Melodic
Big data cloud

Title:

# Workload optimisation recommendation and adaptation enactment

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:
D3.4

Date:
31 January 2019

Responsible partner:
ICCS

Editor(s):
Yiannis Verginadis

Author(s):
Yiannis Verginadis, Christos Chalaris, Ioannis Patiniotakis, Vasilis Stefanidis, Fotis Paraskevopoulos, Evgenia Psarra, Babis Magoutas, Efthimios Bothos, Evagelia Anagnostopoulou, Kyriakos Kritikos, Paweł Skrzypek, Marcin Prusiński, Marta Różańska

Approved by:
Gregoris Mentzas

ISBN number:
N/A

Document URL:
http://www.melodic.cloud/deliverables/D3.4 Workload optimisation recommendation and adaptation enactment.pdf

Abstract:

This is a report accompanying the initial software release of the following core components of the Melodic Upperware: *CP Generator, Metasolver, CP Solver, Utility Generator, Solver to Deployment, Adapter* and *Event Management System*. These Upperware components refer to Melodic's substantial functionalities that enable the application optimisation recommendation, the initial application placement and the continuous adaptation enactment. Based on these features the Melodic platform is able to offer timely decision-making on optimised cross-cloud data placements and application deployments.

Specifically, in this deliverable we provide a detailed description concerning the approach, the business logic and the implementation details of each of the aforementioned components. The integration of these components constitutes a significant part of the Melodic platform release 2.0, which is being tested and evaluated in the following months, as part of the project's pilot demonstrators.

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

| Document | |
|---|---|
| Period Covered | M4-26 |
| Deliverable No. | D3.4 |
| Deliverable Title | Workload optimisation recommendation and adaptation enactment |
| Editor(s) | Yiannis Verginadis |
| Author(s) | Yiannis Verginadis, Christos Chalaris, Ioannis Patiniotakis, Vasilis Stefanidis, Fotis Paraskevopoulos, Evgenia Psarra, Babis Magoutas, Efthimios Bothos, Evagelia Anagnostopoulou, Kyriakos Kritikos, Paweł Skrzypek, Marcin Prusiński, Marta Różańska |
| Reviewer(s) | Feroz Zahid, Gregoris Mentzas |
| Work Package No. | 3 |
| Work Package Title | Upper ware |
| Lead Beneficiary | ICCS |
| Distribution | PU |
| Version | 2.9 |
| Draft/Final | Final |
| Total No. of Pages | 55 |

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

# Table of Contents



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

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

# List of Tables



# List of Figures



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

# 1 Introduction

This document is intended for the general audience interested in learning about the core components of the Melodic Upperware. Parts of the document require a high-level understanding of the Cloud technologies and the Melodic platform, for which readers are referred to [1].

Specifically, in this deliverable, we discuss the approach, the design and the implementation of a number of core components of the MELODIC platform, which are mainly related to the application optimisation recommendation, initial application placement and continuous adaptation enactment. Essentially, we are referring to the majority of the Upperware components as defined in [1] and depicted in Figure 1, with the exception of the Data Lifecycle Management (DLMS) component which has been presented in its separate deliverables [2], [3] and the LA Solver which is currently under finalization and it is scheduled to be reported in terms of the D3.5 deliverable.

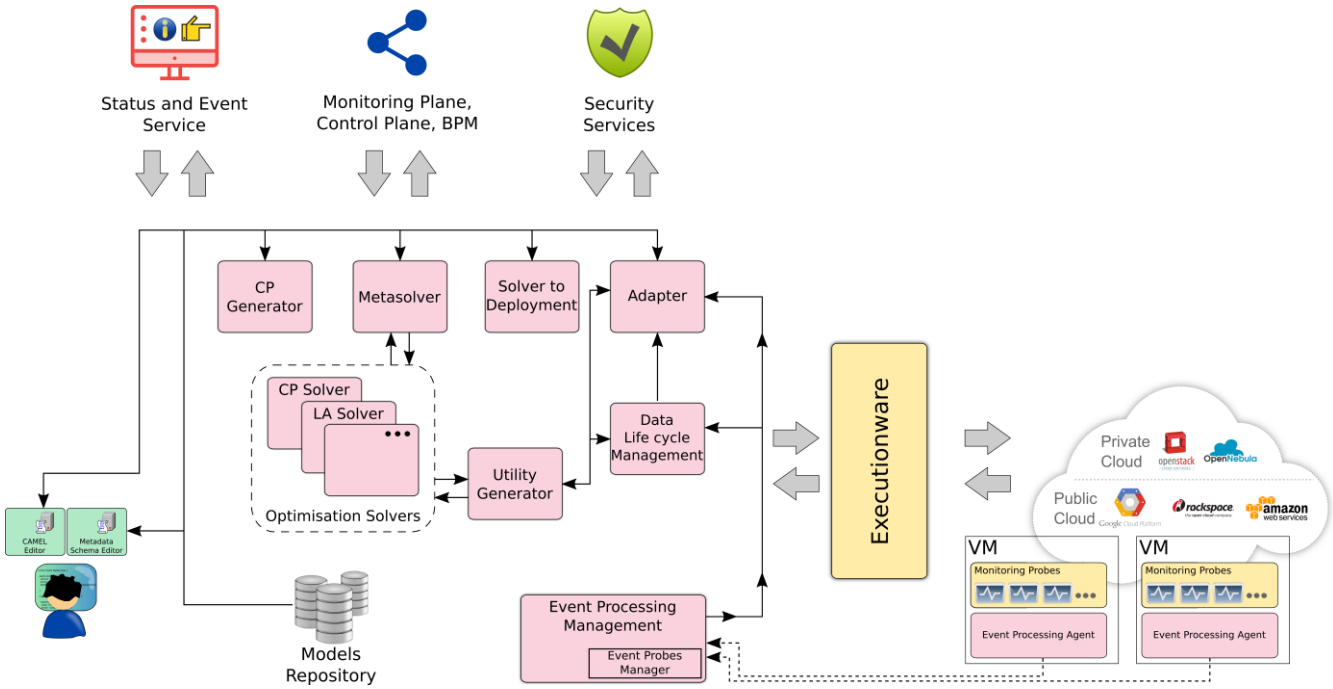


*Figure 1. Overview of the Upperware Components*

As depicted in Figure 1, the Upperware comprises a number of software components that encapsulate all the necessary functionality for making timely decisions on appropriate cross-cloud data placements and application deployments. Specifically, in this deliverable, we present the design and implementation details of the following components:

- CP Generator – for generating a formal Constraint Problem (CP) out of a set of a cloud application placement requirements

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

- Metasolver – for coordinating and supporting the CP solving process and deciding when reconfigurations are required
- CP Solver - for finding, in a stateless manner, optimal cross-cloud resources allocation and application placement according to a set of pre-defined requirements captured as a CP
- Utility Generator – for calculating the utility function value for the deployment solutions proposed by the optimisation solver configuration
- Solver to Deployment – for instantiating the deployment models according to an accepted CP solution in order to allow the deployment process to continue
- Adapter – for creating the target application configuration to be deployed into cross-cloud resources and relaying appropriate instructions to the Cloudiator [4]
- Event Management System – for collecting, processing and delivering monitoring information pertaining to a cross-cloud application, deployed and maintained by the Melodic platform

Each of these Upperware components are discussed in separate chapters of this deliverable by presenting the core aspects of the approach considered for each of them, along with their business logic and technical implementation details.

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

# 2 CP Generator

**Mission**: Create Constraint Problem (CP) to be solved by any of the Solvers.

**Positioning in Melodic**: CP Generator is one of the microservices comprising the UpperWare of the Melodic platform.

## 2.1 Approach

**High-level Approach**: The CP Generator is responsible for creating the Constraint Problem (CP) Model based on the provided CAMEL Model and a set of Node Candidates fetched from Cloudiator according to the hard requirements described in CAMEL.

**Functionalities**:

- Creating hard requirements in order to fetch matching Node Candidates from Cloudiator
- Storing fetched Node Candidates in Cache for further use
- Creating Constraint Problem based on requirements from CAMEL Model and border values of corresponding fields of Node Candidates
- Storing Constraint Problem Model to CDO
- Sending Success or Failure Notification in case of any errors

**Input**:

- CAMEL model
- Node Candidates fetched from Cloudiator

**Output**:

- Constraint Problem Model stored in CDO
- Node Candidates stored in Cache
- Notification sent to the control process

## 2.2 Business Logic (Creating Constraint Problem process)

First of all, the CAMEL Model is loaded from CDO. Then the Software Components are retrieved from the first Deployment Model and based on them and their requirements, the node candidates (i.e. a set of acceptable resources available) are fetched from Cloudiator separately for each component, grouped by provider name. In this form all the received node candidates are stored in cache. The following requirements are considered when looking for a new set of node candidates:

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

- Resource Requirements
  - RAM min, RAM max
  - Cores min, Cores max
  - Disk min, Disk max
  - CPU min, CPU max
- Location Requirement
- Image Requirement
- OS Requirement
- Provider Requirement
- Node Type Requirement (IAAS, Function as a Service (FAAS))

After all node candidates have been successfully fetched, the generation of the CP Model is started. For each Software Component defined in CAMEL, the CP Model is filled with "hard" variables and their restrictions, as follows:

- Required Variables
  - Variable of type CARDINALITY calculated based on Horizontal Scale Requirement or default values are set if missing (0 as a minimum number of instances, 100 as a maximum number of instances)
  - Variable of type PROVIDER calculated based on previously loaded Node Candidates for the actual component
- Optional variables (these variables are calculated only if there is an attribute annotated by special annotation, indicating that this variable should be created). All these variables are created not only for each Software Component, but also for each Provider from the corresponding Node Candidate set. Each variable is prefixed with a function (Provider, 1)=1 <=> (Provider==1) which ensures that one and only one constraint for each variable type will be fulfilled.
  - Variable of type CORES with domain calculated based on Node Candidate hardware → cores field
  - Variable of type RAM with domain calculated based on Node Candidate hardware → ram field
  - Variable of type STORAGE with domain calculated based on Node Candidate hardware → disk field

After that, the CP Model is enriched with Metrics created on the CAMEL Model:

- Raw Metrics
- Composite Metrics
- Traditional Attributes - variables which are both part of current configuration and marked with annotation from "hard requirement" group.

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

Last, the Constraints are created based on Metric Variable Constraints from the CAMEL's Constraint Model. This step completes the generation of the CP model. After that, the CP Model is stored in CDO and a successful notification is sent back to the control process in order to allow for the flow to be passed to the next component (Figure 2).

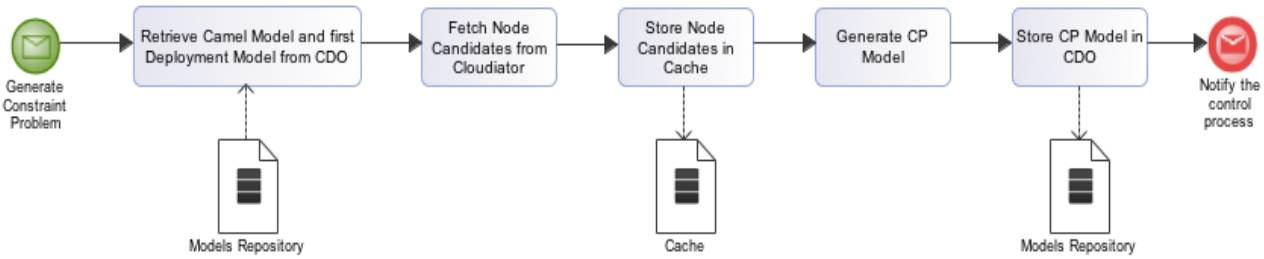


*Figure 2. Generate Constraint Problem BPMN process*

## 2.3 Technical Implementation

### 2.3.1 Architecture

As shown in Figure 3, the CP Generator comprises the following sub-components:

- Controller - provides the REST API of the CP Generator for receiving process calls from the control layer
- Generator Orchestrator - orchestrates the creation of a new Constraint Problem
- CDO Service - retrieves the CAMEL Model from the Models repository and stores the created CP Model to the Models Repository
- New Constraint Problem Service - creates a new Constraint Problem based on information from CAMEL and the fetched Node Candidates
- Notification Service - notifies the control process
- Cloudiator Service - fetches the Node Candidates fulfilling the requirements
- Cache Service - creates a Melodic cache with a map of fetched Node Candidates for each component

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

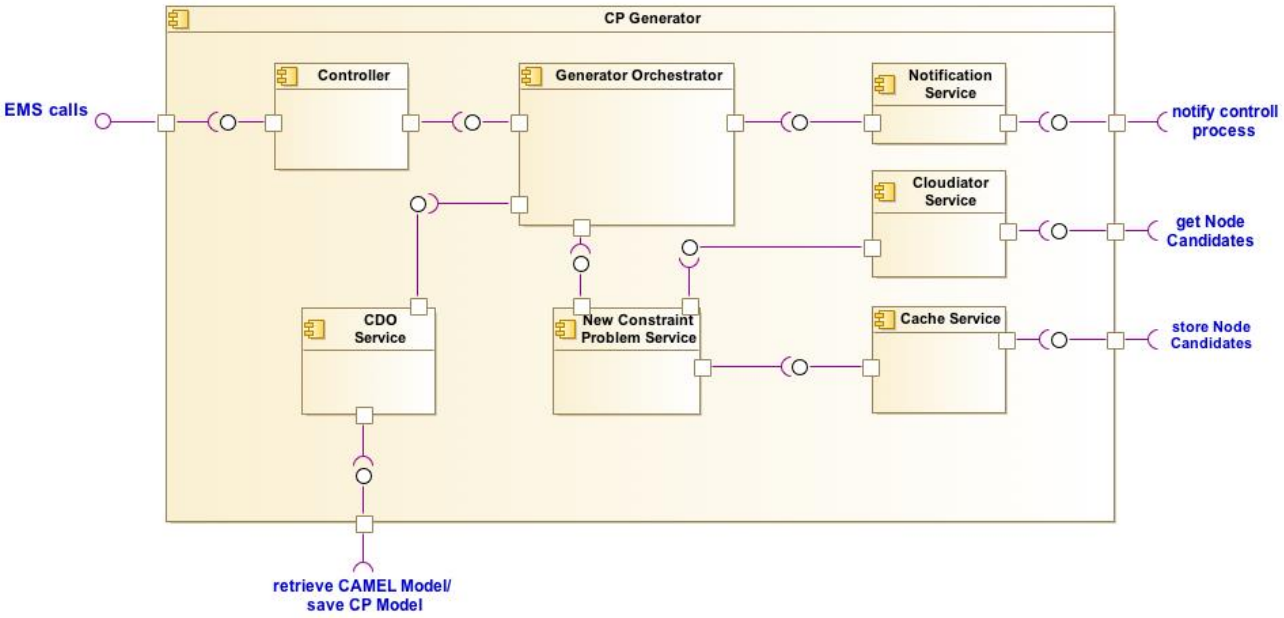


*Figure 3. CP Generator Component diagram*

### 2.3.2 Implementation

The CP Generator has been written in Java 8. The project is built using Maven and thanks to the Maven plugin[1], a Docker image is created which allows running this component as a separate microservice.

The following packages are involved in the implementation of the CP Generator:

**upperware.profiler.generator**: this package is responsible for correctly running the CP Generator as a Spring Boot application.

**upperware.profiler.generator.communication**: this package is responsible for the communication with external systems (i.e. CDO, Cloudiator).

**upperware.profiler.generator.error**: this package is responsible for handling errors dedicated to CP Generator.

**upperware.profiler.generator.notification**: this package is responsible for sending back notifications.

[1] `com.spotify:docker-maven-plugin`

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

upperware.profiler.generator.orchestrator: this package is responsible for orchestrating requests and synchronizing all the involved internal operations.

upperware.profiler.generator.properties: this package is responsible for handling external properties.

upperware.profiler.generator.result: this package is responsible for creating notifications.

upperware.profiler.generator.service.CAMEL: this package is responsible for dealing with the CAMEL Model and CP Model.

upperware.profiler.generator.service.CAMEL.creator: this package is responsible for generating Constraint Problem variables.

upperware.profiler.generator.service.CAMEL.parser: this package is responsible for parsing the utility expression.

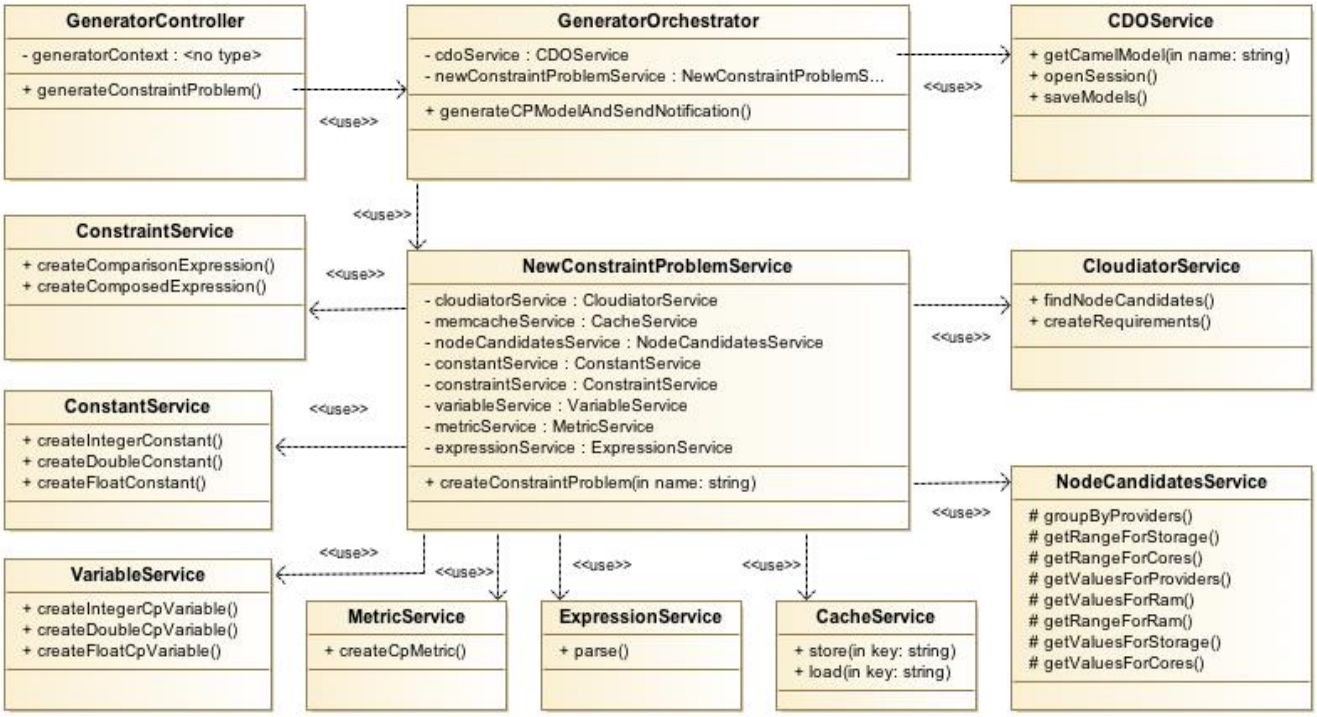


*Figure 4. CP Generator Class diagram*

The CP Generator's source code is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/cp_generator

Dependencies: CDO Client, Melodic cache, Cloudiator Client, MathParser, Spring-boot framework, JWT commons, Melodic commons

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

# 3 Metasolver

**Mission:** Coordinate and support the Constraint Problem (CP) solving process.

**Positioning in Melodic:** Metasolver is one of the microservices comprising the UpperWare of the Melodic platform.

## 3.1 Approach

**High-level Approach:** The Metasolver undertakes the task of selecting an appropriate solver for a given CP problem and subsequently verify that the solution yielded by this solver is significantly better than the currently deployed one. A configurable threshold is used for defining how much better the new solution should be in order to be deployed.

**Functionalities:**

- Select a solver given a CP model.
- In case an initial placement has already been realized, update the CP model with the most current metrics values from the application monitoring infrastructure (EMS – see also chapter 8), as well as with the most recently deployed solution (i.e. description of application deployment topology) from the Adapter (see also chapter 7).
- Evaluate a given CP solution, provided by the selected solver and *accept* or *reject* it, by comparing it to the currently implemented solution (if any). This comparison is based on the utility values of the two solutions.
- Receive events signalling SLO violations from the application monitoring infrastructure (EMS) and trigger a new reconfiguration process.

**Input:**

- CP model
- Monitoring information in the form of metric values from events relayed through EMS
- Reconfiguration Events, signalling SLO violations

**Output:**

- Solver selected for solving a given CP problem
- Acceptance or Rejection of a new solution (yielded by a solver)
- Signal the start of a new reconfiguration process (in case the new solution is better than the current one)

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

## 3.2 Business Logic (Metasolver Processes)

The aforementioned functionalities are mapped onto concrete processes as indicated in the following Table 1.

*Table 1: Functionality to Process mapping*

| Functionality \ Process | Solver Selection | Solution Evaluation | Solution Deployment Notification | Event Subscription Configuration | Metric Value Update | Application Reconfigu-ration |
|---|---|---|---|---|---|---|
| Select solver | ✓ | | | | | |
| Evaluate CP solution | | ✓ | | | | |
| Update CP model | ✓ | | ∗ | ∗ | ✓ | |
| Trigger application reconfiguration | | | | ∗ | | ✓ |

The *Event Subscription Configuration* and *Solution Deployment Notification* processes do not directly fulfil any individual functionality, but they facilitate other processes to achieve their objectives. The objectives facilitated and indirectly achieved, are marked with an asterisk (∗). Specifically, the former process enables Metasolver receiving events from EMS, while the latter updates the CP model when a new solution gets deployed (i.e. it becomes the new *current* solution).

Below we discuss the details of each of the processes mentioned in Table 1 while depicting the tasks involved as a set of UML BPMN diagrams.

Solver Selection process

Upon request by the Upperware control process, the Metasolver selects a suitable solver for solving the application Constraint Programming (CP) problem, captured as a CP model. Specifically, as depicted in Figure 5, it:

- Retrieves the application CP model, generated by the CP generator service of Upperware.
- Updates the metrics (in the retrieved CP model) with the most recent values from the in-memory Metric Value Registry.
- Notifies EMS about the metric value updates in the CP model.
- Selects a stateless solver in order to solve the CP problem and notifies the Upperware Control process. Currently only "CP solver" is considered.

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

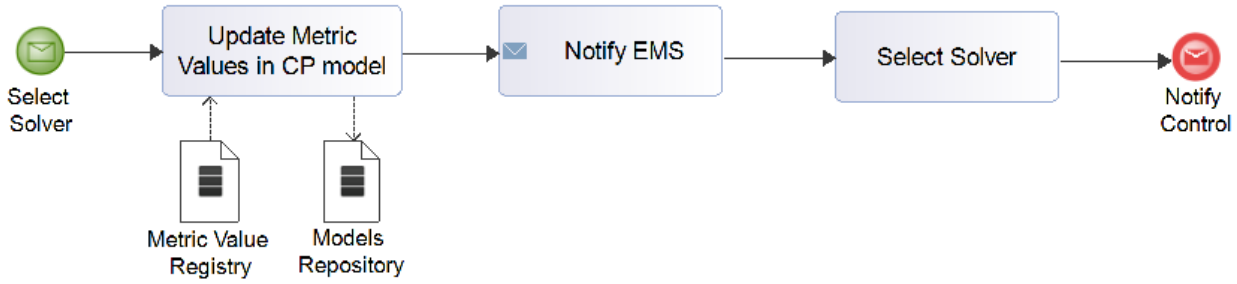


*Figure 5. Solver Selection BPMN process*

## Metric Value Update process

When Metric Value events are received from the Event Broker, Metasolver extracts the corresponding metric name and value, and stores them in an in-memory Metric Value Registry, updating any previous value (Figure 6).

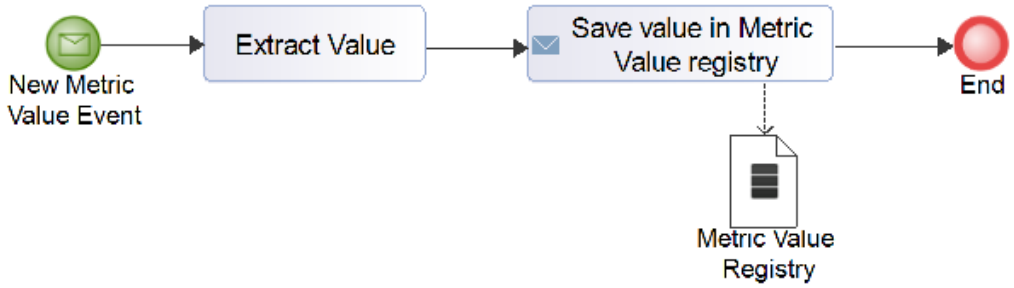


*Figure 6. Metric Value Update BPMN process*

## Solution Evaluation process

Upon request by the Upperware control process, Metasolver evaluates a given CP solution, captured in a CP model (by the selected Solver), and indicates whether it should be realized as the new application VM deployment topology or not (i.e. accept or reject it). This resolution is based on the comparison of the utility value of the calculated (new) solution to the utility value of the currently deployed solution. In order to prevent "frequent" application reconfigurations, Metasolver requires that a new solution must have a utility value better (higher) than the utility value of the currently deployed solution by at least a preconfigured percentage (e.g. 10% or better). Obviously, during the application's first deployment there will be no previously deployed solution and therefore any new solution provided by the Solver will be accepted.

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

The exact steps taken during the new solution evaluation are depicted in Figure 7 and specified next:

- Acquires the application CP model from Models Repository, and extracts the new CP solution, as well as the currently deployed one.
- Compares the two solutions in terms of their utility values.
- If the new solution is significantly better than the current one, or if there is no current solution, then:
    - The Upperware Control process is notified to continue with the deployment of the new solution, thus realizing the application deployment reconfiguration,

  Otherwise, the Upperware Control process is notified about the rejection of the new solution.

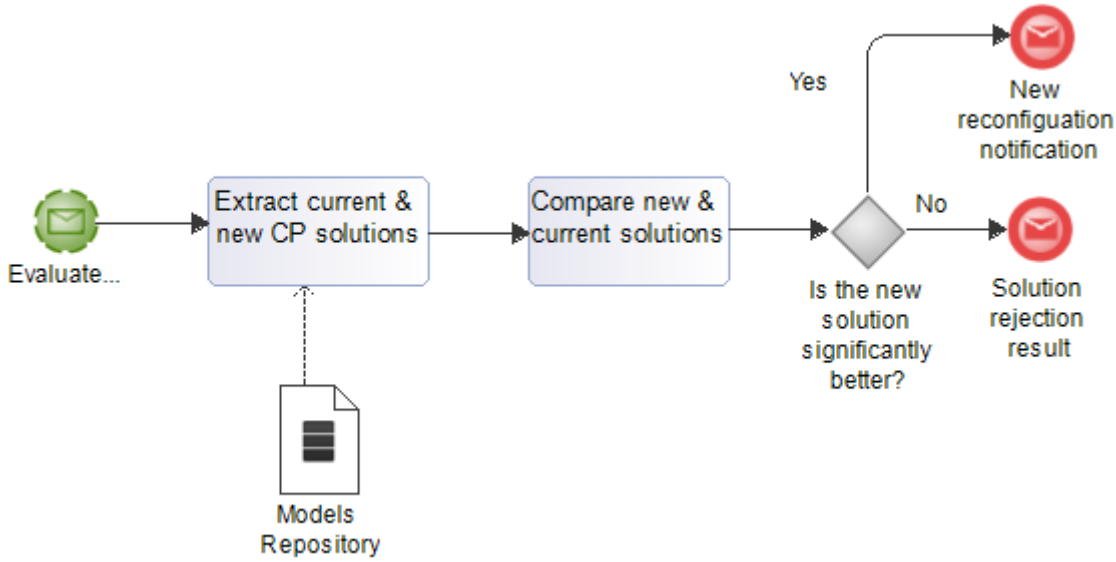


*Figure 7. Solution Evaluation BPMN process*

Solution Deployment Notification process

Upon notification by the Upperware control process, Metasolver updates the CP model with the most recently deployed solution. Thus, the newly deployed solution is marked as the *current* solution in CP model. Specifically, as depicted in Figure 8, it:

- Retrieves the application CP model, which contains the deployed solution.
- Marks the deployed solution as *current* and stores the updated CP model back in the Models Repository.

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

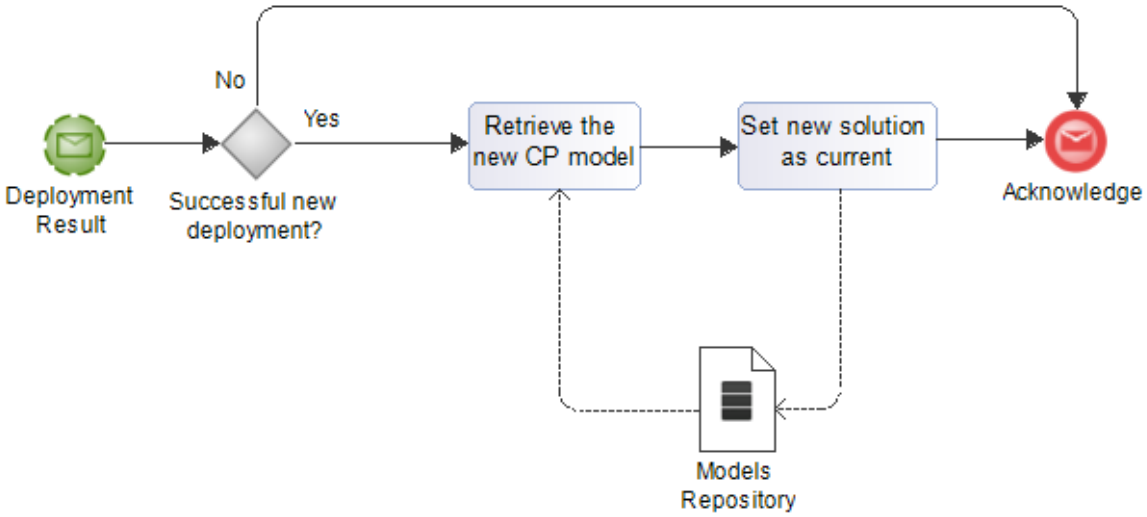


*Figure 8. Solution Deployment Notification BPMN process*

## Event Subscriptions Configuration process

Upon request by EMS, Metasolver receives a new event subscription configuration and applies it. Specifically, as depicted in Figure 9, it:

- Receives a new configuration (by EMS) regarding the event topics it must subscribe to. This configuration also indicates the event broker where each topic has been registered to, as well as whether each event topic provides *Metric Value* events or *SLO violation* events. Specifically, two types of events are considered:
  - *Metric Value* events, which convey the most recent / updated values of certain metrics, based on the measurements of sensors in VMs hosting application components;
  - *SLO violation* events, which signal that the application deployment topology needs to be reconfigured. For instance, when an SLO has been violated the application must be reconfigured in order to cope with this violation.
- Unsubscribes from any old event topics that it has previously been subscribed to.
- Subscribes to the configured new event topics and starts receiving relevant events.

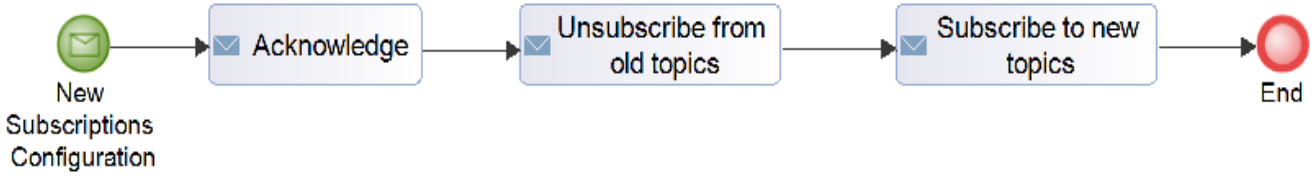


*Figure 9. Event Subscriptions Configuration BPMN process*

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

Application Reconfiguration process

As shown in Figure 10, when SLO violation events are received from the Event Broker, Metasolver notifies the Upperware Control process to start a new iteration of application reconfiguration (i.e. to start solving the Constraint Problem from the top).

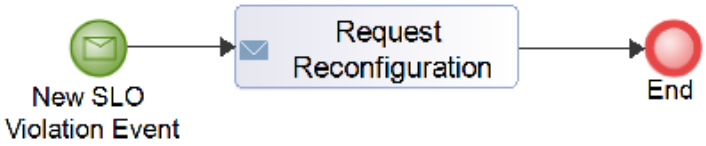


*Figure 10. Event Subscriptions Configuration BPMN process*

## 3.3 Technical Implementation

In this subsection, all the technical details on the Metasolver implementation are discussed (e.g. language, frameworks, 3rd party libraries, code structure etc.).

### 3.3.1 Architecture

A high-level depiction of the Metasolver architecture is given through the following UML component diagram (Figure 11).

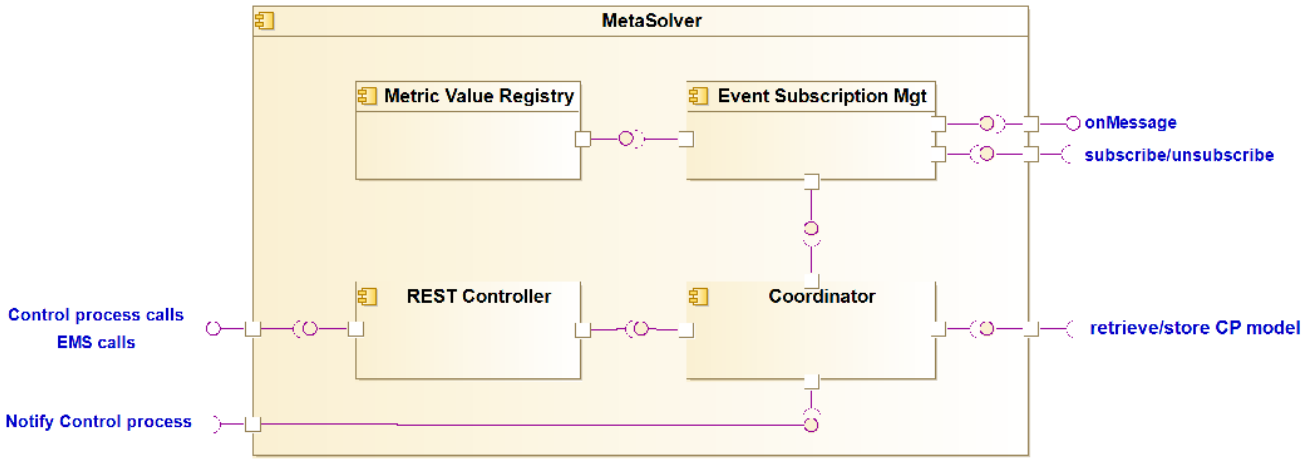


*Figure 11. Metasolver Component diagram*

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

As shown in Figure 11, the Metasolver comprises the following sub-components:

- **REST Controller:** provides the REST API of the Metasolver for receiving process calls from the control layer (see [5] for more details)
- **Coordinator:** orchestrates the functioning of the whole component.
- **Event Subscription Management:** subscribes to the configured event topics registered in the Event Broker of EMS. It is also responsible for unsubscribing when the configuration changes or when Metasolver shuts down. It uses the Broker Client library provided by EMS in order to communicate with Event Broker and receive events.
- **Metric Value Registry:** maintains an in-memory catalogue of metric values. The values are extracted from the corresponding events, received from the Event Broker. When the Metasolver is requested to select a solver (for application reconfiguration), it first updates the corresponding CP model with the most recent metric values.

The Metasolver internally uses an instance of the CDO client in order to communicate with the Upperware Models Repository and retrieve or modify the application CP model. Moreover, it uses an instance of the Broker Client in order to communicate with Event Broker of EMS and subscribe to event topics for receiving events. Both CDO and Broker clients are imported as dependencies from other Melodic-platform modules, namely *CDO client* and *Broker Utils* of EMS.

The following Table 2 indicates the Metasolver sub-components involved in each process.

*Table 2: Sub-component to Process mapping*

| Process / Sub-component | Solver Selection | Solution Evaluation | Solution Deployment Notification | Event Subscription Configuration | Metric Value Update | Application Reconfigu-ration |
|---|---|---|---|---|---|---|
| REST API Controller | ✓ | ✓ | ✓ | | | |
| Coordinator | ✓ | ✓ | ✓ | | | ✓ |
| Event Subscription Mgmt. | | | | ✓ | ✓ | |
| Metric Value Registry | ✓ | | | | ✓ | |
| CDO client (external) | ✓ | ✓ | ✓ | | | |
| Broker client (external) | | | | ✓ | ✓ | ✓ |

### 3.3.2 Implementation

Metasolver has been implemented using the Java programming language, version 8. It has been developed as a Spring-boot application for making its maintenance quite straightforward. It is built and bundled, using the well-known Maven system, into a single fat JAR, containing

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

Metasolver classes and dependencies. It is also bundled (during its building with Maven) as a Docker container and subsequently been added in the Melodic-platform component swarm.

In Figure 12, we provide further implementation details, using a UML class diagram.

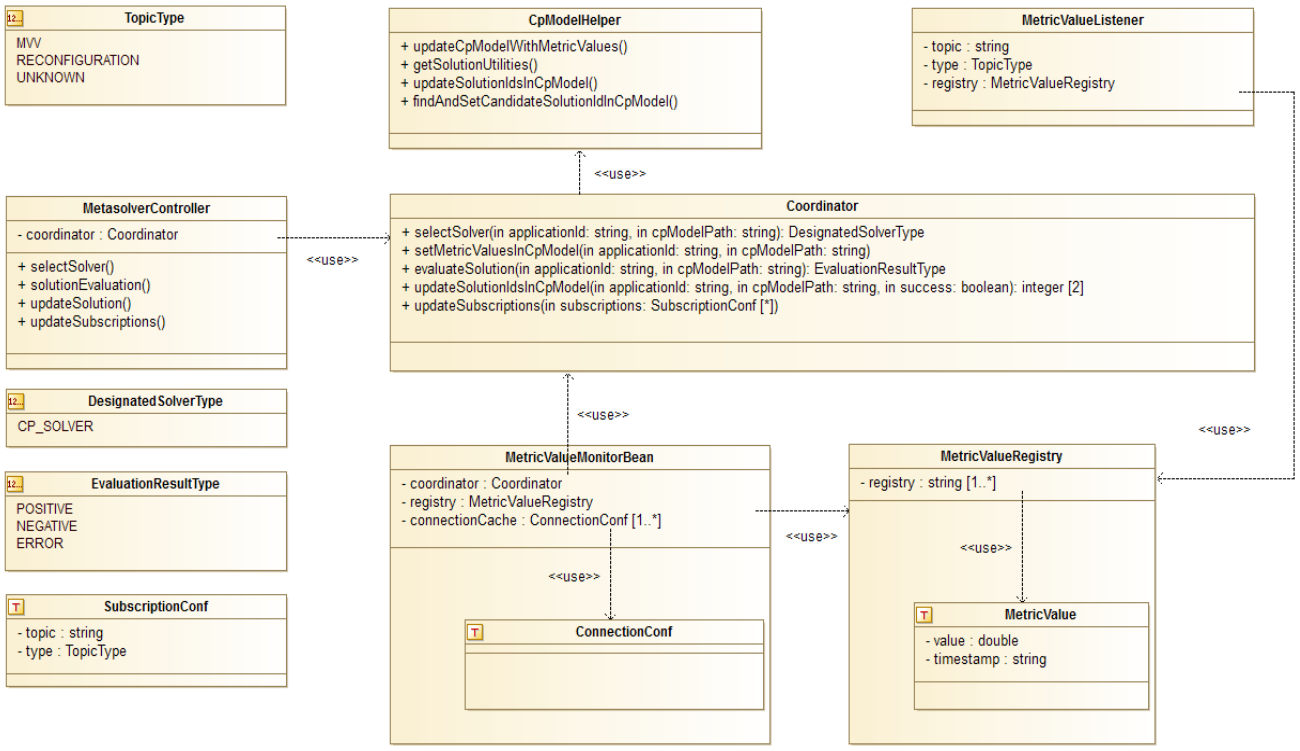


*Figure 12. Metasolver Class diagram*

The main classes of the Metasolver code are depicted in Figure 12 and briefly presented next. Only the most important of their methods and fields are explained and depicted.

**MetasolverController:** This class corresponds to the REST API Controller sub-component and it is the "entry point" for the functionality of Metasolver. Its mission is to accept REST calls from the Control process or EMS and extract the information relevant to the Metasolver. Subsequently this information is passed to the Coordinator sub-component to carry out the actual processing. For this purpose the MetasolverController class has a reference to Coordinator through the field "coordinator". Moreover, it offers four public methods that implement the REST API endpoints; namely selectSolver(), solutionEvaluation(), updateSolution() and updateSubscriptions().

**Coordinator:** This class corresponds to the Coordinator sub-component and is responsible for performing the processing requested through REST API Controller. For this purpose, it calls other classes and combine their results. It offers five methods to MetasolverController; namely selectSolver(), setMetricValuesInCpModel(), evaluateSolution(), updateSolutionIdsInCpModel()

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

and updateSubscriptions(). Coordinator is also responsible to notify the Control process of Upperware to start a new reconfiguration iteration when an SLO violation event is received (see MetricValueListener below).

**MetricValueRegistry:** This class implements an in-memory catalogue containing the latest values of the monitored metric events. Its values are updated with the reception of a new event. It includes a HashMap field called "registry," which provides the actual catalogue. The metric value (extracted from events) are stored along with a timestamp. For this reason, the inner class "MetricValue" is defined. Therefore, "registry" stores pairs of Metric names and MetricValue instances.

**MetricValueMonitorBean:** It implements the Event Subscription Management sub-component. It retains references to the Coordinator sub-component, to MetricValueRegistry and an internal connection cache. It provides methods to subscribe to and unsubscribe from topics in the event broker. For each subscription, it retains the relevant connection information as an instance of ConnectionConf in the internal connectionCache. The ConnectionConf instances contain information like the address of the corresponding event broker and the sessions opened to it.

**MetricValueListener:** It is also part of the Event Subscription Management sub-component. An instance of this class is created for every subscription and is responsible for receiving events from EMS Event Broker, extracting their values and updating the MetricValueRegistry. In case of SLO violation events it notifies Coordinator.

**CpModelHelper:** It is an auxiliary class that provides methods for accessing and updating a specified CP model. It offers the following methods; updateCpModelWithMetricValues(), getSolutionUtilities(), findAndSetCandidateSolutionIdInCpModel() and updateSolutionIdsInCpModel().

**Metasolver's source code** is available in Bitbucket at: https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/meta_solver?at=refs%2Fheads%2FRC2.0

**Dependencies:** Broker-Client library, CDO client, Spring-boot framework

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

# 4 CP Solver

**Mission:** Solve a CP model after it has been generated by the CP Generator or updated by the Metasolver, in cooperation with the Utility Generator.

**Positioning in Melodic:** The CP Solver, part of the Upperware module, is available in a micro-service form.

## 4.1 Approach

**High-Level approach:** The CP Solver is one of the solvers available in the Melodic platform utilised for solving a CP model. Such a solver can be used both for performing initial application deployment reasoning as well as application redeployment reasoning. Once the CP model is received, it is transformed into an internal representation which is fed into the CP solving engine. During CP model solving, the CP Solver cooperates with the Utility Generator in order to compute the utility of the currently examined candidate solution. Once the CP model solution is produced, it is incarnated inside the CP model in a certain specialised part.

**Functionalities:**

- Solves a CP model
- Cooperates with the Utility Generator during the CP model solving
- Registers the discovered solution within the CP model manipulated

**Input:**

- CP Model (path to that model within the CDO Model Repository)

**Output:**

- CP Model enhanced with the solution computed

## 4.2 Business Logic

The aforementioned functionalities are mapped to a single process, the CP model solving process, which is depicted in Figure 13. This process performs the following steps:

1. Get CP Model based on its (CDO) Model Repository path given as input
2. Transform the CP model into internal representation of the constraint problem based on the current CP solving engine exploited (Choco solver[2]). This internal representation

[2] www.choco-solver.org

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

concerns an in-memory Java object that represents the constraint problem which contains other solver-specific elements (like constants, constraints and variables).

3. If the problem is infeasible then notify solution infeasibility.
4. Compute optimal solution
    a. Compute first candidate solution. If the problem is infeasible, notify:
    b. Otherwise:
        i. invoke Utility Generator to produce the current candidate solution's utility
        ii. Get next candidate solution (by also updating the CP model to include an additional constraint indicating that the utility should be greater than that found for current candidate solution)
            1. if no new solution is found, then previous solution is optimal. It is written back to (CDO) Models Repository and a respective notification is produced
            2. else go to 3.b.i

The above process, as depicted in Figure 13, incorporates a sub-process named Compute Optimal Solution, which is depicted in Figure 14. This sub-process maps to part 4 in the above description.

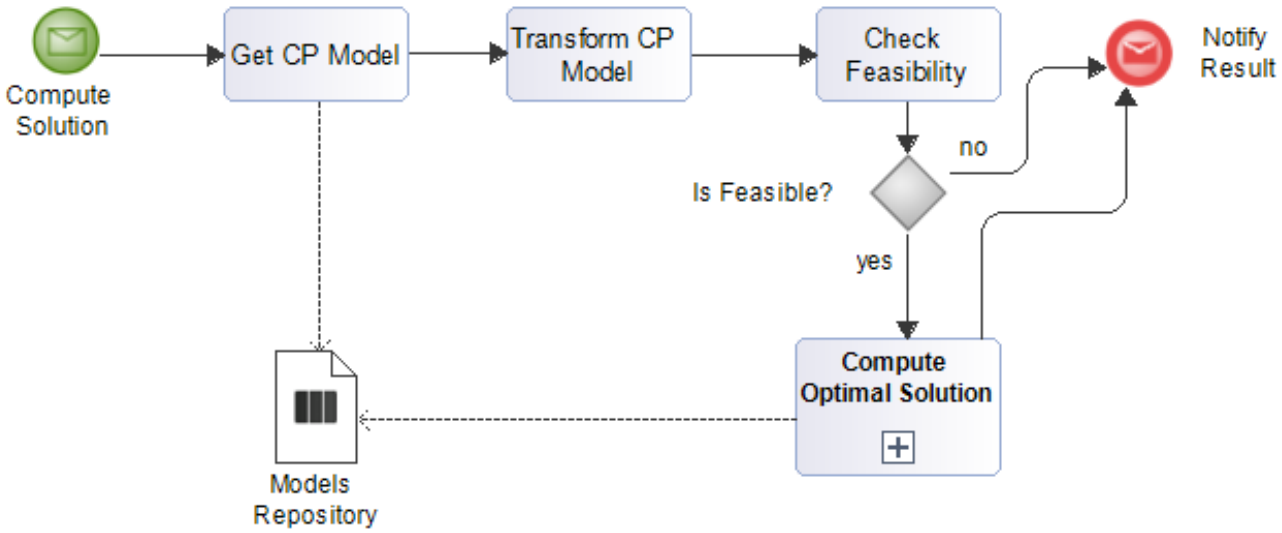


*Figure 13. The overall CP solving process*

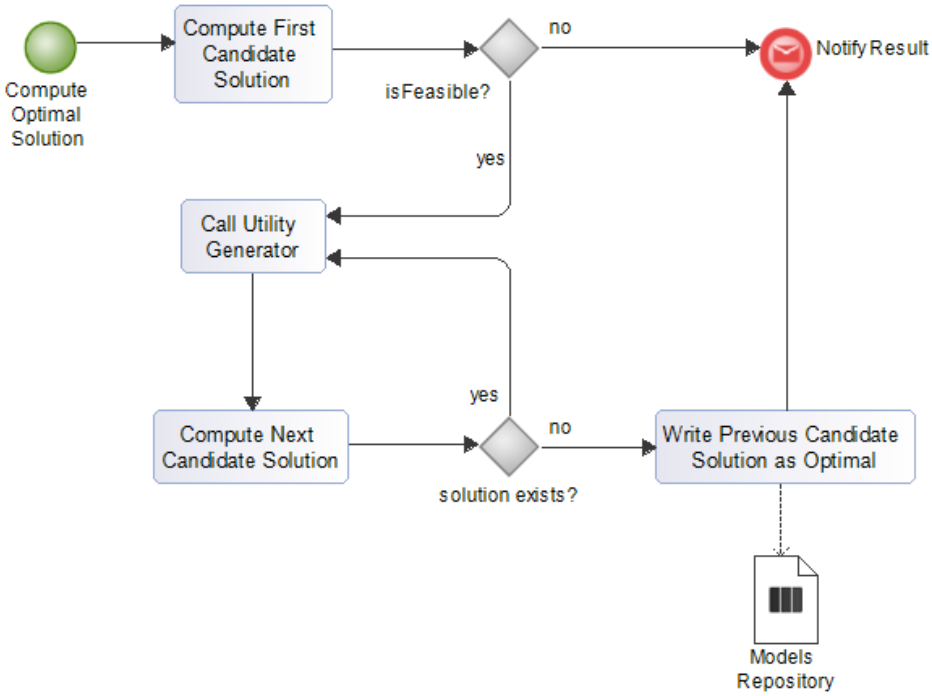


*Figure 14. The optimal solution computation sub-process*

## 4.3 Technical Implementation

In this subsection, the technical details of the CP Solver's implementation are discussed.

### 4.3.1 Architecture

The internal architecture of the CP Solver is shown in the component diagram of Figure 15. As it can be seen, this component comprises the following three sub-components:

- CP Solver: this sub-component is responsible for computing the optimal solution to a CP model and storing that solution, if it exists (i.e., the problem is feasible), in the (CDO) Models Repository. The whole functionality is encapsulated in the form of a single method called *solve* that takes no input parameter (as all relative parameters are given as input to this sub-component/class constructor) and returns as a result a boolean parameter indicating whether an optimal solution has been produced or not (i.e. if the problem is infeasible).
- CP Solver Executor: this sub-component is responsible for manipulating the CP Solver component (i.e., utilise it to compute the optimal solution) and notify back the (successful or unsuccessful) result produced. This whole functionality is encapsulated in the form of one method called *generateCPSolution*. This method does not return any result and takes

as input the parameters relevant for the CP model solving (such as the application ID, the path to the CDO Models Repository where the CP model is situated and the callback URI (part) for the notification).

- REST Controller: this sub-component encapsulates the solving process interface (comprising one core method called *applySolution*) in the form of a REST service. The sole method realised does not return any result and takes as input a ConstraintProblemSolutionRequestImpl object which encapsulates contextual information from the call that is then applied over the sole CP Solver Executor sub-component's method in the form of its input parameters (see above description).

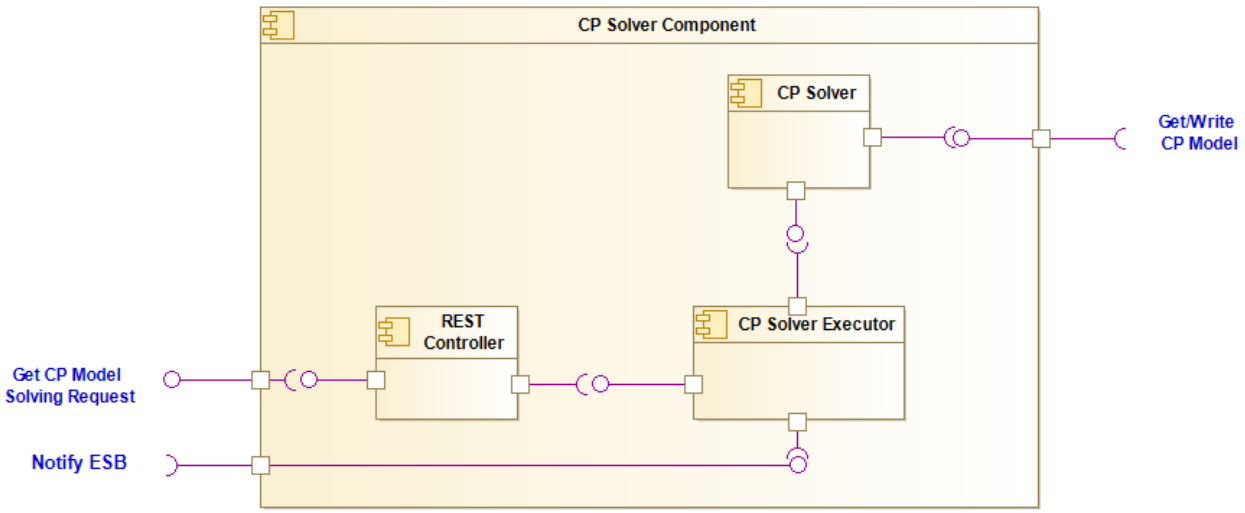


*Figure 15. CP Solver component diagram*

### 4.3.2 Implementation

The CP Solver, having its class diagram depicted in Figure 16, has been implemented in Java as a Spring-boot application. It can be built and bundled via Maven in a form of either a fat JAR file or a Docker image. The latter form facilitates its integration into the Melodic's platform swarm.

Internally, the CP Solver exploits the Choco Constraint Programming solving engine as well as the CDOClient in order to retrieve a CP model for solving, plus writing back to it, the respective solution found. In addition, the Utility Generator is utilised for computing the utility of candidate solutions.

The respective classes of the CP Solver code more or less map to the sub-components introduced above while they mainly offer just one method each, which has already been explicated.

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

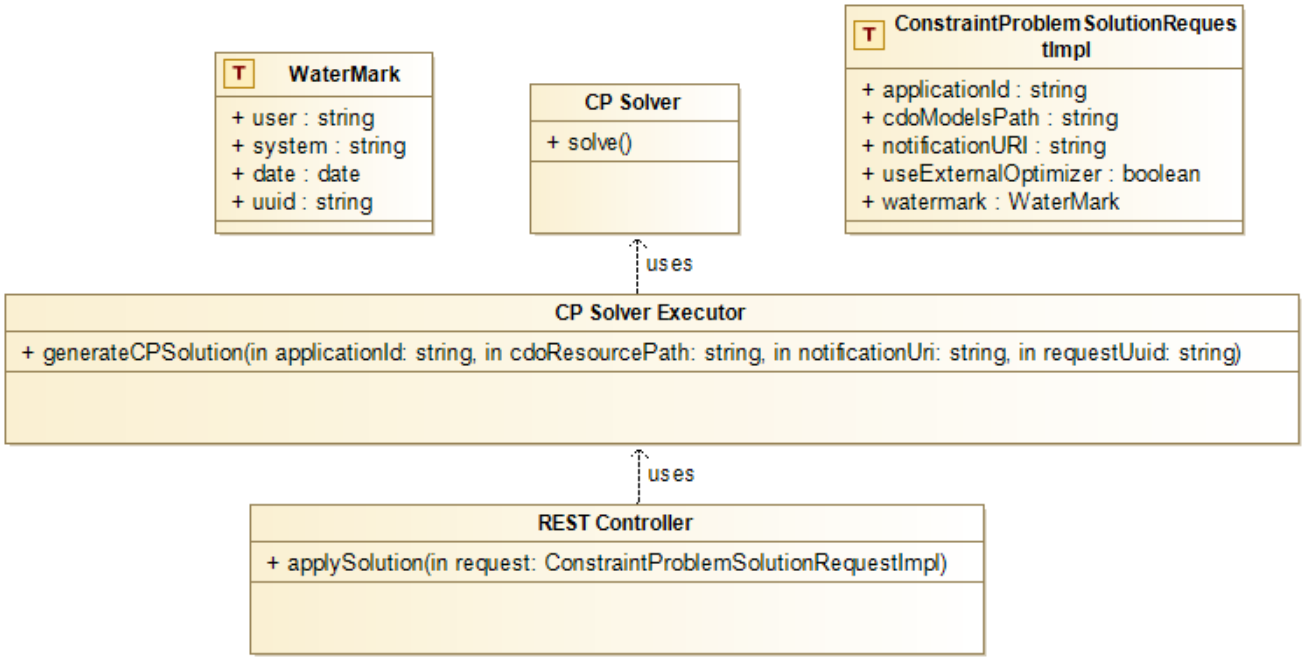


*Figure 16. Class diagram for the CP Solver component*

**The CP Solver's source code** is available in Melodic's bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/cp_solver?at=RC2.0

**Dependencies:** CDOClient, Utility Generator, Spring-boot framework, Choco solver

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

# 5 Utility Generator

**Mission**: Calculating the utility function value for a configuration proposed by an Upperware solver.

**Positioning in Melodic**: The Utility Generator is a library responsible for evaluating each solution found by Upperware solvers of the Melodic platform.

## 5.1 Approach

**High-level Approach**: The Utility Generator class is instantiated for each reasoning separately. The Utility Generator exposes the method to evaluate the proposed solution.

**Functionalities**:

- Calculating the utility function value for the proposed by a solver solution/configuration

**Input**:

- Constraint Problem model
- CAMEL model
- Current measurements (metric values)
- Cache with Node Candidates

**Output**:

- Utility function value

## 5.2 Business Logic (Utility Generator Processes)

Creating of the Utility Generator Application object

Upon creating the Utility Generator Application object, the Utility Generator extracts the Constraint Problem Model to get variables, constants, and metric values. The exact steps taken during the creation of the Utility Generator object are specified next and depicted in Figure 17:

- Acquires the application CAMEL model from Models Repository, and extracts the Metric Model
- Acquires the application CP model from Models Repository
- Extracts from the CP model variables and metric values
- Extracts from the Metric Model the utility function formula and all objects used in this formula, like Node Candidates attributes, DLMS Utility attributes, metrics, variables, and values of the currently deployed configuration

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

- If the optimization requirement and hence the utility function formula does not exist in the CAMEL Model, it creates the default formula which optimizes the cost

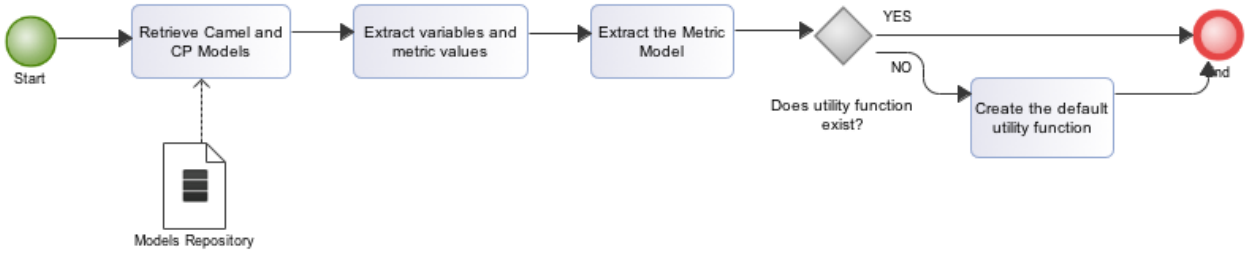


*Figure 17. Creating of the Utility Generator Application object sub-process*

## Solution Evaluation process

Upon invocation by a solver based on a new proposed solution, the Utility Generator transforms it into a configuration for evaluation. This is done by selecting the cheapest one from the available Node Candidates, while fulfilling the criteria from the proposed solution. If one of the DLMS utility attributes is used in the utility function formula, the Utility Generator invokes the DLMS Utility library to get the DLMS Utility. Then, it collects all values used in the utility function formula, calculates the value of the utility function formula, and returns this value as a result.

Specifically the following tasks, depicted in Figure 18, are involved in this solution evaluation process each time a solver suggests a possible configuration:

- Maps the solution to the Node Candidates configuration
- If no Node Candidate fulfils these criteria, returns 0.0 as the utility value
- Invokes the DLMS Utility library to get the DLMS Utility if needed (used in the utility function formula)
- Gets the Node Candidates attributes used in the utility function formula
- Converts all needed values to arguments of the utility function
- Calculates the utility function value and returns this value to the solver

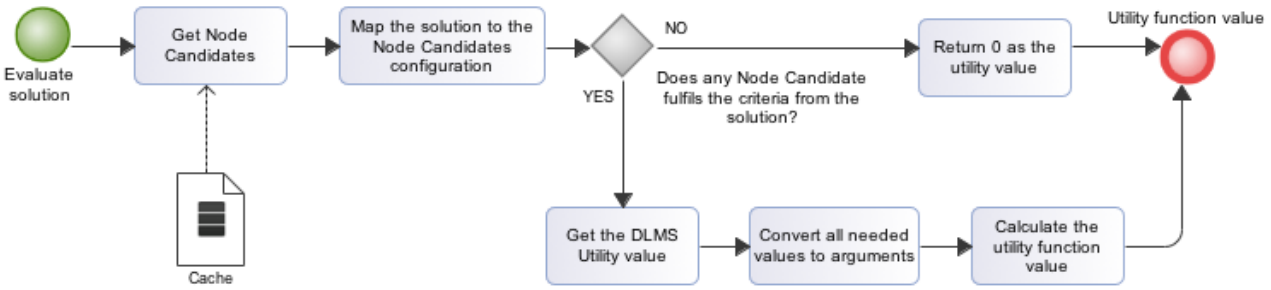


*Figure 18. Utility Generator's Solution Evaluation sub-process*

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

## 5.3 Technical Implementation

### 5.3.1 Architecture

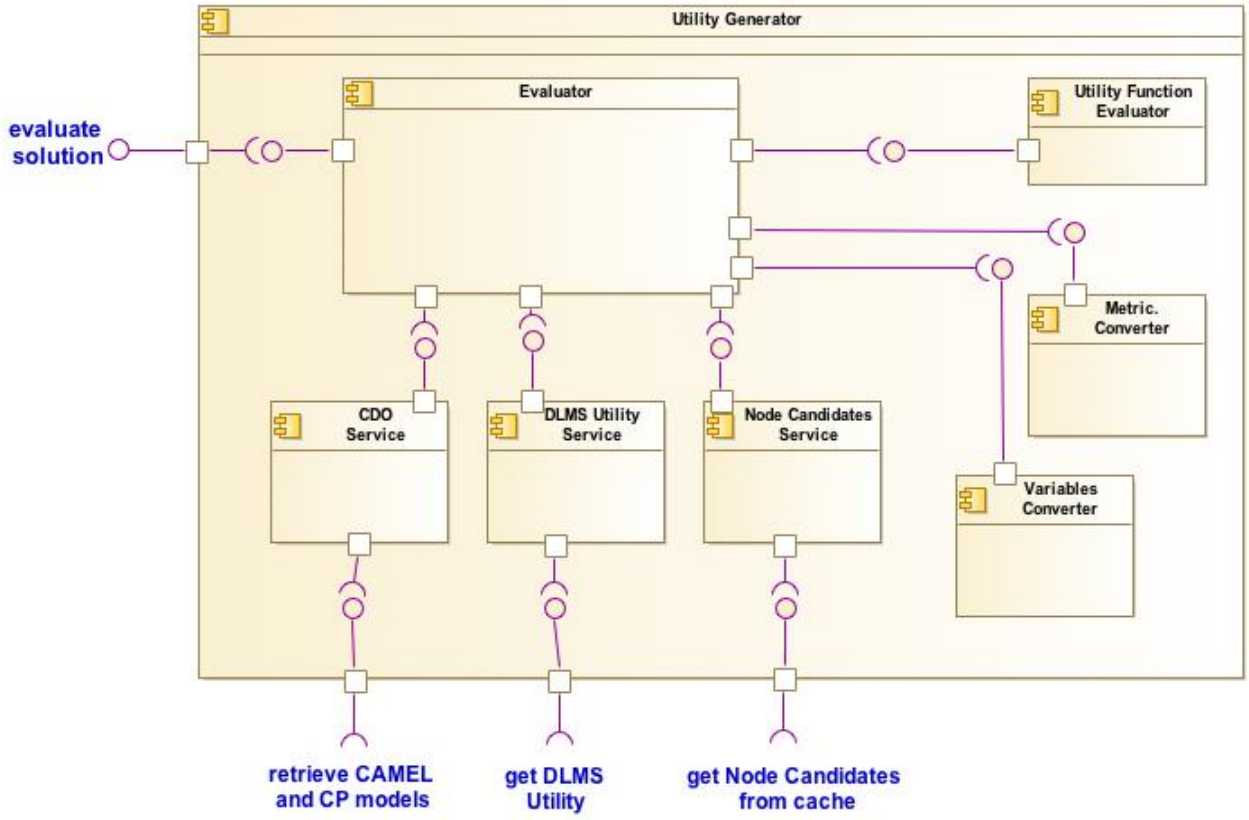


*Figure 19. Utility Generator Component diagram*

In Figure 19, the architecture of the Utility Generator is depicted which comprise the following main components:

- **Evaluator:** the main module of the Utility Generator library, responsible for getting Node Candidates, collecting arguments and invoking the Utility Function Evaluator to get the utility function value
- **Utility Function Evaluator:** the module, which uses the MathParser library, responsible for calculating the utility function formula
- **CDO Service:** the module responsible for retrieving CAMEL and CP models from the Models repository (CDO)
- **DLMS Utility Service:** the module responsible for calling the DLMSUtility library to get the DLMS utility and converting the DLMS utility object to the arguments needed by the Utility Function Evaluator module

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

- **Metrics Converter:** the module responsible for converting metrics to the arguments needed by the Utility Function Evaluator
- **Variables Converter:** the module responsible for converting solution variables values to the arguments needed by the Utility Function Evaluator
- **Node Candidates Service:** the module responsible for converting from the configuration with Node Candidates attributes used in the utility function formula to the arguments needed by the Utility Function Evaluator

### 5.3.2 Implementation

The Utility Generator has been implemented using the Java programming language, version 8. It has been developed as a Java library. Its main classes are depicted in Figure 20. Only the most important methods and fields are depicted, and then explained below the figure.

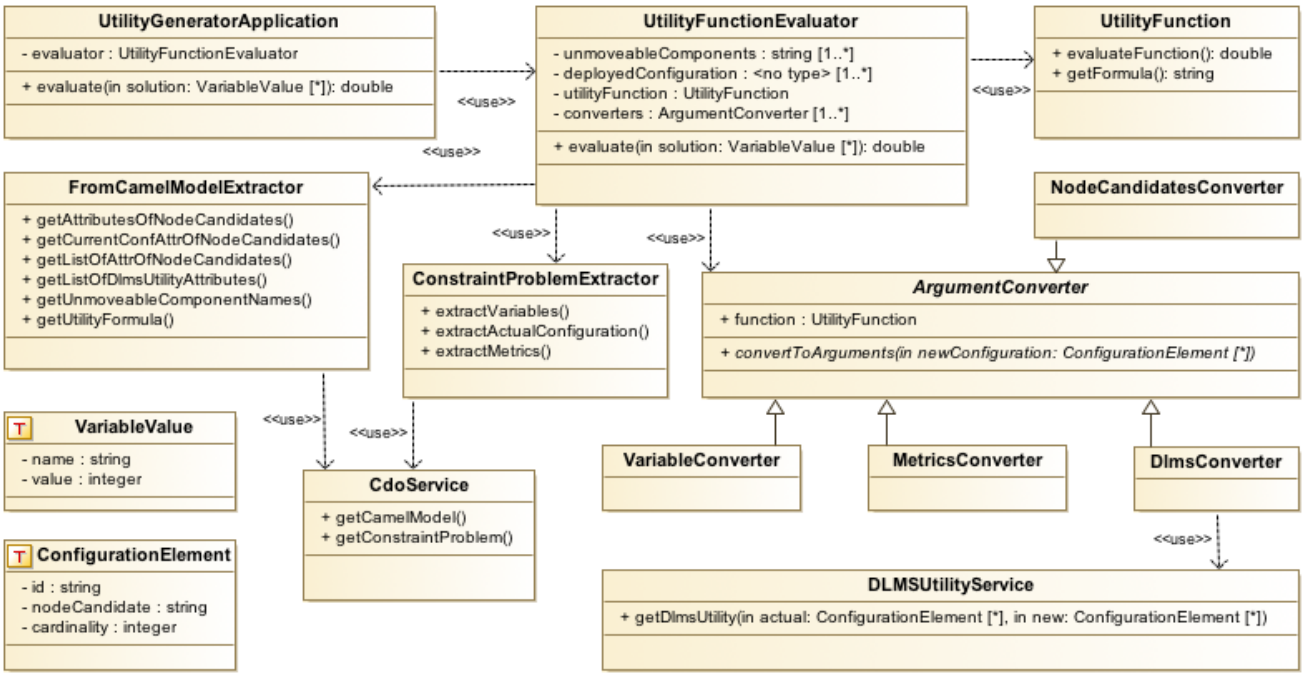


*Figure 20. Utility Generator Class diagram*

**UtilityGeneratorApplication** - The main class of the library. It is created by the solvers and invoked for evaluating the solution.

**UtilityFunctionEvaluator** - This class is responsible for coordinating the whole process of calculating the utility function value. It offers one method: *evaluate(Collection<VariableValue>)* and it includes a list of components which cannot be moved, currently deployed configuration, utility function and list of *ArgumentConverters*.

**UtilityFunction**- This class represents the actual utility function. It offers methods to get the utility function formula and method for evaluating the utility function.

**ConfigurationElement**- This type is used to represent the deployment configuration.

**FromCamelModelExtractor** -This class offers methods to get the needed elements, like attributes of Node Candidates, DLMS Utility Attributes, current-config attributes of Node Candidates, and utility function formula from the CAMEL Model.

**ConstraintProblemExtractor** - This class offers methods to get variables, metrics, and actual configuration from the Constraint Problem.

**ArgumentConverter** - This is an abstract class with one abstract method: *convertToArguments*. It is used for converting the proposed configuration to the arguments of the utility function.

**NodeCandidatesConverter** - This class implements the *ArgumentConverter*. It converts the proposed configuration to the values of node candidates attributes needed by the Utility Function Evaluator.

**VariableConverter** - This class implements the *ArgumentConverter*. It is responsible for converting solution variables values to the arguments needed by the Utility Function Evaluator.

**MetricsConverter** - This class implements the *ArgumentConverte*r. It is responsible for converting metrics to the arguments needed by the Utility Function Evaluator.

**DlmsConverter** - This class implements the *ArgumentConverter*. It uses the DLMSUtilityService to get the DLMS utility value and converts solution to the arguments needed by the Utility Function Evaluator

**DLMSUtilityService** - This class offers the method: getDlmsUtility which invokes the DlmsUtility library to get the DLMS utility value.

**CdoService** - This class offers the methods *getCamelCAMELModel* and *getConstraintProblem* from the Models repository.

**The Utility Generator's source code** is available in Bitbucket at: https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/utility-generator?at=refs%2Fheads%2FRC2.0

**Dependencies:** CDO Client, Melodic cache, Cloudiator Client, DLMS Utility, MathParser, Melodic commons

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

# 6 Solver to Deployment

**Mission:** Creating Deployment Instance Model from the Constraint Problem solution for deployment process.

**Positioning in Melodic**: Solver to Deployment is one of the microservices comprising the UpperWare of the Melodic platform.

## 6.1 Approach

**High-level Approach:** Solver to Deployment is responsible for transforming the solution of Constraint Problem to the Deployment Instance CAMEL Model.

**Functionalities:**

- Creating the Deployment Instance Model from the solution of the Constraint Problem
- Storing the CAMEL Model with Deployment Instance Model to the CDO repository
- Sending success or failure notification

**Input:**

- CAMEL Deployment Model
- Constraint Problem solution

**Output:**

- CAMEL Deployment Instance Model stored in the CDO repository, which contains information about:
  - Software Components - components of deployed application,
  - Communications - communication between application's components,
  - Geographical Regions - distribution of application's components.

## 6.2 Business Logic (Apply Solution process)

Solver to Deployment exposes the REST API that can be invoked by the Upperware control process whenever there is a new solution accepted by the Metasolver that should be deployed. The incoming message contains information about the *application id* and the *Constraint Problem id*. Upon request by the Upperware control process, Solver to Deployment retrieves the CAMEL Model and the CP Model from the CDO repository. Then, it gets the cheapest Node Candidate fulfilling the criteria from the Constraint Problem solution for each software component. It prepares the new software component instances using the information from the chosen Node Candidate.

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

Then, it prepares all new communication instances and new geographical regions (if needed) and registers them in a new Deployment Instance Model in the CDO repository. The process is depicted in Figure 21.

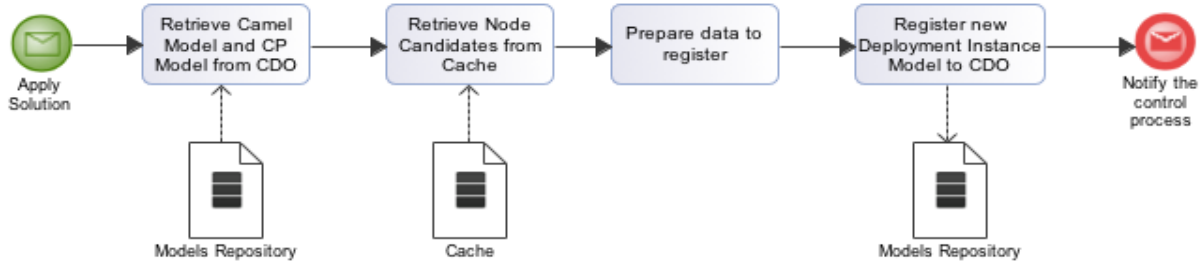


*Figure 21. Apply Solution BPMN process*

## 6.3 Technical Implementation

### 6.3.1 Architecture

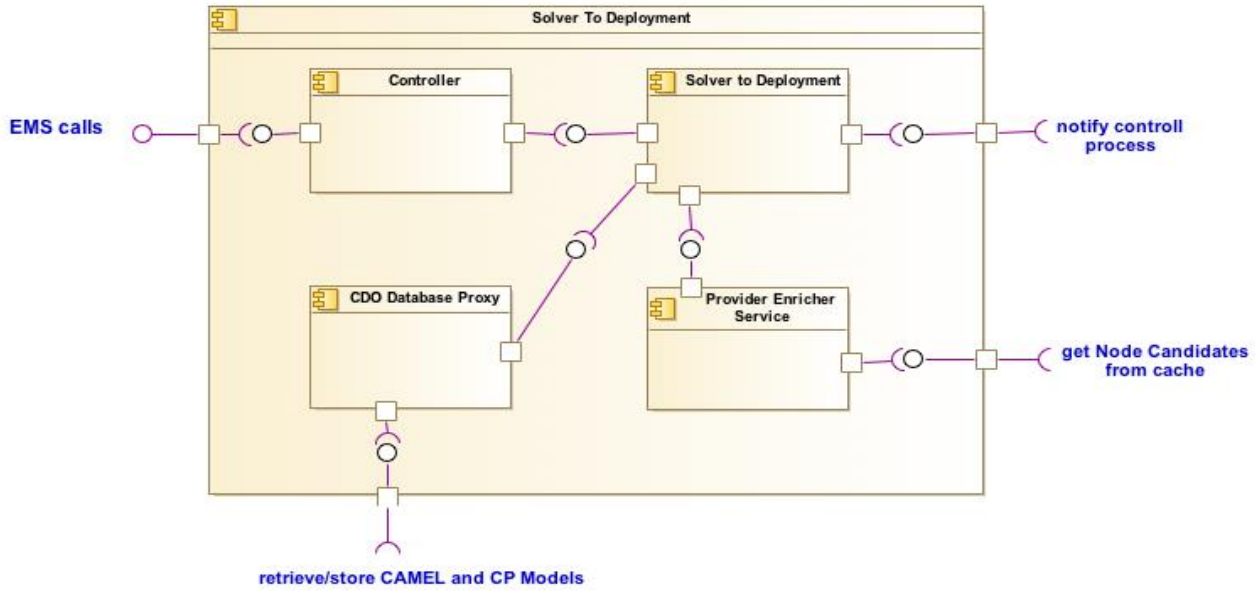


*Figure 22. Solver to Deployment Component diagram*

In Figure 22, the architecture of the Solver to Deployment is depicted which comprise the following main components:

- **Controller:** module, which providers the REST API of Solver to Deployment

- **Solver to Deployment:** module responsible for managing the process of instantiating the accepted solution
- **CDO Database Proxy:** module responsible for communicating with CDO
- **Provider Enricher Service:** module, which enriches information about application's components (i.e. Software Components) by adding data from Node Candidates

### 6.3.2 Implementation

Solver to Deployment has been implemented using the Java programming language, version 8. It has been developed as a Spring-boot application for making its maintenance predictable. It is built and bundled, using the well-known Maven system, into a single fat JAR, containing Solver to Deployment classes and dependencies. It is also bundled (during its building with Maven) as a Docker container and subsequently been added in the Melodic platform component swarm.

The components of Solver to Deployment comprise the following corresponding classes, also depicted in Figure 23:

**SolverToDeploymentController:** collects requests about deploying a new solution and relays them to the SolverToDeployment class for further analysis.

**SolverToDeployment:** It analyses requests about deploying a new solution from the SolverToDeploymentController class and manages the process of creating the CAMEL Deployment Instance Model, involving the following steps:

- Retrieve the CAMEL Model and Constraint Problem from CDO
- Save a new empty Deployment Instance Model in CDO
- For each application's component, compute data concerning the Deployment Instance Model such as information about:
  - communications
  - software components (by adding additional data from Node Candidates, using the Provider Enricher Service)
  - geographical regions
- Update the Deployment Instance Model in CDO
- sending notification about result

**CDODatabaseProxy:** It saves and updates the Deployment Instance Model in CDO.

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

**ProviderEnricherServiceImpl:** It analyses information from Node Candidates and creates from them the necessary attributes, which are required by each Software Component.

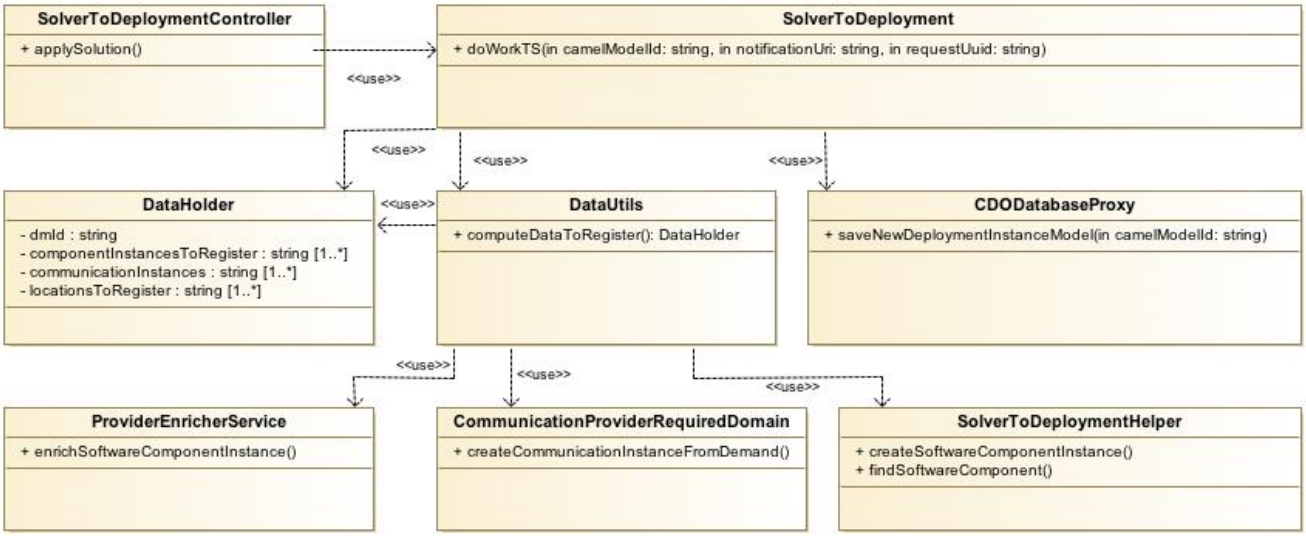


*Figure 23. Solver to Deployment Class diagram*

**Solver to Deployment' source code** is available in Bitbucket at: https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/solver_to_deployment?at=refs%2Fheads%2FRC2.0

**Dependencies:** Melodic Commons, CDO client, Memcache, JWT Commons, Spring-boot framework

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

# 7 Adapter

**Mission**: Create target application configuration to be deployed into cross-cloud resources.

**Positioning in Melodic**: Adapter is one of the microservices comprising the UpperWare of the Melodic platform.

## 7.1 Approach

**High-level Approach:** Adapter is responsible for preparing a complete plan of application reconfiguration for an accepted (by the Metasolver) new deployment. The plan includes a series of tasks to be sent for execution to the Cloudiator, following a specific order. To fulfill this requirement, a graph structure is maintained internally by the Adapter to reflect the application structure along with the dependencies among tasks.

**Functionalities:**

- Analyse and verify the new CAMEL deployment model
- Compute the difference between a currently running application (topology) and the new proposed solution given by the solver
- Producing the reconfiguration plan
- Validate the plan
- Apply the plan to the running system by calling the Cloudiator REST API

**Input**:

- CAMEL Deployment Model

**Output:**

- Series of action tasks instructions for execution by the Cloudiator in correct and efficient order
- Notification of successful/unsuccessful deployment for the control layer

## 7.2 Business Logic (Application Deployment process)

The Adapter exposes a REST API that can be invoked by the Melodic control process whenever there is a new Deployment Model prepared by the Solver to Deployment. The incoming message contains information about the *application ID* that is being (re)deployed and indicates where the

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

Adapter needs to send the deployment notification after finishing the job. With the use of a common CDO Client, the two main artifacts can be collected from the database:

- CAMEL Deployment Model for the currently deployed solution
- CAMEL Deployment Model for the new solution produced by the solver, i.e. the one to be deployed

Based on the existence of the currently deployed model, the Plan Generator (i.e. Adapter's sub-component) creates a configuration or a reconfiguration plan (in case an initial placement has been achieved before), as depicted in Figure 24. The plan contains a series of CREATE or DELETE tasks that reflect Cloudiator API calls in order make a new or update (i.e. to modify nodes, jobs, processes etc.) an existing topology (i.e. cross-clouds where application components have been deployed).

To manage the proper order and dependencies between tasks, a graph structure was adopted inside the Adapter. While some tasks don't depend on each other and are being deployed in parallel (like starting multiple VMs), some may require another task to complete first (e.g. deploying an application component requires a virtual machine to be up and running). Such a graph is by its nature a perfect form of maintaining the dependencies between actions, thus allowing very efficient, in terms of time, application deployment.

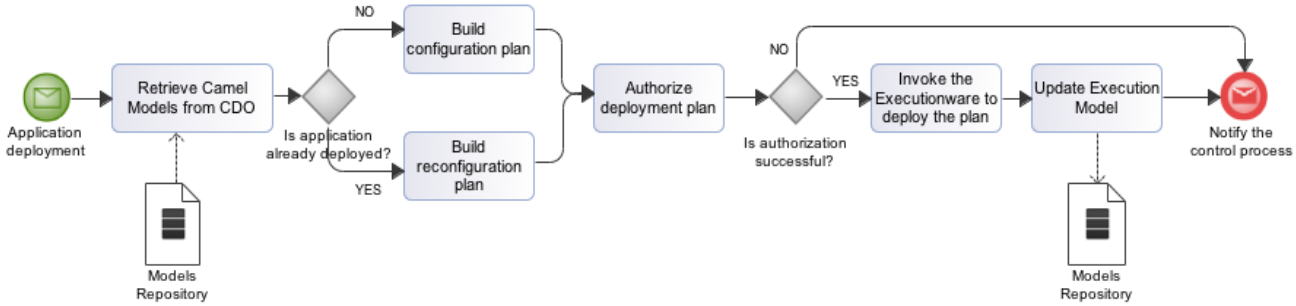


*Figure 24. Application Deployment BPMN process*

The generated plan is then verified by the Authorization Service [6] in order to check if the configuration that is to be deployed aligns with the defined security policies. For that reason, Adapter uses Authorization Service Client and a configurable set of extractors (which collect authorization related data) to contact the Authorization Server, seeking for an authorization decision which will permit the process to continue. Based on the outcome of this authorization request, the deployment actions can be continued or stopped.

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

Each action task contains the data needed to create a JSON object and passed to Cloudiator via its REST API. The actual deployment is yet preceded with refreshing the information about the artifacts that are already deployed within the Cloudiator (context) to assure that if the application changed since the last deployment, the changes will be considered under the current deployment. After completing the deployment, the history record is stored inside the CDO for always persisting the currently deployed model information. The last step is to inform the Melodic's control plane of the result of the deployment.

## 7.3 Technical Implementation

### 7.3.1 Architecture

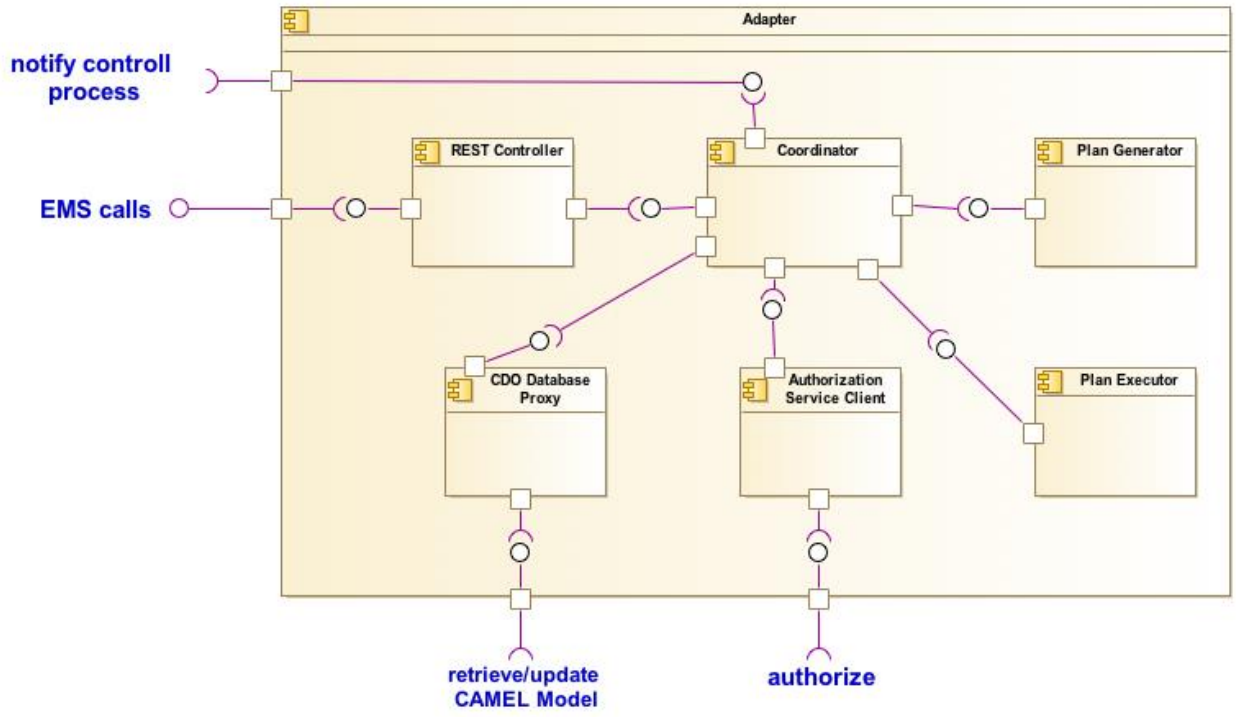


*Figure 25. Adapter Component diagram*

In Figure 25, the architecture of the Adapter is depicted which comprise the following main components:

- **REST Controller:** part of the application responsible for exposing REST endpoints
- **Coordinator:** main sub-component responsible for coordinating all the actions during the (re)configuration process

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

- **Plan Generator:** part of the application responsible for generating the (re)configuration deployment plan (i.e. a set of tasks which must be executed by Cloudiator to finish (re)configuration)
- **Plan Executor:** part of the application responsible for invoking the previously generated deployment plan
- **CDO Database Proxy:** part of application responsible for the communication with CDO Database
- **Authorization Service Client:** client for external Authorization Server which should be invoked to check privileges to (re)configure deployable application according to defined security policies.

### 7.3.2 Implementation

The Adapter component has been developed in Java, version 8. The project is built by Maven and thanks to the Maven plugin[3], a Docker image is created which allows to run this component as a separate microservice. The following classes, depicted in Figure 26, are involved in the implementation of the Adapter:

**AdapterController** - collects requests about applying a new solution and relays them to the *AdapterCoordinator* class for further analysis.

**AdapterCoordinator** - analyses requests about applying a new solution from the *AdapterController* class and manages the process.

**Validator** - validates the *DeploymentInstanceModel*.

**PlanGenerator** - is responsible for generating the configuration and reconfiguration deployment plan (i.e. a set of tasks which must be executed by Cloudiator to finish (re)configuration). It uses the *GraphGenerator* class to generate a graph and *CamelModelConverter* class to convert the Deployment Instance CAMEL Model to Comparable Model.

**GraphGenerator** - offers methods to generate the configuration graph and to generate the reconfiguration graph.

**CamelModelConverter** - is the abstract class which has five implementations: *ScheduleConverter*, *MonitorConverter*, *ProcessesConverter*, *RequirementsConverter*, and *JobConverter*. It offers a method to convert the CAMEL Model to the one needed by the Cloudiator classes.

**Plan** - represents the deployment plan.

**Plan Executor** - invokes the previously generated deployment plan.

[3] `com.spotify:docker-maven-plugin`

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

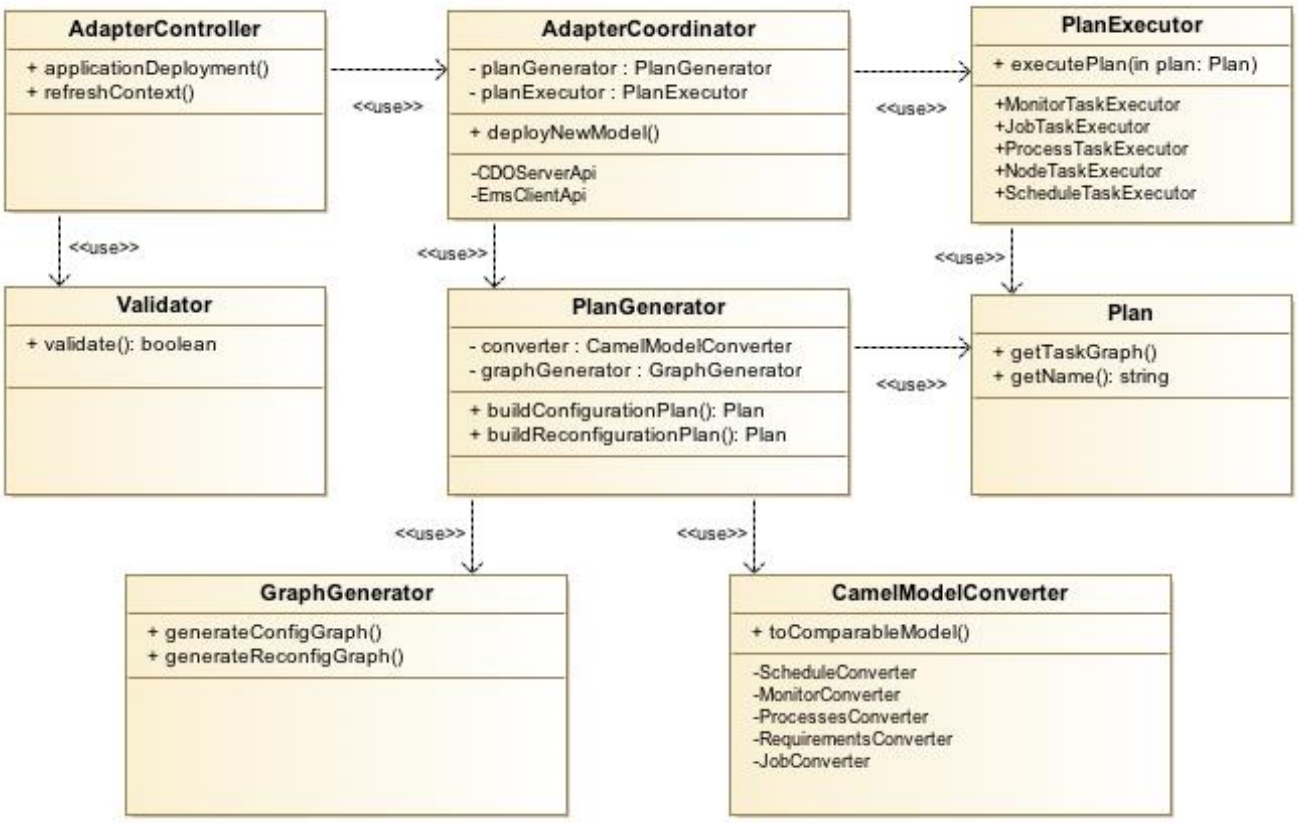


*Figure 26. Adapter Class diagram*

**Adapter's source code** is available in Bitbucket at:
https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/adapter

**Dependencies:** Spring-boot framework, CDO Client, Cloudiator Client, Melodic commons, JWT commons, Authorization Service Client

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

# 8 Event Management System

Event Management System (EMS) is a distributed application monitoring system, which is used by Melodic Upperware for monitoring the operation of the cross-cloud application it deploys, in order to take appropriate actions when certain constraints are violated; i.e. to reconfigure the application.

**Mission:** Collect, process and deliver to interested parties, monitoring information pertaining to a distributed, cross-cloud application, according to CAMEL model specifications.

**Positioning in Melodic:** The EMS server, called Event Processing Manager (EPM), is one of the microservices comprising the UpperWare of the Melodic platform. EMS clients, called Event Processing Agents (EPAs), reside inside each VM that hosts a cross-cloud application component. The EMS server exposes APIs and endpoints, used while interacting with EMS clients or other Upperware components.

## 8.1 Approach

**High-level Approach:** Deploy a network of agents for collecting monitoring information from the sensors (i.e. monitoring probes) as events, process them using distributed event processing techniques, and forward results to the interested parties (e.g. Metasolver). A CAMEL model specifies the needed monitoring information and the kind of processing required.

**Functionalities:**

- Acquire and analyse the application CAMEL model. It yields an abstract form of those parts of the CAMEL model related to monitoring, capturing and processing information, as well as other auxiliary structures. This abstract form is a multi-root Directed Acyclic Graph (DAG).
- Generate complex event processing rules for each EPA and the EPM node. Rule generation is based on the traversal of the DAG resulted from the CAMEL model analysis.
- Deploy and manage the network of EPAs. Deployment is actually carried out by Cloudiator, which runs an EPA-specific installation process. Upon activation, EPAs connect to the EPM node (specifically to the network orchestration module of EPM) and receive their configurations, which encompass the event types to be collected, the event processing rules to be enforced and the event types (raw or generated) to be propagated to another EPA or to the EPM node.
- Event brokering. Each EMS node (EPM node or EPA) encompasses an event broker. Locally captured or generated events, as well as events forwarded from other nodes, are published there. Events are organized in topics according to their source or type. Some of these topics

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

can be configured to forward their events to other EMS nodes. Typically, EPA brokers are private (internal to EPA). However the EPM node broker is open to be accessed (for consuming events) by any interested Melodic platform component.

- Complex Event Processing (CEP). Each EMS node (EPM or EPA) encompasses a CEP engine, which receives events as they arrive to the local event broker, applies the configured event processing rules and publishes the generated events back to the local event broker. Typically, event processing that occurs at cross-cloud application nodes, refers to filtering and aggregating local events (i.e. those collected by local sensors) and pertain solely to that application node. Event processing at the EPM node typically aggregates and filters events from all application nodes. Some EPAs can be designated as intermediary event processors (filters or aggregators), hence resulting in a multitier distributed event processing hierarchy.

Input:

- Raw monitoring information from sensors
- Distributed application topology (i.e. deployment model)
- CAMEL model of a cross-cloud application.

Output:

- Monitoring information as simple or complex events
- Configurations for other Upperware components (i.e. MetaSolver).

### 8.1.1 Event Processing Network

As already mentioned, EMS is a distributed application monitoring system that comprises of a server integrated in Melodic Upperware, named *Event Processing Manager* (EPM), and several clients, named *Event Processing Agents* (EPAs). EPM and EPAs formulate a network of nodes for distributed event processing, called Event Processing Network (EPN). This network is orchestrated and controlled by EPM.

There are several operations involved in event processing. Some of the most common are:

- Event collection (from monitoring probes)
- Event filtering (selecting events that meet certain conditions)
- Complex event generation (creating new events based on the values of other events)
- Event aggregation (creating complex events by aggregating the values of other events)
- Event pattern detection (finding predefined motives of event occurrences)
- Event propagation/delivery (sending events to other processing nodes or destinations).

Furthermore, these operations might be applied in different fashions and scopes. For instance, they might be applied per application node/VM, per cloud provider or for the whole application. In

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

Melodic we have devised a hierarchical model of scopes, where event processing can occur locally (i.e. in each application VM), across the application nodes deployed in a single cloud provider, or across the whole application (cross-clouds). The event processing results of one scope are propagated to the higher-level scope.

For example, RAM usage events can be collected per application VM and averaged per minute. Average value events are then propagated to the cloud level. In cloud scope, all average RAM usage events from the VMs deployed in the same cloud provider are collected and filtered. Events might be checked, if they exceed a specified threshold. Such events might then be propagated to the Application level (i.e. global) in order to signal that a certain condition occurred (i.e. a node has been overloaded).

In Melodic we have defined the following scopes, which we call *groupings*:

- Per Instance: processing involves events from a single application instance executed in a VM.
- Per Host: processing involves events originating from the same local VM. It may also aggregate events generated and propagated from the Per Instance grouping.
- Per Zone: processing involves events originating from VMs in the same cloud availability zone.
- Per Region; processing involves events originating from the same cloud region.
- Per Cloud; processing events originating from all application nodes deployed in the same cloud provider.
- Global; processing events originating from any application nodes. Events might be received from any subordinate grouping. This is the terminal grouping.

For each grouping, different event processing operations might be required. These can be expressed as sets of event process rules and event propagation flows between groupings.

An important aspect of the hierarchical grouping model of Melodic, is that it requires the event processing to occur as close to the event generation point as possible. This statement means that the scope processing must occur in the same VM/location where input events are created. For example, in per host grouping, event processing must take place locally, in each VM. In per cloud grouping, event processing must take place in application VMs designated for this purpose, per cloud provider. The global grouping event processing occurs in the EPM node.

To implement this approach, an EPA must be deployed in each application VM. EPM configures the EPAs into a cooperating network of event processing nodes, providing all necessary event processing rules and event propagation routes.

This approach is based on the assumption that large numbers of events can be exchanged very fast and much cheaper between VMs running on the same availability zones, regions or cloud infrastructures. Processing events inside the same zone/region/cloud can reduce the number of events that need to be propagated to a central processing node thus reduce the overall network

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

throughput for application management purposes. Moreover, the computational load of the event processing is distributed to all application VMs, hence the central processing node does not require increased computational capabilities or network resources. Of course, if an application requires centralized processing it is possible to reduce the number of groupings into per host and global, thus resulting in events flowing from VMs to EPM.

In the context of EMS, the EPM acts as the global event processing node, whereas EPAs are the per cloud/region/zone/host nodes. EPAs are also responsible for collecting events from local sensors (i.e. installed in the same VM), perform the operations required for the grouping they have been designated to and propagate the results to another EPA or the EPM, which acts as the higher grouping processing node.

Between EPN nodes, two types of connections are established: (i) Control connections, created between each EPA and the EPM, and (ii) event propagation connections, created either among EPAs or between EPAs and EPM. The former connection type is used by EPM to control and configure EPAs, as well as by EPAs to announce their presence to EPM (during VM launching). The latter connection type is used to convey events between the grouping processing nodes.

## 8.2 Business Logic

The EMS design and implementation has the following objectives:

- Analyse the CAMEL model of a cross-cloud application in order to extract the required (by other Melodic-platform components) monitoring information along with the processing needed.
- Deploy (through Cloudiator) EMS clients, which are called Event Processing Agents (EPAs) to each distributed application node that hosts an application component (to be monitored).
- Configure each EPA to collect (from sensors) and forward the needed events, and also apply the required complex event processing rules.
- Provide the required information (specified in the CAMEL model), either by updating the application CP model, by publishing events (any interested party may subscribe to receive them), or by requesting Melodic-platform to reconfigure the distributed application (e.g. when certain SLOs are violated).

The aforementioned objectives and the functionalities that these imply, are realized by a number of concrete processes, which are detailed in the following sections.

New CAMEL Model process

Upon loading a new CAMEL model to the Melodic Models Repository, the Upperware control process invokes EPM in order to pass the identifier/path of the new CAMEL model. Subsequently,

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

EPM retrieves and analyses it and prepares the application monitoring system, and generates various configurations for other Melodic components. Specifically, as shown in Figure 27, it:

- Retrieves the application CAMEL model and convert the monitoring related parts into an abstract form called DAG.
- Using the DAG it generates the configurations for all nodes of the monitoring system (EPM and EPAs):
  - Event Topics per node grouping
  - Event Processing Rules per node grouping
  - Event Topic forwarding per grouping
- Using the DAG, it generates the configuration for Metasolver, containing the event topics it must subscribe to and what type of actions to take (i.e. update the CP model with new metric variable values and start a new reconfiguration iteration).
- Using the DAG, it generates the sensor configurations per application component type.
- Configures the EPM event broker and CEP engine.
- Configures and initializes the Baguette server, i.e. the EPA orchestration module.
- Sends the appropriate configuration to each EPA that successfully connects to the Baguette server, based on the role of the application node in the monitoring network.
- Sends configuration to Metasolver.
- Notifies the Upperware control process when EPM is ready to accept EPA connections and process any incoming events.

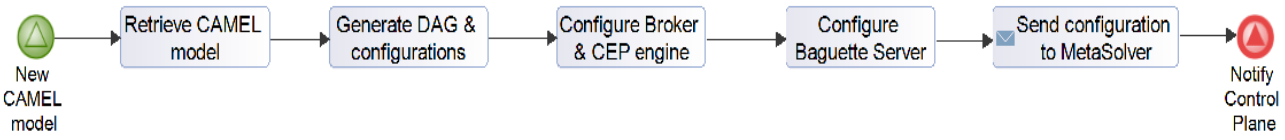


*Figure 27. New CAMEL Model BPMN process*

## New Solution process

When a new application is deployed or an existing is reconfigured, the Upperware control process invokes a solver in order to generate a new application deployment topology (in the form of a constraint problem solution). During this process, the solver assigns values to certain deployment parameters (registered in the corresponding CP model) that might be needed when monitoring the application and processing events; for instance, the storage capacity of all application VMs. For this reason, EMS must be notified every time any of these parameters is updated. This happens through Metasolver, which is notified when a new application topology gets implemented (i.e. application is deployed or reconfigured), signalling that the corresponding, updated parameters become effective. Metasolver subsequently notifies EPM of the new parameter values in effect, passing the identifier/path of the corresponding CP model.

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

Upon notification, EPM will retrieve the new parameter values and propagate them to all event processing nodes (i.e. EPM and connected EPAs). Specifically, as depicted in Figure 28, it will:

- Retrieve the CP model and extract parameter values.
- Apply new parameter values to the EPM CEP engine.
- Send parameter values to each connected EPA.
- Each EPA immediately applies new parameters to its local CEP engine.
- EPAs that might connect or reconnect after parameter values update, will get the most recent values along with their initial configuration.

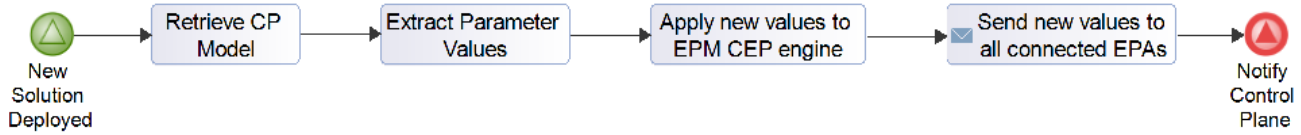


*Figure 28. New Solution BPMN process*

Event Processing Agent process

When an application node is launched, the locally installed EPA is activated, and it immediately tries to connect to EPM. If a connection is established, then the EPA announces itself to EPM and sends its unique id. EPM subsequently designates EPA to a grouping and sends the related configuration (i.e. event processing rules and event propagation routes). If in the future there is any change in the application configuration, EPM will again send updates to EPAs. EPAs, upon receiving their configurations or updates, apply them immediately. Updates can either be slight changes in the values of application deployment parameters (see the "New Solution" process) or completely new configurations.

Each EPA encompasses a local event broker that receives events from local sensors as well as from subordinate EPAs, and a local Complex Event Processing (CEP) engine, responsible for applying event processing rules to the incoming events. The CEP engine subscribes to the necessary local event broker topics in order to receive (input) events. The outcomes of the CEP engine (i.e. new events) are sent to the local event broker. Moreover, the events sent to configured topics are also propagated to the higher-level grouping EPA.

The exact process of EPA operation is depicted in the following diagram (Figure 29).

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

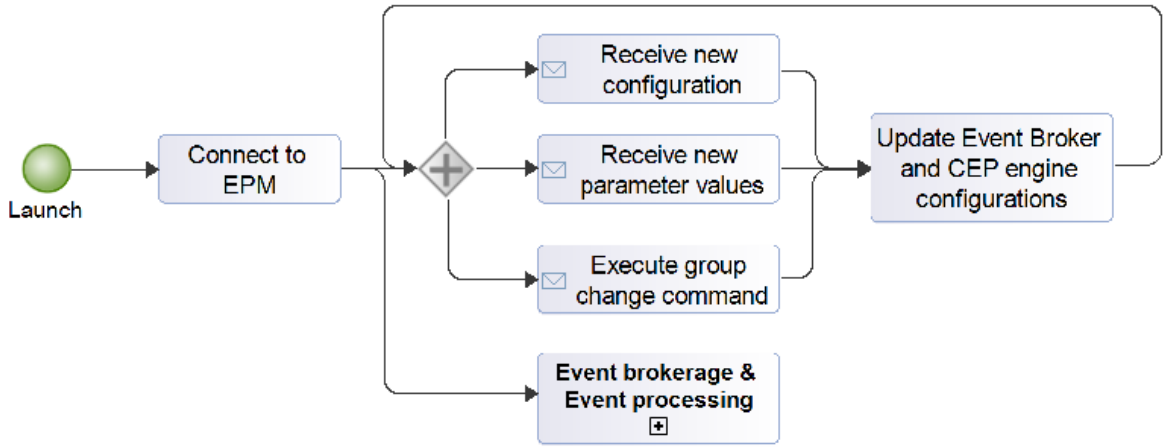


*Figure 29. EPA BPMN process*

## New Event Processing process

Every event delivered to EPM or to an EPA is being handled and processed according to the "New Event Processing" process. Specifically:

- If the event is published to an event topic used in an event processing rule, a copy of the event is immediately handed to the CEP engine, thus triggering a new CEP iteration.
- If an event is published to an event topic configured for propagation to next grouping EPA, a copy of the event is immediately published to the next EPA.
- If the CEP engine generates a new event, it is immediately published to the appropriate topic in the local event broker. This action may also result in publishing the new event to next grouping EPA (if configured).

The next diagram, Figure 30, depicts the process present above.

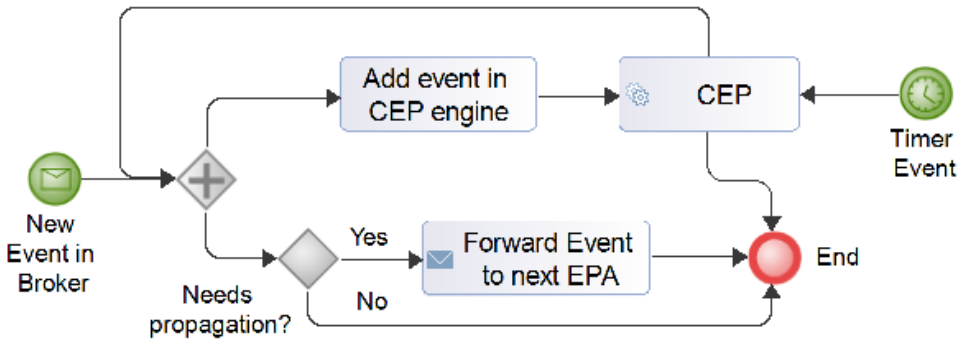


*Figure 30. New Event Processing BPMN process*

## 8.3 Technical Implementation

In this subsection, all the technical details of the EMS implementation are discussed (e.g. language, frameworks, 3rd party libraries, code structure etc.). It essentially comprises the following modules which are explained in the next sub-sections:

- **Event Processing Manager (EPM)** node or EMS server, responsible to analyse the CAMEL model, deploy and manage the whole monitoring network of EPAs, and interact with the rest of the Melodic platform (by providing interfaces and invoking interfaces of others).
- **Event Processing Agents (EPAs)**, deployed to each distributed application node. They are responsible to contact EPM node for taking their configurations (i.e. events to collect, event processing rules and where to propagate events (raw or processed)).

### 8.3.1 Architecture of EPM (EMS server)

A high-level depiction of the EPM architecture is given through the UML component diagram depicted in Figure 31.

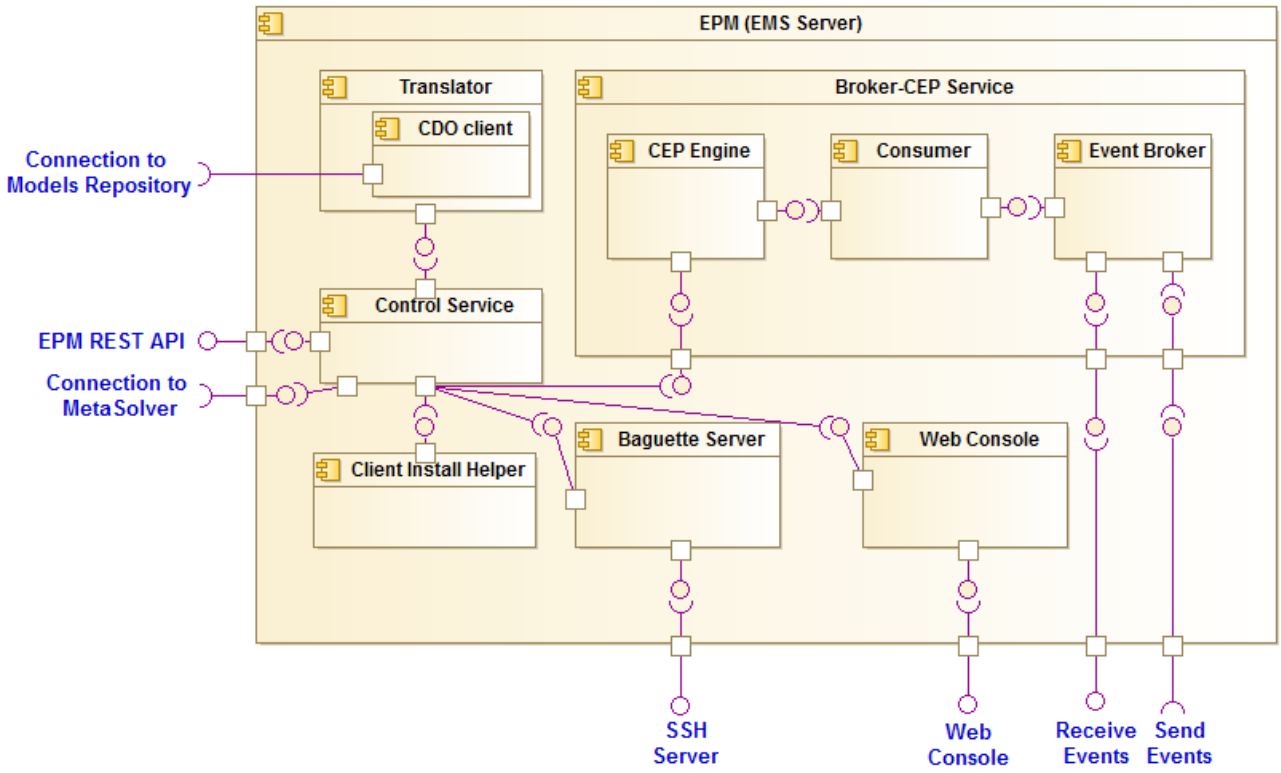


*Figure 31. EPM (EMS server) Component diagram*

Based on Figure 31, the EPM comprises the following sub-components:

- **Translator:** provides a two-step process involving the analysis of the CAMEL model to produce a multi-root Directed Acyclic Graph and also the Generation of EPL rules and other related information.
- **Client install component:** provides the necessary instructions on how to install an EPA to the application VM defined in the application deployment model.
- **Baquette Server:** is responsible for the deployment and management of the Event Processing Network. Specifically, it designates EPAs installed in each application VM, to the appropriate grouping, by sending the corresponding configuration. It also collects VM identification information sent from EPAs. The Baguette server encapsulates an SSH server used to accept incoming connections from EPAs. These connections are used to send configurations or other commands to EPAs.
- **Control Service**: Coordinates and oversees the functioning of EPM. It also interacts with the Upperware control process through the REST API and furthermore offers a few EPM management and debugging functions (as REST endpoints as well).
- **Web console component:** Provides a dashboard for monitoring the functioning of the local event broker.
- **Broker-CEP Service:** encapsulates an event broker instance and a CEP engine instance, appropriately wired to implement the process presented in Figure 30, hence Broker-CEP provides event brokerage and complex event processing capabilities. The Consumer sub-component depicted inside Broker-CEP is used to forward the event broker messages into the CEP engine in order to be processed.

### 8.3.2 Architecture of EPA

A high-level depiction of the EPA architecture is given through the following UML component diagram, shown in Figure 32.

Based on Figure 32, the EPA comprises the following sub-components:

- **SSH Client:** provides secure communication with the Baquette server (of the EPM) through an EPM connection interface.
- **Executor component:** Configures and starts the Broker-CEP Service based on the instructions by the Baguette Server.
- **Broker-CEP service:** provides the local Complex Event Processing engine (i.e. Esper) along with the local Event Broker (i.e. ActiveMQ) that collects and propagates data messages to other Virtual Machines components in the distributed hierarchy of Virtual Machines in our cloud environment. This client can undertake the role of per instance, per host, per zone, per region or per cloud grouping as explained in section 8.1.1.

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

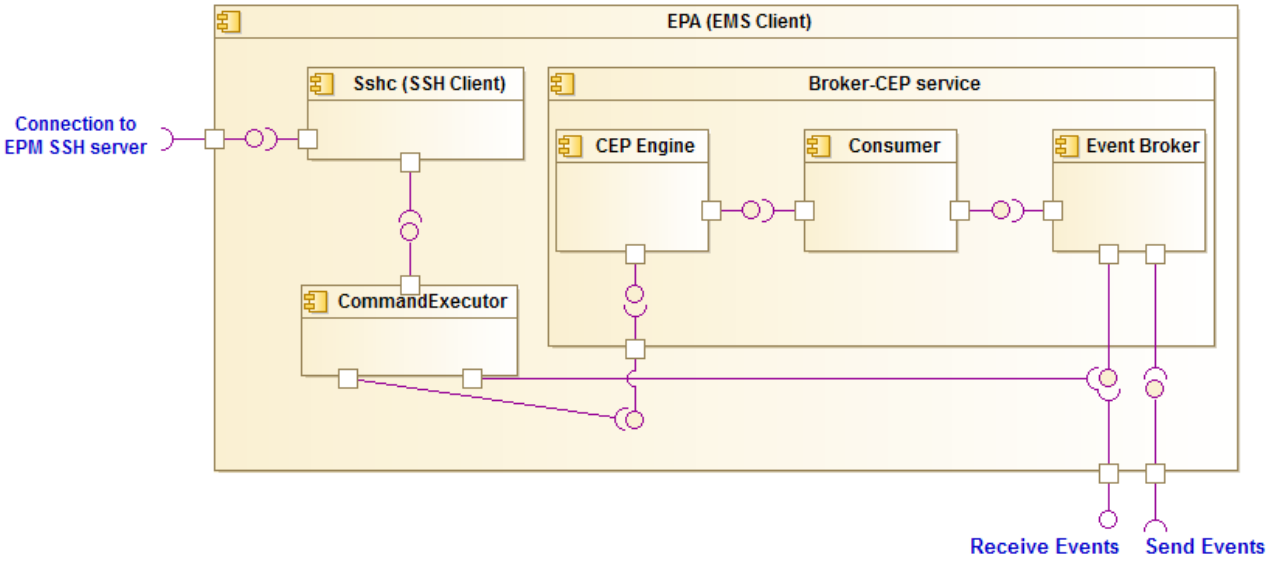


*Figure 32. EPA Component diagram*

### 8.3.3 Implementation

All EMS modules have been implemented using the Java programming language, version 8, and almost all of them are Spring-boot applications, components or configurations, thus making their maintenance quite predictable. EMS is delivered as a set of software packages; one for the EPM node, one for EPAs, and one for the Broker client library.

All EMS parts (EPM node, EPAs and Broker client library) are built and bundled using the well-known Maven system. Due to license incompatibilities, the third-party libraries used are not bundled with the EMS code but are kept separately. Therefore an EMS package is a collection of JAR files (containing the EMS code and the compiled third-party libraries), configuration files and launch scripts.

Following, more information on each EMS package are given:

- **EPM package:** provides a "fat" microservice encompassing several EMS modules. EPM can also be bundled (during building with Maven) as a Docker container image and subsequently be added in the Melodic platform component swarm.
- **EPA package:** provides a lightweight, low-consumption application, since it is collocated with the application node software and it must not compete with it for system resources. It is bundled as a Zip file including all needed JAR files, configuration and launch scripts. During application deployment, Cloudiator uses that Zip file to install EPA in each application node.
- **Broker Client:** has been implemented as a lightweight, non-Spring library. Therefore it can be used in any kind of Java application, either Spring-boot or not.

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

More insights on the EMS implementation are provided by the following class diagram (Figure 33) and the accompanying class descriptions. Only the most important classes have been included for better understanding of the implementation approach:

Control Service

- ControlServiceController: It encompasses methods that map to EMS REST API endpoints and provide the respective functionality. These methods relay the actual processing to the Control Service Coordinator and take care of any HTTP request conversions into the internal representations of EMS, and vice versa.
- ControlServiceCoordinator: It implements the business logic of the EMS Control Service component and it is mainly invoked by the Control Service Controller. This Coordinator initializes and utilizes features realized in other EMS components (Broker-CEP, Baguette Server and Translator).

Broker-CEP service

- BrokerCepService: It provides the API of the Broker-CEP service and encapsulates the event brokerage and complex event processing functionality. It holds pointers to the actual implementations of the Event Broker and the CEP engine and communicates with them in order to fulfil the API calls.
- BrokerService: It provides the Event Broker functionality. This class is part of the ActiveMQ event broker.
- CepService: It encapsulates the Complex event processing functionality. It initializes an instance of the Esper CEP engine and registers the required extensions. It is also used to register event processing rules and the respective listeners that receive the outcomes of the rules.
- BrokerCepConsumer: It is an event processing rule listener used to redirect the events generated in the Esper engine into the event broker.

Baguette Server

- BaguetteServer: It is a wrapper class that provides the API of the Baguette Server component. It initializes an instance of the Sshd class and an instance of ServerCoordinator, according to the configuration.
- Sshd: It encapsulates an SSH server, used to communicate with EPAs, and takes care of the session establishment and communication details.
- ServerCoordinator: It is an interface defining the required functionality of the Server Coordinator objects. Server coordinators are responsible to implement the event processing network topology. To this end, they must designate the appropriate roles to every EPA connected and send the corresponding commands.

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

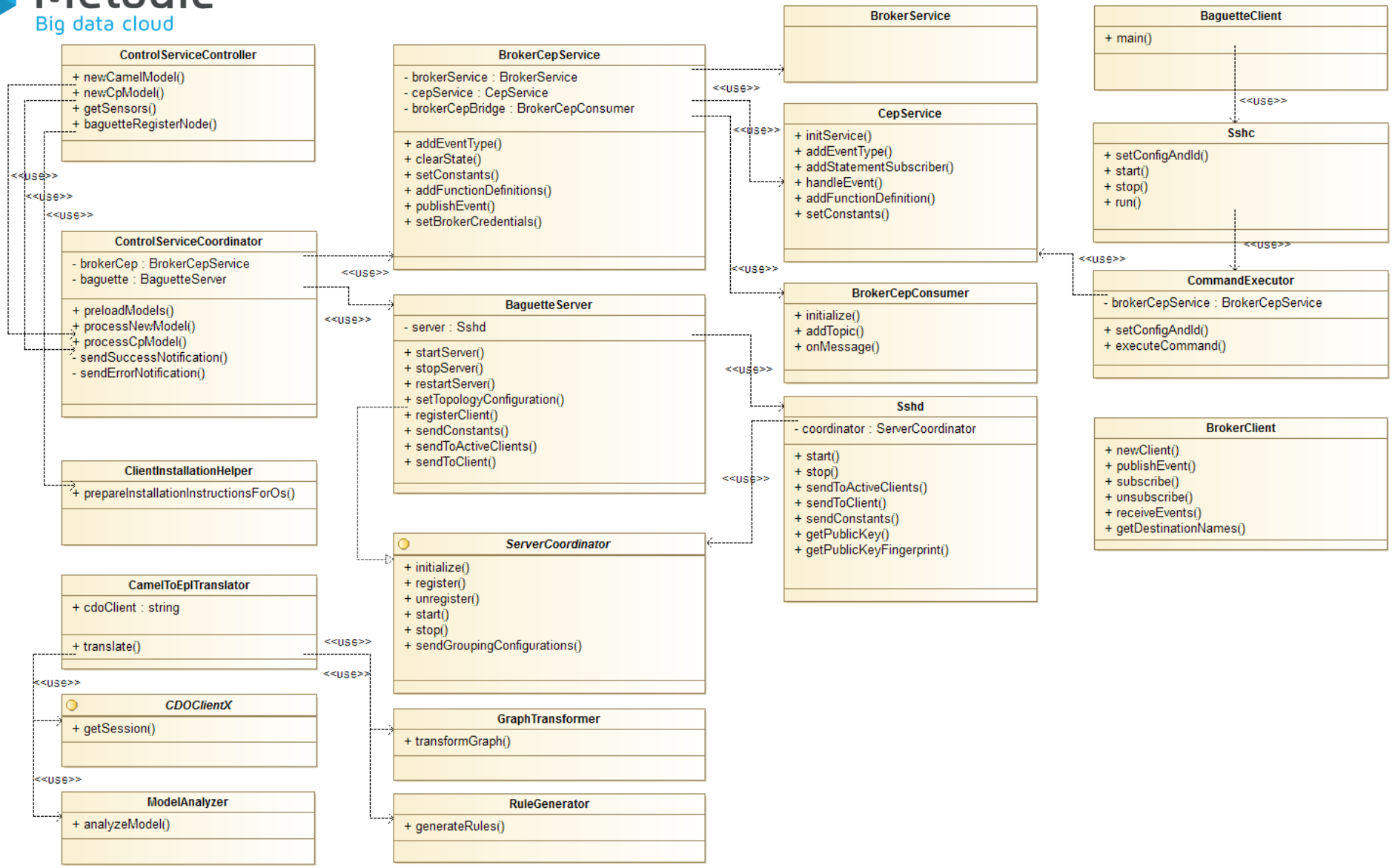


*Figure 33. EMS Class diagram*

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

Translator

- CamelToEplTranslator: It encapsulates the Translator component functionality. It instantiates all related classes (Model Analyzer, Graph Transformer, Rule Generator and CDO client) and invokes them accordingly.
- CDOClientX: It provides a CDO client used to connect to the Melodic Models repository in order to retrieve the CAMEL and CP models.
- ModelAnalyzer: It provides methods that parse a given CAMEL model and yields an abstract Directed Acyclic Graph representation of it, as well as a set of configurations for various Melodic or application components (e.g. Metasolver and Monitoring probes).
- GraphTransformer: It improves or optimizes the DAG generated by the Model Analyzer.
- RuleGenerator: It generates event processing rules based on the transformed DAG. The rules are generated using rule templates and information from DAG. The current version of this class generates EPL rules, compatible to the Esper engine used in the Broker-CEP service. The rule templates are defined in the *application.yml* configuration file bundled with the Translator code.

Broker Client Installation

- ClientInstallationHelper: It is used when registering a new application node in order to provide the needed EPA installation instructions. These instructions are relayed to the Cloudiator to carry them out.

Classes of EPA

- BaguetteClient: It is the application executed when launching an EPA. It loads the configuration and instantiates the Sshc class.
- Sshc: It provides an SSH client used to connect to the EMS Baguette server. It also takes care of the communication details. The commands received from the server are replayed to the Command Executor.
- CommandExecutor: It carries out the command received from the server. It instantiates a local event broker and a CEP engine, and subsequently uses them to process the events received.

Classes of Broker Client library

- BrokerClient: It is used to connect to a configured event broker and publish or receive events.

**The most recent version of EMS (RC2.0)** is available in Bitbucket at:

https://bitbucket.7bulls.eu/projects/MEL/repos/upperware/browse/event-management?at=refs%2Fheads%2FRC2.0

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

**Dependencies:** Spring-boot framework, ActiveMQ library, Esper CE library, CDO client, Apache MINA SSH library, JGraphT library, and Apache Commons libraries (lang3 and text).

# 9 Conclusions

This deliverable encapsulated a detailed description concerning the approach, the business logic and the development of each one of the following core components of Upperware: *CP Generator, Metasolver, CP Solver, Utility Generator, Solver to Deployment, Adapter* and *Event Management System*. These Upperware components are essential for the application optimisation recommendation, the initial application placement and the continuous adaptation enactment. Their operation brings the necessary functionality to the Melodic platform, for making timely decisions on appropriate cross-cloud data placements and application deployments, starting with the generation of a formal Constraint Problem model and resulting to a target application configuration the captures the optimised deployment into cross-cloud resources.

All these Upperware components have been integrated with the rest of the platform components, consolidating a major part of the Melodic Release 2.0. The next steps involve a number of comprehensive real-world tests, across the project's pilot demonstrators. According to their evaluation results, the Upperware components will be further improved and fine-tuned as part of the finalisation development work which will lead to the final release of 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, G. Horn, K. Kritikos, F. Zahid, D. Baur, P. Skrzypek, D. Seybold, M. Prusiński, S. Mazumdar, "D2.2 Report on Architecture and Initial Feature Definitions", Melodic Deliverable, 2018.

[2] F. Zahid, D. Pradhan, "D3.2 Report on Business logic for supporting the complete data and data-intensive application life-cycle management", Melodic Deliverable, 2019.

[3] F. Zahid, K. Kritikos, S. Mazumdar, D. Seybold, Y. Verginadis, "D2.5 Report on Data Placement and Migration Methodologies", Melodic Deliverable, 2018.

[4] D. Baur, D. Seybold, "D4.3 Report on Resource Management Framework Prototype", Melodic Deliverable, 2018.

[5] F. Zahid, Y. Verginadis, G. Horn, K. Kritikos, D. and E. G. Gran, "D2.3 Report on Final framework and external APIs", Melodic Deliverable, 2019.

[6] P. Skrzypek, I. Patiniotakis, Y. Verginadis and C. Chalaris, "D5.3 Report on Security requirements & design", Melodic Deliverable, 2018.

[7] Kritikos, K., Magoutis, K., & Plexousakis, D. (2016). Towards Knowledge-Based Assisted IaaS Selection. In Proceedings of the 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxembourg, 12-15 December (pp. 431-439).

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