gathered during TJob execution so problem troubleshooting is easier than looking to plain log
Test case execution	As ElaSTest user, I want to review easily all information gathered during one specific test (logs, events and files) so I can focus on information related to a test (possible failed)

TestLink info management	As ElasTest user, I want to see TestLink projects, test cases, suites, builds and test plans in ElasTest interface so I can see that information integrated with other TJobs and projects
TestLink Test plan execution	As ElasTest user, I want to execute TestLink Test plans using browsers provided by ElasTest and recording all information from SUT and browsers so I can associate all that information to a bug report in case of test failure
Test Engines	As ElasTest user, I want to start, use and stop a Test Engine so I can start the engine only when needed
Show platform information	As ElasTest admin, I want to see the version and compilation date of ElasTest components so I can see if platform is updated or not
Show logs and metrics in real-time	As ElasTest user, I want to see logs and metrics from SUT and Tests execution so I can know what happen with SUT and Tests in case I want to solve any problem

Table 1. ElasTest tests manager features

3.3. Baseline concepts and technologies

ETM is composed internally by several sub-components. The main sub-component is ETM Core, a service providing 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 architecture³ (SPA). ETM Core is the responsible to coordinate the rest of the ElasTest components and other internal sub-components.

ETM Core is implemented in Java language with Spring Boot framework⁴. ETM GUI is implemented in TypeScript language with Angular framework⁵. Other sub-components used in ETM are:

- Logstash⁶: Used to retrieve and process logs and metrics during TJob execution. This information is the stored in Elasticsearch⁷ provided by ElasTest Data Manager component (EDM). Logstash and Elasticsearch are part of elastic stack⁸, the leading open source stack used to gather, process, register and analyze logs, metrics and any kind of KPI of Internet applications.
- RabbitMQ⁹: Used to send real time information from ETM-core to the frontend my means of WebSockets. RabbitMQ is a leading message queue software well integrated with Spring technologies used in ETM-core.

³ https://en.wikipedia.org/wiki/Single-page_application

⁴ <https://spring.io/projects/spring-boot>

⁵ <https://angular.io/>

⁶ <https://www.elastic.co/products/logstash>

⁷ <https://www.elastic.co/products/elasticsearch>

⁸ <https://www.elastic.co/>

⁹ <https://www.rabbitmq.com/>

In the rest of the section, the interactions between ETM Core and the rest of the subcomponents of ETM is being described.

3.4. Component architecture

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

3.4.1. Component diagram

ETM is composed by the following sub-components:

- **ETM Core:** Service backend that coordinate all other internal sub-components and interacts with the rest of ElasTest components.
- **ETM GUI:** is the graphical interface with which the user interacts. It is called Angular GUI to clarify the technology used to build it.
- **RabbitMQ:** is a messaging broker used to send logs and metrics in real time to the user interface.
- **Logstash:** is a server-side data processing pipeline that ingests received data, transforms it, and then sends it to RabbitMQ and Elasticsearch. Elasticsearch is a sub-component of ElasTest Data Manager (EDM) component used to register all information gathered during test execution.
- **Dockbeat** and **Filebeat:** this services are used for log retrieval and monitoring of TJobs executed as docker containers. They are well integrated with the elastic stack.
- **TestLink:** ETM includes an instance of this project to manage manual tests.

These sub-components are illustrated in Figure 1.

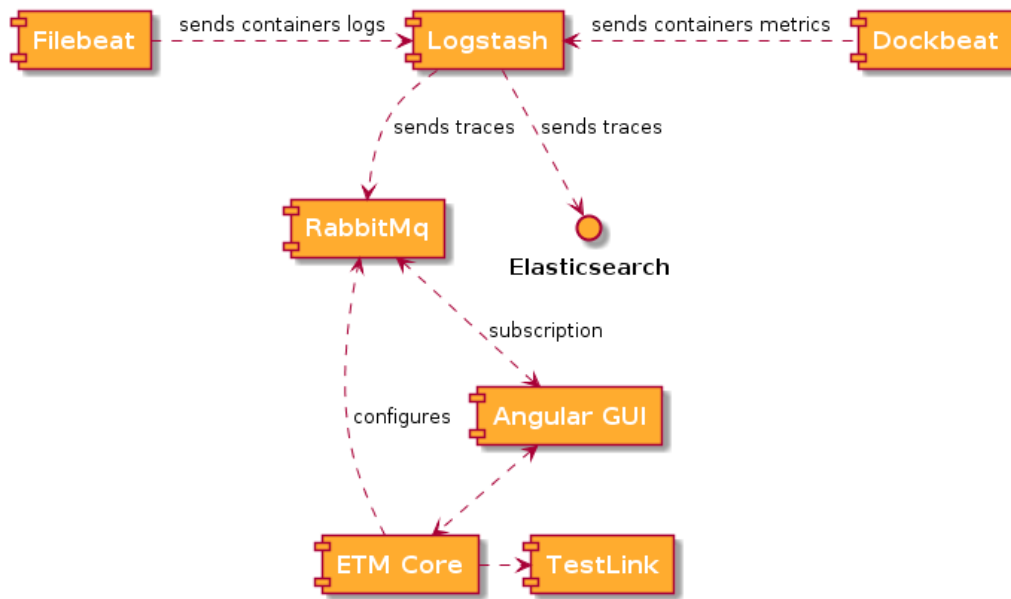


Figure 1. ETM sub-components

3.4.1.1. Modules used for TJob execution

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 the control of execution using the following services:

- **EsmService:** interacts with ElasTest Service Manager (ESM) component to manage the Test Support Services (TSSs) associated to the TJob. It is performed using EsmServiceClient through ESM API.
- **SutService:** used in case of TJob has associated SUT.
- **DockerService:** making use of Docker, this service will initialize and start the necessary containers for TJob execution. The containers started are Dockbeat (to get execution metrics), the container for to execute the tests and, if TJob has specified 'SUT Deployed by ElasTest', SUT container. It is also responsible for obtaining the result files and copying them to the user's filesystem through FileServices.
- **DockerComposeService:** used when TJob has associated 'SUT Deployed by ElasTest from Docker Compose' to start/stop docker-compose SUT services.

The interaction of these modules during the execution of the TJob is reflected in Figure 2.

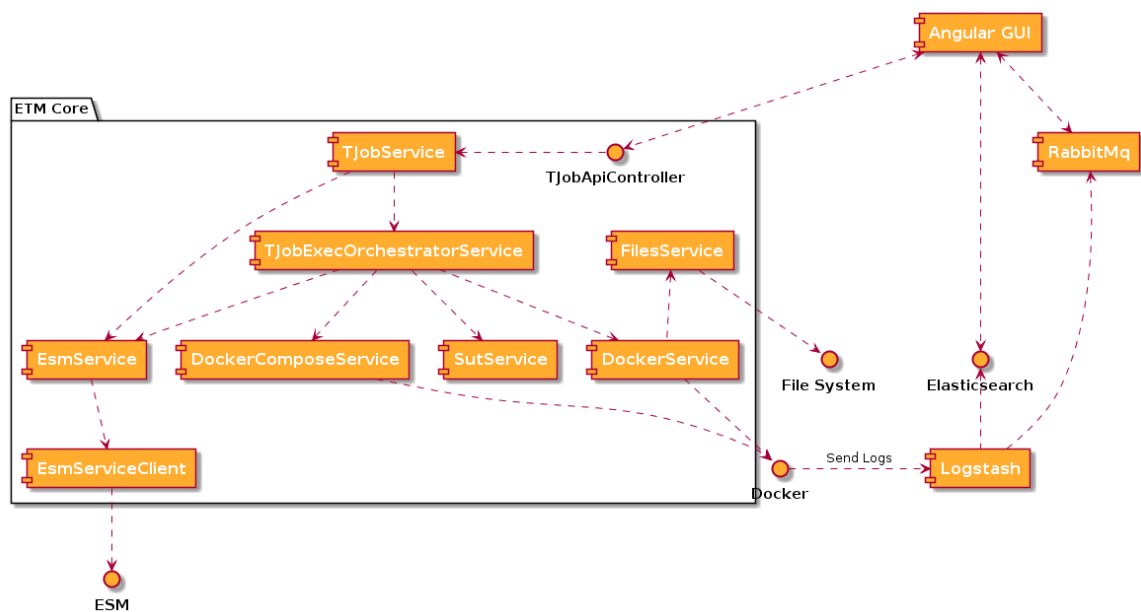


Figure 2. ETM Core modules used to execute TJobs

In the GUI side, the main modules are the following:

- **DashboardComponent:** the main component. It 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:** TJobs are managed by this service to populate the GUI with them.
- **TJobExecService:** It updates the interface in real time during the execution of TJobs. For that, it implements a pooling strategy.
- **ElastestRabbitmqService:** creates the necessary connections with RabbitMQ to obtain logs and metrics in real time when the TJob is executing.
- **EtmMonitoringViewComponent:** is responsible for managing everything related to metrics and logs, making use of the information obtained from RabbitMQ and, occasionally, from Elasticsearch.

Figure 3 shows the relationship between these components:

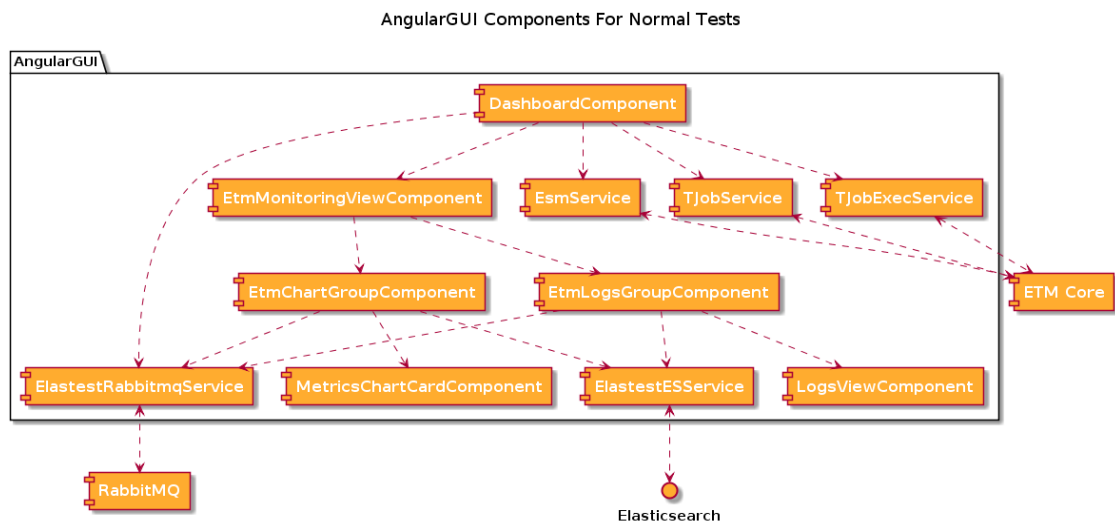


Figure 3. ETM GUI components

3.4.1.2. Modules used in LogAnalyzer

LogAnalyzer is a part of ETM that allows the user analyze logs retrieved during tests execution. It allows the user mark and filter log entries with certain patterns or contents. Figure 4 shows the main GUI modules of LogAnalyzer. Let's see details for each of them:

- **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 **TJobService**, **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.
- **ElastestESService:** this service is used to make queries to get the logs from Elasticsearch.

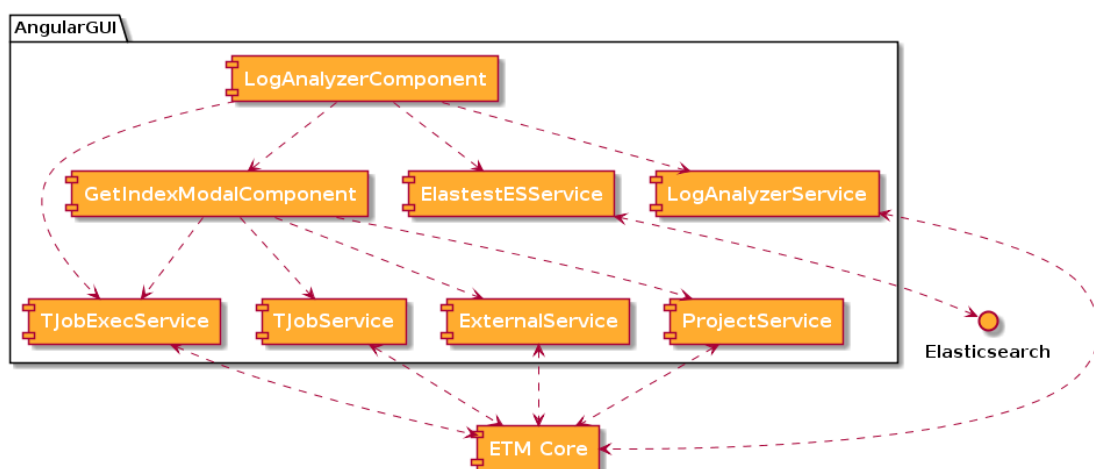


Figure 4. LogAnalyzer modules

3.4.1.3. Modules used in TestLink integration

ElasTest allows users to manage manual tests with its integration with TestLink. A user can execute the test cases of a test plan defined in TestLink and gather information during the execution. All that information (logs and metrics) is available for later inspection for problem troubleshooting.

To implement this functionality, ETM GUI interacts with **TestLinkApiController** in ETM Core to get TestLink information. The controller makes use of the TestLink API through **TestLinkService**. TestLinkService depends on DockerService to get TestLink container information such as the IP.

ElasTest offers the user the ability to run a test plan. A TestLink test plan is associated with an ExternalTJob. When the user runs a test plan, ETM GUI interacts with both **ExternalApiController**, for managing data associated with ElasTest, and TestLinkApiController, for managing data associated with TestLink.

ExternalApiController uses **ExternalService** to process the request received. ExternalService needs EsmService to start/end a EUS that provides a browser to the user for the tests.

The interaction of the modules when TestLink is used can be seen in Figure 5.

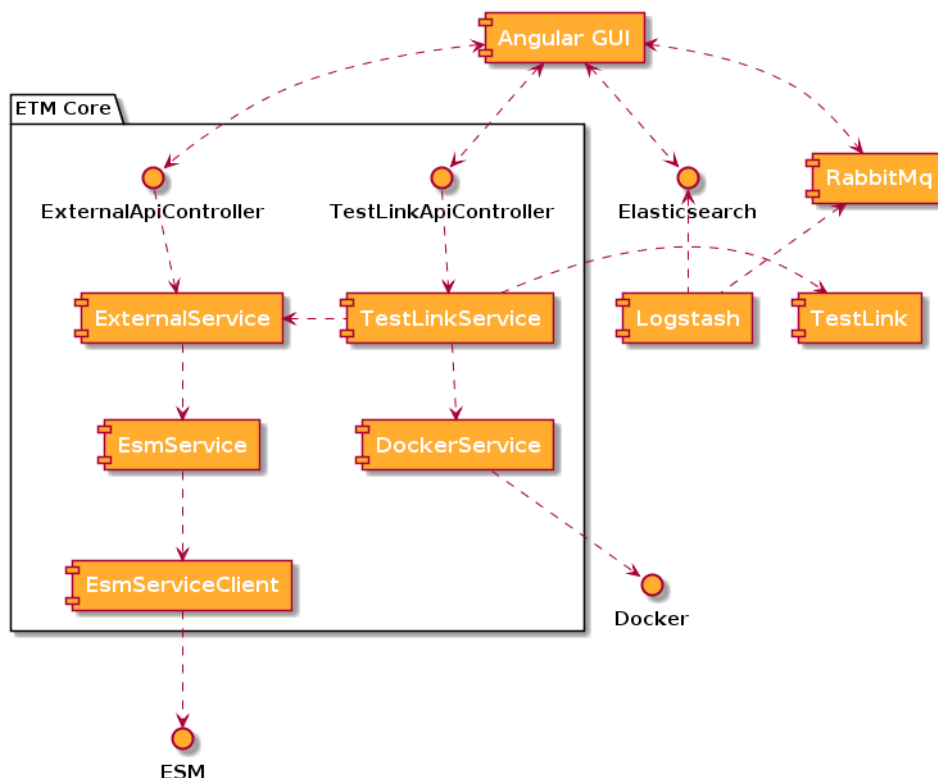


Figure 5. ETM Core modules for TestLink integration

3.4.2. Metrics and logs

ElasTest can show metrics and logs that, as we have seen before, are received and processed by Logstash. These are then forwarded to Elasticsearch for persistent storage and to RabbitMQ for real time visualizations during the execution.

Logstash has five input ports configured to receive data:

- **5000**: TCP port to receive logs using syslog format from TJob containers.
- **5001**: Beats port to receive both logs and metrics from EMS.
- **5003**: Http port to receive both logs and metrics.
- **5037**: Beats port to receive only metrics from ETM Dockbeat.
- **5044**: Beats port to receive both logs and metrics.

Depending on the type of trace received (log or metric) and its input port, Logstash will process it differently. The result of the processing has a set of common main fields that are necessary for ElasTest to interpret the traces:

- **@timestamp**: full date of the trace.
 - e.g.: '2018-05-31T12:56:37.668Z'
- **exec**: The execution index where Elasticsearch will store the trace.
 - e.g.: '37', 's1_e37'...
- **component**: represents the component or service from which the trace is collected.
 - e.g.: 'sut', 'test', 'sut_fullteaching'...
- **stream_type**: the type of the trace.
 - e.g.: 'log', 'composed_metrics' or 'atomic_metric'
- **stream**: it's a way to classify the trace.
 - e.g.: 'default_log', 'console', 'et_dockbeat'

Figure 6 shows an example of visualization of metrics and logs obtained from the execution of a TJob in ElasTest.

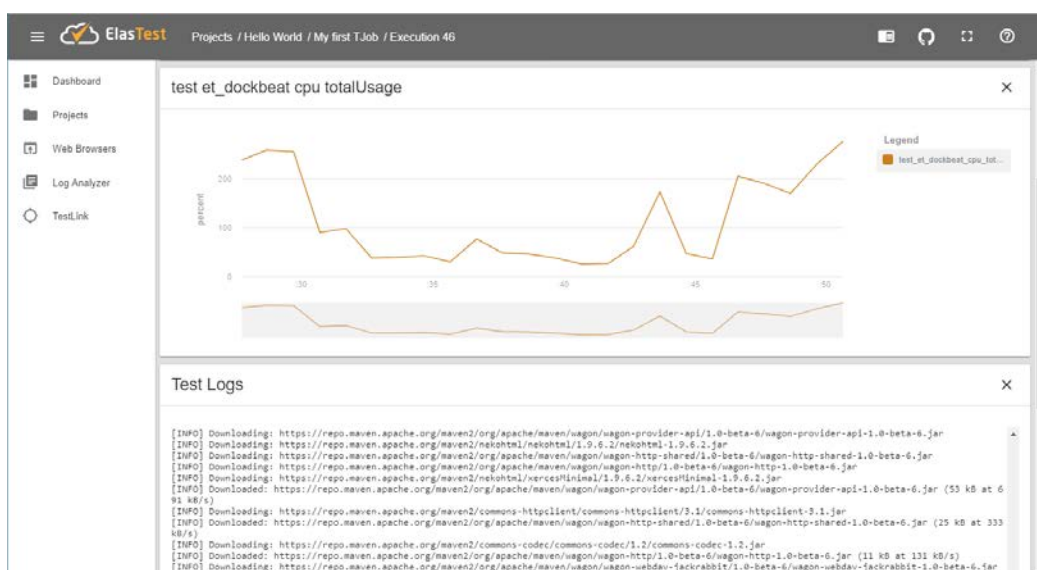


Figure 6. Metrics and logs in ElasTest

3.4.3. Data model

The ETM Core works with a unified data model, this means that the presentation model, the logical model and the persistent model are the same. Figure 7 shows this data model. In this class diagram it can be seen all the entities that make up the ETM model and their relationships.

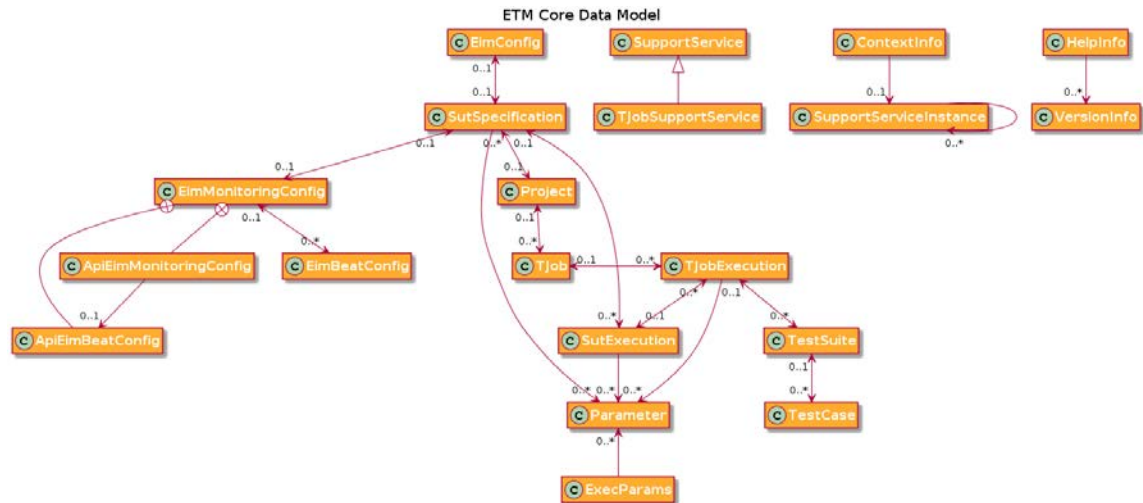


Figure 7. ETM Core Data Model

3.4.4. Use cases

In this subsection, several sequence diagrams are shown to describe how subcomponents of ETM collaborates to perform the most important ElasTest use cases.

3.4.4.1. TJobs

The main use case for ETM is define a TJob and execute it, as can be seen in Figure 8.

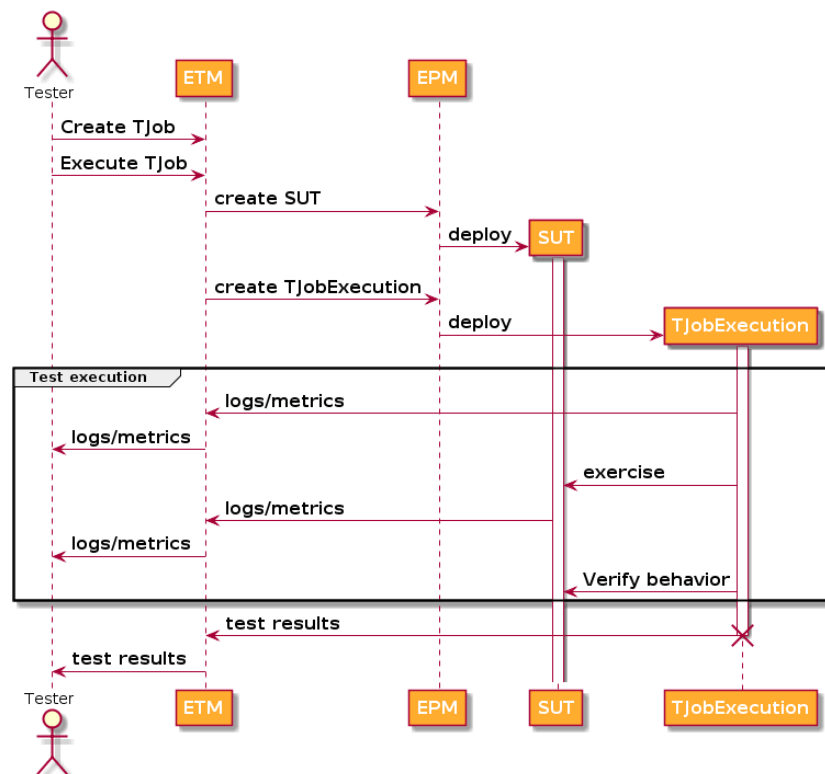


Figure 8. Define a TJob and execute it

Figure 9 shows the process of using a Test Support Services (TSS) in the TJob.

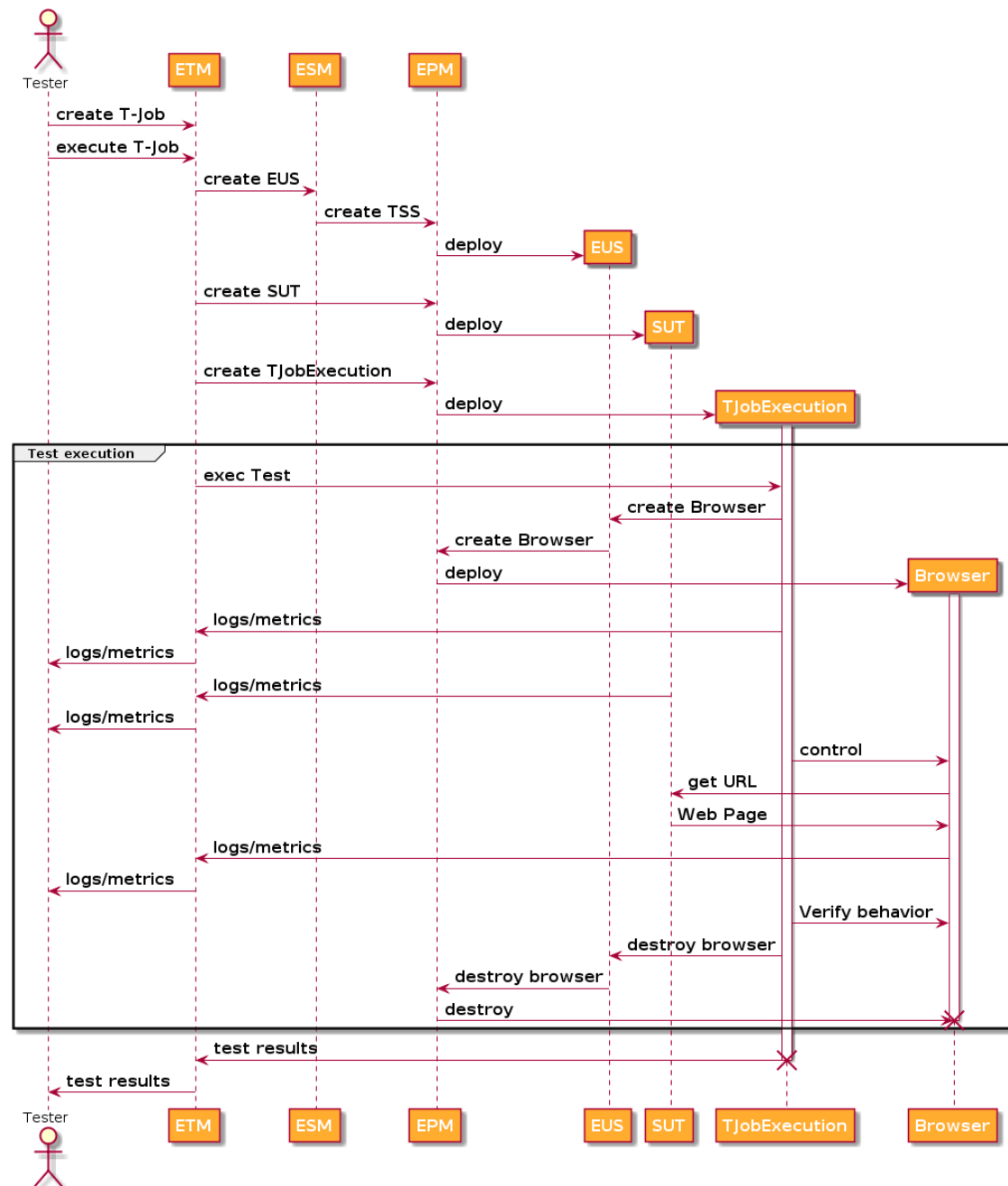


Figure 9. Define a TJob with TSS and execute it

The sequence diagram shown in Figure 10 represents the creation of a TJob in more detail, focusing on the sub-components of ETM Core used to do that.

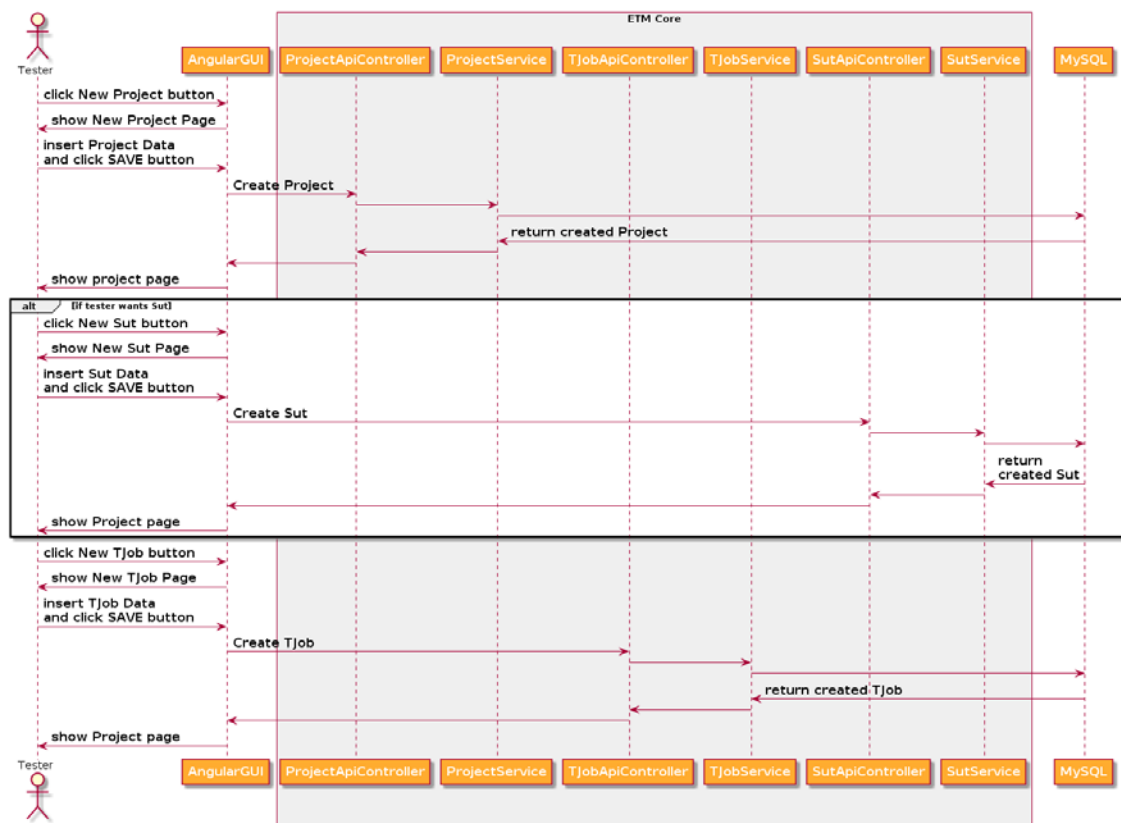


Figure 10. Create a TJob

As it can be seen in the diagram, user must first create a project. Afterwards, he can create a SUT and assign it to the project. Lastly, he can create a TJob into the project.

Once the TJob is created, user can execute it as seen in the diagram of Figure 11.

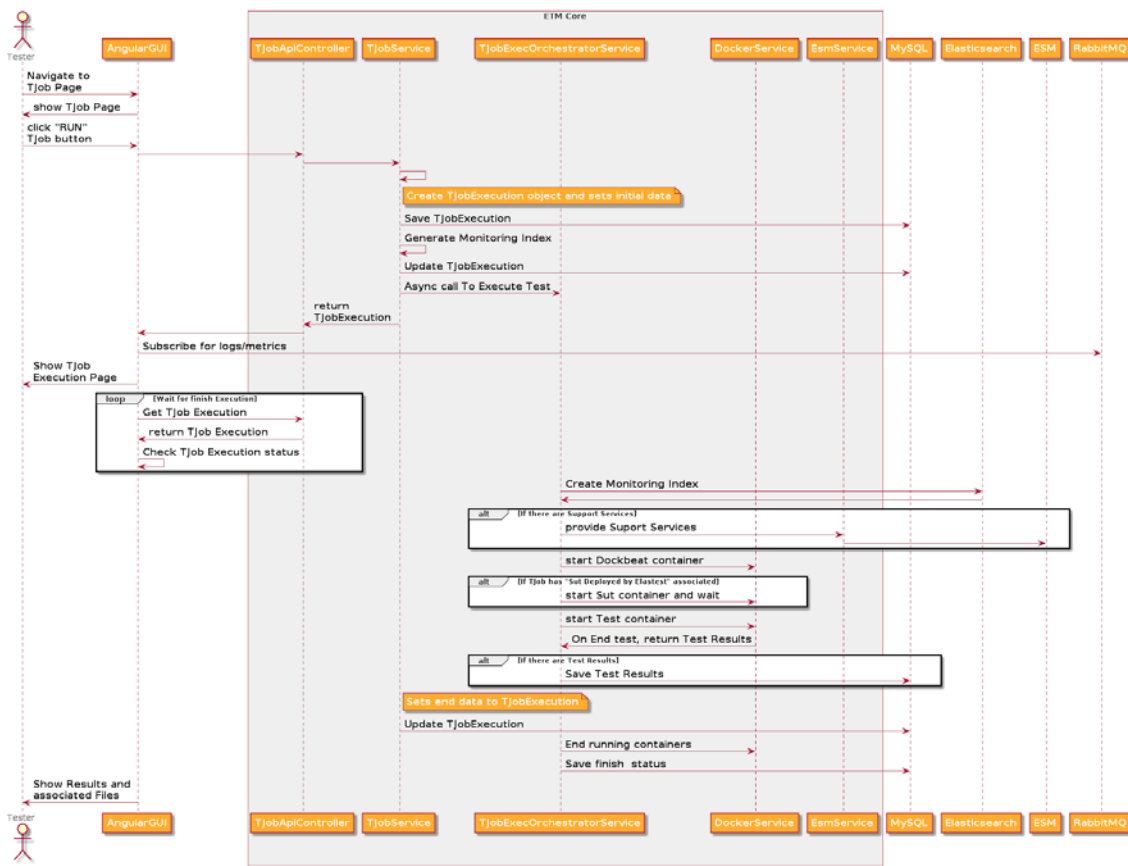


Figure 11. TJob execution

When user runs a TJob, a new TJob Execution is created and stored into MySQL database. Next, a monitoring index is generated and TJob Execution Object is updated in the database. Then, the execution is launched asynchronously, and the Execution object is returned to the ETM GUI.

The async process perform the following actions:

1. First it creates the generated monitoring index into Elasticsearch, where metrics and logs will be stored.
2. If the TJob has Test Support Services (TSSs) selected, it calls to EsmService to provision them.
3. It starts Dockbeat container to send metrics of tests and SUT (if applies).
4. If there is a 'Deployed by ElasTest SUT' associated to TJob, the process starts it.
5. It starts the test container and wait until it ends.
6. If there are Test Results, copies and saves them into MySQL and associates them to TJobExecution.
7. Updates TJob Execution into MySQL.
8. Ends started containers.
9. Saves finish status.

On the other hand, when ETM GUI receives the TJob execution information:

1. Shows the TJob Execution page

2. Subscribes to RabbitMQ to get metrics and logs in real time
3. Pools ETM Core periodically for the status of the execution

When the test is finished, the test results are shown. In addition, files generated by the execution (for example, browser recordings generated by EUS) are shown in the GUI.

3.4.4.2. LogAnalyzer

Figure 12 shows how ETM sub-components interact to allow users to analyze execution(s) logs.

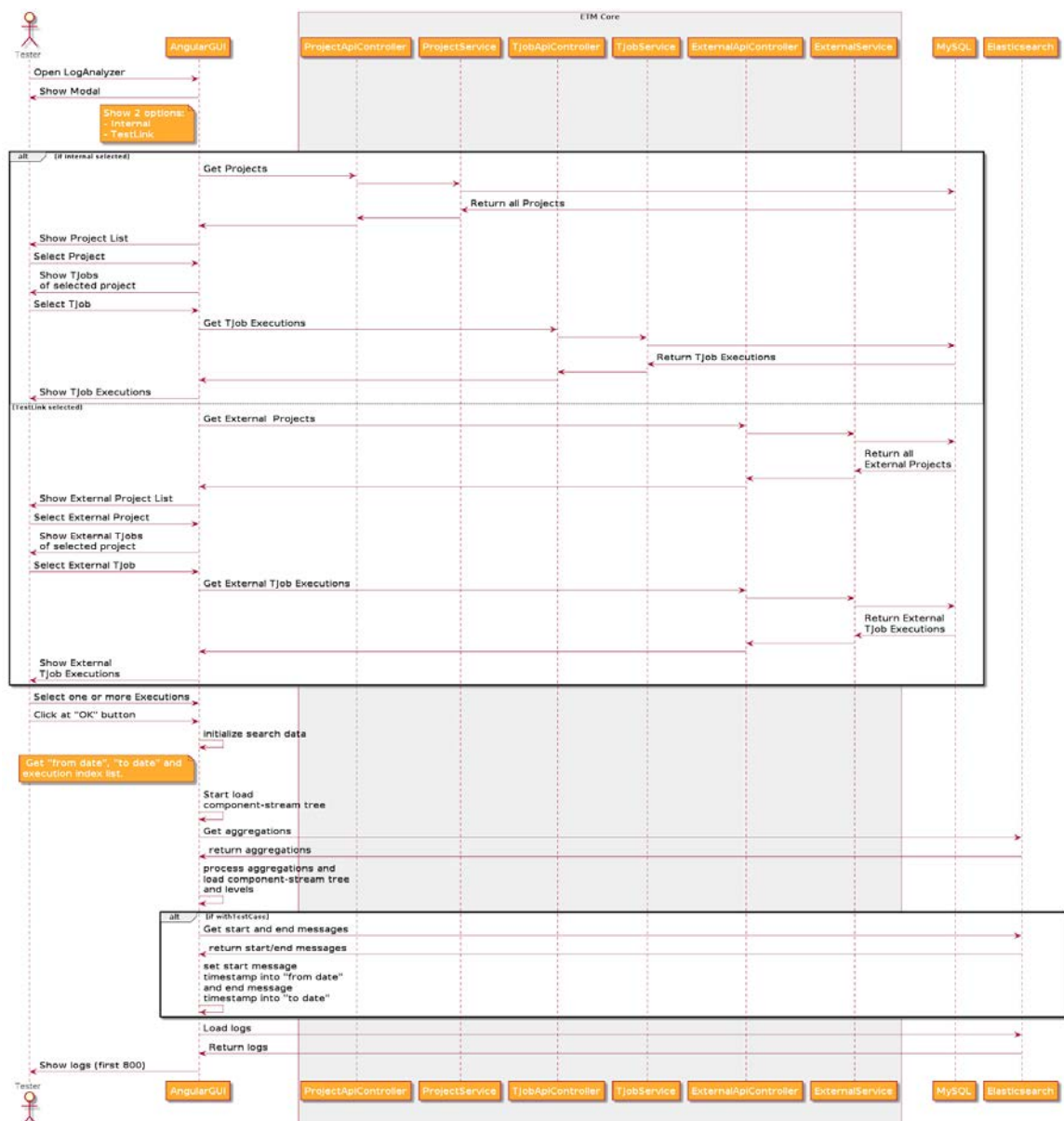


Figure 12. Search logs with LogAnalyzer

As can be seen, user can select first between two options: Internal (normal TJob Executions) or TestLink (Test Plan Executions). Once chosen, he can select a Project and a TJob for it. Lastly, one or more Executions can be selected to search through logs.

Logs will be shown, and user will have a panel on the right side that will allow him to create filters or mark log entries for matching words.

LogAnalyzer has several filters:

- **From date/To date:** Narrow the search from “from date” to “to date”.
- **Tail:** If this option is checked, “To date” will be ignored and LogAnalyzer will load logs periodically.
- **Components/Streams Tree:** show only logs from selected Component/Stream nodes.
- **Levels:** show only logs from selected level.
- **Message:** Filters by matching phrases in the message.
- **Nº Entries:** Number of logs to load (Max 10.000).

In addition to that filters, the user has three options:

- Reload logs with selected filters.
- Load logs with selected filters from last loaded trace.
- Load logs with selected filters from selected trace.

Figure 13 illustrates this process.

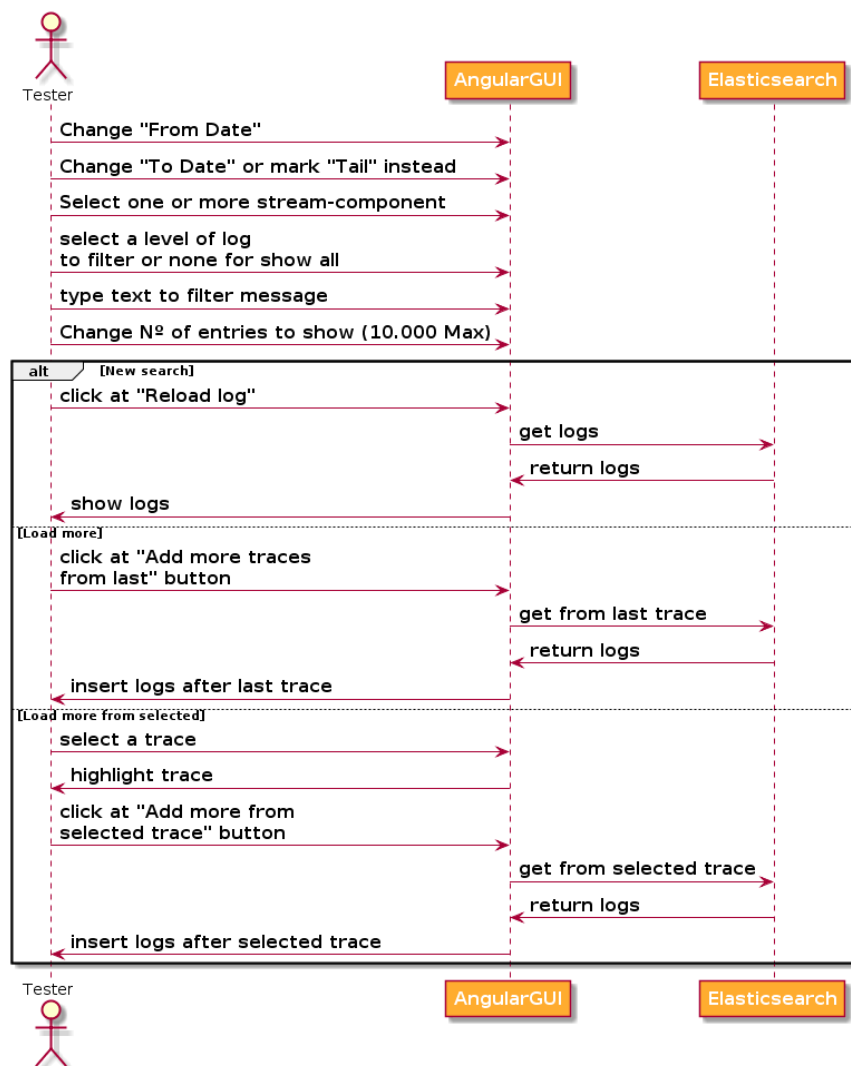


Figure 13. Filter logs in LogAnalyzer

Into the “Mark” tab, the user can match a word or phrase to mark (with a color) rows that match them and navigate the specific ones. The Mark feature process is represented in Figure 14.

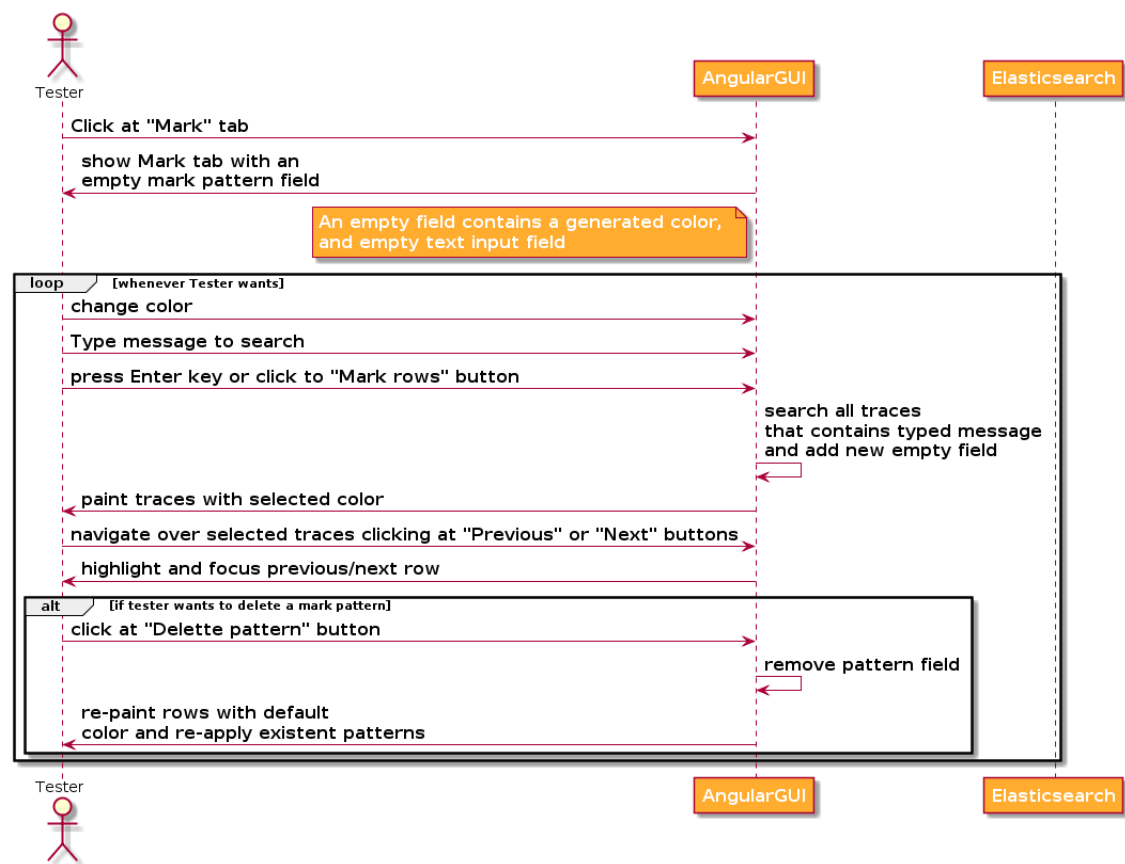


Figure 14. Mark logs in LogAnalyzer

3.4.4.3. TestLink

As previously introduced, ElasTest offers an interface to visualize the data created in TestLink. It also allows users to execute a test plan and register data generated during the execution such as logs of the application or browser videos.

Figure 15 shows the process of executing a test plan. In this diagram it is assumed that the user has already created all the necessary data in TestLink and has synchronized them in ElasTest.

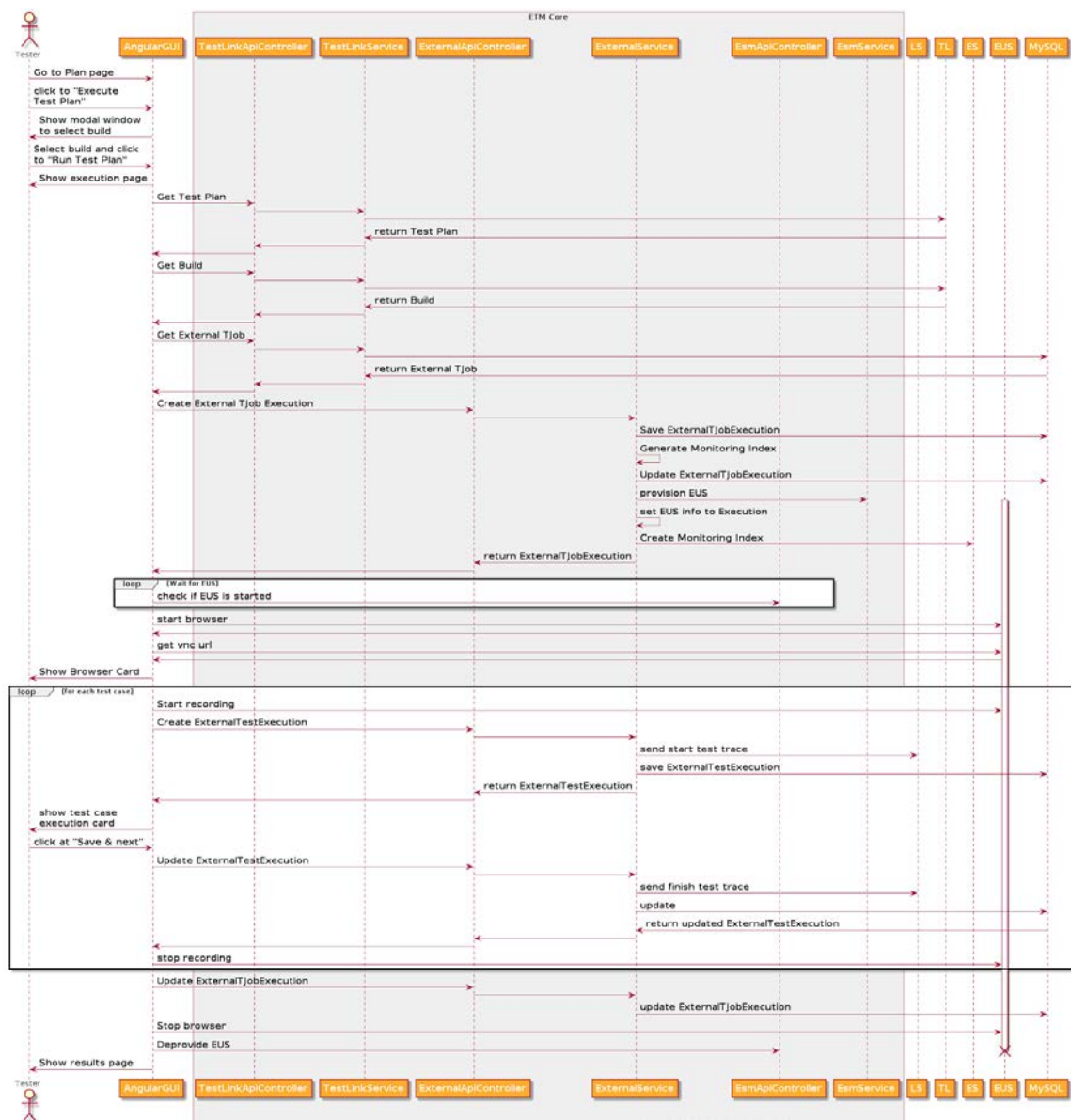


Figure 15. Test Plan Execution

3.5. Code links

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

The development of ETM is being carried in the open using the GitHub repository:

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

3.5.1. Validation

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 of TJobs execution coordinating the rest of ElasTest components.
- An extensive number of unit, integration and end to end tests have been implemented and are executed in the continuous integration system. These tests evidence the features implemented in ETM are behaving as expected and regressions are detected quickly.
- ElasTest platform (coordinated by ETM) is being used to implement and execute end to end tests of Kurento¹⁰, an open source WebRTC platform used to implement videoconference web applications.

3.5.2. Discussion

ETM is now mature enough to be used in real projects. This new phase is very interesting for ElasTest project because allows to gather feedback from real users. In the following months, new features will be designed with the collaboration of real users using ElasTest. In the first experiences with ElasTest, one important issue have been detected: It needs very important computational resources to be executed due to its microservices architecture. We are right now designing a new “reduced footprint version” of ElasTest. This version, will allow users to try ElasTest in the development machine and, if fit their needs, the full fledged version can be installed in the appropriate dedicated servers.

3.6. Research results and plans

ETM is the main entry point of all ElasTest features. In the first project phase (M18, June 2018), ETM development have been focused in providing the glue code to coordinate the rest of ElasTest components in a cohesive way. One of the main challenges was to design the technical procedures to host third party components (Test Engines, TE, and Test Support Services, TSS) into ElasTest. That integration was designed with different aspects like GUI integration, communication, artifact downloading, lifecycle management, etc.

Other important feature provided by ETM is the gathering, storing and analyzing of logs and metrics generated during TJob execution. To implement that, some components of the Elastic stack have been used (ElasticSearch, Logstash and Beats agents). The innovation here comes, again, from the integration of these generic tools with the ElasTest components and with the docker technology used to execute tests and internal SUTs.

This technical foundation allows ElasTest to include easily new third party components with ease and all types of information can be associated to test executions. Based on that, in the second phase of the project (from M18 to M36) new research areas will be explored. The main area will be the automatic analysis of the information gathered during the execution. For example, in case of regression, comparing logs and metrics

¹⁰ <http://www.kurento.org/>

obtained from failed tests with the information of the same tests when succeeded. Another research line of this area will be 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.

4. ElasTest orchestration engine

4.1. Introduction

The concept of test orchestration is one of the three main principles of the project and it is specified in the ElasTest Description of Action (DoA) document [1]:

*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 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. 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. To this aim, a test orchestration notation should be defined. The idea is that testers define the different TJobs (edges) and checkpoints (vertices).
2. Test augmentation. This concept consists on introducing new TJobs to the original one 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.

This section is devoted to report the advances on the design, implementation, and validation of the concept of test orchestration within the ElasTest platform at milestone M18 of the project lifecycle (i.e. June 2018). The rest of this section is structured as follows. Section 4.3 derives a comprehensive snapshot of existing work in related areas to the concept of test orchestration presented in this deliverable. Next section 4.4 presents a detailed description of the design proposal to the concept of test orchestration in ElasTest. Afterwards, section 4.5 describes the current status of the implementation in the ElasTest Orchestration Engine (EOE) component. Then, section

4.5.1 summarizes a case study carried out as experimental validation of the current status of the implementation. Finally, and due to the fact that this work is on progress at the time of this writing, section 4.5.2 provides a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis aimed to drive the progress of this task in the final part of the project lifecycle.

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 some kind of 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 create orchestration topology so that users can define a TiL by aggregating different TJobs
EOE DSL parser	EOE is able to parse Jenkins notation
EOE communication manager	EOE is able to support data-driven orchestration approach
EOE proxy	EOE intercept requests from 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-drive approach)

Table 2. Orchestrator requirements

4.3. Baseline concepts and technologies

The concept of “test orchestration” as it is understood in the context of the ElasTest project is completely novel in the current literature. Nevertheless, there are similar approaches that deserves to be reviewed before the actual design and implementation of our ideas in ElasTest. This section provides a summary in three closely related areas,

namely: i) test composition, i.e., existing approaches to combine tests; ii) test parallelization, i.e., run test cases in parallel; and iii) orchestration languages, i.e., exiting notations to describe processes, pipelines or workflows.

4.3.1. Test composition

The concept of test composition with the aim of increasing testing effectiveness while reducing the overall costs and effort has already been addressed in the literature. For instance, [4] and [5] are based on letting developers create elementary test cases involving simple predicates (e.g., “insert authentication pin”, “user is authenticated”, “user is blocked”). Then, the testing system composes the execution of these test cases for deducing the validity of logical formulae, which are also provided by the tester (e.g., “if after inserting authentication pin, authentication fails three times, user should be blocked”). This type of approach enables testers to reduce test code to the test cases and the formulae. However, there are not well-established methodologies on how to generate such cases and a strong theoretical background (e.g., temporal logic) is requested from developers to do it. In addition, the computational complexity may be prohibitive for large systems where the number of cases may be huge. Due to this, compositional testing has traditionally only been used for testing small software systems.

Combinatorial testing [6] aims at reducing the testing complexity and costs through an approach involving: i) modeling the SUT as a set of input factors; ii) generating a sample of the possible combinations of factors and values; and iii) creating and executing test inputs corresponding to that sample. Although combinatorial testing is being applied in relevant application domains [7] it still has relevant limitations preventing its seamless use in the testing of large software systems. Notably, it does not provide any notion of composition or sequencing of tests, and the problem of evaluating combinatorial explosions of factors in terms of testing cost or time is only recently investigated, e.g., by Demiroz and Yilmaz [8]. However, cloud resources are leveraged to perform combinatorial tests execution in parallel and identify faulty interactions through concurrent test algebra execution and analysis [9].

4.3.2. Test parallelization

Modern software codebases contain lots of individual test cases. The execution of these test suites takes relevant amounts of time, and as a result, development and release procedures tend to be time-consuming. In order to solve this issue, test parallelization has been proposed as a solution. Recently, Candido et al. [10] conducted an empirical survey on the impact of test suite parallelization in open source projects. The authors reported that only 19.1% of the projects analyzed use parallelization, being the major deterrent to its adoption the resistance concerning concurrency issues.

Existing approaches on test parallelization assume, either implicitly or explicitly, independence among tests being executed. This assumption is not always true in practice, since test executions in parallel can produce non-deterministic outcomes.

Zhang et al. [11] investigate the existence of dependent tests in 5 popular open source projects, finding a total of 96 dependent tests, 95 of which would result in a false negative when executed out of order.

Additional current research efforts on test parallelization are focused on test dependency. Gambi et al. [12] present Cloud Unit Testing (CUT), a tool for automatically executing unit tests in distributed execution environments. This work is continued in PRADET, another tool for detecting problematic dependencies in a reasonable amount of time for projects with thousands of tests [13].

4.3.3. Orchestration languages

Regarding orchestration languages (not strictly related to testing), we can find different approaches. First, we could use TOSCA¹¹ (Topology and Orchestration Specification for Cloud Applications). TOSCA is an OASIS (Organization for the Advancement of Structured Information Standards) language to describe a topology of cloud-based web services, their components, relationships, and the processes that manage them. Its first version is based on XML. Moreover, TOSCA implements a profile based on YAML. This profile has been adopted by several solutions, such as in:

- Cloudify¹² is an open source software cloud orchestration product. It implements DSL configuration files called blueprints which define the application's configurations, services and their dependencies. The Cloudify blueprint files describe the execution plans for the lifecycle of the application for installing, starting, terminating, orchestrating and monitoring the application stack. Cloudify also supports configuration management tools like Chef, Puppet, or Ansible for the application deployment phase, as a method of deploying and configuring application services.
- Alien4Cloud¹³ (Application Lifecycle ENabler for Cloud) is an open source TOSCA based designer and Cloud Application Lifecycle Management Platform. At the moment of this writing, the topology definition in Alien4Cloud can be done using simple profile in YAML v1.0 and also with the Alien4Cloud 1.3 DSL.
- Ubicity¹⁴ is a Model-Driven Service Management technology aimed to simplify service management on cloud stack. Ubicity is also based on TOSCA YAML profile for describing the topology of cloud-based services.

Regarding workflow definition, a promising alternative is the Common Workflow Language¹⁵ (CWL), which is a specification for describing analysis workflows and tools in a way that makes them portable and scalable across a variety of software and hardware environments. CWL documents are written in JSON or YAML. CWL documents are made

¹¹ <https://www.oasis-open.org/committees/tosca/>

¹² <http://cloudify.co/>

¹³ <https://alien4cloud.github.io/>

¹⁴ <https://ubicity.com/>

¹⁵ <http://www.commonwl.org/>

up of different parts to define the workflow: metadata, environment, input and output parameters, and steps. This structure could fit with our rich notion of test orchestration. Nevertheless, at the moment of this writing, CWL does not allow advanced workflow steps, such as loops, conditional, or parallel tasks. Similar features are planned in the CWL backlog for next future releases.

Another relevant language to describe cloud infrastructures is AWS CloudFormation¹⁶. It is based on JSON and provides a common language to describe and provision all the infrastructure resources in AWS cloud environments. Moreover, the OpenStack Foundation has defined Heat¹⁷, a project which implements an orchestration engine to launch multiple composite cloud applications based on templates. The latter are conceived as text files that are readable and writable by humans, and can be checked into version control, diffed, etc.

Finally, one approaches related to our concept of test orchestration is implemented in the Jenkins pipelines. A Jenkins Pipeline¹⁸ is made up of several *steps*, and each step tells Jenkins what to do, serving as the basic building block for both declarative and scripted pipeline syntax. A Jenkins Pipeline is written using a Domain Specific Language (DSL) syntax based on Groovy [14]. Typically, the definition of a Jenkins Pipeline is written into a text file (called *Jenkinsfile*) implementing the test workflow, including checking out the project's source control, executing tests, reporting, deploying, etc.

4.4. Component architecture

In ElasTest, test orchestration is understood as the interconnection of different TJobs expressed as a graph. The precise form of the graph (i.e., first one TJob, then this other one) is specified somehow by the tester. We propose two different types of test orchestration, which we refer to as *verdict-driven* and *data-driven*.

Figure 16 depicts the ***verdict-driven*** approach. This notion of orchestration does not assume any constraints or model availability neither on the TJobs nor in the execution topology nor in the SUT. Therefore, tests are seen like black boxes (Figure 16a). Internally, TJob exercise the SUT with some custom logic and assertions, and as a result provide a verdict (test passed or test failed).

TJobs are managed inside ElasTest (Figure 16b). We propose a custom notation (see next section for implementation details) to select some of these TJobs, ordering the execution as a graph. Moreover, we introduce conditional paths based on the TJob verdict (i.e., passed or failed) about previous verdict in the graph (Figure 16c). Finally, as an advanced feature of this mode, we can also parallelize the execution of a number of tests, as depicted in Figure 16d. Again, we can use the verdicts of the parallel TJobs to feed a conditional, using logic operators (*OR*, *AND*, etc.) to create richer conditions.

¹⁶ <https://aws.amazon.com/cloudformation/>

¹⁷ <https://wiki.openstack.org/wiki/Heat>

¹⁸ <https://jenkins.io/doc/book/pipeline/>

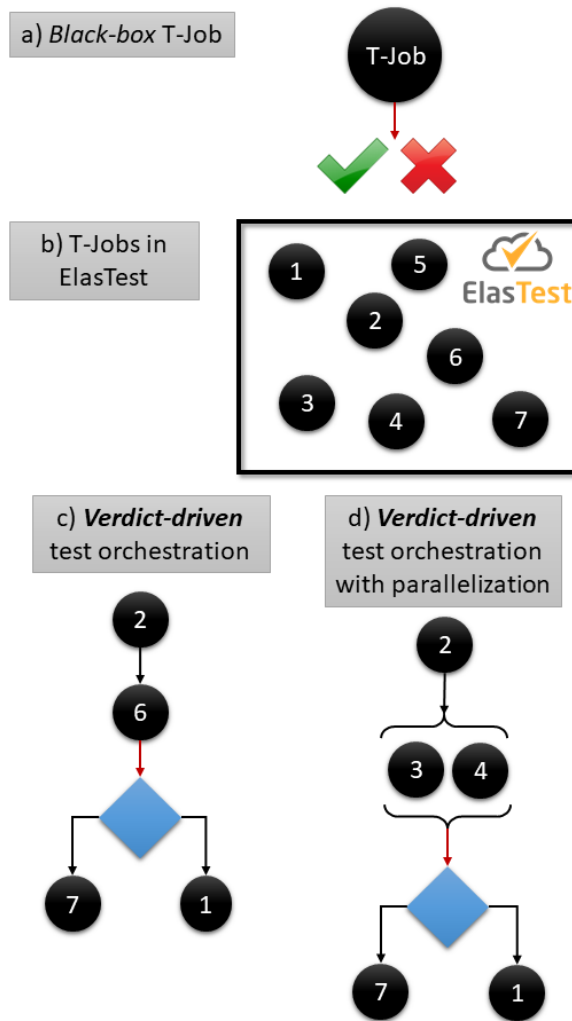


Figure 16. Verdict-driven test orchestration

Then, Figure 17 depicts the second approach, called **data-driven**. This approach is more advanced in the sense that TJobs can be interconnected using its test data (input) and the outcome (output). Therefore, a TJob is modeled as a set of input data which is incoming to the TJob, and as a result of the execution of the specific test's logic, some output data is generated (in addition to the usual test verdict, i.e., pass or fail). This concept is shown in Figure 17a, where the test is colored as green to differentiate to the black-box TJobs, used in the previous approach and represented as black colored circles. Both types of TJob can coexist inside the same ElasTest instance (Figure 17b).

With this schema in mind, the test orchestration is richer in several ways. First, the output data of each TJob can be used to feed the next TJob in the resulting graph. This is described in Figure 17c. Notice that the output data of each stage is used to feed the input of successive TJobs. Moreover, the output data can be used to control the workflow in conditional statements. In other words, not only the TJob verdict can be used to create logic conditions in the workflow, but richer operator conditions can be employed by comparing the test output with custom oracles. Moreover, constraints can

be specified to the input or output data within the test, adding extra assertions to the TJob. Finally, tests can be parallelized in this approach as well as depicted in Figure 17d.

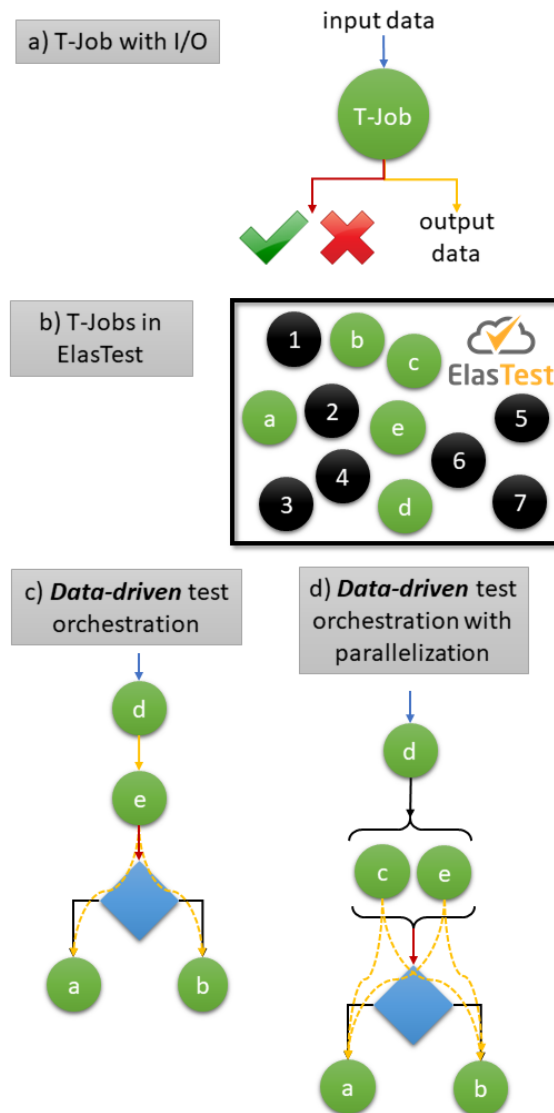


Figure 17. Data-driven test orchestration

Both approaches or orchestration (verdict-driven and data-driven) are supported by a component called ElasTest Orchestration Engine (EOE), following the ElasTest naming conventions. This component is an individual microservice and lives together with the rest of the ElasTest components. As usual, EOE is deployed as a Docker container within ElasTest. The structure and relationship with other components within ElasTest is illustrated in Figure 18 and it is explained in the next paragraphs.

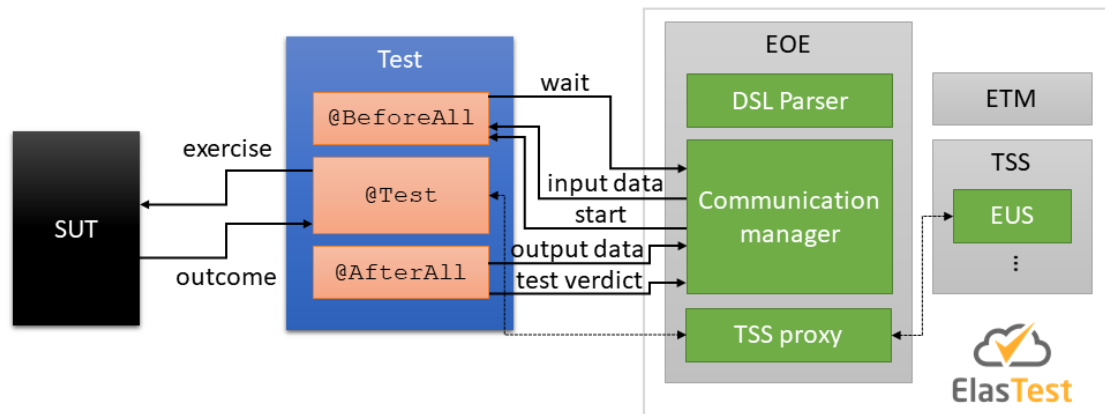


Figure 18. EOE schema

EOE is in charge of handling test orchestrations, both verdict and data-driven within ElasTest. To that aim, EOE uses as input a DSL orchestration language. As a result, EOE is aware of the number of tests to be executed and its relationships in terms of conditional paths and test data (in the case of data-driven). After parsing the DSL notation, EOE performs in a different way for verdict and data-driven orchestration.

Regarding verdict-driven orchestration, EOE basically starts TJobs in sequence in synchronous fashion. That means that it starts the first test, wait until it finishes, and then the next one. EOE is also capable of executing tests in parallel if required.

Regarding data-driven orchestration, EOE works in a more complex fashion. In this case, TJobs can be composable, and for that reason, test executed inside the TJobs need to be created beforehand following some guidelines, namely:

- Just before the actual test starts, the test sends a message to EOE asking for permission to execute the test logic. In other words, the test is paused until EOE gives the grant to be started. At this point, EOE also injects the input data in the test.
- Just before the test instance is disposed, the test sends a message to EOE informing the output data together with the test result.

The idea is that EOE starts all TJobs at the beginning of the execution. This way every test is able to resolve its dependencies, pausing the execution just before the actual test. After that, EOE sends the proper signal to start the test execution in the proper order (established in the DSL workflow). Before this signal, the input data is injected in the test. If the test is intermediate, this input data will be provided by the output data of the previous test. Both output data and verdict should be sent at the end of the TJobs. This data can be used in the EOE (according to the workflow) to decide next TJob. Of course, the SUT is always the same among the different TJobs executions. The idea is that the state of the SUT is evolving from some initial condition through the different steps according to the DSL orchestration.

Moreover, EOE behaves as a proxy for ElasTest's services, called Test Support Services (TSS) in the ElasTest jargon. The idea is that EOE intercepts these calls to share sessions

between all the tests. For example, and supposing that the tests in the orchestration are using a browser provided by the ElasTest User Impersonation Service (EUS), the browser is shared between all the tests. In terms of the W3C WebDriver protocol [15], this simply implies to create a browser session at the beginning (identified uniquely by an identifier, *sessionId*), and this identifier is shared among all requests in different tests. Only in the last tests (those that end at the leaves of the graph) this session will be closed.

4.5. Code links

In order to implement the concept of orchestration as designed in previous section, first of all we need to select a strategy to define a graph of interconnected TJobs. As introduced previously, there are different alternatives for creating workflows and orchestration languages. Due to its flexibility, we leverage the DSL notation of Jenkins pipelines, both for verdict and data-driven approaches. Concretely, we have implemented a Jenkins *shared library* which exposes a simple API to orchestrate jobs. Job is the name given to single execution units in a CI server such as Jenkins, typically composed by one or several tests. The orchestration Jenkins library has been implemented in Groovy language. It is open source and available on GitHub¹⁹. It provides a high-level class called *orchestrator* which exposes the methods as described in the following table.

Method	Description
<code>runJob(String jobId)</code>	Method to run a Jenkins job given its identifier (<i>jobId</i>). The execution of the will be declared as a stage in a Jenkins pipeline. This method returns a boolean value: <i>true</i> if the execution of the job finishes correctly and <i>false</i> if fails.
<code>runJobDependingOn(boolean verdict, String job1Id, String job2Id)</code>	This method allows to run one job given a boolean value (typically a verdict from another job). This boolean value is passed in the first argument (called <i>verdict</i> in the method signature). If this value is <i>true</i> the job with identifier <i>job1Id</i> is executed. Otherwise it is executed <i>job2Id</i> .
<code>runJobsInParallel(String... jobs)</code>	This method allows to run a set of jobs in parallel. The jobs identifier are passed using a variable number of arguments (<i>varargs</i>).

Table 3. Orchestrator API

Moreover, the *orchestrator* class can be configured using the following different options. First, different exit condition for the orchestration can be selected. To that aim, a Groovy enumeration with the following options is provided, as described in Table 3.

¹⁹ <https://github.com/elastest/elastest-orchestration-engine/>

Method	Description
EXIT_AT_END	The orchestration finishes at the end (option by default). This means that even though an intermediate job fails, the orchestration continues until the end of the graph.
EXIT_ON_FAIL	The orchestration finishes when any of the TJobs fail.
EXIT_ON_PARALLEL_FAILURE	The orchestration finishes when any a set of parallel TJobs fail.

Table 4. Orchestrator exit condition alternatives

Finally, the condition used to give a verdict about parallel jobs can be also configured. There are two options:

Method	Description
AND	Using this option, the verdict of a set of jobs executed in parallel is <i>true</i> only if all the jobs finish correctly. This is the default option.
OR	Using this option, the verdict of a set of jobs executed in parallel is <i>true</i> when at least one of the jobs finishes correctly.

Table 5. Orchestrator parallel jobs verdict conditions

An example of orchestration notation using the orchestrator Jenkins library is shown in the following listing. In this example we can see how the library is configured at the beginning. After that, the graph of jobs is declared. A job identified as *myjob1* is executed first place. According to the next sentence, if the verdict of the execution of this job is success, then *myjob2* is executed. Otherwise *myjob3* is executed. After that, a set of jobs is executed in parallel: *myjob4* and *myjob5*. The result of this execution is computed when both jobs finished, and in this example, it will be based using the *OR* boolean operation (as configured at the beginning of the orchestration). To conclude, a manual condition is defined using the result of the previous parallel job execution.

```
@Library('OrchestrationLib') _

// Config
orchestrator.setContext(this)
orchestrator.setParallelResultStrategy(ParallelResultStrategy.OR)
orchestrator.setExitCondition(OrchestrationExitCondition.EXIT_ON_FAIL)

// Graph
def result1 = orchestrator.runJob('myjob1')
orchestrator.runJobDependingOn(result1, 'myjob2', 'myjob3')
def result3 = orchestrator.runJobsInParallel('myjob4', 'myjob5')

if (result3) {
    orchestrator.runJob('myjob6')
    orchestrator.runJob('myjob7')
}
```

```
else {  
    orchestrator.runJob('myjob8')  
}
```

Snippet 1. Test orchestration example

4.5.1. Validation

In order to carry out an initial experimental validation of the presented approach, we have carried out a case study using a real application as target. Concretely, we use an application called *Full Teaching* application, which is educational web platform based on OpenVidu²⁰, an open source videoconferencing framework based on WebRTC.

Full Teaching is assessed using a complete test suite implemented in JUnit 4 with different types of tests, including unit, integration, and end-to-end. At the time of this writing, the total number of tests in *Full Teaching* is 87. This large test suite is good news for the *Full Teaching* team in terms of coverage and level of confidence to avoid regressions in the codebase. On the other side, it has a relevant side-effect which impacts directly to the agility of the development process. Due to the fact all tests are executed in the Jenkins server supporting the CI process, developers need to wait until one patch is merged in the codebase.

To avoid this problem, our orchestration library has been used. One of the benefits of using the ElasTest orchestrator library as a Jenkins DSL pipeline is that it can also be used outside ElasTest, directly in a Jenkins instance. In this example, and as shown in Figure 19, a Jenkins pipeline implementing an orchestration has been created. In this orchestration, a group of tests has been selected. A *smoke test* is going to be the first one. A smoke test case is the first to be run by testers before accepting a build for further testing. Failure of a smoke test case will mean that the software build is refused. The name of smoke testing derives electrical system testing, whereby the first test was to switch on and see if it smoked. This type of tests is done for accepting a build for further testing. A failure of this test will mean that the software build is refused, due to the fact that the orchestration has been configured using the *EXIT_ON_FAIL* option. After that, a group of relevant functional tests has been selected. These tests, executed as Jenkins jobs, is executed in parallel using the method *runJobsInParallel* of the orchestrator library. To rest of the initial tests of the *Full Teaching* test suite is executed using another job configured using a nightly job.

²⁰ <http://openvidu.io/>

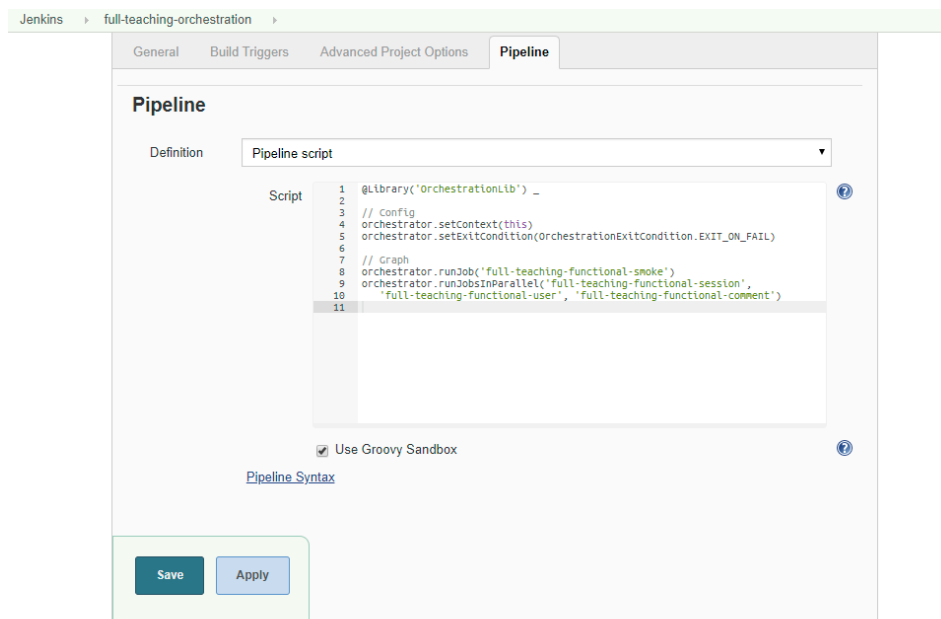


Figure 19. Using ElasTest orchestration Jenkins library

As a result, the orchestrated job shows a relevant reduction of time to be executed compared to the complete test suite. The proper selection of the smoke test together with critical functional test cases allows to the *Full Teaching* team to have a good level of confidence to merge patches in the development branch in a short amount of time. As can be seen in Figure 20, all the orchestrated test takes less than 2 minutes to be completed. In addition, if the smoke test fails at the beginning, no further tests are executed and the job is declared as failed.

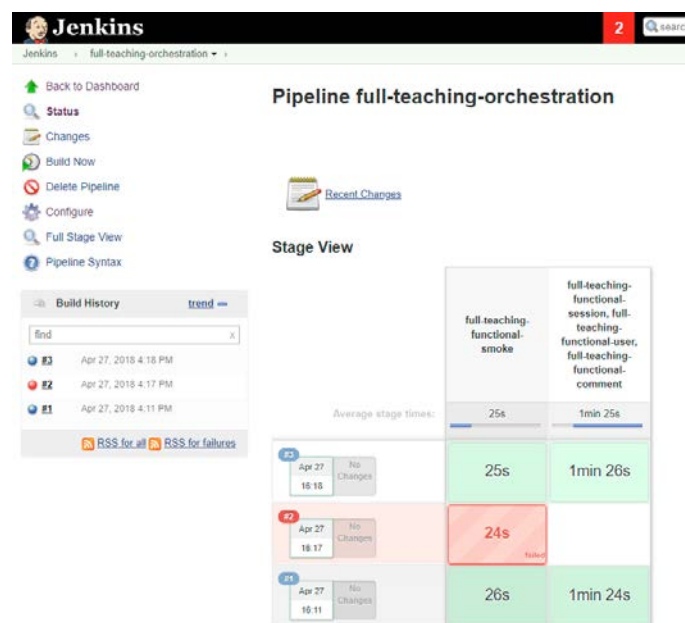


Figure 20. Execution of the orchestration in Full Teaching application

4.5.2. Discussion

In order to analyze the contributions of this work, this section presents a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis of the current proposal on test orchestration.

- Strengths:
 - o Our concept of orchestration is aligned with current trends in software testing research, at least in the parallelization domain.
 - o To implement verdict-driven orchestration, existing test codebases can be reused for selecting and parallelizing tests (TJobs in ElasTest).
- Weaknesses:
 - o To implement data-driven orchestration, tests need to be implemented specifically. In other words, we cannot reuse existing codebases since tests need to be composable in terms of data input and output.
 - o In order to inject input data and extract output data in the data-driven approach, only one test is supposed to be contained in a TJob. Otherwise some extra effort need to be done to organize tests inside the same TJob.
- Opportunities:
 - o Our view of test orchestration is novel in the state of the art, and we aim to create a complete theory around this concept.
- Threats:
 - o The concept of orchestration still need to prove its value for practitioners. At the moment of this writing there is only a preliminary validation of the approach based on a single case study, but further effort in this domain is required.

4.6. 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.

In the future, it is expected to include more contributions to this list as EOE development advances and incorporate new features.

5. ElasTest cost engine

5.1. Introduction

Executing tests are not free. Public cloud resources cost money. Private cloud installations need energy, procurement and maintenance to operate. When not optimally designed, tests could cause waste of resources and incur unnecessary financial costs. It is in the interest of a test designer of a SiL to know the cost projections well in advance so that s/he can perform test optimization and prevent the bill shock which usually follows when financial aspects are ignored in the beginning.

From the DoA, the significance of the cost engine can be ascertained from these sentences:

“If ElasTest does not consider these aspects, then although the generated tests may deliver from a technical perspective, it risks of not being financially sustainable. This is significantly important for ElasTest as some test orchestration mechanisms may produce combinatorial explosions whose cost should be well known by developers before taking the decision of using them.”

Some of the key functionalities (among others) as outlined in the DoA which are covered in this deliverable are:

- model for specifying costs
- mechanisms enabling estimation of costs
- mechanisms enabling calculation of true cost of test executions

There are few additional functionalities outlined in the DoA which is not covered in this deliverable but will be included in future iterations of this report.

5.2. Features

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

Requirement	Description
Receive TJob information from ETM	ECE should be able to get the list of TJobs from the ETM
Receive TJob information from ESM	ECE should be able to get the service type cost definitions from ESM
Static Estimation of a TJob cost	ECE should be able to estimate the cost of execution of a TJob statically using the cost model definitions received from the ESM
Retrieve monitoring information	ECE should be able to query and get the actual monitored data capturing the events and resource consumption for a TJob execution
Actual calculation of the cost of execution of a TJob	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.

Extend cost model to support all ElasTest support service	ECE task-force should define the cost models for relevant ElasTest services using meaningful metrics.
---	---

Table 6: Cost Engine Requirements

5.3. Baseline concepts

Cyclops²¹ is a general-purpose accounting and billing framework which was developed in previous European projects namely Mobile Cloud Networking²² and TNOVA²³. The DoA outlines use of this framework towards facilitating the real cost calculations in ElasTest. The existing framework is microservice based itself and was primarily designed for true usage-based accounting and model-based billing supported by multiple rule engines. The requirements gathering stage in ElasTest has revealed that using a full featured framework such as Cyclops is an overkill now and a much more lightweight approach has been adopted for cost estimation and computation. The possibility to use Cyclops at a later stage if the situation mandates remains an option on the table.

For cost estimation, one requires two piece of information, the cost model and the usage model. Once these two models are available, it is possible to perform static estimation analysis for execution of tests. In the next subsection, the general cost model and usage models will be presented and explained in depth.

To keep things reasonably realistic, two pricing models have been considered:

- **pay-as-you-go**: in this model, the service provider specifies the per unit cost of use of their service by users, this could be based either on the duration of the use of an instance, or even based on the service specific metric being instrumented within the service instance and somehow getting exposed for external accounting module to use.
- **subscription**: in subscription mode, the billing is expected to be done at subscription boundaries and generally is agnostic of usage volumes up to a specified limit in the subscription.

Even though, theoretically, it is possible to use numerous pricing models such as: *time-based, volume-based, QoS based, flat-rate, Paris-metro model, priority-based, smart-market model, edge, responsive, proportional-fairness, cumulus, session-oriented, one-off and time-of-day based*, the two considered above continue to be widely popular choices in ICT domain.

5.4. Component architecture

The design of ECE is keeping in mind the interfaces offered by ETM and ESM. The cost model is tightly coupled with operational capability and TJob orchestration workflow supported in ElasTest.

²¹ Cyclops framework: <https://github.com/icclab/cyclops/>

²² Mobile cloud networking, FP7 project: <http://mobile-cloud-networking.eu/site/>

²³ TNOVA FP7 Project: <http://www.t-nova.eu/>

5.4.1. Architecture and workflows

In ElasTest, ECE is implemented as an on-demand accessible service. The TJob developers can access the engine when they wish to see the cost analysis of using one or more support services. Two options are presented to the users: Analyze & True Cost.

- **Analyze:** when chosen, this presents a static cost estimation for executing a TJob in ElasTest platform by fetching TJob definition from ETM APIs and support service cost definition and plan offerings from ESM. It presents an estimation based on the projected length of TJob execution without complex usage models in this iteration. A comprehensive usage model will be developed as part of final release that will allow ECE to perform more complex estimates for multiple TJob execution scenarios.
- **True Cost:** this option when chosen allows the TJob developer to assess the actual costs of past executions based on the measured system and process parameters by ElasTest monitoring subsystem - ETM inbuilt capabilities as well as in conjunction with EMP.

Figure 21 below highlights the key components of ECE, more details about the architecture can also be found in ElasTest D2.3 deliverable.

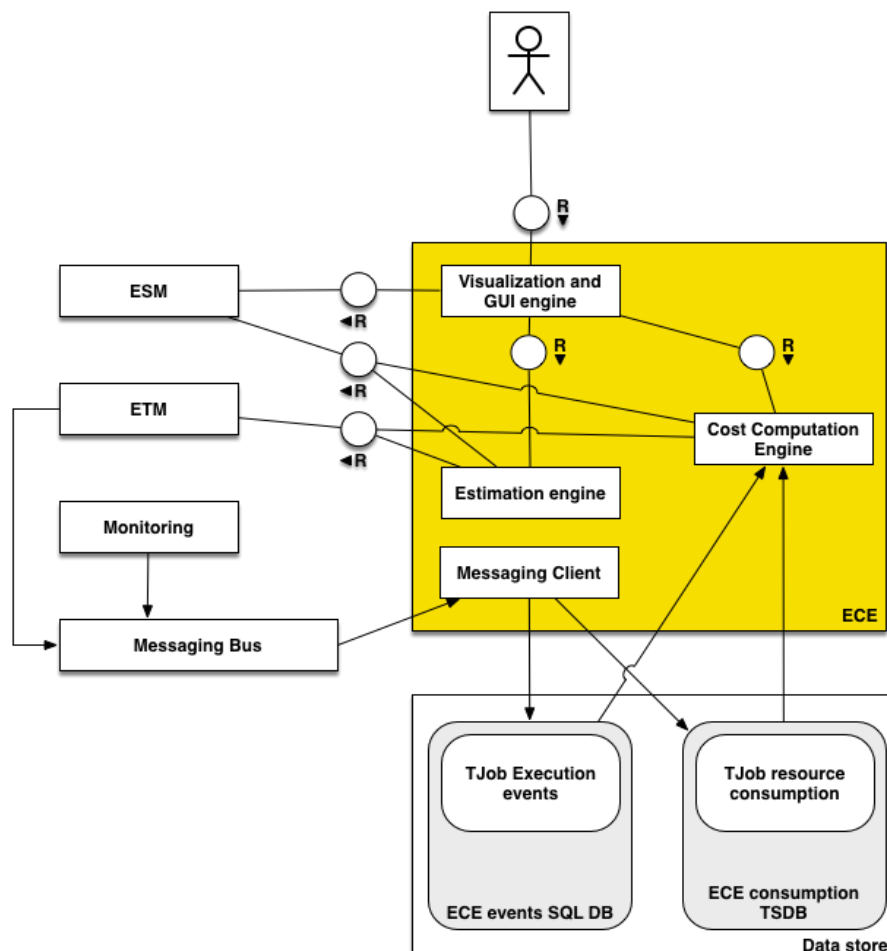


Figure 21. ECE FMC Architecture

The main components of the architecture in the diagram above are:

- Visualization and GUI engine: this component allows user interaction with the engine, it fetches the list of registered TJobs with ElasTest ETM and allows users to initiate estimate or calculation of actual cost analysis for the selected TJob
- Estimation engine: this module computes the estimated cost for running a TJob together with requested support services using the cost model defined by various services.
- Cost computation Engine: this module gets all execution run list of a particular TJob and using actual execution parameters, resource consumption metrics observed during execution, and defined cost models, it computes the true cost of running the test.
- Messaging Client: execution events (start/stop) and monitored metrics are sent to messaging bus, this module fetches the messages off the queues and persists them to relational DB or time-series data store based on the nature of the data.

The static estimation is based on the cost model (5.4.2) together with the usage model (implicit model is assumed until 0.9.0 release) that hints at projected trend at the time of TJob definition.

Figure 22 below shows the schematic that enables true cost calculation for TJob executions. Figure 23 describes the flowchart with steps involved in computation of true cost by the ECE. Whenever a TJob is executed, the start and end events are sent by the TJob orchestrator to ECE marking the begin and stop of an execution run. While the tests are under execution, the relevant metrics against the meters as defined in the cost model (5.4.2) are measured and sent via messaging infrastructure into ECE as well.

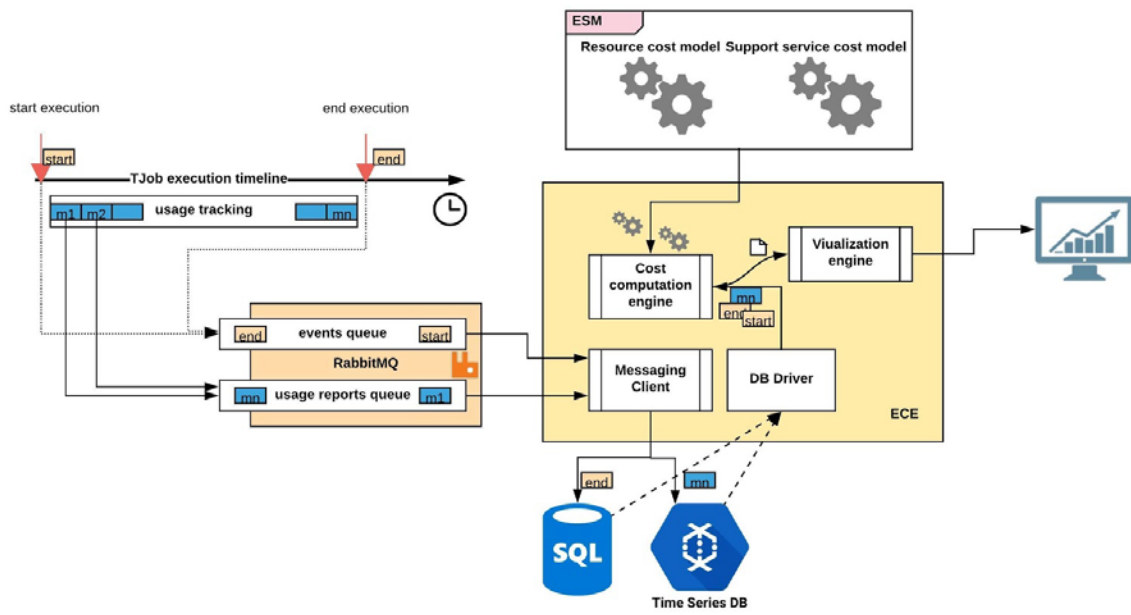


Figure 22. Schematic showing execution events and resource usage metrics flow enabling real cost estimation

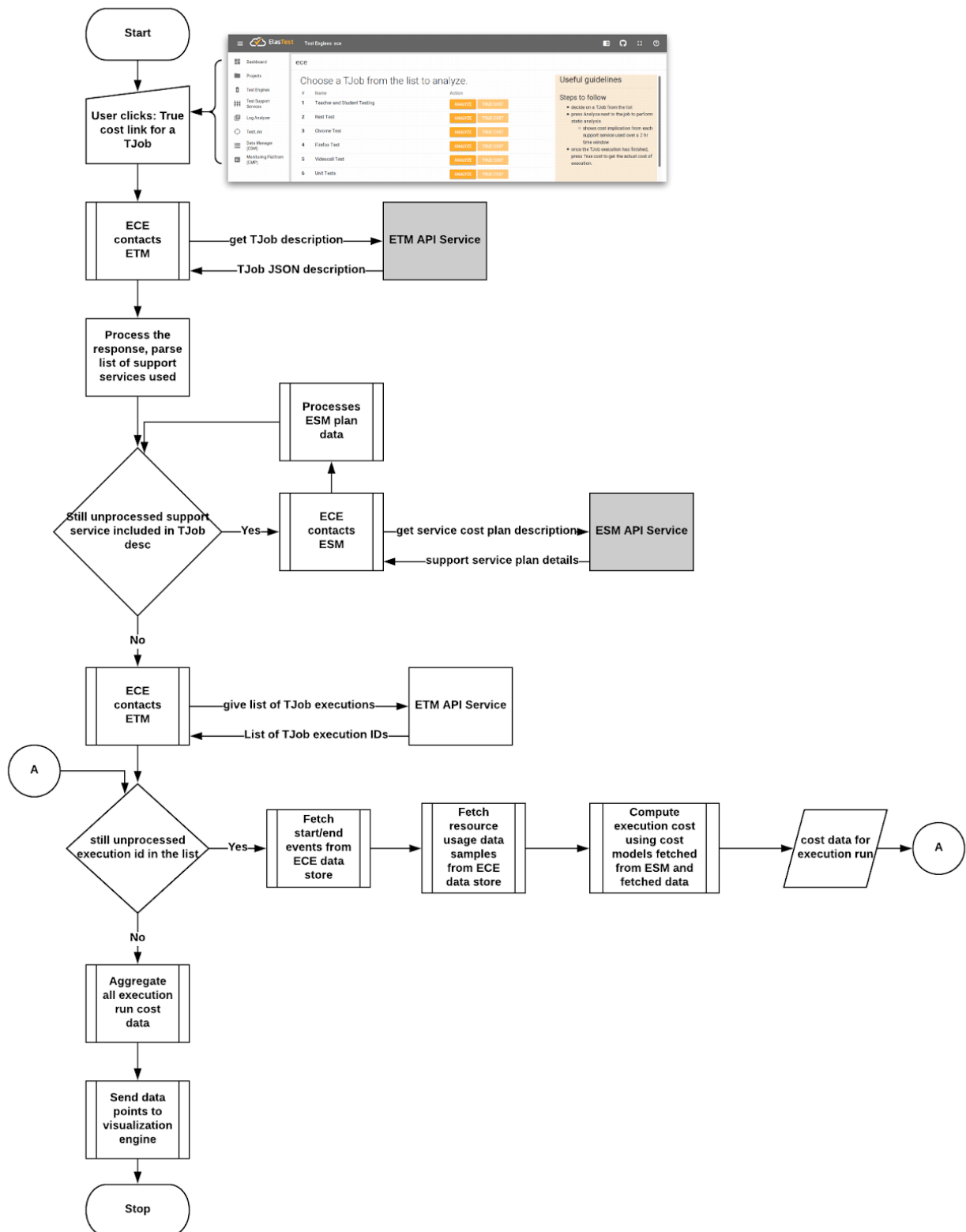


Figure 23. Flowchart showing steps in true cost computation

5.4.2. Cost model elements

The cost model has the following structure as shown in Snippet 2:

```
{
  "description": "some description",
  "currency": "eur",
  "model": "pay-as-you-go",
  "model_param": {
    "setup_cost": 0
    ...
  },
  "meter_list": [
    {
      "meter_name": "ram",
      "meter_type": "counter",
      "unit_cost": 2.5,
      "unit": "gb-hour"
    },
    ...
  ]
}
```

Snippet 2: ECE cost model

The model elements are described next.

- **description**: a string free form value describing the model purpose as human readable text
- **currency**: ISO currency value
- **model**: whether **pay-as-you-go** or **subscription** type is defined
- **model_param**: set of parameters relevant for the model type
 - **setup_cost**: one-time cost associated with starting the service for use by TJob execution run, this value can be provided irrespective of model type
 - **duration**: valid when model type is **subscription**, it denotes the duration of the subscription
 - **auto_renew**: flag telling if the **subscription** is to be auto-renewed or not
 - **pro_rata**: flag telling if the pro-rata calculation is allowed where the **subscription** is not starting on the natural invoice boundary
 - **natural_invoice**: flag telling if the invoice in **subscription** mode is to be generated at natural invoice boundary
- **meter_list**: list of meters associated with a particular service along with the cost specification for the meter
 - In case where the model is **subscription**, this list can be empty, if not empty, the cost computation based on meter list will also be considered and thus will set the model as one of mixed mode.
 - **meter_name**: the name that identifies a metric belonging to a particular meter.
 - **meter_type**: nature of the data collected by the metering process for a particular meter

- delta: actual usage of the resource since the last report was generated by the metering service.
- gauge: current value of the meter when the value was read
- cumulative: increasing meter, total volume observed since the start of measurement, actual usage is usually the difference between the latest two reported values.
- unit_cost: non-negative floating value representing the cost of consumption of the resource per unit
- unit: unit of the reported metric from the metering subsystem

A few example cost models are shown next (Table 6) for illustration purposes:

Subscription: Mixed mode	Pay-as-you-go
<pre> { "description": "cost model for torm", "currency": "eur", "model": "subscription", "model_param": { "duration": "M", "auto_renew": "Y", "pro_rata": "Y", "natural_invoice": "Y", "setup_cost": 3.5 }, "meter_list": [{ "meter_name": "tjob", "meter_type": "delta", "unit_cost": 1.25, "unit": "tesTJobs" }, { "meter_name": "log_size", "meter_type": "cumulative", "unit_cost": 5, "unit": "gb-hour" }] } </pre>	<pre> { "description": "cost model for epm", "currency": "eur", "model": "pay-as-you-go", "model_param": { "setup_cost": 0 }, "meter_list": [{ "meter_name": "ram", "meter_type": "gauge", "unit_cost": 2.5, "unit": "gb-hour" }, { "meter_name": "cpu_cycles", "meter_type": "delta", "unit_cost": 0.025, "unit": "giga-ops" }, { "meter_name": "disk", "meter_type": "delta", "unit_cost": 1, "unit": "gb-hour" }, { "meter_name": "nw_out", "meter_type": "cumulative", "unit_cost": 0.00125, "unit": "gb" }] } </pre>

Table 7: Illustrative examples of cost models

Using the specified cost model, it is possible to define a concrete cost model for any support service in ElasTest. A supporting metering algorithm is needed that provides

usage data from raw monitored values based on the cost model definition. Currently the metering functionality is planned to be part of the cost computation engine itself. In the future architecture revision, it may be separated into a standalone module within ECE.

5.5. Implementation and code links

ESM provides an implementation of the OSBA reference model whereby every service is registered with it and provides one or more service plans. Every plan consists of service offer details along with the cost parameter specification using the model described in section 4.2.2. Without this data, ECE cannot perform estimation nor compute true cost of execution of a TJob.

Furthermore, ECE is also dependent on ETM to provide the list and description of all registered TJobs, as well as providing monitored metrics for calculation of true cost post execution.

The key technology parameters that characterizes ECE are:

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

The implementation exposes RESTful interface through a class *Controller.java* that allows the ElasTest GUI ask ECE for cost estimation for a given TJob ID. The table below (Table 7) presents the list of methods exposing interfaces to the users of ECE

Method	Description
showIndex	This method fetches the list of registered TJobs and presents in a page displayed to the user with the ElasTest GUI.
showStaticAnalysis	Once a user has clicked on a TJob asking for static analysis, this method is called where the TJob data is sent as a post body.

Table 8 ECE REST interface methods supporting key GUI functions

A programmatic API is not included in the current implementation (release 0.9.0) but could be implemented if a need arises in the final release.

The ECE is packaged as a Docker container which needs key parameters to be provided as part of environment. Table 8 lists the necessary parameters.

Method	Description
ET_ETM_API	endpoint for ETM API service, this is needed to fetch the list of all TJobs, and also details definition of a specific TJob.
ET_ESM_API	endpoint for ESM API service, this is needed to get the plan definition of a particular TJob that contains the cost definition based on which the static cost analysis is done.

Table 9 ECE container necessary environment parameters

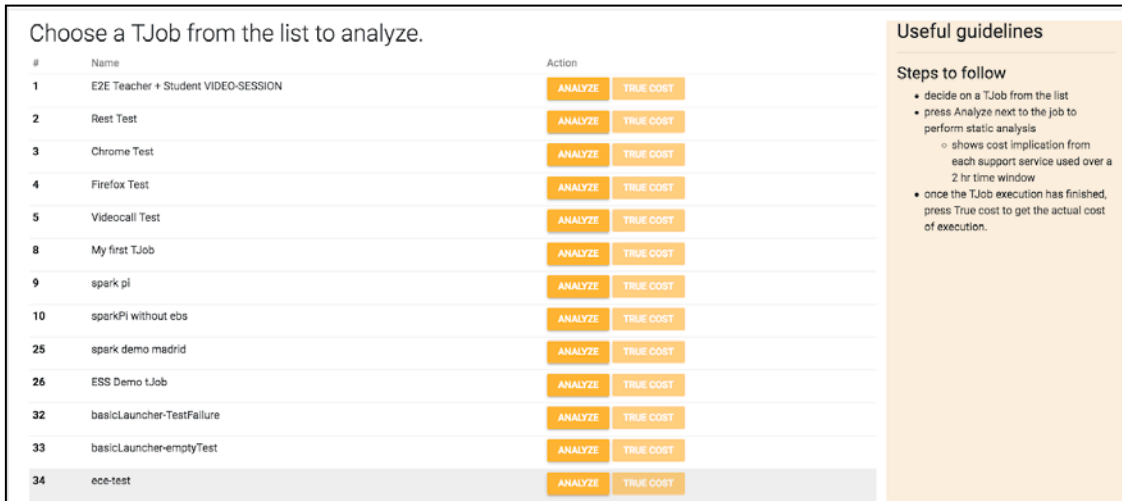
5.5.1. Validation

The current implementation is already integrated with the ElasTest dashboard and is available to users of ElasTest for cost estimation. For validation, a TJob titled ece-test has been created in ElasTest as part of ECE-Test project and the following support services have been selected as part of the TJob definition:

- EBS
- ESS
- EUS

Each of these services have specified a test cost model while registering with the ESM.

Once the cost engine is started, the user navigates to the list of TJobs and choose TJob named ece-test for static analysis (Figure 24).



Choose a TJob from the list to analyze.

#	Name	Action
1	EZE Teacher + Student VIDEO-SESSION	ANALYZE TRUE COST
2	Rest Test	ANALYZE TRUE COST
3	Chrome Test	ANALYZE TRUE COST
4	Firefox Test	ANALYZE TRUE COST
5	Videocall Test	ANALYZE TRUE COST
8	My first TJob	ANALYZE TRUE COST
9	spark pi	ANALYZE TRUE COST
10	sparkPi without ebs	ANALYZE TRUE COST
25	spark demo madrid	ANALYZE TRUE COST
26	ESS Demo TJob	ANALYZE TRUE COST
32	basicLauncher-TestFailure	ANALYZE TRUE COST
33	basicLauncher-emptyTest	ANALYZE TRUE COST
34	ece-test	ANALYZE TRUE COST

Useful guidelines

Steps to follow

- decide on a TJob from the list
- press Analyze next to the job to perform static analysis
 - shows cost implication from each support service used over a 2 hr time window
- once the TJob execution has finished, press True cost to get the actual cost of execution.

Figure 24. ECE landing page and selection of ece-test TJob as part of validation

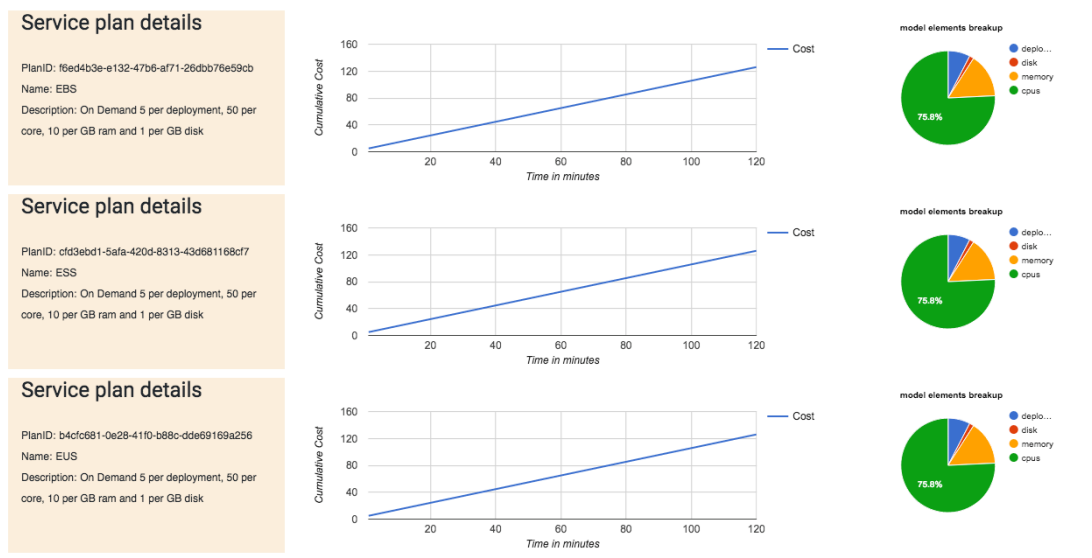
Once the Analyze button is clicked, ECE fetches related cost models of selected support services from ESM and computes the cost projection. A snippet of the engine logs shows a part of this process below (Figure 25).

```
c.s.cab.costengine.support.RESTDriver : GET response-code: 200 : URL: http://nightly.elastest.io:37006/api/tjob
ch.splab.cab.costengine.Controller : Call to TORM API: status OK
ch.splab.cab.costengine.Controller : from TORM received TJob count: 45
o.s.web.servlet.PageNotFound : No mapping found for HTTP request with URI [/favicon.ico] in DispatcherServlet with name 'dispatcherServlet'
o.s.web.servlet.PageNotFound : No mapping found for HTTP request with URI [/favicon.ico] in DispatcherServlet with name 'dispatcherServlet'
ch.splab.cab.costengine.Controller : Support services list for TJob ece-test: [{"id":"fe5e0531-b470-441f-9c69-721c2b4075f2","name":"EDS","selected":
ch.splab.cab.costengine.Controller : Static analysis request received: job-id:34 job-name:ece-test services-string: [{"id":"fe5e0531-b470-441f-9c69
ch.splab.cab.costengine.Controller : Starting analysis, services list count: 5
c.s.cab.costengine.support.RESTDriver : GET response-code: 200 : URL: http://nightly.elastest.io:37005/v2/catalog
ch.splab.cab.costengine.Controller : Call to ESM API: status OK
ch.splab.cab.costengine.Controller : from ESM received catalog entries: 5
ch.splab.cab.costengine.Controller : Found an enabled service for selected TJob: EBS id: a1920b13-7d11-4ebc-a732-f86a108ea49c
ch.splab.cab.costengine.Controller : Catalog matching entry for support service: a1920b13-7d11-4ebc-a732-f86a108ea49c with short-name: EBS located.
ch.splab.cab.costengine.Controller : Proceeding to analyze cost with plan-id: f6ed4b3e-e132-47b6-af71-26dbb76e59cb
ch.splab.cab.costengine.Controller : Found ece cost model: On Demand 5 per deployment, 50 per core, 10 per GB ram and 1 per GB disk
ch.splab.cab.costengine.Controller : Fix cost element: deployment:5.0
ch.splab.cab.costengine.Controller : Var rate element: disk:1.0
ch.splab.cab.costengine.Controller : Var rate element: memory:10.0
ch.splab.cab.costengine.Controller : Var rate element: cpus:50.0
ch.splab.cab.costengine.Controller : Found an enabled service for selected TJob: ESS id: af7947d9-258b-4dd1-b1ca-17450db25ef7
```

Figure 25: ECE log sample

The cost projection result is shown to the user post computation by the engine as shown in Figure 26.

Static cost analysis for TJob: ece-test (TJob ID: 34)



① The static analysis was carried out with these assumptions: the model defines cost for each resource in price/*-hr basis. Additionally, the cost unit is assumed to be Euro.

Figure 26. Cost analysis result page

The process is validated by the fact that 3 support services were selected in the TJob definition and the resultant shows 3 projection for each selected support service. In the figure above, cost definitions incidentally in all 3 services were similar and therefore the projections looked similar over the period.

The validation of ECE-ETM integration is performed using Selenium tests which tests the starting of the engine and checking if the list of TJob page appears or not. The test is done as a periodic job in ElasTest Jenkins CI server. Figure 27 below shows the latest run snippet captured from ElasTest CI server dashboard for ece-e2e-test.

Pipeline ece-e2e-test

Full project name: elastest-cost-engine/ece-e2e-test
end 2 end test



Figure 27. ElasTest Jenkins CI server stages for ECE end-to-end integration test pipeline

5.5.2. Discussion

ECE remains under active development at the time of writing of this document. This document presents the state of work until software release 0.9.0. The cost estimation provides a key differentiator to ElasTest and adds significant value for the test developers. The cost model design while keeping ElasTest needs in mind is intended to be generic in nature which would allow ECE models to remain relevant in future projects even after ElasTest.

5.6. Research results and upcoming plans

To maximize the utility of ECE, a few modules need to be further developed: metering process, and design of a usage model to be populated by TJob at the time of its registration. Currently, cost estimation is based solely on time basis against use of support services ignoring the associated infrastructure costs. An explicit usage pattern intention declaration in addition to including an infrastructure cost model declaration will help tighten the cost estimation significantly. These remain as backlog tasks at the time of write up of this deliverable, and updated architecture as well as details on backlog items will be included in the future planned deliverable in the series.

The cost model definition along with the workflow for estimation of execution cost and true cost post execution will be analyzed statistically for accuracy of the prediction engine and the findings will be published in a reputable conference in the 2nd half of this project's duration.

6. Conclusions and future work

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

Regarding ETM, its main objectives are: i) Allow the execution of end to end tests against complex distributed applications coordinating the rest of the ElasTest components and ii) Gather, register and analyze the information generated during test execution. These two objectives have been accomplished. 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 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 know 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*. It is more complex due to the fact that tests within TJobs are supposed to be composable, meaning that the test data (input) and test outcomes (output) are imported and exported by tests. The inconvenient 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 [16]. This extension is not released at the time of this writing, although we can anticipate here 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 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. 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 [17]) applied to the output data in the data-driven orchestration approach.

Regarding ECE, cost estimation engine 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, but the complexity and challenges of accounting remains. As the 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] Falcone, Y., Fernandez, J.C., Mounier, L. and Richier, J.L., 2007. A compositional testing framework driven by partial specifications. In *Testing of Software and Communicating Systems* (pp. 107-122). Springer, Berlin, Heidelberg.
- [5] Daca, P., Henzinger, T.A., Krenn, W. and Nickovic, D., 2014, March. Compositional specifications for ioco testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on* (pp. 373-382). IEEE.
- [6] Kuhn, R., Lei, Y. and Kacker, R., 2008. Practical combinatorial testing: Beyond pairwise. *It Professional*, 10(3).
- [7] Orso, A. and Rothermel, G., 2014, May. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering* (pp. 117-132). ACM.

- [8] Demiroz, G. and Yilmaz, C., 2016. Using simulated annealing for computing cost-aware covering arrays. *Applied Soft Computing*, 49, pp.1129-1144.
- [9] Tsai, W.T. and Qi, G., 2017. Combinatorial Testing in Cloud Computing. In *Combinatorial Testing in Cloud Computing* (pp. 15-23). Springer, Singapore.
- [10] Candido, J., Melo, L. and d'Amorim, M., 2017, October. Test suite parallelization in open-source projects: a study on its usage and impact. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (pp. 838-848). IEEE Press.
- [11] Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D. and Notkin, D., 2014, July. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (pp. 385-396). ACM.
- [12] Gambi, A., Kappler, S., Lampel, J. and Zeller, A., 2017, July. Cut: Automatic unit testing in the cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 364-367). ACM.
- [13] Gambi, A., Bell, J. and Zeller, A., 2018. Practical Test Dependency Detection.
- [14] Ghosh, D., 2010. *DSLs in action*. Manning Publications Co.
- [15] Stewart, S. and Burns, D., 2012. WebDriver. *Working draft, W3C*.
- [16] B. García, *Mastering Software Testing with JUnit 5*. Packt Publishing, 2017.
- [17] Barr, E.T., Harman, M., McMin, 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.