Melodic
Big data cloud

Title:

Test Strategy and Environment

Abstract

This document presents a strategy for testing, acceptance criteria, test related products and responsibilities related to quality assurance tasks in Melodic. The document should be use as a guideline for performing all activities within the testing process. Also, it should be used in conjunction with the 'D5.10 Quality Assurance Guide' deliverable, which contains detailed descriptions on how to accomplish test related tasks and should be use as a manual for those tasks. The deliverable's intended audiences is as follows: It could be used by all participants in the project. Especially it should be use by development teams (the unit/integration tests and the bugs definition), test teams (the whole document), architects (bugs and test cases processes), use case application users (test reporting and bugs handling) and managers (management of the releases and the testing process).
In detail, this document defines the test strategy, and describes various levels of testing (e.g. unit testing, integration testing, functional testing, and non-functional testing). The next main topic covered is the environments configurations, including their purposes, followed by test process descriptions. For the test process descriptions, the document explains the process of testing, how to use it in the project, the purpose of the process, and where to find more information. Furthermore, this document describes test related products delivered within the Melodic project, including a Test Plan, Test Cases with scenarios, and a Test Report. Finally, the document covers communication and responsibilities related to quality assurance in the project. Altogether, this deliverable should represent a complete and comprehensive guide for the most important quality assurance tasks with regards to testing in the project.

Multi-cloud Execution-ware for Large-scale Optimized Data-Intensive Computing

H2020-ICT-2016-2017
Leadership in Enabling and Industrial Technologies; Information and Communication Technologies

Grant Agreement No.:
731664

Duration:
1 December 2016 -
30 November 2019

www.melodic.cloud

Deliverable reference:
D5.6

Date:
31 May 2018

Responsible partner:
7bulls

Editor(s):
Katarzyna Materka

Author(s):
Katarzyna Materka,
Michał Semczuk

Approved by:
Ernst Gunnar Gran

ISBN Number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D5.6 Test Strategy and Environment.pdf

This project has received funding from
the European Union's Horizon 2020 research
and innovation programme under grant agreement No 731664

Document	
Period Covered	M6-12
Deliverable No.	D5.06
Deliverable Title	Test Strategy and Environment
Editor(s)	Katarzyna Materka
Author(s)	Katarzyna Materka, Michał Semczuk
Reviewer(s)	Kyriakos Kritikos, Ioannis Patiniotakis
Work Package No.	5
Work Package Title	Integration and security
Lead Beneficiary	7bulls
Distribution	PU
Version	Final
Draft/Final	Final
Total No. of Pages	35

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table of Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Index of Tables

Index of Figures

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

1 Introduction

Testing is a key ingredient to the success of implementing software systems. This document is designed to address the test strategy in the Melodic project. The goals of the Test Strategy in the Melodic project are the following:

- deliver a high-quality system;
- prove that the system is fit for service;
- minimize the risk of failure in a live environment.

The scope of this document is described in more detail in section 1.1 "The Purpose and Objective", while the main assumptions as well as the intended audience are provided in section 1.2 and section 1.3, respectively. In general, the document analyses the overall testing process, the phases of testing, the types of testing and all testing activities. Together, these subjects constitutes a test strategy that defines the general approach to quality assurance of testing in the Melodic project.

The Melodic deliverable 'D5.10 Quality Assurance Guide' [1] is a manual focusing on the usage of Quality Assurance (QA) related tasks, prescribing how these tasks should be done. The usage (especially in the JIRA[1] system) of concepts and ideas included in this document are described in chapters 3 and 4 of D5.10.

The main system to support all QA related activities will be Atlassian JIRA[1], described in more detail in D5.10. JIRA has been chosen as a tool for supporting the project's QA activities after careful evaluation of many alternatives. JIRA is currently the most popular tool for ticket management, development and test process support, is feature rich, and supports plenty of additional plugins. Atlassian has offered JIRA licenses for free for the Melodic project, which represents added value. More advantages of JIRA are presented in the D5.10 deliverable. The test team will be created and led by 7bulls. It will mostly comprise members of 7bulls, but other project participants are encouraged to participate in the tests.

1.1 The purpose and Objective

1.1.1 Purpose

The purpose of the test strategy is to document a certain test approach to be used for testing in the Melodic project. This approach relies on standards, and especially on the

[1] https://www.atlassian.com/software/jira

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

ISTQB[2] standard methodology for software testing. The methodology for testing outlined in this document will be used throughout the entire project lifecycle. This document will ensure that the project can be consistent with the production of test deliverables, documentation, procedures, and execution cycles.

1.1.2 Objective

The objective of this document is to define the strategy that will be used to test the Melodic platform, which includes:

- detailing the activities required for preparing and conducting the various levels of testing;
- communicating to all parties the tasks that they need to perform and the schedule to be followed in performing these tasks;
- documenting the test products and reports;
- explaining the levels of testing that will be followed during the entire project life cycle;
- defining the roles and responsibilities of all participants involved in the testing process in the Melodic project
- outlining at a high level the issue tracking procedures; and
- outlining at a high level how changes in the scope of the project (requirements, non-functional requirements and technical assumptions) and in the code base/documentation will be managed.

Specific test objectives apply for particular types of testing that will be supported in the project, such as Unit and Smoke testing. Such objectives will be addressed in the respective test plans for these testing types. The project coordinator, or persons designated by the project coordinator, will be responsible for the final acceptance of the results of the tests (the test reports).

1.2 Assumptions

The project lifecycle includes the main phases of the project:

- analysis and requirements gathering,
- design,
- development,
- acceptance test and
- pilot validation/evaluation

[2] http://www.istqb.org/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Pilot validation/evaluation is the stage where the development and testing phases are jointly called as implementation. The fulfilment of the following set of assumptions, which refer to all testing activities conducted throughout the entire project life cycle, is to assure the highest possible level of quality of the system to be delivered by the project. The phase-specific assumptions and guidelines will be documented in their respective test plans, whenever appropriate.

Assumptions:

- During the implementation phase (which includes both development and acceptance test phase), regular meetings will be scheduled between all parties involved in the implementation to track progress;
- Regular status meetings will be held between all parties involved in the implementation during the Acceptance Testing phase as appropriate;
- All issues with severity one and two will receive immediate attention from all parties involved in the implementation.

1.3 Intended Audience

This document is intended for all participants of the Melodic project, and in particularly for:

- Management of the project – the most important chapters are chapter 5 'Test Process', chapter 6 'Test related products', and 7 'Communication', as these chapters are highly related and impact the management activities of the project
- Development and architecture teams – the most important chapters are chapter 3 'Levels of Testing', chapter 4 'Test Environments', and chapter 5 'Test Process', as these chapters are most related to the technical side of the project
- Use case partners – the most important chapters are chapter 5 'Test Process, chapter 6 'Testing related products', and chapter 7 'Communications', as these chapters are most related to the results of the testing and the final quality of the project.
- Test teams – the whole document is important.

1.4 Document Structure

This document comprises the following chapters:

- Chapter 2, Test Strategy – this chapter supplies the definition, purpose and content of the Test Strategy.
- Chapter 3, Levels of Testing – this chapter involves a detailed description of all levels of testing which will be used and executed in the Melodic project.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- Chapter 4, Test Environments – this chapter provides the description, configuration and purpose of all environments in the Melodic project.
- Chapter 5, Test Process – this chapter explicates the test process that will be used in the Melodic project.
- Chapter 6, Testing related products – this chapter supplies a list and definitions of all testing-related products in the project.
- Chapter 7, Communication – this chapter explains the approach for communication in the QA related area of the project, the responsibilities of each participant involved in the testing activities and how information sharing will be performed. We would like to highlight section 7.3, which supplies a Responsibility-Acceptance-Consult-Information (RACI) matrix for all QA related activities.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2 Test Strategy

A test strategy is a collection of methods, phases, procedures, rules, techniques, tools, documentation and management of the test process (described in chapter 5 'Test Process') for the test team(s). It consists of a number of components, specifying:

- how the project test scope can be broken down into different stages;
- what are the tasks to be carried out, at what stage, and by whom;
- what deliverables are expected;
- test management and control;
- what tools to be used for each type of testing.

A more detailed description of the most important testing activities during the project is presented below.

2.1 Definition of Project Testing

Project testing is focused on all testing activities within the project, starting from business analysis and test strategy preparation, through unit testing, system testing, integration testing, and user acceptance testing, till production deployment.

The main QA related activities in the project, also depicted in Figure 1, are the following:

- Preparation of the Test Strategy and QA Guidelines as a base for all QA related activities in the project - this maps to the analysis phase
- Preparation of a Test Plan and Test Cases for functional and non–functional testing - this maps to the development phase
- Execution of Smoke Tests (a subset of the test cases, whose purpose is to validate if the system is ready for the Acceptance tests), Functional Tests and Non-functional Tests for each version of the software delivered per Melodic Release. There could be many versions of software due to the bug fixing process during the Acceptance Testing phase.
- After each Melodic release, the Summary Test report will be delivered, which will contain the results of all the tests executed in that release. This report is still within the Acceptance Testing phase.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

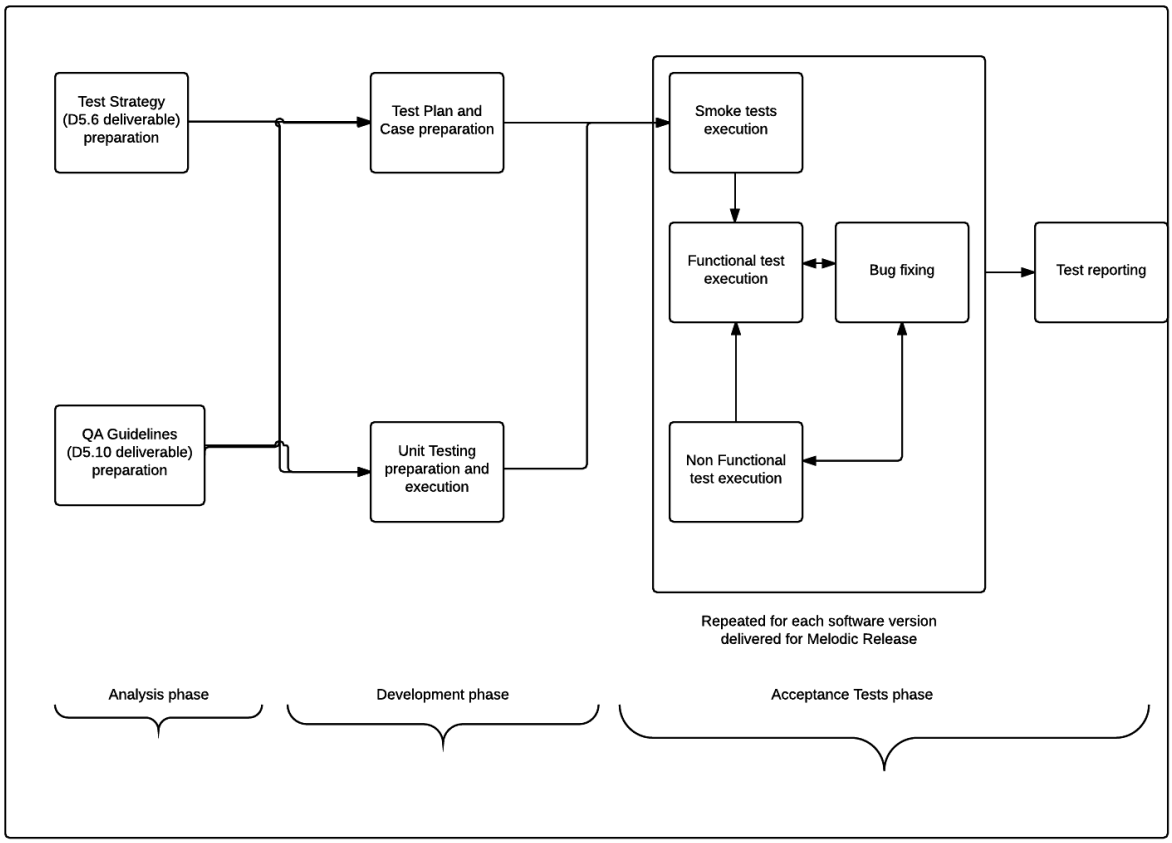

Figure 1 Diagram of the QA related activities

In Figure 1, the testing activities are shown in the rectangles while arrows between rectangles represent the dependencies between the respective activities.
Based on the results of the Analysis phase, from the QA point of view, the following deliverables have been created:

- D5.6 "Test Strategy and Environment" – this deliverable
- D5.10 "Quality Assurance Guide" [1] – the deliverable correlated to this one (see Chapter 1 - Introduction)

Based on the results of the Development phase, from the QA point of view, the following artefacts will be generated:

- Test Plan and Test Cases
- Unit and integration tests delivered by development teams

Based on the results of the Acceptance Test phase, the following deliverables will be produced:

- D5.7 "Integration release and initial environment" – for the first release of Melodic
- D5.8 "Platform prototype release" – for the second release of Melodic
- D5.9 "Melodic final release" – for the third release of Melodic.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

2.2 Planning of Test Activities

Test activities occur throughout the implementation phase. Accordingly, throughout the entire project duration, 7bulls will ensure that a test team works together with the development teams of all participants, resulting in an improvement of test documentation, test cases and code. This section highlights the most important testing activities within the overall implementation phase of the project.

2.2.1 Development

During the Development phase, Unit and System testing is conducted by the respective development teams. The 7bulls testers will work with developers to assist in the creation and review of the data used for unit and system testing where applicable. Unit testing is generally the sole responsibility of the development team; the test team only gives input as and when required. The test team input will be delivered in form of recommendations about what should be added/improved in unit or integration tests. Moreover, the test team could consult unit or integration tests creation. The 7bulls test team will also start preparations for the execution of the Functional testing. The preparation includes the creation of Test cases, scenarios and test data.

2.2.2 Acceptance Test

The Acceptance Test phase will verify that the correct functionality has been delivered. Acceptance tests will be executed by the 7bulls test team, and results will be accepted by the project coordinator or persons designated by him.

2.2.3 Scope of Testing

The objectives of testing are:

- Verification of the interaction between system components;
- Verification that all requirements as defined in the Analysis and Design phases in the project lifecycle have been correctly implemented;
- Identification of and guarantee that bugs are well addressed prior to the deployment of the software.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3 Levels of Testing

This section describes various types of tests, which will be executed in the Melodic project. These test types are as follows:

1. Unit testing – code level testing prepared and executed by development teams, described below in the "Unit Testing" section
2. Smoke testing – subset of test cases from Functional testing, executed at the beginning of the Functional phase to ensure that the system is stable, described below in the "Smoke Testing" section
3. Functional testing – main Acceptance Tests of the system, prepared and executed by the test team, described below in the "Functional Testing" section
4. Regression testing – testing of previous features of the system (not delivered in current release), to ensure that these features still work properly, prepared and executed by the test team, described below in the "Regression Testing" section
5. Non-functional testing – testing related to non-functional requirements of the system (like performance, security and so on), prepared and executed by the test team, described below in the "Non-Functional Testing" section

Each section below supplies the definition of a particular type of test, its objectives and the tools recommended to conduct it.

3.1 Unit Testing

3.1.1 Definition

Unit Testing (UT) seeks to test the building blocks of an application, typically in isolation from the application's other units and components. A building block is defined as a single class or method of program, i.e., it is a logical program unit. Units are generally the atomic elements of an application, while components may be composed of one or more units. Unit Testing is typically executed by the developers, and involves the testing of individual classes, or small clusters of classes (a package). Its main purpose is to ensure high quality in the design and implementation of classes, checking that these behave as expected, and identifying bugs prior to integrating these pieces of the code into the rest of the system. Identifying bugs from the earliest stages of development is recognized as being the most cost-effective method by the industry, as the cost of bug identification and fixing in later stages is significantly higher. Apart from reducing costs, it also allows to fix bugs faster, thus giving time benefits. Both are essential aspects of a successful implementation realized within time and budget constraints. UT also helps to ensure that when code is

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

promoted to the next level of testing, the code is reliable and testable, i.e. it does not break at the first test instance. UT is executed in a separate development environment, without external connections.

3.1.2 Objectives

Unit Testing aims at:

- verifying branches of a method (in the code unit);
- verifying boundary conditions in the method;
- verifying combinations of state transitions for a particular object when applicable within the program, according to system specification and technical design;
- verifying that each individual class meets its responsibilities as defined in its specifications;
- verifying correct behaviour or preconditions, post conditions and invariants;
- verifying that different modules can call each other successfully, thus performing a kind of high-level inter-module testing;

This kind of testing enables obtaining clean and reliable modules that can be promoted to the next level of testing.

3.1.3 Tools

For the purpose of unit testing, the basic JUnit[3] components – JUnit for non-UI code, and HttpUnit[4] for web-UI code – should be used. These components are integrated into a continuous integration environment that includes executing unit tests during the build process. The continuous integration environment contains the deployed system and the platform for automatic system deployment, based on a given source code repository. Changes (commits) to the source code repository will trigger a new system's version deployment on the environment. In case of more complex methods or code units, or when it is impossible to test single methods or code units, the Spock[5] framework should be used. The Spock framework is a complete testing framework for unit and integration tests. It supports tests written in many languages, though mostly used with Java, Groovy and other VM related languages.

[3] http://junit.org/junit4/
[4] http://httpunit.sourceforge.net/
[5] http://spockframework.org

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.2 Smoke Testing

3.2.1 Definition

Smoke testing is a non-exhaustive software testing method, ascertaining that the most crucial functions of a program work and that the system is stable enough to be deployed on environments other than the development one. A Smoke test generally consists of a collection of tests that can be applied to a newly created or modified computer program. In smoke testing, only a few, chosen positive execution paths/processes of the system are tested (they are called Happy Day's flows). They ensure that the software is in a stable enough condition to be promoted to the test environment(s), but the goal is not to make a full test with all possible test data and cases. The preparation and execution of Smoke testing will be the responsibility of the test team.

3.2.2 Objectives

Smoke Testing aims at:

- validating code changes before the changes are checked into the larger product's official source code collection;
- verifying that the latest changes have not caused any crucial function to fail;
- verifying that the software is in a stable enough condition to be promoted to the functional test environment(s).

3.2.3 Tools

JUnit or Spock framework tests are executed by the Continuous Integration pipeline implemented on Atlassian Bamboo CI[6] during the software build. Bamboo is a Continuous Integration software delivered by Atlassian, tightly integrated with JIRA and other tools used in the Melodic project. The automation of the Smoke tests is very important, because the purpose of the smoke tests is to quickly verifying if the system is ready for the Acceptance tests.

[6] https://www.atlassian.com/software/bamboo

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

3.3 Functional Testing

3.3.1 Definition

Functional testing validates the features and operational behaviour of software to ensure that they correspond to its specifications. Functional Testing is an iterative process and will be broken down into *System Testing, End-to-End* (E2E) *Integration Testing* and *UI Testing*, where appropriate, as some elements of the system do not necessarily require all types of functional testing. For example, if there is no GUI, then UI testing is not applicable. Test results from the test cycles will be reported back to the development team, allowing them to correct any issues found for a next functional test cycle. Depending on the particular function that is being tested, different types of testing need to be employed. The scope of functional testing will be decided individually for each system module, process, feature or flow. It will be described in particular test cases. JIRA will be the tool used for problem reporting and management. Details concerning the usage of JIRA in the test process are described in the D5.10 Quality Assurance Guide [1]. In the following sub-sections, we provide definitions for the different types of functional testing.

UI Testing

Currently there is no plan to test the UI for the system. *UI testing* focuses on the ability to perform defined operations and activities via a UI, see defined data from the system and so on. It also includes the checking of security, UI usability and alignment with general UI creation standards. UI testing will be optional, according to the chosen UI development method.

System Testing

The system will be built based on feature and requirement specifications, as well as architecture design. We assume that during the system testing phase, all components of the Melodic system should be tested separately, according to earlier prepared test cases by the test team. The objective of this testing is to ensure that each component of the system works properly in isolation, and is ready to start E2E Testing.

Integration E2E Testing

During *Integration E2E Testing*, all software is installed on selected hardware, and connectivity is established with each subsystem (where possible) and external systems (e.g., Cloud Providers). Once integrated, the system is tested to ensure that it functions as

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

designed. The objective of Integration E2E Testing is to ensure that all interacting components are operating correctly together. Integration E2E Testing will test scenarios which contain critical end-to-end functionality, in conjunction with the system specification document of the Melodic project (D2.1) [2] and the architecture of Melodic (D2.2) [3], to ensure that each feature/functionality of the Melodic system can be successfully executed. The scenarios will use numerous Test Cases associated with different Use Cases to allow for testing based on predefined functional flow and end-user scenarios. The functional flow and end-user scenarios cover the following Positive and Negative testing processes:

Positive process testing:

- Main Flow: Standard flow for each interaction process (feature/functionality) of the Melodic system, across all components involved in the tested functionality.
- Alternate Flow: Variations on the main standard flow for each interaction process

Negative process testing:

- Exception Flow: This testing is to ensure all validation implemented within the system is successfully working, by submitting invalid data inputs to trigger certain validation rules.

3.3.2 Objectives

System Testing aims at:

- verifying that each component is implemented according to design;
- verifying that each component tested in isolation meets all requirements specified in the system specification.

Integration E2E Testing aims at:

- verifying that the software is correctly installed (database scripts, the docker image(s) and other components are properly created and running) and connectivity is successfully established between each subsystem and the external systems;
- verifying system features/functionality based on predefined functional flow and end-user scenarios;
- verifying that data can be received and transmitted to/from the other subsystems;
- verifying real-time processing between the subsystems, if there will be a requirement to have real-time interaction between particular components;
- verifying audit stamps, error messages, and files exchanged between the subsystems;

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- verifying bug reports, which will be available in JIRA, to ensure errors are identified;

UI Testing aims at:

- verifying that the actual results match the expected results when valid data are used;
- verifying that appropriate error or warning messages are displayed when invalid data are entered;
- verifying proper application of defined business rules;
- verifying that navigation through the system properly reflects business functions and requirements, including window-to-window, field-to-field, and use of access methods (tab keys, mouse movements);
- verifying that UI objects and characteristics, such as menus, size, position, state, and focus conform to a carefully selected standard.

3.3.3 Tools

Functional Testing will be supported using Atlassian JIRA (global test management, manual test cases, bugs handling, change management and reporting). Automated functional tests will be prepared using Spock framework or Soap UI[7]. The Soap UI is a feature-rich software for SOAP/REST interface/method testing. It is designed to test backend systems which expose APIs and due to this it is considered a suitable choice for Melodic system tests. The right tool will be chosen by the test team during the test plan preparation, based on testing needs per particular feature.

3.4 Regression Testing

3.4.1 Definition

Regression Testing is defined as the selective retesting of a software system that has been modified, in order to ensure that:

- every bug has been resolved and fixed,
- no other previously working functions have failed as a result of the reparations, and
- newly added features have not created problems to previous versions of the software

Also referred to as verification testing, Regression Testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced errors. It is a quality control measure to ensure that the

[7] www.soapui.org

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

newly modified code still complies with its specified requirements and that unmodified code has not been affected by the code modifications performed.

3.4.2 Objectives

Regression Testing aims at:

- Verifying that previously working features still work after a software change, whether that change is a bug fix, an enhancement or a new feature.

3.4.3 Tools

As much as possible, regression test cases will be automated using automation test tools. Non-automated test cases will be executed manually, in the same manner as manual functional tests. Automated regression tests will be prepared using the Spock framework[8] or Soap UI[9]. The selection of one of these two tools will be done by the test team, depending on the test case and its respective needs.

3.5 Non-Functional Testing

3.5.1 Definition

Non-functional testing includes various types of tests, such as performance tests, stress tests, failover tests and security tests. Each type of test could be applicable based on the project's needs, and the final decision about which test types to use would be taken before starting the Acceptance Testing phase.

During *Performance Testing*, pre-defined requirements on response time and other non-functional attributes are measured and evaluated. This also includes exposing the system to varying workloads to measure and assess the performance and ability of the system to continue to function properly under these different workloads.

Stress testing aims at finding errors due to low resources or resource contention. Low memory or disk space may reveal bugs in the system that are not apparent under normal conditions. Other bugs might result from the competition for shared resources, like database locks or network bandwidth. Stress testing can also be used to identify the peak workload that the system can handle.

Failover and Recovery Testing objectives map to checking if the system works properly after (even unexpected) restart, failure of one or more of its components, or other non-

[8] Better Code Hub
[9] Sonarqube

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

typical situations. We will especially focus on checking the stability of the system after unexpected situations arise, via this type of testing.
The last element of Non-functional testing is *security testing*. Security testing is a process which intends to find flaws in the security mechanisms of a system that protect data and maintain functionality as intended. During *security testing* we will focus on testing and verifying the following areas of the system to be delivered:

- security of the communication between the components
- authorization at the API/system methods level
- user authentication and authorization
- checking if the system is affected by any attack related to stack/buffer overflow (sql injection, script injection and so on)
- verifying if the data are encrypted according to the system specification (deliverable D2.1 [2]) or architecture (deliverable D2.2 [3]) documents of the Melodic project (if applicable)
- checking that the system is not vulnerable via basic penetration tests, like IP/TCP spoofing, session hijacking, DNS spoofing, and SSL man-in-the-middle attacks. The detailed scope of the test will be prepared based on the requirements and needs in the project.

3.5.2 Objectives

Non-Functional Testing aims at:

- verifying the system's behaviour and performance under normal anticipated workload;
- verifying the system's behaviour under varying workload conditions;
- identifying areas where code and database optimization can be tuned for efficiency;
- verifying the system's behaviour under stress conditions, e.g., limited resources.
- verifying security of the system
- verifying system stability under expected and unexpected situations.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4 Test Environments

4.1 Development environment (Unit Testing Environment)

Unit testing is typically executed in the same environment in which the development takes place. This implies that the unit tests are run within the development IDE (i.e., on the developer's workstation), while the web server, application server and database server can be running either in the development environment or on a remote development server(s).

4.2 Integration environment (Smoke Testing Environment)

Smoke Testing is typically executed in the same environment in which the code is built. Typically, this implies that the smoke tests are executed with a 'full' environment, with the web server(s), application server(s) and database running on remote servers. Smoke tests will be executed by the developer teams. The details of the typical integration environment is provided in Table 1.

Table 1 Integration environment

CPU	x86_64/20 cores
RAM	40 GB
HDD	0 GB
SSD	800 GB
Open stack version	VM on Aruba Cloud
Virtualization	VMware
Host OS	Linux 14.10 for chef/16.04 for Docker
Network interfaces	1Gb/s
External IP/URL address of Open Stack API	94.177.187.234

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

4.3 Acceptance environment (Functional Test Environment)

At least one separate test environment needs to be set up to allow the test team to perform the Functional Testing of the project platform. The environment is used to verify that the built software satisfies the defined requirements. This is an architecturally complete environment that uses scaled-down computing components, as the main purpose is to verify functioning and interoperation, and not validate performance characteristics. The test team will be responsible for the preparation of this environment, with characteristics as detailed by Table 2.

Table 2 Acceptance environment

CPU	x86_64/20 cores
RAM	40 GB
HDD	0 GB
SSD	800 GB
Open stack version	VM on Aruba Cloud
Virtualization	VMware
Host OS	Linux 14.10 for chef/16.04 for Docker
Network interfaces	1Gb/s
External IP/URL address of Open Stack API	94.177.187.234

4.4 Non-Functional Test Environment – optional

We propose to create a separate environment for non–functional testing, with key elements to facilitate system measuring and evaluation. This should allow for efficiency in tuning of the environment that will be taken into production. Otherwise, an existing environment will be exploited to perform this type of testing. The decision concerning the creation of an additional environment should be taken at a later stage in the project. The test team will be responsible for preparing the environment, with a hardware and network configuration as provided by Table 3.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table 3 Non-Functional environment

CPU	40 cores
RAM	80 GB
HDD	250 GB
SSD	0 GB
Open Stack version	ocata
Virtualization	kvm (in planning Xen and Docker hosts)
Host OS	CentOS 7
Network interfaces	external connection: 10GbE
External IP/URL address of Open Stack API	https://omistack.e-technik.uni-ulm.de

5 Test Process

The test process starts at the very beginning of the project life cycle. During the Analysis and Design phase, the test team will start producing a Test Plan, as this should be prepared as early as possible. A test plan contains test cases, which are described in detail in the 'D5.10 Quality Assurance Guide' deliverable [1]. The Test Plan should also contain dependencies between Test Cases (if needed), which specify which Test Cases should be executed before others. This process will have three main benefits:

- It will allow the test team to understand the system to be developed;
- It will serve as a review of the system specifications and requirements;
- It will ease solving issues, as all parties (the test team and development teams) have the same base data (test data - input parameters for test cases, necessary to execute test cases and to reproduce bugs if occurring during test case execution).

Once the Analysis and Design phase is finished, the Implementation phase will start. The development teams start building the application, and at the same time, the test team completes the Test Plan. The Test Plan is then reviewed by the relevant parties (development teams, architects, analysts and the test team) and verification takes place to ensure that all aspects of the Test Cycle have been fully covered. After Test Plan finalization, the test scenarios are designed and developed within JIRA Test Cases in the following manner:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

- Test cases for each User Story will be created in a separate JIRA test project;
- The test cases are developed and stored in the test cases tab;
- Test cases are grouped into test scenarios in JIRA;
- Progress and statistics will be based on JIRA reporting;
- The detailed processes and procedures for JIRA are defined in Quality Assurance Guide deliverable;
- Each test case will be given a priority according to the importance of the tested feature related/linked with it. Prioritization will be based on the priority of the corresponding requirements, if they are available. In case of lack of priority, the following default rules for test case prioritization will be used:
    i. *High* – A feature tested by the test case is critical for proper system functioning and has to successfully pass all tests in all possible cases, including corner and extreme cases.
    ii. *Medium* – A feature tested by the test case is important for proper system functioning and has to successfully pass all tests in typical situations. In corner and extreme situations, improper behaviour is acceptable.
    iii. *Low* – A feature tested by the test case is not critical for proper system functioning.

  The default prioritization of test cases will be done by the Test Team Leader.

5.1 Test cases and test scenario relations

This chapter supplies the definitions of *test case* and *test scenario* and explains the relations between each other.
A test scenario covers one execution flow of a use case, so for each use case there will be many test scenarios with alternative ways to handle the flow, e.g., positive scenarios, negative validations, technical error handling, and business error handling. On the other hand, a test case is a single, complete activity within a test scenario, which relates to one complete function/feature that needs to be tested.
The following relations between test scenarios and test cases exist:

- Each test scenario consists of one or many test cases;
- Test cases can be reusable as each test case could be used by many test scenarios;
- Each test scenario has its own parameters for execution (for example when testing the uploading of a camel model to CDO/MDDB, the sample camel model should be attached as a parameter for the test scenario); each test case defines the needed parameters and determines which ones should be inherited from the test scenario and which ones are defined for this test case only;
- Each test scenario and test case defines the prerequisites needed for the execution.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

5.2 Test acceptance criteria

This chapter contains those acceptance criteria which when fulfilled indicate the ability to promote the given version of a system to the next environment and stage of testing.

5.2.1 For migration to the Integration Environment

To promote system code to the Integration Environment, the criteria listed below should be fulfilled:

- A code review of a pull request should be done by at least one person – to discuss the organization of the code review process, especially code review by different participants. The responsibility for code review is on the developer.
- All ten rules of Better Code Hub[10] should be fulfilled and marked as green – This item should be discussed and confirmed with other participants.
- Preparation and execution of unit tests for delivered code. Minimum coverage of unit tests should be 40%, counted using prepared Sonarqube[11] rules. – the responsibility is on the developer. The value of minimum coverage is based on 7bulls' experience in commercial software development.

The test team has the right to review the unit testing and suggest improvements or adding more tests to the development teams.

5.2.2 For migration to the Acceptance Environment

To promote the system code to the Acceptance Environment, the criteria listed below should be fulfilled:

- The system should work properly; all components should properly interoperate – the responsibility is on the architect and developers which deliver the code for release.
- All integration tests should be prepared and executed. Minimum combining coverage of unit and integration tests should be 50%, counted using prepared Sonarqube[11] rules. The responsibility is on the developers.
- Smoke tests should be executed with positive effect – the execution of the tests is in the responsibility of the test team, while fixing discovered bugs is in the responsibility of the developers.

[10] https://bettercodehub.com
[11] https://www.sonarqube.org

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

The test team might review the integration testing and suggest improvements or adding more tests to the development teams.

5.2.3 For migration to the Final Acceptance Environment

To promote the system code to the Final Acceptance Environment, all the criteria listed below should be fulfilled. Some criteria will be checked via automatic verification, while others need a manual verification of their fulfilment.

- All test cases related to the Melodic framework with priority critical and major should be passed.
- All non-functional requirements related to performance and security should be fulfilled and tested positively.
- All test cases related to the core functional flow of the Melodic platform should be passed.
- The optimization of the deployment architecture should be verified in various scenarios.
- Evaluation and validation of delivered features by use case partners.
- The use case applications should be installed and tested; all major features of these applications should work properly.

5.3 Bugs Tracking

The purpose of this chapter is to describe at a high level the Bugs Tracking Process. The 'D5.10 Quality Assurance Guide' deliverable [1] provides low level processes and procedures targeting the implementation teams.

5.3.1 Bugs Logging Procedure

Test activities comprise of executing test scripts and comparing actual with expected results. Whenever an inconsistency is detected between actual and expected results, this inconsistency should be logged in a bug report for further follow up. The originator records his/her name and gives a description of the inconsistency. Sufficient details (covering the process followed and configuration information) should be provided for investigating the item to be able to recreate the process used when the inconsistency was first discovered. The logging is performed directly in JIRA, which is the only official channel for inconsistency reporting for this project. The reason to use only JIRA is to avoid blurry communication via mails/chats or other means, but rather adhere to one unique source of evidence.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Please note that an inconsistency is the general term used for anything that requires further investigation. Once an inconsistency has been validated to be a proper fault in the software, the term bug is used. The bug handling process is analysed in detail in D5.10.
The priority of the bug is set by the originator of the bug. The priority should only be changed by the originator, test coordinator or technical manager of the project. In particular, it is not allowed for the developer, assigned to fix the bug, to change its priority.

Overview

The JIRA bug-tracking module will be implemented, and the test team will enter any inconsistencies found (either related to the Project code or any documentation) into JIRA. All inconsistencies will be logged in JIRA; even minor or intermittent bugs that cannot be reproduced can still provide valuable information. Bug reports should contain enough details as to how the bug was detected so that an attempt can be made to reproduce the problem. A high level of detail will assist with replication of the problem, and the respective resolution of the bug.
After bugs/issues are resolved, and the (bug) resolution is released via the build cycle, testing will be conducted to verify that the bug resolution is successful and to ensure that the system functions correctly. The testing of the bugs will be based usually on already created test cases; only when there will be a need to create dedicated test cases for a particular bug, new test cases will be created.
Not all errors or bugs require a respective code fix by the software development team. Some may be caused by errors in configuring the test software to match the development or production environment. Some bugs may be resolved by documentation corrections. Others might be deferred to future releases of the software, depending of the severity/importance of the bug. There are yet other bugs that may be rejected by the development team (of course, by supplying a suitable rationale for this) if it deems them inappropriate to be called as bugs.

Bug analysis and resolution

During its entire lifecycle, a bug will change states depending on the progress being made in resolving the respective issue. Note that for the project, the number of different states is kept to a minimum, in order to keep it as straightforward as possible. The bug state enumeration, the allowed transitioning between bug states and the actual bug handling process is described in detail in the D5.10 deliverable. The combination of status and assignee, together with an extra note when required, should provide sufficient information to do a proper follow-up on a bug.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Additionally, it should be indicated that independently of its state, an inconsistency shall remain indefinitely in the bugs database in JIRA. Please note that it is essential that all circumstances, steps taken and events prior to the inconsistency occurring, are logged as detailed as possible from the outset. When all this information is complete and available, the steps will be carefully followed. In case an inconsistency is still not reproducible, all system logging and audits could be examined. At this point, if there is no indication as to what the problem might have been, the inconsistency will be returned to its originator with the status Open. It is then the responsibility of the originator to carefully monitor and repeat the steps to establish a well-documented sequence. If this is successful, the inconsistency will follow the normal flow as defined above. Otherwise, if the bug is still irreproducible after a subsequent release, the bug will be deemed Rejected. This procedure will minimize effort on fruitless "needle in a haystack" type of investigations for all parties involved. The detailed workflow for bugs handling is further described in D5.10.

6 Testing related products

The purpose of this chapter is to summarize the project's products related to the testing process and quality assurance.

6.1 Code Management

All project source code is maintained in a GIT repository on Bitbucket.

6.2 Test Management

JIRA will be used as a test management and control tool to keep track of all test cases and scenarios, their execution, and possible modifications to test cases, in order to enable tracking of what functionalities have been tested against what software version.

6.3 Quality Assurance Guidelines

The "D5.10 Quality Assurance Guide" [1] is a documentation manual with detailed tasks and quality assurance activity descriptions. This document should be used as a guideline for implementing all tasks related to testing and quality assurance in the project.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

6.4 Test Strategy

This document, the 'D5.6 Test Strategy and Environment' is the document that defines the standard test approach for project (system) implementation. This document defines test deliverables, documentation and procedures.

6.5 Test plan

The document that explains how testing of a given level will occur and what will be tested. The test plan is written to identify, document and communicate the test cases, resources, objectives, activities and goals that are required and will be achieved by implementing the plan. The test team will provide this document to cover the functional and non-functional test phase; this test plan will be an internal deliverable per iteration.

6.6 Test Scenarios

The document that outlines the different steps to be performed when testing a function or Test Case, that allows validating actual results against expected results. Each function (or Test Case) will comprise several smaller test steps, which can be combined into one or more larger scenarios. Each level of testing will have its own designated test scenarios, where some test scenarios may be derived from an earlier level of testing. For instance, Integration Testing test scenarios may take a subset of the UI Testing test scenarios, and will have their own specific test scenarios dedicated to their specific types of testing. Therefore, one test case could be used for various types of testing. For example, the same test case could be used for UI testing and Integration testing. The test scenarios are managed and stored within JIRA, although if required, reports detailing these test scenarios can be generated.

6.7 Test Summary Report

A document that will summarize the test results of the Functional Testing at the end of every release of the Melodic system. This report will detail the tests that have been executed, any outstanding bugs identified, and possible workarounds suggested for existing bugs. After executing Non-Functional Testing, which usually succeeds Functional Testing, the Test Summary Report will be extended to add the respective testing results for this testing type.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7 Communication

7.1 Test Deliverables

A first level of communication will be delivered by means of the test deliverables defined in this document. These documents are intended to give the project team members a better understanding of the processes and procedures in place, while others, like Test Summary Reports and Test Plan, created during the project's life cycle, will inform the project team members of the progress being made in the testing of the system. The Functional Test Summary Reports will be shared with all participants. Also, other testing related deliverables (like Test Plan and Test Cases) will be available for all participants. The rules for sharing testing deliverables are described in the RACI matrix (section 7.3). During the entire lifecycle of the project, the bug tracking tool (JIRA) will also be a useful source of communication between the test and development teams. This also allows the project management to extract reports from the database, thus verifying the progress of the project.

7.2 Information Sharing

A Confluence Website[12] (a content collaboration tool) will be set up as a central source of information. All participants are welcome to contribute to the body of knowledge. This is an 'organic' source of information which evolves over the project cycle. It includes:

- Set-up help – a manual explaining how to setup an environment for the system delivered by the Melodic project
- Installation help – a manual on how to install the system delivered by the Melodic project
- Best practices – best practices in various areas of the project
- Patterns – patterns (useful for development, devops, testing and so on) used in the project
- Current environment details – detailed description of each environment
- Knowledge Base – contains useful information (e.g. definitions, articles on how to handle common problems and so on) for the project
- Links to all useful resources – links to external web pages related to the project.

[12] https://confluence.atlassian.com/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

7.3 RACI Matrix

RACI matrix[13] is a way of defining responsibilities, acceptance, consulting and information flow in a project. For each task or product, respective parties are defined in the matrix with the following roles/responsibilities:

- R – Responsible: the person or team responsible for executing a task or delivering the respective product
- A – Accepting: the person who accepts the product or results of the task
- C – Consult: the person or team who could be consulted and supports the execution of the task, involved in the delivery of the results of the task, under the supervision of the person/team responsible for delivery.
- I – Inform: the person or team who should be informed about the task results and activities.

All roles are defined in accordance with the general project management of Melodic. The main roles involved in quality assurance related activities are as follows:

- Project Coordinator – the person responsible for coordinating the execution of actions and tasks between participants of the project.
- Chief Architect – the person responsible for the overall architecture of the software delivered within the project.
- Development team leaders – the persons who are the leaders of the related development work packages in the project, i.e., WP3 and WP4, which should lead the Executionware & Upperware modules development teams, respectively.
- Test team leader, who is the quality assurance leader in WP5 and responsible for all activities related to quality assurance in the project.

The test team will be created and led by 7bulls. It will mostly comprise of members of 7bulls, but other project participants are encouraged to participate in the tests. Table 4 shows the RACI matrix for all high-level QA-related tasks in the Melodic project.

[13] https://www.projectsmart.co.uk/raci-matrix.php

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

Table 4 RACI matrix and responsibilities

Id	Task name	Description	R	A	C	I
1	Test strategy preparation	Preparation of the test strategy – see chapter 6.4 Test Strategy	Katarzyna Materka	Project Coordinator	All participants	All participants
2	QA Guidelines preparation	Preparation of the QA Guidelines deliverable – see deliverable D5.10	Małgorzata Jakubczyk	Project Coordinator	All participants	All participants
3	Jira test case type and workflow configuration	Configuration of Melodic JIRA to add issue type test case and workflow for test cases – see deliverable D5.10	Małgorzata Jakubczyk	Chief architect	All participants	All participants
4	Jira bug type and workflow configuration	Configuration of Melodic JIRA to modify default bug type and default workflow for bugs – see deliverable D5.10	Katarzyna Materka	Chief architect	All participants	All participants
5	Unit test	Preparation of unit tests for delivered software – see chapter 3.1 Unit Testing	Dev team leaders	Katarzyna Materka	Test team	None
6	Integration test	Preparation of integration tests for delivered software – see section in chapter 3.3 on Integration Testing	Dev team leaders	Katarzyna Materka	Test team	None
7	Functional tests	Preparation and execution of functional test – see chapter 3.3 Functional Testing	Małgorzata Jakubczyk	Project Coordinator	Test team	All participants
8	Non-Functional tests	Preparation and execution of non-functional tests – see chapter 3.5 Non-Functional Testing	Katarzyna Materka	Project Coordinator	Test team	All participants

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

9	Test plan and test cases	Test plan and test cases preparation for each Melodic release – see chapter 6	Małgorzata Jakubczyk	Chief architect	All participants	All participants
10	Test automation	Preparation of automated functional test cases – see Tools sections in chapter 3.1 and chapter 3.3	Katarzyna Materka	Chief architect	None	None
11	Test report	Preparation of the report after each release of Melodic. See chapter 6.7 Test Summary Report	Katarzyna Materka	Project Coordinator	All participants	All participants
12	Test related meetings	Organization of test related meetings. See chapter 7 Communication	Katarzyna Materka	None	None	All participants

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

8 Summary

This 'Test Strategy and Environment' deliverable contains the strategy for testing in the Melodic project. In the document the following information has been described:

- Test strategy definition
  The description and definition of a test strategy with a brief presentation of the most important elements.
- Levels of testing
  The levels of testing in the project, starting from unit and integration tests during development phase, through smoke tests, till functional and non-functional testing during the acceptance phase.
- Environments which will be using in this project
  The environments used for testing in the 'Melodic' project, including Development Tests, Integration Tests and Acceptance Tests environments. The Development Tests environment will be used by developers, the Integration Tests environment will be used by both developers and the test team, and the Acceptance Tests environment will be used by the test team only.
- Test process
  The description of the test process is provided. The purpose of the test process is explained. Each element of the process is presented and elaborated.
- Test related products
  Test related products delivered within the project are described. The most important products are described and their purpose is presented.
- Communication and responsibilities
  In the last chapter the communication in the project is described. The tools used for communication are listed with a brief explanation on how to use them. Also, the RACI matrix is provided for the test related activities. For each activity, the responsible person or role is assigned, also the persons and roles which could be consulted, should be informed, and should accept particular task or product are presented.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664

9 References

[1] M. Jakubczyk and M. Prusiński, D5.10 "Quality Assurance Guide". Melodic deliverable

[2] Y. Verginadis, W. Żołnierowicz, P. Skrzypek, D. Seybold, K. Kritikos, S. Mazumdar, A. Schwichtenberg, F. Zahid, J. Domaschka, G. Horn, E. G. Gran, D. Baur, H. Masata and P. Góra, D2.1 "System Specification". Melodic deliverable

[3] Y. Verginadis, G. Horn, K. Kritikos, F. Zahid, D. Baur, P. Skrzypek, D. Seybold, M. Prusiński and S. Mazumdar, D2.2 "Architecture and Initial Feature Definitions". Melodic deliverable

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731664