Deployment artefact manager

Modelling and Orchestrating Heterogeneous Resources and Polymorphic Applications for Holistic Execution and Adaptation of Models in the Cloud

H2020-ICT-2018-2020
Leadership in Enabling and Industrial Technologies: Information and Communication Technologies

Grant Agreement Number
871643

Duration
1 January 2020 – 31 December 2022

morphemic.cloud

Deliverable reference
D5.3

Date
30 June 2021

Responsible partner
7bulls

Editor(s)
Michał Semczuk

Reviewers
Marta Różańska, Alexandros Raikos

Distribution
Public

Availability
morphemic.cloud

Executive summary

In this deliverable, we present the artefact manager of MORPHEMIC, utilizing as a baseline and extending the respective component of MELODIC. The idea of this deliverable is to provide an architectural overview of the Artefact Manager component's design and implementation for the MORPHEMIC Pre-processor software platform and its underlying frameworks.

This document also describes the enhanced version of the deployment artefact management layer of the Executionware. It shows a holistic approach to building a solution that meets the requirements of supporting two platforms: MELODIC and the MORPHEMIC Pre-Processor. The artefact management layer has to: (i) propose possible deployment options to the MORPHEMIC platform, (ii) allocate, manage, and release the selected resources, as well as (iii) monitor them.

Furthermore, this document gives an overview of the enrichment of Activeeon's ProActive Scheduler, making it the full implementation of the new artefact manager, and describes its functionality and integration with the MORPHEMIC platform. ProActive Scheduler is the workflow engine to automate and orchestrate business processes and will replace MELODIC's Upperware called Cloudiator used hitherto.

Author(s)

Łukasz Szymański, Ali Fahs, Maroun Koussaifi, Chris Kachris, Mohamed Khalil Labidi, Paweł Skrzypek

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643

Table of Contents

List of Tables

List of Figures

1 Introduction

This document provides an architectural overview of the Artefact Manager component's design and implementation for the MORPHEMIC Pre-processor software platform and its underlying frameworks. The purpose of the Artefact Manager is to manage all artefacts needed for the cloud deployment of an application and its life cycle management architecture of the Artefact Manager, and the changes needed to be adjusted or developed, will be provided in this deliverable. The most important objectives of the MORPHEMIC project are to provide two types of advanced adaptation of cloud applications: proactive adaptation and polymorphic adaptation. The goal of the proactive adaptation (also known as proactive reconfiguration) feature is to provide the ability to optimize the application deployment in a given time horizon in the future. It means that the application can be optimized for the current execution context or for future execution conditions. The goal of the polymorphic adaptation is to provide the ability to optimise the (multi-cloud) application's deployment by changing the application architecture at runtime. Both of these features need the ability to manage deployment artifacts and provision them to the cloud. That led to the design and implementation of the Artefact Manager component analysed in this deliverable.

This document will focus on:

- The Artefact Manager implementations in the underlying frameworks.
- The requirements of the Artefact Manager in MORPHEMIC and the scope of implementation of this component.

1.1 Structure of the document

This document intends to present all needed modifications to the MELODIC version of the Artefact Manager that will allow the implementation of the Artefact Manager in the MORPHEMIC platform. What is more, technical requirements and assumptions for each key feature of such a component are described. Subsequently, the overall high-level architecture of the Artefact Manager with the underlying frameworks is presented. The architecture is presented in the form of diagrams, which show the interactions between key, high-level components. Also, the interactions with the underlying frameworks are presented; assuming the communication between the MORPHEMIC Pre-processor and these frameworks is conducted through exposed interfaces.

- Chapter 2 Artefact manager requirements – this part of the document describes how the artefacts are handled by Activeeon's [1] solution named "ProActive Scheduler [2]".
- Chapter 3 – a chapter describing how the artefacts are designed and handled by the MELODIC platform – a platform developed in the MELODIC project which is now being developed and expanded with new elements by the MORPHEMIC project.
- Chapter 4 ProActive Artefact Manager – a chapter gathering all requirements for the Artefact Manager which will initially be designed and drawn as an architecture model with all the dependencies from other elements of the platform and then developed and integrated with the platform.
- Chapter 5 Design and architecture – a chapter devoted to describing the design and architecture of the Artefact Manager and showing how the requirements gathered in the previous chapter are aligned to interact with all dependent elements of the platform.
- Chapter 6 Conclusions and next steps – chapter summarizing all the work conducted for the design, evolution, development, and integration of the Artefact Manager with the MORPHEMIC platform.

[1] https://www.activeeon.com/

[2] https://www.activeeon.com/products/workflows-scheduling/

1.2 Target Audience

This document is directed towards those who are interested in the development process, functionality, and usage of the Executionware and the data processing layer

2 Artefact manager requirements

2.1 Requirement collection methodology

The methodology of collecting and describing integration requirements was as follows:

1. The first step of the methodology is to carefully review the objectives of the MORPHEMIC project provided in the project's *"Description of Action" (DoA)* and to create the initial list of Artefact Manager requirements.
2. The second step is to align and extend the created list of Artefact Manager requirements with deliverable *D4.1 Architecture of pre-processor and proactive reconfiguration*. The requirements in D4.1 were based on the generic requirements for cross-cloud data-intensive applications.
3. The third step is to review the use-case application requirements as well as to adjust and refine the list of the integration requirements based on them. The conditions of use case applications are described in the *D6.1 Industrial requirements analysis* and *D6.2 Validation framework design* deliverables. The missing requirements will be identified and added to the list.
4. The final step of the methodology is to apply the cloud computing principles and requirements for modern cloud management solutions to further improve and finalise the list of Artefact Manager requirements.

These steps are illustrated in Figure 1.

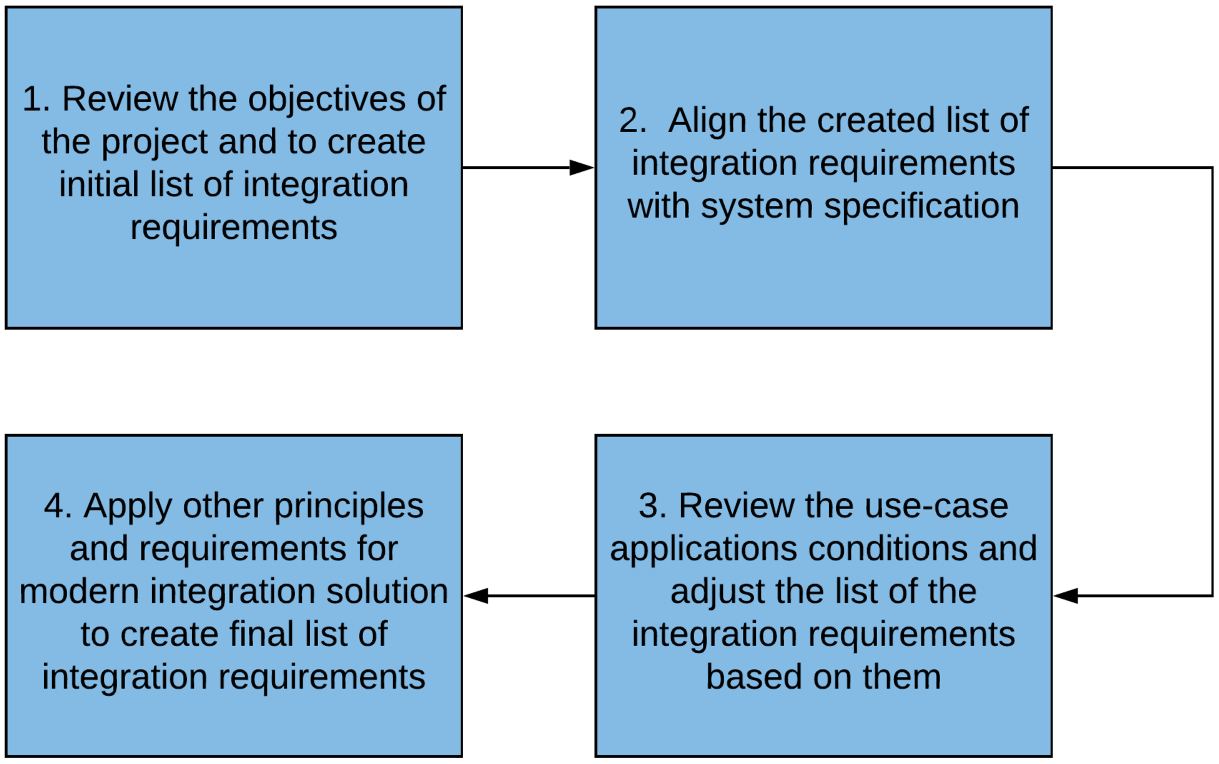

1. Review the objectives of the project and to create initial list of integration requirements
2. Align the created list of integration requirements with system specification
3. Review the use-case applications conditions and adjust the list of the integration requirements based on them
4. Apply other principles and requirements for modern integration solution to create final list of integration requirements

Figure 1 - The methodology of collecting and describing integration requirements

2.2 List of requirements

This section contains integration requirements collected using the methodology described in section 2.1. The requirements are presented in Table 1 below. The structure of the table is as follows:

- Requirement's Name – The unique name of the requirement.
- Requirement's Purpose – Purpose of introducing the requirement for the Artefact Manager component of MORPHEMIC Pre-processor.
- Requirement's description – More detailed description of the requirement.
- Priority – Priority of the given requirement. Possible values: must-have, should-have and nice-to-have.

Table 1 -List of requirements for Artefact Manager

Req. Id	Requirement's Name	Requirement 's Purpose	Requirement's description	Priority
1	Cloud resource discovery - VMs	To provide the available resources (VMs) to MORPHEMIC to allow proper reasoning and selecting an optimal application deployment solution.	The list of the available resources should be fetched from defined cloud providers. The resources should be filtered against the set of constraints, which define what type of resources should be provided to MORPHEMIC.	Must-have
2	Security groups and rules management	Security groups configuration for specific cloud providers.	Ability to flexibly configure security groups based on the application requirements.	Must-have
3	Cloud resource cache	To cache available resources and offerings and allow MORPHEMIC to use them in the reasoning process.	The cache of the resources should be available and should provide fast access to the data. The cache will be used by MORPHEMIC Pre-processor reasoning part	Must-have
4	Cloud resource pricing data collection	Detailed pricing information should be collected for cloud resources. Such information should include various options for pricing, including contracts, usage discounts and more.	The pricing information should be stored in cache and provided to the MORPHEMIC reasoning part to complete the reasoning process.	Must-have
5	Cloud images management	The available list of cloud images per cloud provider should be collected so as to be supplied to the reasoning process of MORPHEMIC.	The detailed list of images with their attributes should be collected from defined cloud providers. The images should be collected based on predefined filtering constraints. It should be possible to also collect	Must-have

Req. Id	Requirement's Name	Requirement 's Purpose	Requirement's description	Priority
			the relation between the images (dependencies) if exists.	
6	Life cycle management	Supporting the complete life cycle of application usage.	Ability to manage the complete life cycle of application (infrastructure provisioning, application deployment, application deployment termination, infrastructure decommissioning)	Must-have
7	Deployable binary artefacts management	The binary artefacts are application elements to be deployed to the cloud.	It should be possible to manage binary, deployable artefacts of the application to allow smooth deployment to the cloud.	Should-have
8	Docker images management	The docker images are application elements to be deployed to the cloud.	It should be possible to manage docker images of the application to allow smooth deployment to the cloud.	Should-have
9	Cloud resource discovery – containers, serverless, hardware accelerated resources)	To provide the available resources (containers, serverless components, hardware accelerated resources) to MORPHEMIC to allow proper reasoning and, selecting an optimal deployment solution.	The list of the available resources should be fetched from defined cloud providers. The resources should be filtered against the set of rules which define what type of resources should be provided to MORPHEMIC Pre-processor.	Must-have

3 MELODIC Artefact Manager

MELODIC[3] is a multi-cloud optimization platform and automatic deployment solution of applications to different cloud providers without changing the application configuration – a fully cloud agnostic approach. The selection of cloud providers and cloud resources is fully optimized by considering various factors like price, performance, and reliability. After its initial deployment, the application is being continuously monitored, and the operation optimized. MORPHEMIC offers two adaptation forms:

- MORPHEMIC is a polymorphic adaptation that lies in the extension of the MELODIC project in order to support live architecture reconfiguration. Polymorphic adaptation is enabled when a component can run in different technical forms or using available hardware accelerators, i.e., in a Virtual Machine (VM), in a container, as a big data job, or as serverless components, etc. As such, the technical form of deployment is chosen during an optimization process to fulfil the user's requirements and needs. The quality of the deployment

[3] https://melodic.cloud/

is measured by a user-defined and application-specific utility. Depending on the application's requirements and its current workload, its components could be deployed in various forms in different environments to maximize the utility of the application deployment and subsequently user satisfaction.

- Proactive adaptation is not only based on the current execution context and conditions but aims to forecast future resource needs and possible deployment configurations. This ensures that adaptation can be done effectively and seamlessly for the users of the application.

The Artefact Manager in the MELODIC platform relies on Cloudiator's (MELODIC's original *Executionware*) Discovery Service for IaaS that allows the discovery of compute resources required to create virtual machines. The implementation of the Discovery Service interface is realised using drivers, each providing a specialized mapping logic required for the API of a particular Cloud Service Provider (CSP). The Discovery Service retrieval functionality comprises of the following methods:

- to get a single entity of the discovery model (e.g., Hardware getHardware(id: string))
- to list all available entities of one class (e.g., List[Image] listImages())

The discoverable entities that are part of the discovery model are presented and explained in Table 2 below.

Table 2 - Discoverable entities

Entity name	Explanation
Hardware	Cloud Service Provider's computational resource, which is characterized by number of cores, disk space and memory.
Image	Cloud Service Provider's resource that represents the underlying operating system of the final IaaS node. The operating system is defined by its version, architecture (e.g., 64-bit), family (e.g., AIX) and type (e.g., UNIX).
Location	Cloud Service Provider's location that represents a region where a particular resource offering is available. Locations follow a hierarchical relationship, i.e., the node is allocated in a particular availability zone that has a parent Location representing a region; they also contain geographical information, that is country, latitude and longitude of the CSP's respective data centre.
Price	A price to pay for one hour of usage of a particular combination of following resources: Location, Hardware and Image.
Virtual Machine	The virtual machine allocated in a particular Cloud Service Provider. It provides access information in terms of its public and private IP addresses as well as the user credentials.

Internally, the Artefact Manager in the MELODIC platform is based on the *Node Candidates* concept; node candidates are essentially possible virtual machine configurations, available BYON (Bring Your Own Node) nodes or serverless compute services in the form of a function (FaaS - Functions as a Service). When Node Candidates are known, the process of selecting the best-match configuration is executed and, as a result, the chosen configurations are used to allocate the resources and the deployment of the requested applications. The Upperware is collaborating with the Executionware to get all of the valid combinations of Node Candidates with respect to constraints imposed by the Cloud Service Providers, e.g., providing specific hardware (instance type) provided by available cloud providers. Since there might be a large number of possible solutions/combinations, the Discover Service provides the possibility to enforce requirements that constraint the target solutions for particular components; this allows for limiting the amount of resource combinations considered, e.g., requirement for a minimum amount of memory or CPU for the virtual machine that the application needs.

In summary, the Node Candidates generation and retrieval process is comprised of two steps:

1. The Discover Service is filtering out discovery entities (Hardware, Image and Location) by applying the constraints passed from the Upperware (CP Generator) and generating all eligible configuration combinations, while also taking into account the constraints imposed by the Cloud Service Providers.
2. The generated Node Candidates are returned to the Upperware which will select the best suited configuration based on constraint problem solving algorithms and pass it back to the Executionware to finally start the allocation of the chosen resources.

4 ProActive Artefact Manager

ProActive Workflows & Scheduling is a java-based cross-platform *Workflow Scheduler* and *Resource Manager* that is able to run workflow tasks in multiple languages and multiple environments. The Resource Manager of ProActive is responsible for the automatic management of cloud resources. It establishes a single point of contact between the orchestration and a wide set of infrastructures.

ProActive Resource manager is capable of supporting edge nodes since it provides node agents compatible with different system architectures including ARM, as well as hardware-accelerated nodes that can be connected through the provided node sources. As a result, ProActive Resource manager was selected to replace Cloudiator as the MELODIC's Executionware.

In this section, we explain the role of the Executionware first, then we describe how ProActive fulfils this role for MELODIC. Finally, we demonstrate how the artefacts of ProActive are managed.

4.1 The Role of the Executionware

An orchestration process that evolves resources aggregated from different cloud providers and infrastructures should cope with the particular way to communicate decisions to each one of them. This design complexity can be eased by a separation of concerns where the decision-making component select the action and then pass it to the Executionware component that translate the decision according to the targeted infrastructure.

Consequently, the orchestration of resources requires three separate steps, starting by a decision made to execute a certain action on the infrastructure. This decision is then transmitted to the Executionware which in turn contacts the controller of the affected infrastructure.

As a result, the Executionware should provide two main features:

1. The support for a wide array of cloud providers and infrastructures.
2. A single point of contact where the Upperware transmit decisions using unified instructions regardless of the targeted infrastructure.

4.2 ProActive as MELODIC's Executionware

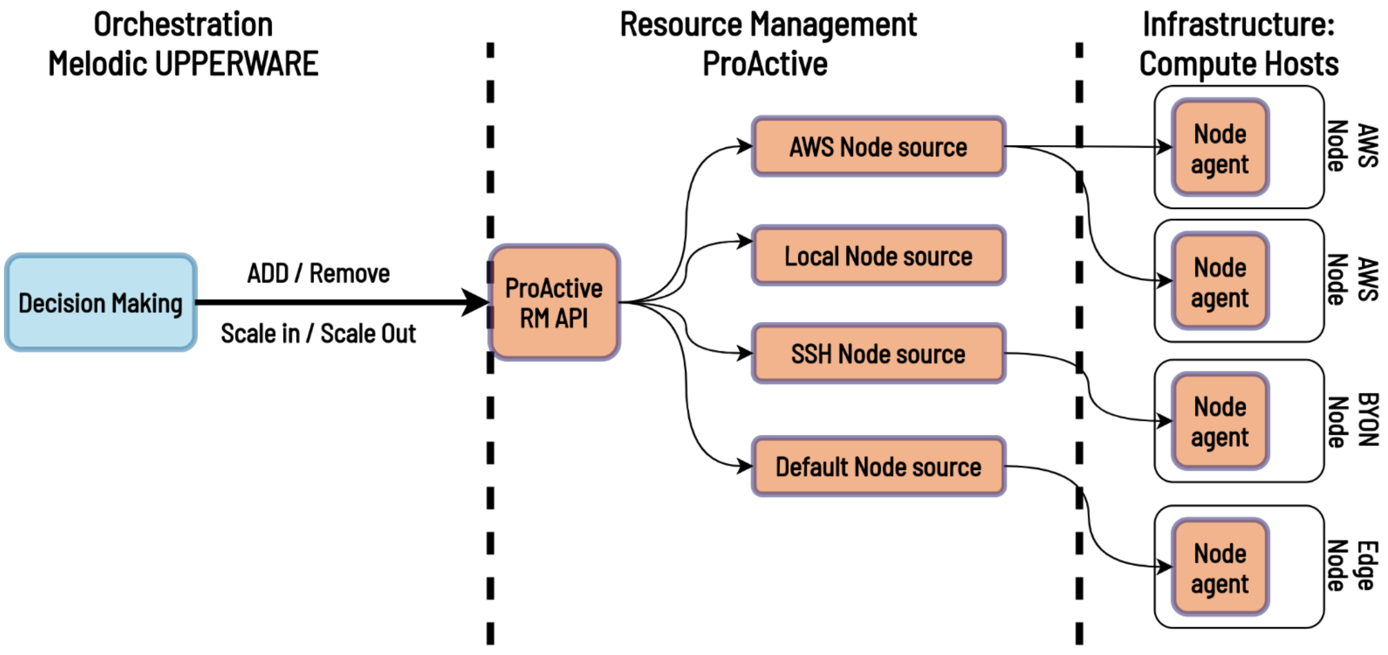


Figure 2 - A simplified flow diagram of ProActive as MELODIC's Executionware

The ProActive Resource Manager is a core element of ProActive that is responsible for organizing and monitoring the available resources. It controls the computational resources (compute hosts) as a set of *Node Sources*. Each node source controls a certain provider such that all the nodes that belongs to the same node source share the same deployment mechanism and policy. In other words, each node source is designed to handle the communication with its associated infrastructure.

The process of adding a node to the ProActive Resource Manager involves the creation of a node agent. The deployment of the node agent depends on the target machine's operating system, its architecture, and the underlying infrastructure. However, the process of communication to the connected nodes is then unified through the node agents. As shown in Figure 2, using the node sources and node agents the ProActive Resource Manager creates an abstraction layer on top of heterogeneous resources allowing the Upperware to control the nodes using unified instructions regardless of their operating system, architecture, or provider.

Also, from the Figure 2, we can conclude how ProActive can fulfil the role of MELODIC's Executionware, first, by providing the Upperware with the resource manager API acting as a single point of contact to all the infrastructures and second, by managing the communication with a wide array of infrastructures.

4.3 Managing ProActive's Artefacts

The ProActive artefacts to be managed consist of: (a) the server that is a central component where the API and node sources are created, and (b) the node agents which are distributed components created on the compute hosts. The ProActive server can be installed on all the mainstream operating systems (Linux, MacOS, and Windows). This can be done by downloading a ProActive archive [4] that contains all of the required dependencies with no extra prerequisites. Since the basic version of ProActive is open source[5], the ProActive server can also be built from sources. In Addition,

[4] https://www.activeeon.com/myaccount/in/

[5] https://github.com/ow2-proactive

the ProActive server can be deployed using a Docker image, a virtual machine image, an Azure Marketplace offer, or an AWS image. More details on the installation of the server can be found in the Appendix Section 7.1.

MELODIC supports node candidates that are heterogenous in terms of their architecture and provider. ProActive makes these nodes available for the Upperware by connecting them using different node agents. For example, cloud nodes are deployed using a cloud node source that communicates with the cloud API. In contrast, BYON nodes are connected to the server directly over SSH. Table 3 sums the methods used to connect the compute hosts to the ProActive Server.

Table 3 - Different methods for deploying ProActive nodes and their Descriptions

Method	Requirements	Type of Node	Description
Deploying nodes locally	ProActive Server in the same compute host	Local nodes	Each ProActive server comes with nodes deployed locally; this allows the user to run jobs without attaching external compute hosts.
Using node.jar	node.jar file and an access key	Edge Nodes	Support for all the architectures, including ARMv7
Cloud node sources	Cloud Provider Access	Cloud providers, orchestration engines	Allows users to deploy on-demand machines on the cloud. This solution is available for a range of cloud providers
SSH agents	SSH connection	BYON	Using SSH, the server will automatically start remote nodes.

However, the process of creating the node agents and connecting them is done in a transparent manner. Once the node sources are created, the node agents are automatically installed on the newly created nodes.

5 Design and architecture

The purpose of the Artefact Manager is to manage all artefacts needed for cloud deployment of an application and its life cycle management. A major pre-requisite should be the collection of the requirements that will guide the scope of features for this component. These requirements should be used during the testing process to verify the completeness of the implementation. The general high-level scope of the component has been presented in deliverable *D4.1 Architecture of pre-processor and proactive reconfiguration*.

5.1 Implementation

The Scheduling Abstraction Layer (SAL) is a component that handles the translation between MELODIC Upperware and ProActive Resource Manager. Figure 3 presents an overview of the integration of the ProActive server with the MELODIC Upperware and the infrastructure composed of heterogenous resources. The artefact manager of ProActive offers various capabilities that can be used via REpresentational State Transfer Application Programming Interface (REST API) endpoints. These endpoints are then called by the Upperware using SAL.

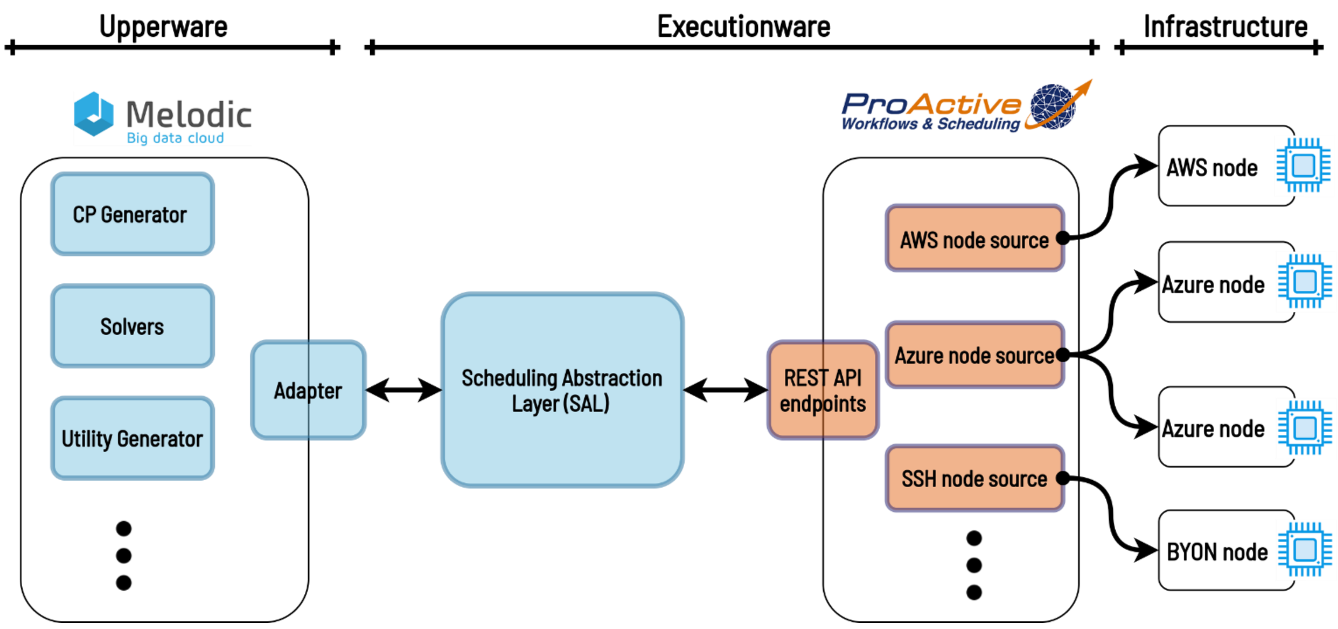


Figure 3 - MELODIC: from decision making to the underlying infrastructure.

The following sections shows how the ProActive capabilities answer all the requirements listed in Section 2.2.

5.1.1 Adding a cloud provider

The MELODIC Upperware has the capability to support several types of infrastructures based on the application's requirements. The application provider chooses the infrastructures to be considered by the Upperware, which consequently sends an Add Cloud request to SAL to execute the respective action.

The Procedure of adding a cloud infrastructure is detailed in Figure 4. After receiving the Add Cloud request, the SAL communicates with the ProActive Resource Manager API to create a node source for the targeted cloud; the API in turn creates an infrastructure controller on the cloud provider. This controller is capable of retrieving information about the available node candidates and deploying nodes once requested. When the node source is deployed, an acknowledgment traverses the pipeline to confirm its creation.

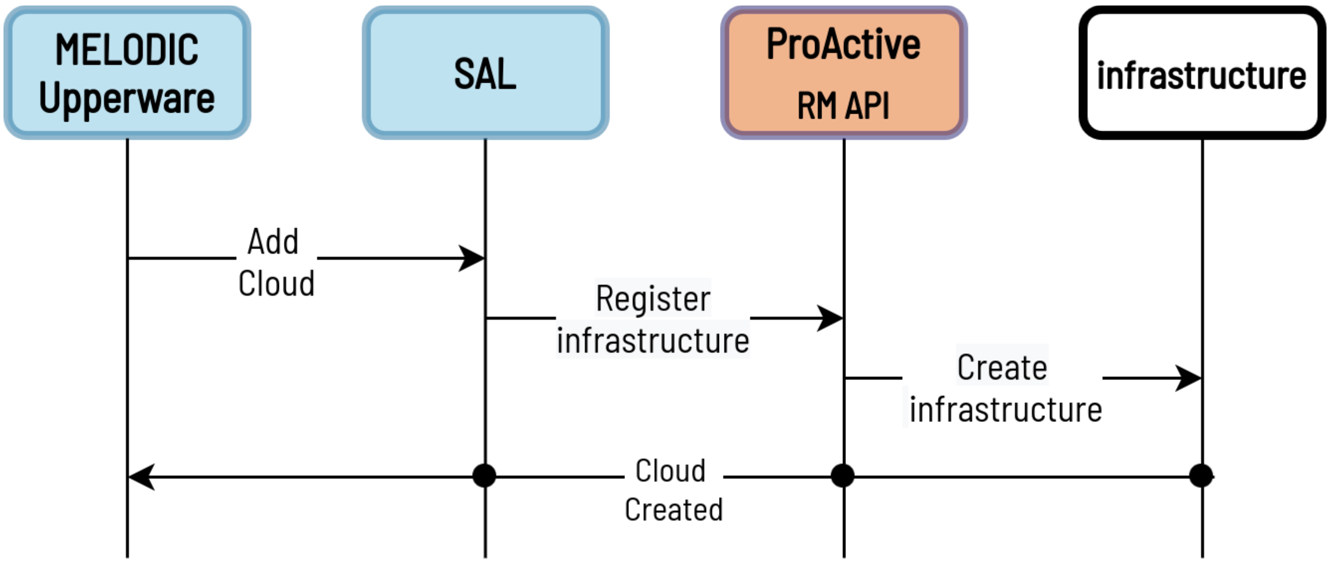


Figure 4 - The procedure for adding a cloud

5.1.2 Discovering and caching node candidates

Once a new cloud is added, the SAL sends a set of requests to collect information about the available node candidates of the newly added cloud. The information consists of the node regions, images, and hardware. Once the information is returned, the SAL will compile a list of available node candidates and store them in the SAL Database.

The information regarding the Node Candidates is vital for the operation of the Upperware. Based on the list of available nodes and on the application requirements, the Upperware filters the list of node candidates selecting the nodes that are compatible with the application to be deployed. As a result, the list of node candidates is provided to the Upperware using the SAL Database and without the need of contacting the underlying infrastructure.

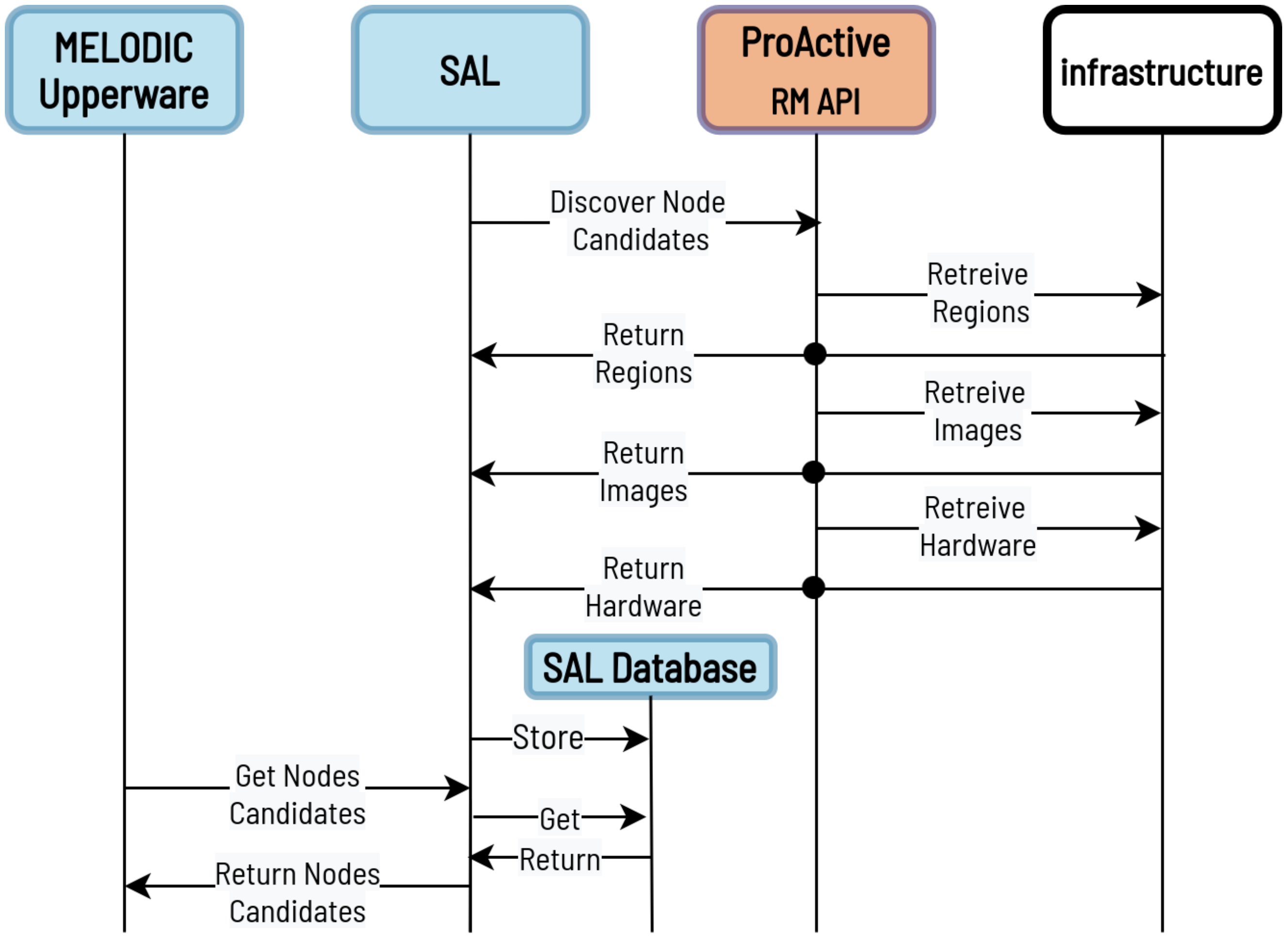


Figure 5 - The procedure of discovering node candidates

The Procedures discussed in Section 5.1.1 and Section 5.1.2 provide the means for the Upperware to fulfil the requirements of: Req-1 resource discovery, Req-3 resource caching, Req-4 resource pricing, and Req-5 image management.

5.1.3 Deployment and life cycle management

The deployment and life cycle management requirement includes the management of the resources and the applications after their deployment. From the application point of view, the ProActive Resource manager supports the ability to scale the application's resources according to instructions from the MELODIC Upperware. For example, if the application requires more resources than the ones provisioned, a scale-out instruction is sent to the Executionware to add more nodes assigned for this application. Similarly, the Executionware should support scale-in instructions that will decrease the resources assigned to the application, and eventually a stop-application instruction that will remove all the resources allocated for the application.

On the other hand, ProActive carries out these actions from the infrastructure point of view by using endpoints responsible for adding or removing nodes from the targeted infrastructure. Finally, the last step of ensuring the life-cycle management is an endpoint responsible for removing the node source of the cloud.

All of the above-mentioned endpoints operate in a similar fashion. The decision for the action is made by the Upperware and then traversed via SAL to ProActive Resource Manager that will execute the action on the target infrastructure (more on that in Section 0).

5.1.4 Security groups and rules management

The SAL layer allows the flexible configuration of security groups based on application's requirements. In fact, when a cloud is registered using the add cloud endpoint, the Upperware has the ability to set a specific single security group for the cloud. If set, all the acquired VMs will use the same security group. If not, each acquired VM will use an auto-generated security group. This security group will be generated according to the communication requirements of the application component that will be deployed on the related VM. For more details about the deployment security, you can refer to D4.2 *Security design and implementation.*

5.2 Hardware accelerators support

Hardware accelerators, e.g., Field-Programmable Gate Arrays (FPGAs), can be used to speed up significantly computationally intensive applications. Specifically, FPGAs are programmable processing platforms that can be configured with specialized architectures tailored-made on the specific applications. For example, FPGA-based accelerators for encryption can contain multiple processing elements (kernel units) optimized with digital blocks for encryption instead of having processors. In that way hardware accelerators can speed up the specific applications while also they can help reduce the total energy consumption.

However, FPGAs need to be configured with the configuration file before being available for the processing of the data. The configuration file is different for each FPGA vendor, device, or even FPGA firmware. Therefore, an Artefact Manager repository for the hardware accelerators is required for easy deployment and integration with the rest of the MORPHEMIC framework. This repository is working closely with the accelerator resource manager and it will be presented in detail in the deliverable *D5.4 Accelerator resource manager and accelerators* and allows seamless integration with the ProActive Scheduler.

The configuration file of the FPGAs is called bitstream. A bitstream is a binary sequence that contains the programming information for an FPGA. The bitstream is typically provided by the hardware designer who builds an accelerator. The major difference in the domain of FPGAs, compared with the CPU or the GPU world, is the compilation complexity of these binaries that makes it a necessity to build accelerators ahead of time.

Bitstreams are dependent on both the hardware architecture and the software stack, and need to be recompiled for each FPGA vendor, each device, and even different firmware versions. This makes building, shipping and running FPGA accelerators much more cumbersome in the data centre, where you may have different kinds of FPGA boards from various vendors, nodes with incompatible drivers, and a large portfolio of accelerators to maintain. To tackle this issue, MORPHEMIC introduce a unified bitstream packaging format accompanied by the essence of a repository for FPGA binaries.

A bitstream repository is a central place comprised of bitstream artefacts which are kept and maintained in an organized way. The concept of such a repository facilitates the development and deployment of FPGA binaries and eliminates cumbersome procedures of manually tracking them. Bitstream repositories manage the end-to-end bitstream lifecycle, being the source for bitstream artefacts needed by an accelerated application, and a target to deploy FPGA binaries generated in the build process. The use of package repositories is a common way to distribute software application binaries. Software packages are widely used as they offer several benefits, like continuous integration and delivery of upgrades, especially in large scale deployments.

Bitstream artefacts can be installed to any (local or remote) repository with a single command. The software developer can list available bitstreams in the target repository, but also get more details about the accelerators inside them.

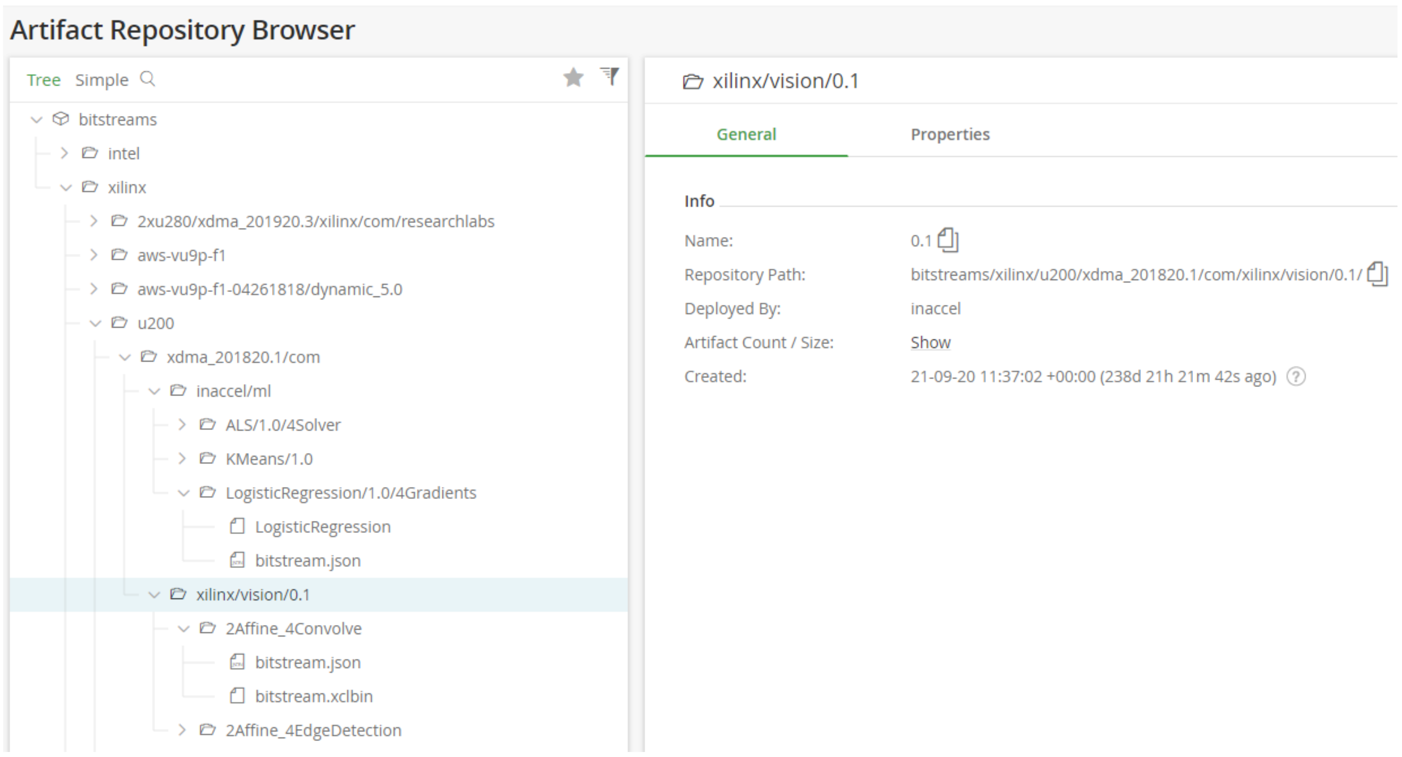
Like in every packaging system, the idea is to have a local storage on every target machine (VM or BYON node), that basically aggregates artefacts from other (public or private) remote sources. For example, an organisation may host a private central repository to distribute in-house FPGA binaries and at the same time also use 3rd-party bitstreams from public central repositories [6]. Packaging bitstream files in such an organized way makes grouping, versioning and generally maintaining a large number of bitstreams totally trivial. Apart from that, having a runtime able to manage this information allows applications to be agnostic of details like the location of the FPGA binaries in the filesystem or other platform-dependent configuration parameters. Bitstreams are now transformed into dependencies that need to be installed locally before application execution.

InAccel has developed a unique bitstream package manager that allows easy artefact management of hardware bitstream files. The bitstream manager used for FPGA is complementary to the Artefact Manager. The FPGA bitstream manager is used only in cases where the application is selected to run on accelerators. The use of a bitstream package manager that decouples these two worlds (software and hardware) is crucial for flawless deployment of FPGA accelerated applications at a data-centre scale. It makes FPGA accelerated libraries more generic and easier to maintain, almost like any other software library, but also creates the notion of hardware packages that need to be installed as dependencies and can be integrated and delivered autonomously. It also introduces the accelerator abstraction that enables the use of a unified API, as far as the application is concerned, even in cases in which the underlying kernels do not share the same properties.

The following figure depicts an example of a public central repository that stores different kind of bitstreams for several FPGA platforms. The bitstreams are organised by:

- The FPGA vendor (e.g., Intel or Xilinx)
- The board name (e.g., Alveo U200, U250, U280)
- The target firmware version
- The package identifier (usually reverse-DNS [organisation.project])
- The artefact version

[6] https://store.inaccel.com/



Artifact Repository Browser

- bitstreams
  - intel
  - xilinx
    - 2xu280/xdma_201920.3/xilinx/com/researchlabs
    - aws-vu9p-f1
    - aws-vu9p-f1-04261818/dynamic_5.0
    - u200
      - xdma_201820.1/com
        - inaccel/ml
          - ALS/1.0/4Solver
          - KMeans/1.0
          - LogisticRegression/1.0/4Gradients
            - LogisticRegression
            - bitstream.json
        - xilinx/vision/0.1
          - 2Affine_4Convolve
            - bitstream.json
            - bitstream.xclbin
          - 2Affine_4EdgeDetection

xilinx/vision/0.1

Info

Name:	0.1
Repository Path:	bitstreams/xilinx/u200/xdma_201820.1/com/xilinx/vision/0.1/
Deployed By:	inaccel
Artifact Count / Size:	Show
Created:	21-09-20 11:37:02 +00:00 (238d 21h 21m 42s ago)

Figure 6 - Artefact Repository Browser

The Artefact Repository for the FPGA-based accelerators will be integrated to the MORPHEMIC artefact manager through the FPGA resource manager (see *D5.4 Accelerator resource manager and accelerators*). Specifically, the FPGA resource manager will be interfacing with the Proactive Scheduler that will allow the easy deployment of FPGA-based hardware instances.

If the MORPHEMIC decides that an application component needs to be deployed on an FPGA-based instance, the Executionware communicates with the FPGA-based resource manager that is in charge of the FPGA deployment. In that case, the FPGA resource manager is interfacing with the FPGA artefact manager to allow the configuration of the FPGA with the right bitstream. The detailed architecture and sequence flow will be presented in *D5.4 Accelerator resource manager and accelerators*

By using the FPGA-based artefact repository, the integration of hardware accelerators become much easier, and it allows a faster deployment and the continuous integration of the FPGA-based accelerators.

5.3 Artefact Manager API

5.3.1 Cloud registering and resources discovery

The cloud resources discovery is triggered by the SAL whenever a cloud is registered using the endpoint **addClouds.** Since the number of the retrieved resources is considerable, the discovery is handled in a parallel fashion. For each cloud, a separate thread is used to retrieve its available resources.

The endpoint's signature is the following:

```
public int addClouds(JSONArray Clouds)

Parameters:
  Clouds: A JSON array that contains a list of Clouds to be added. Each array item is specific to one
      Cloud and contains information about:
            CloudID: An ID of the added Cloud.
            Cloud provider: the provider of the Cloud.
            Cloud credentials: the access credentials for the Cloud.
            Security Groups: the set of protocols and ports that should be exposed.

Returns: 0 if Clouds have been added properly. Greater than 0 value otherwise.
```

All the cloud providers control the application's inbound and outbound traffic using a set of security groups that acts as an instance level firewall. The SAL's layer allows the flexible configuration of security groups based on application requirements. In fact, when a cloud is registered using the **addClouds** endpoint, the flexible configuration of security groups based on application requirements is done as stated previously in Section 5.1.1.

Once retrieved, the SAL stores the cloud resources in the in a local java persistence database. The resources are mapped into multiple linked database tables (**NodeCandidate, Image, Hardware, Location, ...**) to simplify their access and management in general to the reasoning process. These node candidates are accessible through the **findNodeCandidates** endpoint.

The endpoint's signature is the following:

```
public List<NodeCandidate> findNodeCandidates(List<Requirement> requirements)

Parameters:
   requirements: A list of NodeType (IAAS, FAAS, BYON, PASS, or SIMULATION) or Attribute requirements
   such as the location, the image, the hardware, the Cloud type, the Cloud provider or the price.

Returns: A list of all node candidates that satisfy the requirements.
```

In addition to **findNodeCandidates**, the various information related to a specific cloud image, such as the provider ID, the location and the Operating System can be retrieved through the endpoint **getAllCloudImages**. The latter endpoint's signature is the following:

```
public List<Image> getAllCloudImages(String CloudID)

Parameters:
   CloudID: A valid registered Cloud identifier.

Returns: A list of available images.
```

5.3.2 Deployment binary artefacts management

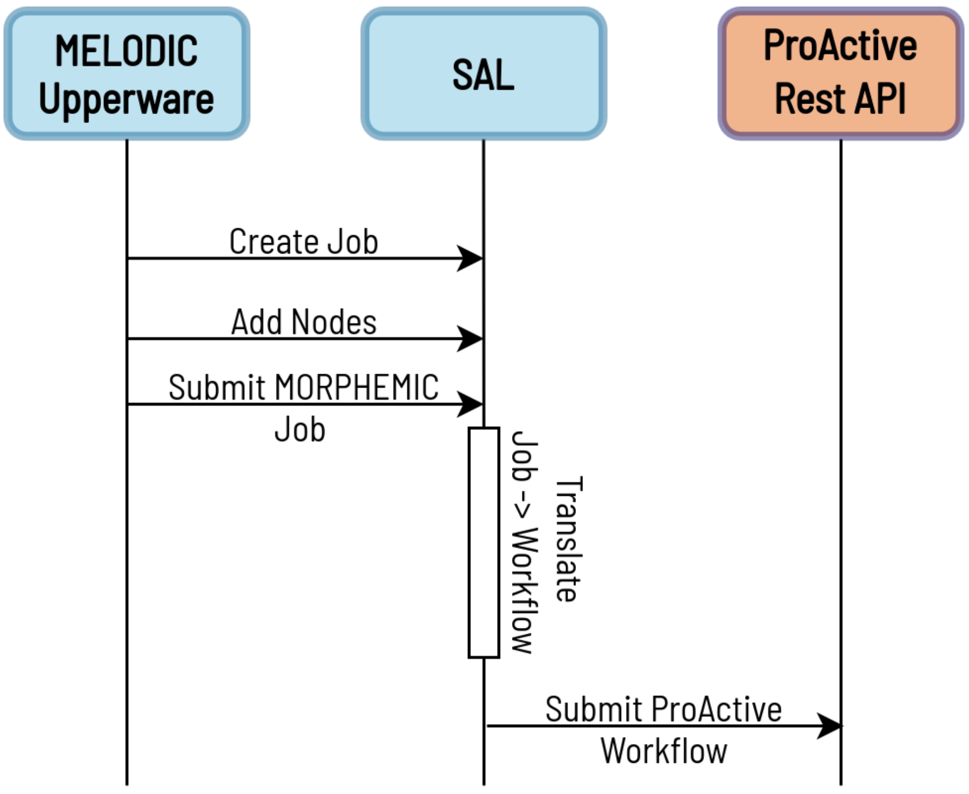


Figure 7 - Deployment binary artefacts management

Once the Upperware decides which nodes to be deployed for an application, the Upperware will first transmit a set of actions required for a proper deployment of the application. This is done by passing a job as a JSON Object to the **createJob** endpoint. The Upperware then calls **addNodes** to assign nodes to application components. Please note here that these steps are conducted locally in SAL without actually contacting the underlying infrastructure. Once SAL receives a submit job signal when the endpoint **submitJob** is called, the job is translated from a MORPHEMIC job to a ProActive Workflow. For each application component, several workflow tasks can be submitted. These tasks are related to the infrastructure provisioning and the application deployment. Basically, these tasks can:

- Acquire the resource on which the component deployment will happen.
- Prepare the infrastructure for the application installation.
- Execute the application’s installation scripts.
- Execute the application’s start scripts.

The endpoint **createJob**’s signature is the following:

```
public void createJob(JSONObject job)

Parameters:
   job: A job skeleton definition in JSON format
```

Notice that for docker containers-based applications, the same strategy is applied. The major difference in this latter scenario is that the installation and starting scripts will be based on a docker image to be pulled, environment variables to be set and communication ports to be configured.

The signatures for **addNodes** and **submitJob** endpoints are the following:

```
public int addNodes(JSONArray nodes, String jobId)

Parameters:
    nodes: An array of nodes information in JSONObject format.
    jobId: A valid job skeleton identifier defined by the user and provided in the JSONObject of the
    createJob endpoint.

Returns: 0 if nodes have been added properly. A greater than 0 value otherwise.
```

```
public long submitJob(String jobId)

Parameters:
    jobId: A valid job skeleton identifier.

Returns: The submitted job id assigned by the ProActive Scheduler.
```

5.3.3 Life cycle management

The Scheduling Abstraction Layer provides several endpoints related to the application life cycle management. The resource provisioning process explained in Section 5.3.1 and the application deployment process explained in Section 0 are both parts of the life cycle management. In addition, SAL provides management processes related to the components scaling, the application components undeployment and the infrastructure decommissioning.

MORPHEMIC allows for dynamic reconfiguration of the deployed applications. Once a running application operational context changes, the Upperware solves an optimization problem to reconfigure the provisioned resources. This may lead to scaling in or out tasks that are carried out by the SAL endpoints **addScaleOutTask** and **addScaleInTask**. On the other hand, the endpoint **stopJob** is the one in charge of undeploying a specific application. Finally, the endpoints **removeNodes** and **removeClouds** will handle the decommissioning of the resources and the registered clouds.

The aforementioned endpoints are analysed as follows:

```
public int addScaleInTask(List<String> nodeNames, String jobId, String taskName)

Parameters:
    nodeNames: A list of nodes to be removed.
    jobId: A valid job skeleton identifier.
    taskName: The name of the task whose nodes are to be removed.

Returns: 0 if the operation went successful, 2 if the operation aborted to prevent last node from being removed.
```

public **int addScaleOutTask(List<String> nodeNames, String jobId, String taskName)**

Parameters:
  nodeNames: Name of the nodes to be created and provisioned.
  jobId: A valid job skeleton identifier.
  taskName: The name of the task whose node are to be allocated.

Returns: 0 if the operation went successful, 1 If the scalling failed because the job and/or the task were not found, 2 If no nodes were defined/added to the component, in other words, if there is no VM’s specification.

public **int stopJob(String jobId)**

Parameters:
  jobId: A valid job skeleton identifier.

Returns: The submitted stopping job id assigned by the ProActive Scheduler.

public **void removeNodes(List<String> nodeNames, Boolean preempt)**

Parameters:
  nodeNames: List of node names to remove.
  preempt: If true remove node immediately without waiting for it to be freed.

public **void removeClouds(List<String> CloudIDs, Boolean preempt)**

Parameters:

  CloudIDs: List of Cloud IDs to remove.
  preempt: If true undeploy node source immediately without waiting for nodes to be freed.

6 Conclusions and next steps

In this deliverable, we have described the approach on how to evolve the artefact manager developed in the MELODIC project and adapt it to specifics of the MORPHEMIC platform without losing any of its original features. The reason for that decision is presented in deliverable *D4.1 Architecture of MORPHEMIC Pre-processor*. The presented approach relies on a specific process that is divided into three parts:

- Description of the current state of current artefact manager,
- Description of the current state of Activeeon 's ProActive Scheduler,
- Description of the future state of the MORPHEMIC's artefact manager.

The third part was further divided into the following parts:

- Requirement's collection backed up by a specific methodology and listing the new features needed to be developed,
- Design and architecture of the new artefact manager explicating the approach needed to design and build the improved solution
- Creation of the MORPHEMIC Artefact Manager’s extended API and life cycle

As such, this deliverable has supplied a holistic approach to building a solution that meets the requirements of the MORPHEMIC form. The solution created as a result of the described work will also be flexible enough to allow for easy and quick addition to the developed service-based component of new functionalities that may appear in the future and may be used by subsequent versions of the MORPHEMIC platform.

7 Appendix

7.1 Activeeon Proactive Server Documentation

7.1.1 Downloading an archived ProActive server

To install and use the ProActive server you need first to go to www.activeeon.com and complete your registration. Once this step is done, you will receive a zip file containing the latest release of the ProActive server (depending on your server OS, you can choose which installation you need). Before launching the server, you need to extract the zip file to your desired location that will be referred to as **PROACTIVE_HOME**.

7.1.2 Launching a ProActive server

7.1.2.1 As a system service

Installing ProActive Server[7] as a system service depends on your operating system; the installation of ProActive can be conducted as follows:

For Windows: Under Windows, it is possible to use the nssm[8] service manager tool to manage a running script as a service. You can configure nssm to absolve all responsibility for restarting it and let Windows take care of recovery actions. In our case, you need to provide to nssm the Path to this script **$PROACTIVE_HOME/bin/proactive-server.bat** to start ProActive as a service.

For Linux: the user has to run the **install.sh** script located under **$PROACTIVE_HOME/tools** folder as root and follow the installations steps provided in an interactive mode. Generally, it is recommended to keep the default configuration (internal account, protocol used, port, etc) unless you would like to setup a particular one.

Once the server is up, ProActive Web Portals are available at **http://localhost:8080/**. The user can access any of the four portals using the default credentials: **admin/admin**.

7.1.2.2 As a Java process

To start the server, you need to navigate your way to the location of the repository containing the server installation. Then you need to go to the bin folder. There are two ways to launch it:

1. If you have a Linux based OS or Mac you can simply run *proactive-server* from your terminal.
2. If you have a windows OS you need to launch the proactive-server.bat execution file.

The server starting time depends on the hardware specifications of your machine. Usually, it should take around 150 seconds to launch the server for the first time (time to extract the samples and install the DB), however, the next times you launch it the time is reduced to ~100 seconds. If you are launching your server for the first time on a Mac OS, you should go to your system preferences, access the “Security & Privacy” page and allow the usage of all libraries that the ProActive server needs.

7.1.3 Accessing the various ProActive services

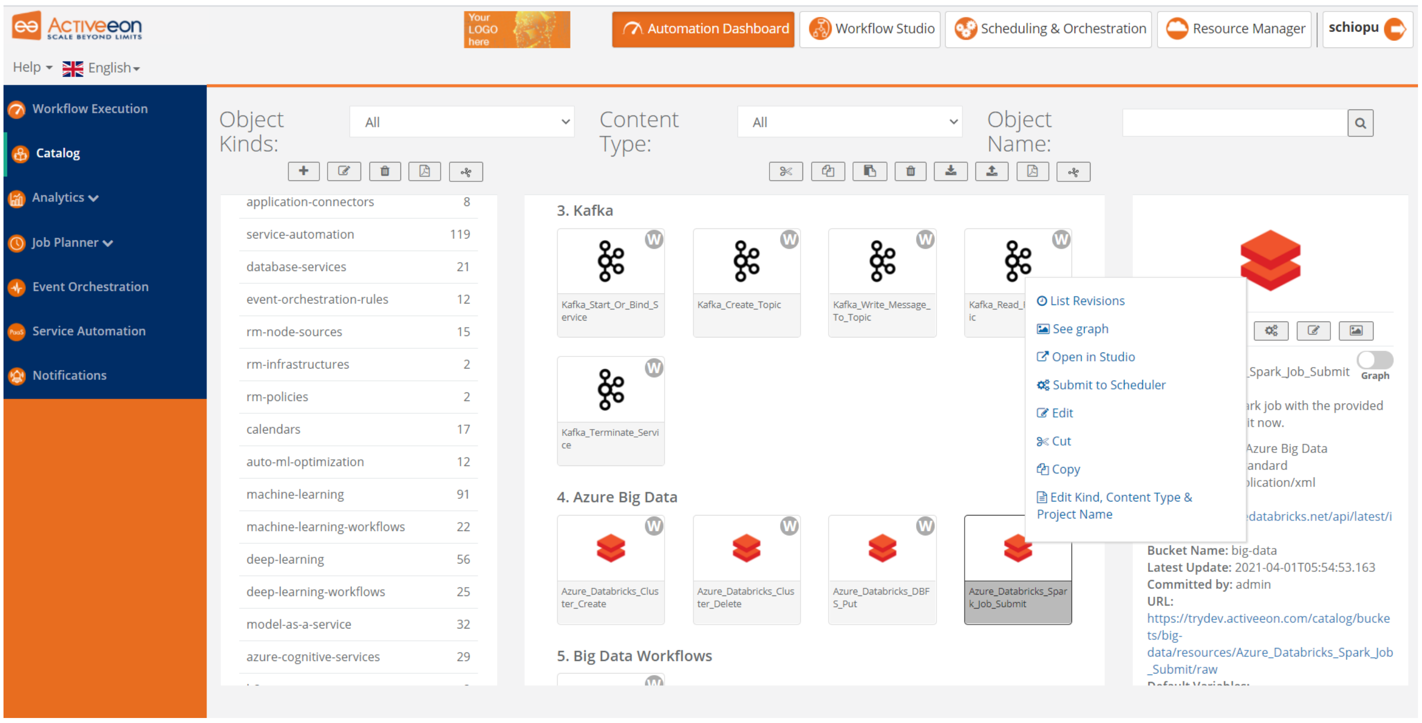
Once you have successfully launched the server, you will receive, through your terminal, the URL that allows you to access the server. Irrespectively of whether you are working on the machine where the server is installed or from a client

[7] https://www.activeeon.com/products/workflows-scheduling/

[8] https://nssm.cc/download

endpoint, you can simply go to your browser and write the complete URL that you received. Once the page is loaded, you can access the various ProActive's services, such as the Scheduler, the Resource Manager and many others.

7.1.4 Usage of the ProActive server



application-connectors	8
service-automation	119
database-services	21
event-orchestration-rules	12
rm-node-sources	15
rm-infrastructures	2
rm-policies	2
calendars	17
auto-ml-optimization	12
machine-learning	91
machine-learning-workflows	22
deep-learning	56
deep-learning-workflows	25
model-as-a-service	32
azure-cognitive-services	29

Spark_Job_Submit
rk job with the provided
it now.
Azure Big Data
andard
plication/xml
databricks.net/api/latest/i
Bucket Name: big-data
Latest Update: 2021-04-01T05:54:53.163
Committed by: admin
URL: https://trydev.activeeon.com/catalog/buckets/big-data/resources/Azure_Databricks_Spark_Job_Submit/raw

Figure 8 - Usage of the ProActive server

To use the ProActive server, you can either use it from the graphical interfaces (Web portals as shown in the figure) that each service offers or from your terminal using specific commands. All the documentation and explanation needed to completely understand how you can use the available services can be found in the ProActive Documentation[9].

7.2 Installing SAL

7.2.1 Gradle/Maven build and installation

To install SAL as a library in your project, it should be added as a project dependency. To do that, first add the following repository:

- In "**build.gradle**" for Gradle:

```
maven { url 'https://nexus.7bulls.eu:8443/repository/maven-public/' }
```

- In "**pom.xml**" for Maven:

```
<repository>
    <id>eu.7bulls</id>
    <name>Melodic 7bulls repository</name>
```

[9] https://doc.activeeon.com/dev/admin/ProActiveAdminGuide.html#_all_documentation_links

```
        <url>https://nexus.7bulls.eu:8443/repository/maven-public/</url>
      </repository>
```

Second, add the SAL dependency:

- For gradle:

```
compile group: 'org.activeeon', name: 'scheduling-abstraction-layer',
version: '3.00-SNAPSHOT'
```

- For Maven:

```
    <dependency>
        <groupId>org.activeeo </groupId>
        <artifactId> scheduling-abstraction-layer </artifactId>
        <scope>compile</scope>
      </dependency>
```

7.2.2 Manual build and installation

To manually build and install the SAL, you need to clone the project morphemic-preprocessor from this Gitlab repository URL: https://gitlab.ow2.org/melodic/morphemic-preprocessor in your local workspace. You need to build your project to generate the SAL library artefact using the following commands:

- Maven:

```
mvn install
```

- Gradle:

```
./gradlew clean build
```

Once the process is finished, you will find in the build folder inside the SAL project the *jar* that allows you to use the project as a library in your projects.

You can find more documentations on the usage and installation by accessing the project ReadMe file available in the following URL:

https://gitlab.ow2.org/melodic/morphemic-preprocessor/-/tree/master/scheduling-abstraction-layer