Melodic
Big data cloud

Title:

Updates to OSS Frameworks

Abstract:
The efficient and appropriate usage of existing open-source software in Melodic needs a careful study with respect to the changes and adaptations required for their integration with Melodic. This deliverable is aimed at identifying such changes. The frameworks reviewed in this deliverable include PaaSage and Cactus/Cloudiator. The review methodology is designed based on existing functional and non-functional features in each framework and the assessment of implementation code for each framework. The functional features are studied based on the objectives of Melodic: multi-cloud deployment of applications, cloud-agnostic description of applications and infrastructures, and support for data awareness and data locality, to name the most important ones. Performance, reliability, maintainability, and serviceability are among the important non-functional features considered in our methodology. The code assessment is performed by verifying the quality of the code against best programming practices as well as quality assurance requirements. We use the Clean Code approach as the best programming practice for evaluation.

Multi-cloud Execution-ware for Large-scale Optimised 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:
D5.02

Date:
1 March 2018

Responsible partner:
7bulls

Editor(s):
Paweł Skrzypek

Author(s):
Antonia Schwichtenberg, Katarzyna Materka, Kyriakos Kritikos, Paweł Skrzypek, Michał Semczuk

Approved by:
Ernst Gunnar Gran

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D5.02 Updates to OSS Frameworks.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	M6-14
Deliverable No.	D5.02
Deliverable Title	Updates to OSS Frameworks
Editor(s)	Paweł Skrzypek
Author(s)	Antonia Schwichtenberg, Katarzyna Materka, Kyriakos Kritikos, Paweł Skrzypek, Michał Semczuk
Reviewer(s)	Daniel Seybold, Amir Taherkordi
Work Package No.	5
Work Package Title	Integration and Security
Lead Beneficiary	7bulls
Distribution	PU
Version	1.0
Draft/Final	Final
Total No. of Pages	63

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

Index of Figures

Index of Tables

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

1 Introduction

The purpose of this deliverable is to provide a review of the existing open-source software from previous research projects that will be adapted and integrated in the Melodic project. The review is based on a comprehensive evaluation of each project, and resulted in a corresponding list of changes required for integrating and reusing components from the reviewed projects in Melodic.

There are two software projects which have been reviewed for integration in Melodic: PaaSage[1] and Cactos/Cloudiator[2]. PaaSage is an open source project that provides an infrastructure-agnostic integrated platform to support model-based development, configuration, deployment and optimisation of Cross-Cloud applications. The second software, Cloudiator, enables deployment of applications in a Cloud-agnostic way to various Cloud providers. Cloudiator was developed within the PaaSage project and extended in the Cactos project. Note that the PaaSword[3] project, which was initially selected for integration in Melodic, is not covered due to licensing restrictions. Furthermore, Spark and Hadoop will act as external systems to the core Melodic platform, thus they are not evaluated in this deliverable.

The methodology used for the review in this deliverable is based on the following considerations:

- Functional requirements – review of the available features in each project
- Non-functional requirements – review of the current implementation of the non-functional features in each project
- Code assessment – assessment of the implementation code in the reviewed projects

1.1 Structure of the document

This deliverable is divided into two logical parts. The first part describes the methods employed for the review of functional and non-functional features of PaaSage and Cloudiator, and respective code assessments. In the second part, the review results are presented. Furthermore, the description of each suggested change is provided, and the mapping between changes and user stories in JIRA[4] is given. Towards the end of the document, a summary with the conclusions of the review is presented.

[1] https://paasage.ercim.eu/
[2] http://www.cactosfp7.eu/
[3] https://www.paasword.eu/
[4] 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

The detailed structure of the rest of the document is as follows:

- Section 2: List of components for review and assessment. This section contains a list of components from the underlying frameworks which have been reviewed and assessed, based on the Melodic architecture described in the Melodic deliverable D2.2 "Architecture and Initial Feature Definitions" [1].
- Section 3: Review Methodologies. The section contains a description of the methodology of review, where separate concrete methodologies for each type of requirement and for the code assessment are provided.
- Section 4: Required functional features. This section lists the functional changes needed to be implemented in each project to be integrated and adapted properly in Melodic.
- Section 5: Required non-functional features. This section lists the non-functional features needed to be implemented in components from PaaSage and Cloudiator.
- Section 6: Code assessment results. This section contains the results of the code assessment of the selected components from PaaSage and Cloudiator.
- Section 7: The components' fulfilment level. This section presents a summary of the fulfilment level per category for each selected component.
- Section 8: Summary. This section includes a summary and the conclusion of this deliverable, with general recommendations for the integration and adaptation of the underlying frameworks.

2 List of components for review and assessment

The following components have been reviewed and selected for use in Melodic according to the content of deliverable D2.1 "System Specification" [2]. The components are also mapped to the corresponding Melodic objectives. The components, as listed in Table 1, have been reviewed against both the functional and non-functional requirements, while an assessment of their code has been performed.

As indicated in D2.1 [2], the most important objectives of the Melodic system are:

- The ability to perform Cross-Cloud deployment of applications
- The ability to describe an application and its infrastructure in a Cloud-agnostic way
- Deployment optimality (according to, e.g., business constraints and utility)
- Support for data awareness and data locality
- Support for extending modelling vocabulary through a metadata schema
- The ability to support real use case applications for commercial and non-commercial organisations
- Support for Big Data processing frameworks

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

The "Support for data awareness and data locality" and "Support for extending modelling vocabulary through metadata schema" are unique to the Melodic project. Therefore, they should either be mapped to new features of existing components in the adopted frameworks, or new components in the Melodic platform. The new features will be added to existing components.

Possible levels of fulfilment are as follows:

- Fully supported – the component's features meet a given Melodic objective
- Partially fulfilled – the component's features meet a given Melodic objective, but some improvements are needed.
- Not fulfilled – the component's features do not meet a given Melodic objective.

Table 1: List of components for review and assessment

Framework	Component	Description, key features	Corresponding Melodic objective	Level of fulfilment
PaaSage	CAMEL	Domain code supporting the parsing, validation and interpretation of specifications in the CAMEL language.	The ability to describe an application and its infrastructure in a Cloud-agnostic way	Partially, functional and non-functional changes need to be introduced, as described in Sections 4 and 5.
PaaSage	CDO Server	Storing and Managing (CAMEL) models.	The ability to describe an application and its infrastructure in a Cloud-agnostic way	Fully supported functional requirements, partially supported non-functional requirements as described in Section 5.
PaaSage	CDO Client	Accessing, storing and updating models in CDO Server	The ability to describe an application and its infrastructure in a Cloud-agnostic way	Fully supported functional requirements, partially supported non-functional requirements as described in Section 5.
PaaSage	CP Generator	Profiling of the application and preparing a constraint programming (CP) model	Deployment optimality	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.

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

PaaSage	Meta Solver	Controlling solvers operating on the CP model.	Deployment optimality	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.
PaaSage	MILP Solver	Solving CP models that only take the form of linear optimisation problem specifications	Deployment optimality	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.
PaaSage	CP Solver	Solving all types of problems encoded in the CP model using gradient descent approach.	Deployment optimality	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.
PaaSage	LA Solver	Solving all types of problems encoded in the CP model using Stochastic Learning Automata approach.	Deployment optimality	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.
PaaSage	Solver to deployment	Transforming CP models encompassing the solution produced by solvers to a provider-specific deployment model.	The ability to perform Cross-Cloud deployment of applications	Fully supported functional requirements, partially supported non-functional requirements as described in Section 5.
PaaSage	Plan Generator	Generating deployment action plan for the Adapter.	The ability to perform Cross-Cloud deployment of applications	Partially, functional and non-functional changes need to be introduced as described in section 4 and 5.
PaaSage	Adapter	Deployment control and adaptation of multi-cloud applications	The ability to perform Cross-Cloud deployment of applications	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.
Cloudiator server part	Cloudiator (server part)	Deploying an application and infrastructure to the Cloud Providers.	The ability to perform Cross-Cloud deployment of applications	Partially, functional and non-functional changes need to be introduced as described in Sections 4 and 5.

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

Cloudiator VM part	Cloudiator (VM part)	Components deployed on the created VMs	The ability to perform Cross-Cloud deployment of applications	Fully supported functional and non-functional requirements.
PaaSage	SRL adapter	Handling scalability rules defined in an SRL model.	The ability to perform Cross-Cloud deployment of applications. Deployment optimality.	Fully supported functional requirements, partially supported non-functional requirements as described in Section 5.

3 Review Methodologies

In this section, we describe the review methodologies used for functional changes, non-functional changes, and code assessments.

3.1 Review methodology for functional changes

The main assumption is that the underlying frameworks, PaaSage and Cloudiator, will provide many of the functional features needed to implement the objectives of the Melodic project, and that new features will be introduced either in the existing components or in new ones. The purpose of the functional review is to assess currently available features and specify the scope of the changes needed to adapt the given frameworks to the Melodic project.

The functional requirements of the Melodic project are described in detail in deliverable D2.1 "System Specification"[2]. In that deliverable the general purpose of each component (new ones plus those coming from the underlying frameworks) in the overall Melodic architecture is presented.

Based on the above assumptions, a methodology for the functional review has been designed and followed, comprising the following steps (depicted by *Figure 1*):

1. In the first step, the selected components (listed in Section 2) are matched with the functional objectives of the project.
2. For each selected component to be used in Melodic, the functional features of the given component have been reviewed by assessing the degree of realisation of each feature with respect to the Melodic objectives and the degree of satisfaction of these objectives.

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

3. After reviewing the functional features of each selected component, the list of functional changes needed to extend current features of the component has been produced as the review result.
4. After the creation of the complete list of functional changes for all components, one of the following two possible priorities is assigned to a change request for each component:
    a. must have – the functional change is required to be implemented in order to be able to use the component in the Melodic project as it is critical with respect to the objectives of the Melodic project.
    b. nice to have – it would be good to implement the given feature as it gives additional value, but it is not critical with respect to fulfilling the objectives. The objectives will still be fulfilled without this feature, but its added-value will be diminished. This might include capabilities to support more ergonomic or efficient usage of the platform, or a reduction of the work that is usually conducted or the exploitation of new resource/service kinds.

    The two levels of priority assignment are based on the importance of given changes to fulfil the objectives of the Melodic project and the effort needed to implement them. The general rule is that the more preferred the change is, the earlier release it is assigned to.
5. Based on the prioritized list of functional changes, one of the Melodic releases is assigned to each change qualified for implementation. A given functional change will be implemented in the assigned release.
6. Each functional change will be registered in the JIRA system, which is used for development process management in the Melodic project. Each change should either have a dedicated story, or be combined with other changes into a single story, if all these changes are logically connected to each other. In this respect, we can have a two-level approach where the top-level maps to a group of changes and the lower level mapped to each individual change. This provides flexibility as well as more precise management and reporting as we can assign a whole story to a certain Melodic release or go to the next level and make assignments at the individual functional change level, i.e. individually map each change to a Melodic release.

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

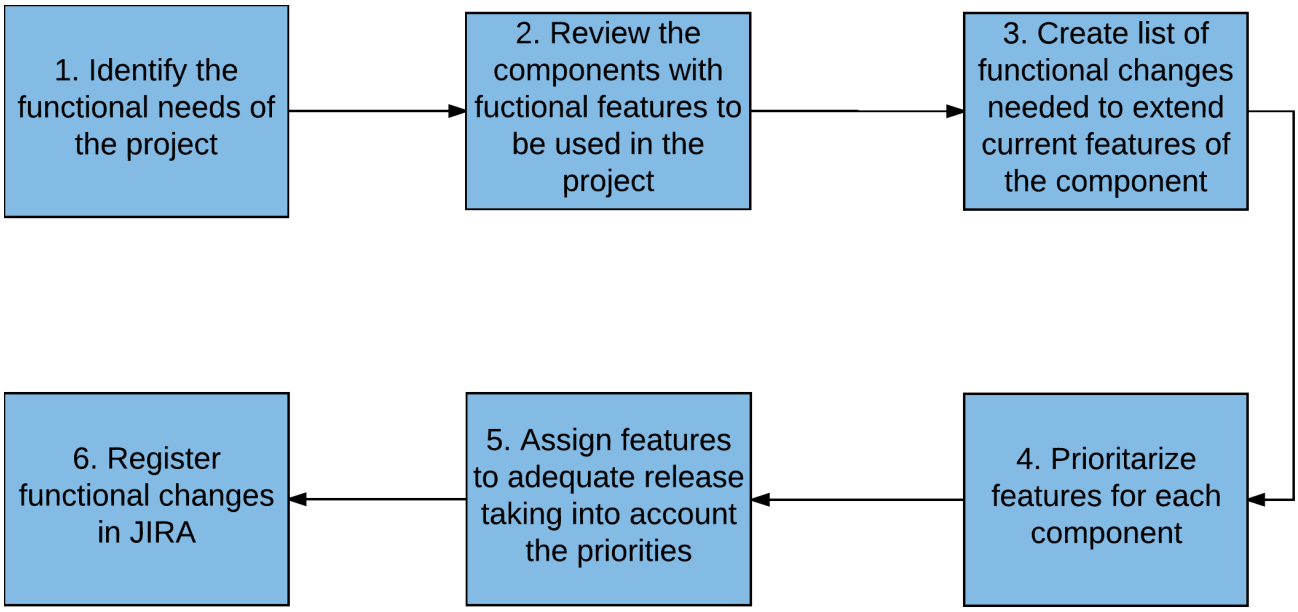

Figure 1: Review methodology for functional changes

3.2 Review methodology for non-functional features

One of the objectives of the Melodic project is the possibility to use the Melodic platform for high-performance systems, like for example real data-intensive applications. Melodic is developed as an open-source product, while the product potentially can be transformed into a commercial one after the end of this project. Thus, the focus of the project is on delivering features that are not of sole commercial or functional nature. Non-functional features are equally important for successful exploitation of the results of the project.

The purpose of the non-functional features review for the two underlying frameworks is to assess how the Melodic objectives are fulfilled in each component. The objectives should be realised and satisfy non-functional requirements as well as the quality requirements described in D5.06 “Test Strategy and Environment” [3].

A more detailed description of these requirements is provided in the Melodic system specification [2]. The non-functional requirements have been collected based on the respective deliverable content (platform requirements) in [2], as well as on professional experience. In addition, for the classification of the non-functional requirements, the CUPRIMDSO methodology has been chosen. CUPRIMDSO is a methodology encompassing a quality model which includes a set of groups that contain certain non-functional criteria. The grouping of criteria is logically performed according to a certain aspect. For instance, the Performance group contains criteria like response time and throughput, which are both related to the performance of an application or system. CUPRIMDSO is a widely used methodology and its set of groups of non-functional attributes are adequate, due to

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

the complexity and scope, for the system under evaluation [4] [5]. The CUPRIMDSO groups of criteria will be used extensively throughout the Melodic project. Moreover, associated metrics can be added to the groups criteria used in the use case demonstrators to assess the quality of the Melodic components. In this document, the non-functional requirements are classified according to the same set of high-level CUPRIMDSO groups of criteria, making the final quality assessment homogenous.

The CUPRIMDSO groups of criteria are as follows:

- Capability - group related to functional aspect of the system for a given usage.
- Usability - group contains all elements related to usability of the system, by humans and by other integrated systems.
- Performance - all performance related aspects of the system.
- Reliability - all elements related to the reliability of the system.
- Installability - the ability to deploy and upgrade the system.
- Maintainability - all topics related to the maintainability of the system.
- Documentation - various types of documentation for the system.
- Serviceability - the ability to properly service system.
- Other - other aspects not covered by the above groups.

For the purpose of this deliverable only the most important elements of this quality model related to non-functional requirements have been chosen, as described below. Due to nature of the underlying frameworks (i.e., academic, non-commercial projects), some elements of the CUPRIMDSO quality model have been left out as they are less relevant (e.g. more suitable to a commercial offering/product). That is the reason why a selection of elements in the CUPRIMDSO quality model have been used. Also, categories and attributes of the model have been extended to better describe requirements for the Melodic project (especially its integration nature). Furthermore, the aspect “usability” is considered mainly as part of the UI design and UI requirements (how well the UI is designed and implemented and how well UI requirements are met), which is briefly covered in section 5.4. All requirements related to the selected CUPRIMIDSO non-functional properties will be registered in JIRA as user stories, and implemented in Melodic releases according to the delivery plan, available effort and time.

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

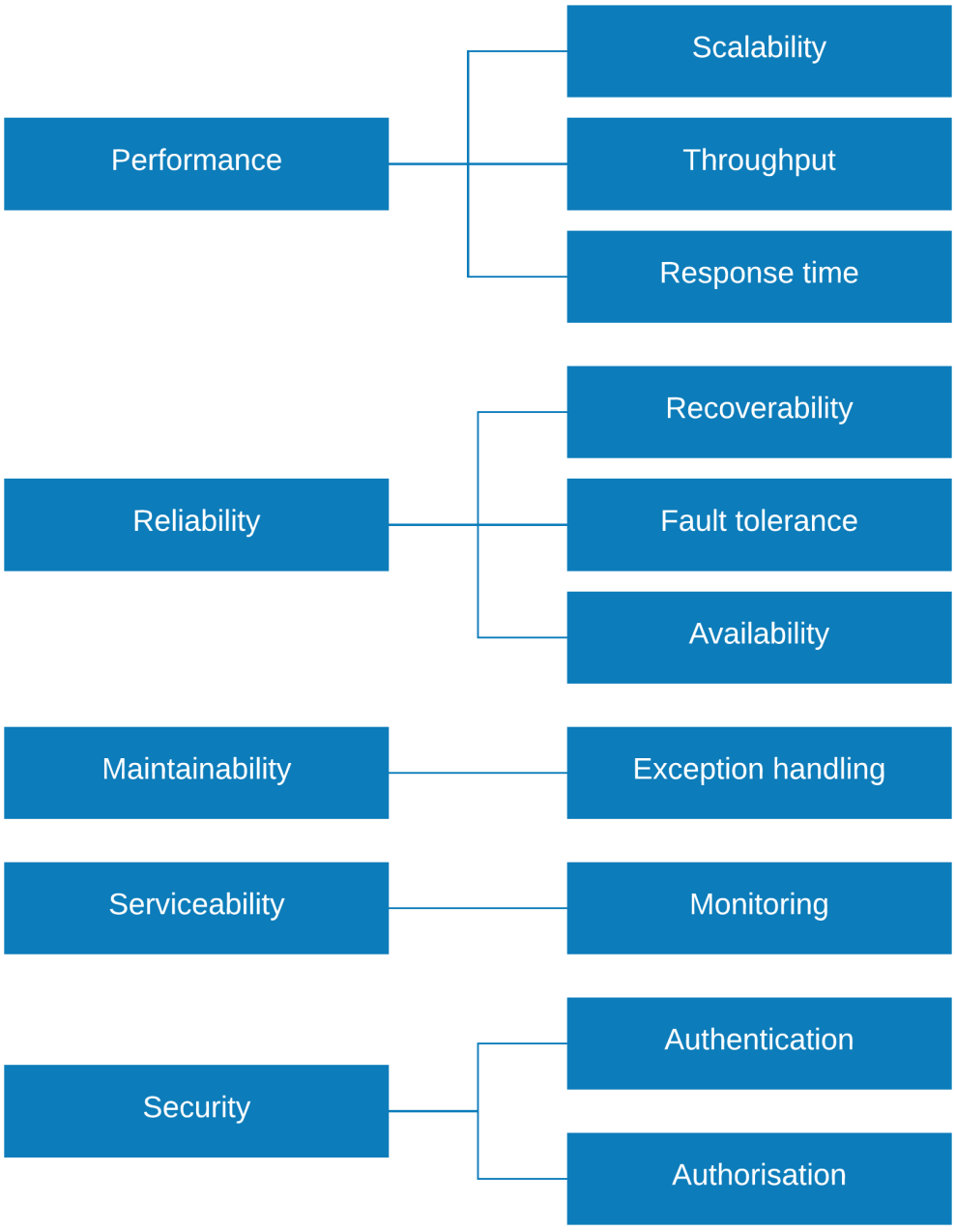

Figure 2: Categories and attributes of CUPRIMDSO methodology

The categories and attributes of the CUPRIMDSO methodology are presented on *Figure 2*. Below are the most important non-functional requirements for the Melodic platform listed and explained in detail based on the CUPRIMDSO quality model. The convention of the naming is as follows: "CUPRIMDSO Category/<Adjective> CUPRIMDSO Attribute$_1$ ('and' CUPRIMDO Attribute$_i$)*". For example, "Reliability/Full Recoverability" indicates that we express a requirement on achieving

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

full system recoverability, where the latter is an attribute of the reliability category in the CUPRIMDSO quality model:

- Reliability/Full recoverability - to achieve reliable flow of the invoked operations, with full control of execution of each operation and returned results. Also support for transactional processing.
- Reliability/High availability – support for highly available, multi-node configurations, at least in active-passive[5] form (active-active[6] configurations will be also explored as an additional benefit). This is related to scalability, but the purpose of multi-node configuration here is to achieve high availability.
- Performance/Sufficient throughput – identification and resolution of performance issues which lead to low throughput, to ensure that performance is not hampered any more or is hampered to the minimum possible degree.
- Performance/High scalability – there are many general and complete definitions of scalability[7], but for the purpose of this deliverable we will use the following one: scalability is the ability to scale the system/platform integration layer both horizontally and vertically. In this sense, scalability is defined as the ability to add new instances for a given component, or increase the hardware resources reserved for that component either in a certain machine or by moving it to another more powerful one. Scalability is related to performance and has an effect on performance as the more resources are available for an application or system, the better performance it might have.
- Performance/Sufficient response time – identification and resolution of performance issues which cause delay in response, to ensure that performance is not hampered any more or is hampered to the minimum possible degree.
- Maintainability/Unified exception handling – maintainability represents the degree of effectiveness and efficiency through which a product or system can be modified in order to be improved, corrected or adapted to changes in environment, and in requirements. For this deliverable, we focus on proper exception handling and retrying of operations. For instance, due to the distributed nature of cloud computing, the communication exceptions need to be properly handled.
- Serviceability/Configurable monitoring – the ability to monitor all components and their operations invoked at the integration layer, with the possibility to dynamically configure monitoring of these operations. This also includes an easily configurable logging mechanism for all of the invoked operations.

[5] Active-passive – multi-node configuration in which one node is active and performs operations, while the other node is passively running but not executing any operation. In case of failure of the primary node, the second node takes over and starts performing operations.

[6] Active–active configuration means that all nodes of a multi-node configuration are running and executing operations.

[7] http://searchdatacenter.techtarget.com/definition/scalability

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

- Security/Basic authentication and authorisation – support for both authentication and authorisation of clients and definition of the access rights to invoke each system operation. The security requirement is mentioned here for completeness, but it will be addressed by a separate new component with new features.

Based on the above analysis, the following methodology for the non-functional feature review is defined:

1. Identify the most important non-functional requirements of the project.
2. Prepare a list of changes. Each selected component of the underlying frameworks (as described in section 2) planned to be used in the Melodic project, will be examined with the main goal to assess the scope and level of implementation of the non-functional requirements analysed above. Based on that, the scope of needed changes to fulfil the aforementioned non-functional requirements will be determined.
3. Prioritize the changes. After creating the complete list of changes for all components related to non-functional requirements, one of the following three possible priorities will be assigned to each one of them:
    a. Must have – the non-functional related change needs to be implemented in order to use the component in the Melodic project.
    b. Nice to have – it would be good to implement the given change as it gives additional value, but it is not critical for fulfilling the objectives of the Melodic project.
    c. To test – while a non-functional requirement is implemented for a certain component, its implementation level might not be clear. To this end, we carefully verify the implementation level and scope for a certain non-functional requirement via testing. In particular, we prepare a dedicated testing scenario. Such scenarios are described in detail in deliverable D5.4 "Integration & testing requirements" [6]. After the execution of the test scenario, the respective decision is taken about whether the corresponding component implementation needs to be improved, according to which priority, and for which future release of the Melodic project.

    The priority assignment is based on the effectiveness of given changes to fulfil the objectives of Melodic project, and the effort needed to implement them. The general rule is that the more preferred the change is, the earlier release it is assigned to.

    The following criteria are used for priority decision/assignment. The criteria should be used together, in a combined way:

    a. The importance of the non-functional requirement with respect to the importance of a non-functional change, depends on the actual system process being applied and the criticality of the component concerned on the level of fulfilment of the requirement by the component. In this respect, if the component is not critical, then

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

the priority will be "nice to have"; otherwise, the priority will be "must have". For example, consider the requirement about performance/response time for the deployment process. In this case, deployment reasoning, due to its heavy computational nature, has a far greater impact on deployment process response time than the functionality of choosing solver for reasoning in the Meta Solver. Thus any change related to this requirement for the solver components will be assigned a "must have" priority in contrast to the case of the "Meta Solver" for which the respective change(s) will be assigned a "nice-to-have" priority.

b. The actual level of a given non-functional requirement fulfilment for a particular component. Based on the actual implementation of the given non-functional requirement, the schedule of the change implementation will be determined. For example, if a given component has no logging capabilities, then adding logging capabilities should be given a higher priority than what would be the case if the given component already has some logging capabilities, but with improvements needed to fulfil requirements.

4. Based on the prioritized list of non-functional changes to existing components, each change, qualified to be implemented, is assigned to one of the Melodic releases. The general rule is that all changes with "must have" priority have to be implemented, and more critical changes need to be handled within the first releases, while less critical changes can wait for later releases. Some changes could be mapped to two releases, if they need more effort for implementation. The "must have" changes should be scheduled according to overall Melodic release planning, and should be aligned with all planned changes. The "must have" changes for an existing component should be planned for first release, if possible, according to available effort and time. The changes with nice to have priority should be implemented according to available resources and time, as it will be planned in Melodic project's delivery plan. The changes to new components are planned with the development of new components, mostly in the 2nd and 3rd releases of the Melodic framework.
5. Each required non-functional change will be registered in the JIRA system as a story, which is used for development process management in the Melodic project. Each change should either have a dedicated story, or be combined with other changes into a single story if all these changes are logically connected to each other. This approach is exactly the same as for the review of functional requirements.

3.3 Review methodology for code assessments

The source code of the components from underlying frameworks, which are qualified to be used in the Melodic project, should be reviewed and assessed. The purpose of the code assessment is to

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

verify the quality of the code against best programming practices as well as quality assurance requirements.

The best programming practices used for evaluation are based on the Clean Code[8] approach. Clean Code is one of the most widely recognized and used approaches for software development. In addition, it is also extensively used by 7bulls.com, meaning that Melodic can exploit the knowledge and experience obtained by 7bulls.com people on this approach. The most important evaluated principles are as follows:

1. Write Short Units of Code — Short units are easier to understand.
2. Write Simple Units of Code — Simple units are easier to test.
3. Write Code Once — Duplicated code means duplicated bugs and duplicating changes.
4. Keep Unit Interfaces Small — Units with small interfaces are easier to reuse.
5. Separate Concerns in Modules — Modules with a single responsibility are easier to change.
6. Loosely Coupled Architecture Components— Independent components can be maintained in isolation.
7. Keep Component Architecture Balanced — a balanced architecture makes it easier to find your way.
8. Keep Your Codebase Small — a small codebase requires less effort to maintain.
9. Automate Tests — automated tests are repeatable, and help to prevent bugs.
10. Write Clean Code — the code should be clean and self-descriptive.

The quality assurance related requirements are described in deliverable D5.06 "Test strategy and environment" [3]. The most important quality assurance requirements related to the code assessments are the following:

- Unit tests preparation with a given code coverage of 40% as described in [3].
- Integration tests preparation with a given code coverage, for all methods exposed for other components.

Based on the above assumptions, a certain methodology is proposed for the source code assessment comprising the following steps (see *Figure 3*):

1. Each qualified component of underlying frameworks (list provided in section 2) is reviewed against best programming practices and quality assurance requirements as described above. A list of requested changes is prepared for each component.
2. The list of changes needed to fulfil the best programming practices and quality assurance requirements is reviewed, and one of the following two possible priorities are assigned to them:

[8] https://bobbelderbos.com/2016/03/building-maintainable-software/

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

a. Must have – quality assurance-based changes are always assigned this priority. On the other hand, only a code assessment change that violates any of the first 5 principles takes the same priority.
b. Nice to have – it would be good to implement a given change because it gives additional value, but it is not critical to fulfil the objectives of the Melodic project. This maps mainly to code assessment changes which are related to the violation of the last 5 principles.

Based on the prioritized list of code assessment changes, each change qualified to be implemented, is assigned to one of the Melodic releases. The general rule is that all changes with "must have" priority have to be implemented, and more critical changes need to be dealt with in the first releases. Less core changes can be handled in further releases. Some changes could be mapped to two releases, if more effort for implementation is needed. The changes with nice to have priority should be implemented according to available resources and time as it will be planned in Melodic project's delivery plan.

For each component, all code assessment changes are registered in the JIRA system as a story based on the two-level approach suggested previously for the functional and non-functional changes in this project.

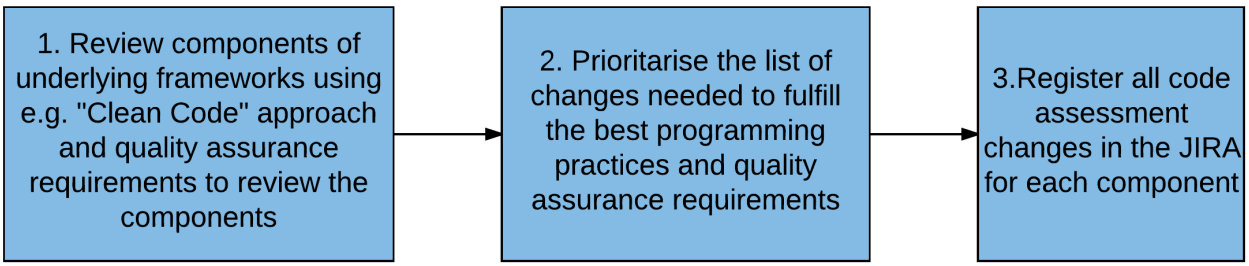

Figure 3: Review methodology for code assessment

4 Required functional features

This section presents the results of the evaluation of functional changes needed to be implemented for the underlying frameworks (PaaSage/Cloudiator). As described in section 3.1, having a careful review conducted, the list of changes for the underlying frameworks is identified. Based on the list of issues, the scope of needed changes per issue is described. Also, the priority for each change is assigned, as well as the exact release(s) of Melodic where this change will be made.

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

The results of the evaluation are presented in the following table. The table focuses mainly on the description of the issue, the scope of the corresponding change, the priority assigned to this change/issue, and the target release. It contains the following set of columns:

- Issue id – unique id of the given issue related to the underlying frameworks.
- Project/component – the name of the project and the components selected from that project related to the given issue.
- Issue name – the name of the issue.
- Issue description – brief description of the issue.
- Scope of change – the scope of the needed change in the underlying framework, mainly in terms of the component(s) involved, to fix or prepare a workaround for the respective issue.
- Priority – the priority of the change, assigned according to methodology described in section 3.1.
- Release – Melodic release in which the change will be deployed.

Table 2 presents the mapping of each change identified in the review process to the respective user story registered in JIRA, as well as to the Melodic release(s) (multiple releases can be involved if the change is gradually applied) in which the change should be implemented. This table contains the initial list of user stories, a list that will be extended and changed during the project lifetime according to identified needs and implementation plan in Melodic releases.

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

Table 2: Mapping of changes to user stories and releases

Issue Id	Project/ component(s)	Issue name	Issue description	Scope of change	Priority	Release
001	PaaSage / Profiler	Merging of CP Generator and Rule Processor	As both the CP Generator and Rule Process sub-components perform filtering of the provider space, it has been decided to merge them into one, single component, i.e., the Profiler one.	The merged component will expose two distinct functions which could map, in the end, to two distinct sub-components: (a) filtering of the provider space; (b) generation of the CP model, i.e., the model to be solved by the PaaSage/Reasoner in order to derive a concrete deployment model for the user application.	Must have	R1
002	PaaSage / Profiler + CP Generator	Full CAMEL Semantics Conformance in Profiler	CAMEL semantics for requirement modelling	Qualitative hardware requirements will be covered similarly to the case of quantitative ones, but linguistic search might be employed due to the nature of the involved requirement values. Security requirements will be covered via the matching of security controls sets (those offered and those required). Handling of data aspects will also be covered by this component.	Must have	R2

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

003	PaaSage / Profiler + CP Generator	Complete & Correct CP Model Generation	CP Model generation needs to be corrected	The CP model to be generated needs to have the right and sufficient number of variables and related constraints, which need to conform to the requirements posed in the user's CAMEL model.	Must have	R1
004	PaaSage / Profiler + CP Generator	Component Co-Location Handling in Profiler	CP Model generation needs to be corrected to support the semantics of the type of communication between application components.	The CP Generator needs to be modified in order to properly handle the different types of communication between application components in the generation of the CP model. REMOTE communication will be handled for now, and LOCAL/ANY communication in the second release.	Must have	R1, R2

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

005	PaaSage / Profiler + Adapter + Solver To Deployment + Executionware + CAMEL	CAMEL – PaaS Extension	PaaS extension of CAMEL to enable the deployment of application components via PaaS as well as the exploitation of PaaS services.	Multiple components need to be updated: (a) CAMEL as it requires some extensions on its meta-model; (b) Profiler to enable the matchmaking of PaaS and the generation of the respective CP model; Solver to deployment to produce extended CAMEL models (d) Adapter to enable validating extended CAMEL models and enforcing the deployment of the respective user applications; (e) Executionware as it requires to support the deployment of application components via PaaS services.	Nice to have	R2

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

006	PaaSage / Profiler	Component Composition	Components can have two different types of (composition) dependencies: (a) one component comprises multiple sub-components; (b) one component hosts multiple components. In both cases, we need to make an assignment of all the dependent components in a respective VM by aggregating the corresponding hardware requirements that need to be satisfied for each component.	The filtering of the provider space in the Profiler should consider the aggregated hardware requirements that need to be collected from the composition dependencies.	Must have	R2

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

007	PaaSage / Profiler + Reasoner + Adapter	Offer Space Update Propagation	Offer space updating (e.g., due to change of pricing of VM offerings) needs to be handled by the platform.	The Profiler should be able to sense the updates and check whether they need to be propagated for the application at hand. In that case, it needs to update the CP model generated and re-initiate the Upperware flow, with the exception that the application needs to be reconfigured and not deployed from scratch. This means that the Reasoner should produce a new solution to the updated CP model and forward it to the Adapter which can then decide whether such a solution needs to be enforced. From now on, reconfiguration should be performed based on the updated CP model.	Must have	R2
008	PaaSage / Profiler	Provider Filtering	The current way the providers are filtered is wrong: (a) it does not rely completely on CAMEL semantics; (b) it does not take into account the providers that can be actually exploited by the user (i.e., the user has an account on them)	The Profiler needs to be updated in order to first filter cloud providers based on the user account context, and then (second) based on the actual provider requirements supplied by the user in the CAMEL model.	Must have	R1, R2

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

009	PaaSage / Profiler + Reasoner	Utility Function Handling	Generation of utility function according to the user preferences and optimisation goals. Exploitation of the function by the Reasoner.	Currently, the utility function is hardcoded in the CP model, and wrongly encoded. In addition, it is now considered whether it can be externalised into a component exploited by the Reasoner during CP model solving.	Must have	R2
010	PaaSage / Reasoner	Uniform Logging in Reasoning	A uniform way of logging across all solvers, realising the functionality of a Reasoner.	This update needs to be applied to all solvers developed in PaaSage to enable a uniform logging experience for the user.	Must have	R1
011	PaaSage / Meta Solver	Solver Selection in Meta Solver	The ability to select one from all possible solvers available in the platform.	The selection functionality, currently missing, should rely on certain criteria, which should include the CP model linearity, as well as some unique solver capabilities. Such criteria could come from the user preferences.	Must have	R2

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

012	PaaSage / Meta Solver	Multiple Solvers in Meta Solver	The ability to invoke multiple solvers in parallel, and receive in response the respective solutions and find the best possible one.	Currently, the Meta Solver has the ability to invoke multiple solvers. However, this has to be done in conjunction with update 011 (above). In addition, the use of multiple solvers leads to concurrency problems (writing on the same CP models in the model repository) which need to be resolved.	Nice to have	R2

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

013	PaaSage / Meta Solver	Deployment Reasoning Strategy & Metric Subscription Scheme in Meta Solver	The Meta Solver needs to dynamically invoke the Reasoner according to a certain strategy in order not to overwhelm the system with too frequent deployment reasoning executions. The metric subscription schema also needs to be updated to cater for the fact that multiple applications can now be handled by a single platform instance; multiple applications can now be handled by a single platform instance in the form of an application agglomeration, i.e., a super-application	The strategy should rely on the frequency of production of the relevant metrics measurements and take into consideration the fact that not all measurements indicate the need to re-solve the CP model of the user application.	Must have	R2

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

014	PaaSage / CAMEL + CP Generator + Adapter + Executionware	Cost Modelling & Sensing	The ability of the platform to specify cost models.	The cost models should be able to model at least the approximated cost of the application.	Must have	R2
015	PaaSage / Adapter	Adapter Re-Engineering	Re-engineer the Adapter according to the new Melodic Architecture, and enable its testing & produce respective technical documentation. Data-awareness must also be realised.	The first part will be realised in the 1st release, while the 2nd part (data-awareness) will be realised in the 2nd release.	Must have	R1, R2
016	PaaSage / Adapter	Cost of the reconfiguration	The Adapter needs to provide the cost of the reconfiguration to Utility Generator.	The Adapter needs to be updated with the capability to provide the estimated cost of reconfiguration.	Must have	R2
017	PaaSage / Adapter	Exception Handling in Adapter	Deployment fault handling seems inadequate: when errors occur, the deployment plan execution is interrupted and the current deployment model / state is updated. In this case, all deployments are undone. However, the previous stable and legitimate state of the running system should also be recovered.	Need to implement proper deployment fault handling in the Adapter as it is critical for the use-case deployment & reconfiguration. The scope and type (local/global) of deployment fault handling needs to be considered. In case of permanent errors, the default behaviour would be to always be able to return to the previous stable and legitimate state of the running system.	Must have	R2

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

018	PaaSage / Adapter	Hardware Details Selection in Adapter	Currently, the hardware details for each application VM are selected randomly. A more informed decision needs to take place according to certain criteria.	Once a solution is validated, it needs to be enriched by the Adapter via the determination of the most suitable hardware details for each application VM to be deployed. Such a determination could rely on selection rules that rely on certain criteria as well as possible user preferences.	Must have	R2
019	PaaSage / SRL Adapter	SRL Adapter Re-Engineering	The functionality of this component has not been properly and completely tested. Moreover, it seems that it does not impose an optimal monitoring infrastructure reconfiguration.	SRL Adapter needs to be re-engineered to enforce an appropriate monitoring infrastructure reconfiguration when application reconfiguration also takes place. The ability to properly test this component should also be realised.	Must have	R1, R2

5 Required non-functional features

In this section, the results of the review of the level of fulfilment of the non-functional requirements for each component are presented. As described in section 3.2, having the careful review conducted, the list of non-functional changes needed to be implemented in components from the underlying frameworks, will be identified. The priority for each change will also be assigned. First, we present the component specific non-functional features. Then, in the following sections, we present the common features for all components, one feature per section.

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

5.1 Component specific non-functional features

In this section, we present in table-based form the list of changes that need to be performed for each component of the underlying frameworks according to the set of non-functional requirements that have been posed. Detailed, specific changes for each component are listed. The columns of each corresponding table have the following semantics:

- Change id – unique id of the identified change.
- Non-functional feature – name of the non-functional feature.
- Change description – the name of the change and brief description.
- Scope and level of implementation – the scope and level of implementation of the needed change in the underlying framework to fulfil a particular non-functional feature.
- Priority – the priority of the change, according to the methodology described in section 3.2.
- Release - Melodic release in which the change will be deployed.

Table 3 lists the non-functional requirements for the Adapter component from the PaaSage project.

Table 3: Non-functional requirements for the component Adapter

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	The ability to retrieve deployment configuration after restart. Transaction handling with CDO Server communication.	The implementation of the retrieval of the current deployment configuration should be performed. The implementation of the writing and retrieval of deployment configuration information should be supported via the use of transactions over the CDO model repository.	Must have	R1
002	Reliability/High availability	The ability to set up multi-instance configurations, with active-passive working nodes.	It shall be possible to set up a multi-instance configuration of the Adapter component, and to use the second instance in case of failure of the first.	Must have	R2

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

003	Performance/ Sufficient throughput	To be verified in performance tests.	The change might be applied after performing the respective tests that highlight the need in question.	To test	R2
004	Performance/ High scalability	Multi threads and parallel operations support	The deployment operations should be done in parallel, via the use of multiple threads, where possible, and should allow for vertical scaling[9] of the component.	Must have	R2
005	Maintainability/ Unified exception handling	Proper exception handling in case of deployment failure	A proper approach for handling unsuccessful execution	Must have	R2

Table 4 lists non-functional requirements for the component CP Generator from the PaaSage project.

Table 4: Non-functional requirements for the component CP Generator

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	Transaction support in communication with CDO.	The transaction support should be implemented in case of storing/retrieving data to/from the CDO Server.	Must have	R1.5
002	Performance	To be verified in performance tests.	The dedicated test scenario is described in the deliverable D5.04 "Integration & testing requirements".	To test	R1.5

[9] Vertical scaling – the ability to improve of given application/component through installing it on a more powerful machine. Usually it requires the ability to parallel multithread processing within the application, with proper application architecture (without bottlenecks by single thread operations).

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

003	Performance/High scalability	The ability to scale vertically.	The CP Generator should be able to scale vertically, where possible.	Nice to have	R3
004	Reliability/Fault tolerance and High availability	Recover after crash or restart.	The recovery process should be checked during the fault tolerance tests.	To test	R1.5
005	Maintainability/Unified exception handling	Proper exception handling.	Proper exception handling in communication with the ESB and the CDO Server should be implemented, with retrying of operations in case of particular error types	Must have	R1.5

Table 5 lists non-functional requirements for the component CAMEL editor from the PaaSage project.

Table 5: Non-functional requirements for the component CAMEL editor

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	Not applicable.	n/a	n/a	
002	Performance/Sufficient throughput	Not applicable.	n/a	n/a	
003	Performance/High scalability	Not applicable.	n/a	n/a	
004	Reliability/Fault tolerance and High availability	Not applicable.	n/a	n/a	
005	Maintainability/Unified exception handling	Exception handling should be implemented.	n/a	n/a	

Table 6 lists non-functional requirements for the component Meta Solver from project PaaSage

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

Table 6: Non-functional requirements for the component Meta Solver

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	Transaction handling and retrying.	Transaction support for communication with the CDO server should be improved, especially with respect to the retry of failed transactions.	Must have	R1
002	Performance/Sufficient throughput	To verify during performance tests.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R2
003	Performance/High scalability	Multithread support.	Support of multithread operations.	Must have	R2
004	Reliability/Fault tolerance and High availability	The ability to support the execution of multiple Meta Solver instances.	This needs to be verified during fault tolerant tests.	Nice to have	R3
005	Maintainability/Unified exception handling	Handling exceptions and retrying in communication with solvers.	The communication with solvers should properly handle exceptions and allow for retrying in case of inaccessibility of a certain solver.	Must have	R2

Table 7 lists non-functional requirements for all the solver components (MILP Solver, CP Solver and LA Solver) from the PaaSage project.

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

Table 7: Non-functional requirements for all the solver components

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	The ability to recover last operation in case of a crash/restart.	Possibility to recover last computation from a given point in case of a crash or restart of the component.	Nice to have	R3
002	Performance/ Sufficient throughput	To be verified during performance tests.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R2
003	Performance/High scalability	The ability to scale vertically, multithread support.	The calculation of the solution should be implemented with multithreads support to allow for vertical scaling.	Nice to have	R3
004	Reliability/Fault tolerance and High availability	The ability to deploy multiple instances of the solver.	It should be possible to deploy multiple instances of a solver in the active-passive mode. In case of a failure of the primary instance, the secondary instance could be used.	Nice to have	R3
005	Maintainability/ Unified exception handling	Proper exception handling in communication with the ESB and CDO Server. Retrying of ESB invocation and storing data in the CDO Server. The scope should be decided during the implementation.	There should be support for exception handling in communication with CDO, especially for storing the CP model. There should also be retrying of the communication to ESB to send notifications about the computed solution.	Must have	R2

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

Table 8 lists non-functional requirements for the component Solver to Deployment from the PaaSage project.

Table 8: Non-functional requirements for the component Solver to Deployment

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/Full recoverability	Transaction handling.	Transaction support for communication with the CDO Server should be implemented.	Must have	R1.5
002	Performance/Sufficient throughput	No changes needed.	n/a	n/a	
003	Performance/High scalability	No changes needed due to lack of heavy computation operation in this component.	n/a	n/a	
004	Reliability/Fault tolerance and High availability	Possibility of multiple instances deployment.	As for the Adapter, it should be possible to implement the Solver to Deployment component in multi-instance deployment, with active-passive model. In case of a failure of the primary instance, the secondary instance could handle the incoming requests.	Must have	R2
005	Maintainability/Unified exception handling	Proper communication with the ESB and CDO Server guaranteed by exception handling and retrying.	The exceptions in the communication with the ESB and CDO Server should be properly handled. A retrying mechanism should be implemented.	Must have	R2

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

Table 9 lists non-functional requirements for component SRL adapter from the PaaSage project.

Table 9: Non-functional requirements for the component SRL adapter

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/ Recoverability (transactions)	The ability to recover after a crash/restart.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R1
002	Performance/Sufficient throughput	Needs to be verified during performance tests.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R2
003	Performance/High scalability	The ability to scale horizontally.	It should be possible to set up multiple instances of the SLR adapter for performance and high availability purposes.	Nice to have	R3
004	Reliability/Fault tolerance and High availability	The ability to install in multi-instances configuration.	See above.	Nice to have	R3
005	Maintainability/Unified exception handling	Proper exceptions handling and the ability to recover communication with other components	Proper exceptions handling and the ability to recover communication with other components should be implemented.	Must have	R2

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

Table 10 lists non-functional requirements for the component CDO Server from the PaaSage project.

Table 10: Non-functional requirements for the component CDO Server

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/ Recoverability (transactions)	See fault tolerance and exception handling (4th and 5th row)	See fault tolerance and exception handling (4th and 5th row)	n/a	
002	Performance/Sufficient throughput	No an issue.	n/a	n/a	
003	Performance/High scalability	Not needed.	n/a	n/a	
004	Reliability/Fault tolerance and High availability	The ability to install a multiple instances configuration.	It should be possible to deploy multiple instances of the CDO Server in the active-passive mode. In case of a failure of the primary instance, the secondary instance should be used.	Nice to have	R3
005	Maintainability/Unified exception handling	Proper exceptions handling and retrying communication with underlying database.	Proper exception handling and retrying communication for custom code created on the top of Eclipse code with an underlying database should be implemented.	Nice to have	R3

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

Table 11 lists non-functional requirements for the component CDO Client from the PaaSage project.

Table 11: Non-functional requirements for the component CDO Client

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/ Recoverability (transactions)	Retrying operations in case of inability to reach CDO Server.	The ability to retry operations, based on a given configuration, in case of inaccessibility of CDO Server.	Nice to have	R3
002	Performance/Sufficient throughput	No known issues.	n/a	n/a	
003	Performance/High scalability	Not applicable.	n/a	n/a	
004	Reliability/Fault tolerance and High availability	Not applicable.	n/a	n/a	
005	Maintainability/Unified exception handling	Proper exceptions handling.	The exceptions should be properly logged with clear error messages. The recoverability aspect of exception handling is described under change 001 in this table.	Nice to have	R3

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

Table 12 lists non-functional requirements for the component Plan Generator from the PaaSage project.

Table 12: Non-functional requirements for the component Plan Generator

Change Id	Non-functional feature	Change description	Review results and scope of change	Priority	Release
001	Reliability/ Recoverability (transactions)	The ability to recover the last operation after a restart or crash.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R1
002	Performance/Sufficient throughput	No known issues.	n/a	n/a	
003	Performance/High scalability	Not applicable.	n/a	n/a	
004	Reliability/Fault tolerance and High availability	HA configuration.	As for the Adapter, it should be possible to implement a Plan Generator component in multi-instance deployment, with an active-passive model. In case of a failure of the primary instance, the secondary instance could handle the incoming requests.	Nice to have	R3
005	Maintainability/Unified exception handling	Exception handling and retrying communication to ESB.	The exceptions in the communication with the ESB should be properly handled. The retrying mechanism should be implemented.	Must have	R2

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

Table 13 lists non-functional requirements for the component Cloudiator.

Table 13: Non-functional requirements for the component Cloudiator

Change Id	Non-functional feature	Change description	Scope and level of implementation	Priority	Release
001	Reliability/ Recoverability (transactions)	The ability to retrieve the current status of invoked method in case of a restart or crash.	Needs to be checked during the tests.	To test	R1
002	Performance/Sufficient throughput	Needs to be verified during the tests.	The dedicated test scenario is described in deliverable D5.04 "Integration & testing requirements".	To test	R2
003	Performance/High scalability	The ability to scale horizontally.	It should be possible to deploy multiple instances of Cloudiator (or sub components of Cloudiator) in active-passive mode (all instances could execute the tasks) to allow horizontal scaling.	Nice to have	R3
004	Reliability/Fault tolerance and High availability	The ability to install in multiple instances.	It should be possible to deploy multiple instances of Cloudiator (or sub components of Cloudiator) in active-passive mode to allow high availability configuration.	Nice to have	R3
005	Maintainability/Unified exception handling	Proper exception handling and retrying communication with Cloud Providers and ESB layer.	Needs to be checked during the tests.	To test	R2

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

5.2 Serviceability and Configurable monitoring

The scope of changes for the Serviceability CUPRIMDSO category in the area of monitoring and logging is very similar for each reviewed component. Implementation of a unified way of logging should be introduced. It should support different logging levels and common configuration of logging per Melodic platform. This change has the "Must have" priority, as without the change, difficulties with regards to monitoring might impair the potential for improvement of the system as well as troubleshooting.

5.3 Overall Security

The security requirements are similar for each component. The method invocation authentication and operation authorization should be introduced at the ESB level. Also, for specific components, the secure transmission protocol SSL should be used. Thanks to that, a minimum level of security will be met.

5.4 Usability of non-functional requirement

The term Usability (as a top level CUPRIMDSO category) covers all elements that have impact on the general usage of the system, making it easy to use, ergonomic, easy to understand and stable. Usability as a non-functional requirement should be reviewed looking at the system as a whole, not at particular components. For this reason, usability is described separately. However, note that usability mainly refers to how the system is used by users and is thus related to the external interfaces of some of the components in the platform. Thus, in the end, usability is required to be reviewed by looking at the system as a whole and the functionality it exposes. Still, it always comes down to checking the external interfaces of some of the components of the platform.

For usability, the most important element is the User Interface (UI) of the system and general ergonomics of usage. Also, it should be kept in mind that other non-functional requirements like reliability, logging, monitoring, exception handling and fault tolerance, have an overall impact on (system) usability. For example, without proper logging of errors in a unified way, it is impossible to effectively use the system on production due to inability to effectively troubleshoot it.

There are several User Interfaces (UI) for the PaaSage platform and Cloudiator. The most important ones are the following:

- CAMEL Editor – Eclipse-based editor for CAMEL authoring.
- Cloudiator UI – web based UI for inspection/retrieval and creation of objects in Cloudiator.

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

- Social Network – UI for social interaction and model sharing.
- PaaSage Rest Client – command line tool to invoke PaaSage operations.

The UIs are created using different technologies and cover different (functional) parts of the system. There is no unified UI for the whole platform. There is also no UI for monitoring and administering the platform as a whole. To this end, it has been decided to try to design, develop and prepare a new UI component which will unify existing UI components from PaaSage and extend them via the addition of missing capabilities. In this respect, as we are dealing with a new component, its evaluation is considered as out of scope of this deliverable. Furthermore, it does not make sense to evaluate any of the existing UI components of PaaSage, as a new, unified UI component will be used to replace them. For the new unified UI component, the detailed scope of UI features will be supplied as a set of requirements and registered in JIRA as user stories. Elements of the new UI related component are described in deliverables D2.4 "Metadata schema management" [7].

5.5 Mapping requirement to JIRA user stories

The JIRA dashboards listed below presents a mapping of the changes identified in the review process to user stories registered in JIRA, and respective Melodic release(s) in which the change should be implemented. This list will be extended and changed during the project, according to identified needs and implementation plans for Melodic releases.

Link to dashboard of all user stories https://jira.7bulls.eu/browse/MEL-10?filter=10020.

Melodic release 1.0 https://jira.7bulls.eu/secure/Dashboard.jspa?selectPageId=10111.

Melodic release 1.5 https://jira.7bulls.eu/secure/Dashboard.jspa?selectPageId=10112.

Melodic release 2.0 https://jira.7bulls.eu/secure/Dashboard.jspa?selectPageId=10113.

6 Code assessment results

In this section, the results of the code & quality assurance assessment for each component are presented. As described in section 3.3, after a careful review that has been conducted, the list of changes needed to be implemented in components of the underlying frameworks has been identified. A priority has been assigned to each change.

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

The results of the review related to the code assessment are presented in the tables below, each table associated with a given component. The tables contains the following set of columns:

- Issue id – unique id of the identified issue.
- Project/Component/Class - the project, component and class in which the issue has been discovered.
- Issue name – the name of the issue.
- Issue description – description of the issue, with relation to programming best practices as well as impact of the issue.
- Suggested resolution – suggested resolution; how to fix the given issue.
- Priority – the priority of the change, assigned according to the methodology described in section 3.3.
- Clean Code principle IDs – IDs of Clean Code principles (listed in section 3.3) violated by the given issue.

Table 14 presents the code assessment results for the component Adapter from the PaaSage project.

Table 14: Code assessment results for the component Adapter

Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	Action interface	Complicated and duplicated code.	Two Map arguments of execute method. More than 90% of the Action class's implementations use only one object, which must be stored under 'execInterfacer' key (which by the way is a magic string).	The Input param should be generic. The Output param should be generic, too, but not as method param, but as method return type.	Must have	1, 2, 3
002	Action interface	No unit tests for implementation of the Action interface.	There are no tests even for implementations with complicated logic.	Add tests.	Must have	9

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

003	PlanGenerator	Too long methods.	E.g., the generatePlanGraph method contains 20 decision points.	E.g., use a chain of responsibility pattern or mapping tool.	Must have	2
004	ExecInterface	One very big class for cloud operation. Lack of proper structure of the code.	More than 4000 lines of code.	Split to smaller classes with bordered logic, e.g., a class for cloud lifecycle, operation on virtual machines.	Must have	2
005	ExecInterface	No unit tests for ExecInterface logic.		Add tests.	Must have	9
006	ZeromqServer	No dependency injection framework is used.	Construction is separated from usage, but in case of a need to change implementation class or even jms queue address we must change the implementation.	Use a dependency injection framework, e.g., Spring.	Must have	10
007	ZeroMQPublisher	No dependency injection framework is used.	Construction is separated from usage, but in case of a need to change implementation class or even jms queue address we must change the implementation.	Use a dependency injection framework, e.g., Spring.	Must have	10
008	ZeroMQSubscriber	No dependency injection framework is used.	Construction is separated from usage, but in case of a need to change implementation class or even jms queue address we must change the implementation.	Use a dependency injection framework, e.g., Spring.	Must have	10

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

009	ModelComparator	Comparing and grouping logic in one big class. No usage of java.util. Comparator.	This class does not contain compared logic only, but also contains grouping logic.	Should be split into two groups of classes: one for grouping logic and the second for comparing. java.util.Comparator should be used.	Must have	1, 2
010	DeploymentModelComparator	Comparing and grouping logic in one big class. No usage of java.util. Comparator.	This class does not contain compared logic only, but also contains grouping logic.	Should be split into two groups of classes: one for grouping logic and the second for comparing. java.util.Comparator should be used.	Must have	1, 2
011	CommunicationAction	Too long methods.	There are methods whose implementation contains more than 300 lines.	Resolution depends on the problem in specific class, but in general, splitting up to smaller private methods by using chain of responsibility or strategy pattern, will be enough (after removing commented lines of code).	Must have	2
012	VMAction	Too long methods.	There are methods whose implementation contains more than 300 lines.	Resolution depends on the problem in specific class, but in general, splitting up to smaller private methods by using chain of responsibility or strategy pattern, will be enough (after removing commented lines of code).	Must have	2

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

Table 15 presents code assessment results for the component Solver to Deployment from the PaaSage project.

Table 15: Code assessment results for the component Solver to Deployment

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	S2D_ZeroMQServer	Too long methods.	Too many decision points in methods.	E.g., use chain of responsibility pattern or mapping tool.	Must have	2
002	S2D_ZMQ_Service	Too long methods.	Too many decision points in methods.	E.g., use chain of responsibility pattern or mapping tool.	Must have	2
003	All classes	Lack of tests in Solver to Deployment component.		Add tests.	Must have	9

Table 16 presents code assessment results for the component Plan Generator from the PaaSage project.

Table 16: Code assessment results for the component Plan Generator

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	PlanGenerator	Too long methods.	E.g., the generatePlanGraph method contains 20 decision points.	E.g., use chain of responsibility pattern or mapping tool.	Must have	2
002	Test classes	Wrong packages name in unit tests.	Test class package name should be equal to the class that is under test.	Fix packages names.	Nice to have	10

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

Table 17 presents code assessment results for the component CP Solver from the PaaSage project.

Table 17: Code assessment results for the component CP Solver

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	All classes of CPSolver module	Lack of package declaration in classes	No package declared in classes	Add package	Must have	10
002	CPSolver	Too big responsibility of methods		Clean code rules.	Must have	5, 7
003	CPSolver	God class in CPSolver module	CPSolver looks like a god class and has too big responsibility	Move some class logic into external classes, e.g., a class for parsing exceptions or factories to create constraints and variables	Must have	2, 5
004	CPSolver	No configuration files	Queues configuration is hardcoded in the CPSolverDeamon class	Move to properties file	Must have	10

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

Table 18 presents code assessment results for the component Meta Solver from project PaaSage.

Table 18: Code assessment results for the component Meta Solver

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	Most classes	Lack of unit tests.	For three modules, only one class (CPModelTool from the Meta Solver module) has test coverage.	Add tests.	Must have	9
002	Meta Solver	No dependency injection framework is used.	Construction is separated from usage, but in case of a need to change implementation class or even jms queue address, implementation must be changed.	Use a dependency injection framework, e.g., Spring	Must have	10
003	App	No dependency injection framework is used.	Construction is separated from usage, but in case of a need to change implementation class or even jms queue address, implementation must be changed.	Use a dependency injection framework, e.g., Spring	Must have	10
004	Mapper	Too big responsibility of methods.		Clean code rules.	Must have	5, 7
005	APP	Daemon does not start properly.	The main method expects the keyword 'deamon' for all three arguments, but the method which starts deamon pass this keyword on only the first two arguments .	Correct it if it is a mistake. Unit tests will be very helpful with verifying number of arguments for a method.	Nice to have	10

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

006	Meta Solver	Not all classes start with an upper-case.	Java class names should start with upper-case.	Use Java naming standard.	Nice to have	10
007	Meta Solver	Method access modifier.	The rule to use method access modifier as restricted as possible is not followed.	Use more restricted access modifiers.	Nice to have	
008	CpModelTool	Lack of java.util usage.	When Java 8 is used, good practice is to use java.util. Optional for method result which can be empty instead of returning null.	Use java.util.	Nice to have	10
009	CpModelTool	Methods used only in unit tests.		Should be moved to test classes.	Nice to have	10
010	CdoTool	Commented code lines.		Remove them. "Remembering" old code is the version control system's responsibility, so source code should be just replaced, not copied.	Nice to have	10
011	most classes	Usage of System.out.print.	There are lots of lines of code where System.out.print is used to log. Especially in the Meta Solver module.	Use Logger with debug level. Print into standard output will be still possible through logger configuration.	Nice to have	10

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

012	APP	Unnecessary synchronized method.	Slowing down application.	Remove unnecessarily synchronized method.	Nice to have	10
013	old classes with extension .old are kept in project	Old classes with extension .old are kept in project		"Remembering" old code is the version control system's responsibility, so source code should be just replaced not copied.	Nice to have	10
014		Not used classes.	Changing app configuration without changing implementation of some classes is impossible. Also adding a new solver in the future will cause a need to change implementation of Meta Solver class.	To remove not used classes.	Nice to have	10

Table 19 presents code assessment results for the Cloudiator.

Table 19: Code assessment results for Cloudiator

Issue Id	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	Lack of unit tests	There is a test package, but no tests written within the Colloseum project.	Provide test scenarios.	Must have	9
002	Lots of TODOs	There are a lot of TODO comments in the code indicating that there is still some work that needs to be done within the code.	Implement TODOs.	Must have	10

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

003	Hardcoded values	There are some values set directly within the code instead of putting those values as configuration parameters, for example in a configuration file.	Move to properties file.	Nice to have	2, 10
004	Error descriptions	Error messages can be more meaningful. Now, in some cases, there are exceptions thrown, but no specific error description is logged.	Add more logging with meaningful descriptions .	Nice to have	10
005	Outdated version of Play framework used	Currently, Colloseum's code uses version 2.4 of the play framework, while the newest version released is 2.5.	Consider moving to the 2.5 version of the Play framework.	Nice to have	10
006	Inconsistency when using annotations	In most of the code, annotations are properly used, but there are still some places that can be improved.	Use annotations .	Nice to have	10
007	Unused parameters	In the application configuration file, the parameter colosseum.scalability.visor.telnet.port is never used.	Delete unused parameters.	Nice to have	10

Table 20 presents code assessment results for the component SRL adapter from the PaaSage project.

Table 20: Code assessment results for the component SRL adapter

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	Execution	Very long methods	Far too many lines of code and comments within a single method.	Separate responsibilities into classes and methods.	Must have	2

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

002	AbstractAdapter	Logger initialized in abstract class	Logger is initialized in the abstract class as getLogger (AbstractAdapter.class). When using it in implementation classes, it will always log AbstractAdapter.class as a source - not the actual implementation class.	Create a logger per implementation class. Use the lombok library and annotate class with @Log annotation[10].	Must have	5, 7
003	*Adapter	Too long methods	Methods above 15-20 lines of code.	Create several smaller methods with one responsibility and call it in adapt method.	Must have	2
004	*Adapter	No dependency injection framework used in the project	There is no dependency injection framework used in this project. All classes are tightly coupled.	Use spring framework.	Must have	10
005	*Adapter	Debugger specific code in production code		Remove debugger specific code from the production code.	Must have	10
006	CompositeMetricContextAdapter	Lots of TODOs	There are lots of TODOs in the code (there are classes were there are more than 10 TODOs). It looks like almost every implementation class is incomplete.	Do TODOs or remove them.	Must have	10

[10] https://projectlombok.org/features/Log.html

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

007	adapt method implementations	God classes	The *Adapter classes does everything to instantiate the adapter. They even check the number of parameters in the configuration. There is no clear separation of responsibilities there.	Split different functionalities into different class hierarchies, use DI container like Spring, and inject beans to class	Must have	2
008	FrontendCommunicator	Functionalities overloaded classes	There are functionally overloaded classes such as RestFrontendCommunicator - it literally does everything.	Refactor functionalities that overload classes.	Must have	1 ,2
009	RestFrontendCommunicator	Complex IF statements	There are complex IF statements that are hard to read and understand.	Create simple methods to compare and return result for two values - it will reduce the code. Encapsulate all the conditions into methods with informative names.	Must have	2, 10
010	CommandLinePropertiesAccessorImpl	Huge switch statements	There are huge switch statements that are not to be maintained.	Create enum with property names. Create method inside enum that reads properties according to the enum's value. Use enum values everywhere inside this class, not hardcoded string values.	Must have	1, 2

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

011		No unit tests in the project	The code in various methods is so complex that introducing unit tests in the current phase is almost an impossible task to accomplish.	The code needs to be refactored.	Must have	9
012	App	Switch statement instead of Factory pattern used	In the main App class there is a switch statement that should be easily refactored to a factory class.	Refactor to factory pattern.	Nice to have	10
013	AbstractAdapter	Redundant static {} scope	Static {} scope is redundant. Initialization of a logger should be done in a field definition.	Move initialization of logger to field definition.	Nice to have	10
014	BinaryEventPatternAdapter	Unused import packages		Remove unused imports.	Nice to have	10
015	BinaryEventPatternAdapter	Constant strings declared as constant variables	Constant variables are used inside methods - they should be declared as constant fields.	Using constants inside code is less maintainable. Using several instances of the same constant makes it redundant.	Nice to have	10

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

Table 21 presents code assessment results for the component Camel from the PaaSage project.

Table 21: Code assessment results for the component CAMEL

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	DeploymentSwitch	Complex switch statement	Unreadable switch statement because of its volume.	Each switch can be replaced with a method. Each method can be implemented in an Enum.	Nice to have	1, 2, 10
002	DeploymentPackageImpl	Big and complex initialization methods	Big methods are hard to read, understand and maintain.	Split into smaller methods.	Nice to have	1, 2
003	MetricFactoryImpl	God classes	There are god classes in the code that have too many functionalities, e.g. factory classes that instantiates other type of classes or make other actions, for instance "convertToString" methods.	Create other factories according to classes functionalities.	Nice to have	2
004	Every Enum class	get(int value) methods in enumerations	There is an unused field "value", which stores the enum ordering, and other static values to the same that are used in a method get(int value).	Use LayerType.ordinal()	Nice to have	10
005	ActionType	Creating array of all enum values inside the enum		Use ActionType.values()	Nice to have	10

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

006	DeploymentFactoryImpl	Complex switch statement		Use Enum class with a method that returns EObject. Implement every method in an enum value and just call Deployment PackageEnum value (eClass.getClassifier Id). getObject().	Nice to have	1, 2, 10
007	DeploymentPackageImpl	Big and complex initialization methods	Big methods are hard to read, understand and maintain.	Split methods to several sub-methods. In the shown methods, there are comments that group functionalities which should be replaced by methods.	Nice to have	1, 2
008	DeploymentAdapterFactory	Complex anonymous class initialization inside other class	Deployment AdapterFactory initializes DeploymentSwitch in a method. It has references to all the methods inside the DeploymentAdapterFact ory without any logic.	Replace it with a class extension.	Nice to have	10
009	DeploymentValidator	Validation messages inside the class with text formatting	Validation messages should be placed in property files or even in localized property files.	Move messages into property files.	Nice to have	10
010	LocationValidator	Code duplication	There are methods that duplicate code.	Create one method to replace the duplicated code, and use it.	Nice to have	3, 10

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

011	RequirementGroupImpl	Complex methods	There are a few complex methods that should be refactored into calls to several sub-methods.	Refactor.	Nice to have	1, 2, 10
012	RequirementValidator	Lack of self-explanatory patterns	There are places where a pattern should be used such as all validate* methods in utility classes **Validator.	Refactor.	Nice to have	10
013	Metric	Badly structured classes	In the package eu.paasage.camel.metric there are classes that can be easily moved to other packages to make structure more clear (for instance: Property, PropertyCondition, PropertyContext, and PropertyType could be moved to the package eu.paasage.camel.metric .property).	Create more concise packages.	Nice to have	1, 2, 10
014	LocationFactoryImpl	Redundant class casting	There are plenty of places (mostly switch statements) where return value is casted to EObject interface, but return class instances are extension of this interface.	Remove it and avoid using it in the future.	Nice to have	4, 10

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

Common quality assurance issues have been identified across all components, and are summarized in Table 22. Priorities assigned to each change are also provided in the table.

Table 22: Quality assurance issues and priorities assigned to each change

Issue Id	Class	Issue name	Issue description	Suggested resolution	Priority	Clean Code Principle ID
001	All components and respective classes	Lack of unit tests and improvement/ enhancement of existing ones	There are no unit tests or not sufficient unit tests in all components and classes.	Implement unit tests according to the rules and coverage level described in deliverable D5.6 "Test strategy and environment".	Must have	9
002	All components and respective classes	Lack of integration tests	There are no integration tests implemented in any components and classes.	Implement integration tests according to the rules and coverage level described in deliverable D5.6 "Test strategy and environment".	Must have	9

7 The components' fulfilment level

In this section, we present the summary of the fulfilment level per category for each selected component. The columns in Table 23 are as follows:

- Component – name of the framework and component.
- Functional req. level of fulfilment – Summary value of fulfilment of the functional requirements for a given component. Possible values are N, L, P, and F, as described below.
- Non-functional req. level of fulfilment – Summary value of fulfilment of the non-functional requirements for a given component. Possible values are N, L, P, and F, as described below.

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

- Code assessment fulfilment – Summary value of fulfilment of the code assessment principles for a given component. Possible values are N, L, P, and F, as described below.
- Required effort – general estimation of required effort to address all identified issues for a given component. Possible values are:
  - Low – below 20 man-days,
  - Medium – between 20 and 50 man-days,
  - High – above 50 man-days.
- Overall status – description of the overall status of component and indication of the plans for the component development.

Possible levels of fulfilment of functional, non-functional and code assessment requirements are:

- N – Not fulfilled,
- L – Low level of fulfilment,
- P – Partially fulfilled,
- F – Fulfilled

Table 23: Summary of fulfilment levels

Id	Component	Functional req. level of fulfilment	Non-functional req. level of fulfilment	Code assessment fulfilment	Required effort	Overall status
001	PaaSage / Adapter	L	L	L	High	Very low level of fulfilment of all types of requirements. Component needs to be rewritten
002	PaaSage / CP Generator	L	L	L	High	Serious changes needed related to functional and non-functional requirements. Component will be heavily rewritten
003	PaaSage / CAMEL	P	F	P	Low	Minor changes needed to be implemented

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

004	PaaSage / Meta Solver	P	P	P	Medium	Some functional changes need to be implemented. Also, some minor non-functional changes should be implemented
005	PaaSage / MILP Solver	P	P	F	Low	Some minor functional and non-functional changes need to be implemented
006	PaaSage / CP Solver	F	P	P	Low	Some minor functional and non-functional changes need to be implemented
007	PaaSage / LA Solver	P	L	F	High	Functional implementation needs to be finished. Some non-functional changes should be implemented
008	PaaSage / Solver to deployment	P	P	P	Medium	Minor functional and non-functional changes should be implemented
009	PaaSage / SRL adapter	P	P	P	Medium	Both functional and non-functional changes should be implemented

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

010	PaaSage / CDO Server	F	P	F	Low	Minor non-functional changes to implement
011	PaaSage / CDO Client	F	F	F	None	Nothing to change
012	PaaSage / Plan Generator	L	L	L	High	Component will be rewritten from scratch
013	Cloudiator (with all subcomponents)	P	P	F	Medium	Functional and non-functional changes should be implemented

8 Summary

In order to achieve its goals, Melodic builds on software components from earlier research projects, and in particular the PaaSage Upperware and the deployment and orchestration layer from PaaSage and Cloudiator (with the extensions created in the Cactos project). More specifically, we have identified 14 different components that are worth building upon.

This document presents a code quality assessment of the components from underlying projects which will be used in the Melodic project, and further elaborates the needed enhancement by Melodic in order to support the desired functionality (defined through functional requirements) and quality (defined through non-functional requirements and code quality). For each component, this document presents a detailed list of enhancements, spanning all three aspects, and will be considered in the planning of the project releases.

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

We use three different, but aligned, methodologies in order to identify required functional changes, non-functional changes, and code quality changes. In the functional evaluation, we identified the gaps of the existing components compared to the functional requirements of Melodic, and compiled them as a prioritised list denoting the enhancements and their priority as either "nice to have" or "must have". Finally, we assigned a Melodic release to each of the enhancements. This led to a total of 19 functional enhancements (16 "must have", two "nice to have", and one "other"); four of them will be tackled in the first release, eleven in the second release, and the other four in both releases.

For capturing non-functional enhancements, we apply the CUPRIMDSO quality model. This methodology covers aspects, such as Reliability, Scalability, Performance, etc. We have followed the same approach as for functional requirements, except that we have verified all non-functional aspects against all components. Furthermore, we allow a third category of prioritisation in addition to "must have" and "nice to have", which is "to test". This category means that the requirement is implemented, but should be carefully tested with respect to its level of fulfilment. The non-functional enhancement methodology has resulted in 15 non-functional features with the "must have" priority, 13 with the "nice to have" priority, and 13 with the "to test" priority.

With respect to code quality, each component was reviewed against best programming practices and quality assurance requirements following the Clean Code principles; this led to a list of 69 changes that were again classified as "must have" (41) or "nice to have" (28).

Each functional, non-functional or code assessment change is registered in the JIRA system, which is used for development process management in the Melodic project. Furthermore, each change has a dedicated story or is combined with other related changes into a single story.

The changes identified will be further enhanced during the project execution and will be integrated with other changes or mapped to new features, e.g., to the security & UI component(s) in the Melodic platform.

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

References

[1] Y. Verginadis *et al.*, "D2.2 Architecture and Initial Feature Definitions", The Melodic H2020 Project Deliverable D2.2, 2018.
[2] Y. Verginadis *et al.*, "D2.1 System Specification", The Melodic H2020 Project Deliverable D2.1, 2017.
[3] A. Schwichtenberg *et al.* "D5.06 Test Strategy and Environment", The Melodic H2020 Project Deliverable D5.06, 2018.
[4] S. H. Kan, "*Metrics and Models in Software Quality Engineering*", 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[5] "ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models." [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. [Accessed: 01-Mar-2018].
[6] Antonia Schwichtenberg *et al.*, "D5.04 Integration & testing requirements", The Melodic H2020 Project Deliverable D5.04, 2018.
[7] Y. Verginadis *et al.* "D3.1 Metadata Schema Management", The Melodic H2020 Project Deliverable D3.1, 2018.

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