Deployment Modelling

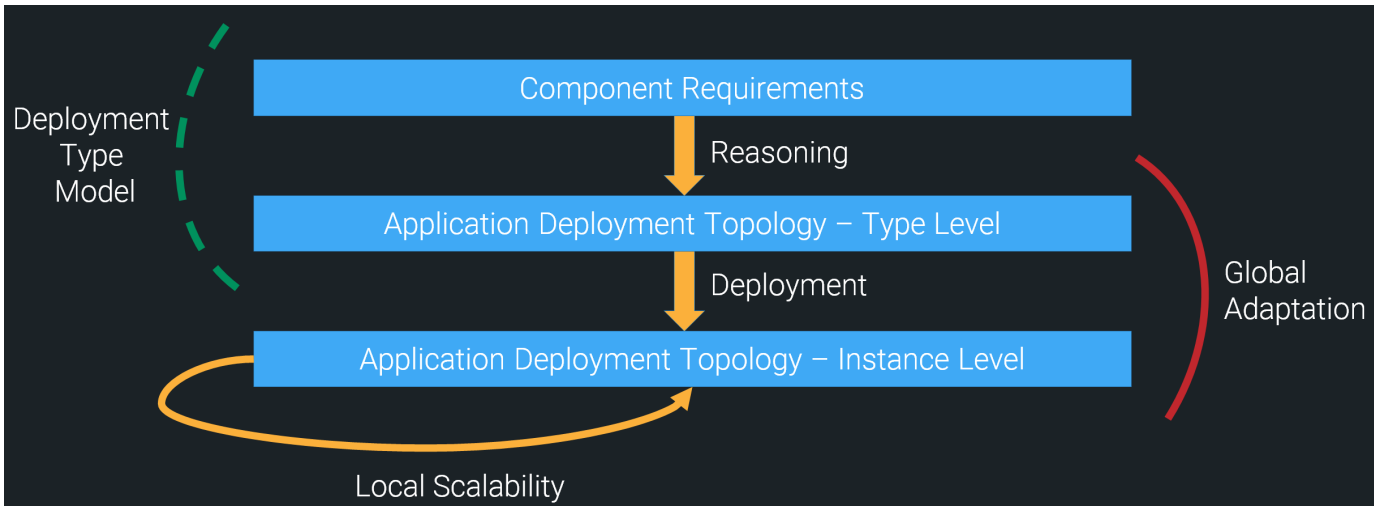
One of the core aspects captured by CAMEL is the deployment one. Through this aspect, it is possible to specify all details that are needed for the deployment of a multi-cloud application. In the past, such deployment details concerned the whole topology of the application at the type level while they did not cover the use of PaaS and serverless services. In CAMEL 2.0, this has changed. In particular, it is not needed any more to specify the whole application topology. This topology is now restrained on the level of application components and does not go down to the level of their hosting. Further, CAMEL now enables to specify application component requirements over both PaaS and serverless services. In the following, we first explain in a high-level what kinds of topologies are expected to be specified by modellers in CAMEL. Then, we attempt to analyse all possible parts of a deployment model by also supplying examples from the textual syntax of CAMEL. Please note that we confine this analysis at the type level. As this is the level that mainly concerns the modeller. The instance level should be rather handled by the respective platform in the context of maintaining the deployment state of a running application.

Deployment Topology

Traditionally and especially in other cloud application modelling languages, the fully topology of an application is covered. This means that the deployment model covers not only what are the application components and how they communicate with each other but also exactly where they are hosted, i.e., in respective VMs or other container-based elements. In other words, both hosting and communication relationships between components are covered. In CAMEL 2.0, we now take the position that hosting relationships do not need to be specified by the modeller. This is due to the following arguments: (a) flexibility: without introducing hosting relationships, we leave the freedom to the platform to decide whether particular application components should be co-hosted or not. By restraining the hosting of application components, we humper such flexibility and do not enable the possible optimisation of the application deployment through the reduction of costs by the sharing of cloud resources; (b) reduced modelling effort: the modeller does not have to specify such relationships so the modelling effort is indeed reduced. This does not mean that these hosting relationships should not be specified. However, this should be done by the platform itself once deployment reasoning has been performed and we know exactly where each application component should be hosted. However, such information also pertains to a certain deployment reasoning solution. As such, such a solution can be invalidated in the next deployment reasoning cycle. In this respect, we believe that there are actually three deployment levels that need to be handled:

1. *Component level*: here we specify the application components, their configuration and requirements, as well as possible location groupings of these components
2. *Type topology level*: here we specify the topology of the application at the type level by also introducing the hosting relationships between application and container-based/hosting components. Such a topology is valid for a certain deployment reasoning session / episode. Which means that it should be modified once application reconfiguration should be performed.
3. Instance topology level: this is the level where the actual application deployment state is covered. This level should conform to the type one, indicating that each instance of a type element should respect the information that has been specified for that element at the type level (especially with respect to its relationships and its scalability bounds). Thus, this level is bound to the type one and should be modified once the type level is also adjusted/adapted. However, this is also the level that might change most frequently with respect to others due to the execution of scalability rules in the context of local/cloud-specific application reconfiguration.

The following figure depicts these three levels and their relationships. As it can be seen, each level drives the next one in the respective abstraction hierarchy. With the main highlight that if the top level is modified, then we actually change the application and/or its requirements and thus we need to initiate a new application deployment episode. From the above analysis, it is apparent that the modeller should specify only the first level while the two lower ones should be only handled by the respective platform. At the model level, this means that in the context of a certain application deployment episode, we will have one component level model (type), multiple type topology models, each pertaining to a different global reconfiguration episode and multiple instance topology levels, each confined to one global reconfiguration episode and possibly evolved due to the execution of local reconfiguration actions / scalability rules.



Core CAMEL Deployment Elements / Parts

Application Components

In CAMEL 2.0, we have a separation between application components and hosting components. Each component kind subclasses the *Componen t* class. Application components map to pure application software modules that can be deployed in the cloud. On the other hand, hosting components, like VMs, containers or PaaS services, can be used for this deployment, i.e., the hosting of application components.

An application component can take different forms, where form can be enforced through the adoption of the respective configuration (kind) specified for that component. These forms are the following: (a) *normal* form: here we denote a component that is deployed via a script or PaaS configuration to a hosting component (VM, container or PaaS); (b) *clustered* form: here the component will be deployed within a big data processing framework which means that it will be clustered in different VMs/hosting elements in which this framework should have been deployed; (c) *serverless* form: here the component takes the form of a serverless component which has been deployed in a serverless platform.

In the past, CAMEL did not cover the data aspect. Now, in CAMEL 2.0, this aspect has been covered which has an impact on the way application components can be specified. In particular, we can have data components which encapsulate a data source and the respective data that it contains. We can also have data processing components. Such components can consume and/or produce data. Otherwise, we can just have a traditional component which does not rely on the use of certain data or data source. Such a component could be, for instance, a UI one. Or it could be a processing component that is not bound to a certain database / data source.

Application components can be also either long- or short-lived. Short-lived application components are components which have a short duration and are usually executed on-demand, normally through the respective invocation of this component in the context of a big data processing framework. On the other hand, long-lived application components can have a long duration. Here we can distinguish between non-clustered components, which have a duration which is equal to the duration of execution of the overall user application. As well as clustered components, which could be executed via a big data processing framework but do have a long execution duration.

Application components could be co-hosted. Such a co-hosting can be performed either in the context of instances of multiple components (see Grouping section below) or of the same component. The latter is actually specified via the supply of a boolean parameter named as *coInstanceHo sting*.

Finally, application components can provide and/or require communication ports. Such ports can then be bound to a certain communication that denotes the exchange of data/information between a pair of components. More details will be ginen in the Communication section below. It should be also noted that now we allow the textual description of a component which can explain the actual functionality of that component. While we also enable the annotation of that component via elements of the meta-data schema. Such annotations could further unveil the semantics of that component.

In the following, we supply examples of components that have been specified in the context of the CRM and PeopleFlow use cases.

Currently, in the CRM use case, the following application component has been specified:

```
software SmartDesign{

        requirements SmartDesignReqs  reference to the component's requirement set (see example in below section)

        provided communication ProviderSmartDesignHTTPPort port 9494  the component provides the communication port with
        number 9494

        script configuration SmartDesignConfiguration{  here we have the script configuration of the component (see its specification
        in the configuration section below)
        ...
        }

        longLived  this indicates that the component is long-lived (absence of this statement will indicate that the component is
        short-lived instead)

}
```

In this component specification (where the component is represented by a 'software' element), we refer to the set of (deployment) requirements that should hold for that component while we specify that this component supplies a provided communication port with a specific number. We also model the component's configuration (more details will be given in the Configuration section below) as well as that component is long-lived. Please note that if we do not specify that the component is long-lived, it will be assumed that it is short-lived. Similarly, as we do not specify a 'coInstanceHosting' statement, this means that it is assumed that instances of this component should be placed in different hosts always.

For the PeopleFlow use case, we will demonstrate the specification of components that are annotated and deal with (manage or manipulate) data. The respective textual CAMEL fragment is given below:

```
software SparkOnDemand{

    [MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataManagement.BigDataProcessing.HybridProcessing.ApacheSPARK]  here we denote that this component is amenable for clustered execution via Apache Spark

    generates data [ PeopleFlowDataModel.LocalStorageData ]  here we refer to the data that this component can generate

    consumes data [PeopleFlowDataModel.TableData, PeopleFlowDataModel.ConfigurationFiles]  here we refer to the data that this component consumes

    required communication HiveRequiredPort port 1892  here we denote that this component needs to communicate in a certain port (1892)

}

software Hive{

    data source PeopleFlowDataModel.HiveDS  reference to the data source managed / encapsulated by this component

    provided communication HiveProvidedPort port 0  here we denote that the component supplies a certain communication port

    required communication HDFSRequiredPort port 0  here we denote that the component requires to communicate via a certain port

    longLived  here we indicate that the component is long-lived

}
```

In this fragment, we specify a data processing and a data (managing) component. The data processing component is annotated with an element from the meta-data schema that denotes that it can be executed in a cluster via Apache Spark. We also refer to the data that this component consumes and generates. Finally, we specify that this component requires to communicate via a port with a certain number. Please note that this component is short-lived due to the absence of the longLived statement.

On the other hand, for the data managing component, we refer to the data source that it encapsulates, its provided and required communication ports while we denote that this component is long-lived.

For both components, we have omitted the reference to their requirement sets for brevity reasons.

Requirement Sets

An application component usually has a set of requirements that need to be satisfied for its deployment and which can guarantee the proper instantiation and execution of that component. Such requirements are confined in the form of a *RequirementSet*. This set includes a reference to various kinds of requirements that must be specified for that component and which need to be modelled in a requirement model. Such requirements include the following:

- Resource requirements: this requirement kind maps to requirements that are posed over the resources to be used for the component deployment / hosting.
- Platform requirements: this requirement kind maps to requirements over the platform used for the hosting of the component. Such a platform can be a PaaS service or a serverless platform.
- Location requirements: this requirement kind maps to requirements over the location in which the component can be hosted. We should highlight here that such requirement kind can be used for guaranteeing conformations to legislation or privacy requirements. It can be specified for both data processing and data (management) components in line with the fact that separate constraints can be posed over the location of storage and processing of data, respectively. As such, it can be actually understood that location constraints on data are indirectly posed over the components that manage or manipulate these data.
- Provider requirements: this requirement kind maps to requirements that restrain the providers/clouds in which an application component can be hosted. The modeller could specify with these requirements, in general, whether he/she prefers a public or private cloud for the deployment. Further, he/she could also specify a set of required providers in the form of a string (encompassing the provider names).
- OS requirements: this requirement kind maps to requirements over the operating system in which the application component should operate
- Security requirements: this requirement kind maps to requirements over respective security controls that should have been realised in the cloud/datacentre in which the application component will be deployed.
- Horizontal scale requirements: this requirement kind maps to requirements over the range of the number of instances of an application component. For instance, it can be specified that a component should always have at least two instances when the respective application is deployed or reconfigured in the cloud.
- Vertical scale requirements: this requirement kind maps to requirements over the respective bounds of the characteristics of resources used to support the deployment and execution of an application component. Please note that this kind of requirement is not currently supported by the Melodic platform.

More details about how all these kinds of requirements can be specified is given in the Requirements Modelling page in Confluence. In the following, we supply an example of a requirement set for the sole component in the CRM use case whole modelling has been exemplified above.

```
requirements SmartDesignReqs{

        resource CRM_DS_Requirement.SmartDesignResourceReqs  reference to the resource requirement holding for the
        SmartDesign component

        os CRM_DS_Requirement.UbuntuReq  reference to the OS requirement holding for the SmartDesign component

        horizontal scale CRM_DS_Requirement.SmartDesignHReq  reference to the horizontal scale requirement holding for this
        component

}
```

In the above fragment, we model the SmartDesignReqs requirement set which has been referenced in the SmartDesign's component specification. As it can be seen, there is a reference to three requirements of different kinds: resource requirements, OS requirements and horizontal scale requirements. These requirements have been detailed in the Requirements Modelling page in Confluence.

Configurations

As it has been stated above, depending on the form of an application component, different configurations might need to be specified for it. In the following, we analyse and exemplify all the kinds of application component configurations that can be specified in CAMEL 2.0.

A script configuration is a configuration that includes the specification of component lifecycle management scripts. Such scripts could be operation system-specific. In this respect, we need to supply the respective OS to which these script conform. Such scripts could be also devops-tool-specific. This means that they are confined for usage in the context of a devops tool which needs to be specified in the form of a string. Currently, the Melodic platform only supports docker-based scripts. Finally, a script configuration could be related to the instantiation of a certain image. We distinguish between two cases here: (a) the specification of a VM image. In this case, such an image can be used to instantiate a certain VM; (b) the specification of a docker image. In this case, such an image can be used to create a certain container instance within a VM instance and could be suitable for the co-hosting of components. Thus, CAMEL supports both VM- and container-based deployment of components through the specification of the identifiers of respective images to be utilised for this deployment.

An example of a script configuration can be seen in the CRM use case and is expressed through the following textual CAMEL fragment.

```
script configuration SmartDesignConfiguration{

        // Here we have a specification of a download command for properly downloading the respective component

        download 'sudo apt-get -y install wget &amp;amp;&amp;amp; wget http://46.16.79.27/vaadin-smartdesign.war &amp;amp;&amp;amp; wget http://4
        6.16.79.27/smartdesign.config &amp;amp;&amp;amp; wget http://46.16.79.27/sd-resources.zip &amp;amp;&amp;amp; sudo wget http://46.16.79.27/
        jre-8u172-linux-x64.tar.gz -O java.tar.gz &amp;amp;&amp;amp; wget http://46.16.79.27/createRamLoad.sh &amp;amp;&amp;amp; chmod +x
        createRamLoad.sh'

        // Here we have an install command that prescribes how the component can be installed

        install 'sudo apt-get update &amp;amp;&amp;amp; sudo apt-get install stress &amp;amp;&amp;amp; sudo apt-get install unzip &amp;amp;&amp;amp; sudo
        mkdir java8 &amp;amp;&amp;amp; sudo tar zxvf java.tar.gz -C java8 --strip-components=1 &amp;amp;&amp;amp; sudo mkdir sd-apps
        &amp;amp;&amp;amp; unzip sd-resources.zip -d sd-resources'

        //Here we have a start command which attempts to initiate the component's execution

        start 'nohup java8/bin/java -XX:+UseG1GC -Xms512m -Xmx2048m -jar vaadin-smartdesign.war -verbose true
        -databaseSuggestionsVisible true -port 9494 -serverUrl http://smartwe.melodic.dcsresearch.cas.de:8080/SmartWe/services -se
        rverWsdlUrl http://smartwe.melodic.dcsresearch.cas.de:8080/SmartWe/eim.wsdl -serverType OPEN -database melodic
        -username &amp;quot;Robert Glaser &amp;quot; > sd.log.out 2>&amp;amp;1 &amp;amp;'

}
```

From the above example, it can be seen that ubuntu-based (composite) lifecycle management scripts are defined for properly downloading, installing and starting the SmartDesign component. It should be noted that for an application component additional scripts could be defined concerning, for instance, the configuration and stopping of this component.

A cluster configuration is a configuration for a clustered component, i.e., of a component that is deployed within a big data processing framework. In this case, we should specify the following pieces of information: (a) the framework of deployment based on the use of an annotation from the meta-data schema; (b) the URL from which the code of the component can be downloaded; (c) framework-specific configuration parameters for the component deployment in the form of a list of attributes (that represent key-value pairs).

A particular example of a cluster configuration comes with the PeopleFlow use case scenario. In the following fragment, we denote the script configuration for the SparkInMemory component:

```
cluster configuration SparkInMemory{

    [MetaDataModel.MELODICMetadataSchema.Big_DataModel.DataManagement.BigDataProcessing.HybridProcessing.ApacheSPARK]

    download URL 'http://www.cet-traffic.com/sparkInMemory'

}
```

The above fragment indicates via an annotation the big data processing framework to be used for the SpartInMemory component deployment while it also supplies its download URL. As it can be seen, no special configuration of the Apache Spark framework needs to be performed.

A PaaS configuration denotes a configuration for the deployment of a component over a PaaS service. For this kind of configuration, the following pieces of information have to be specified: (a) the PaaS API (type) to be exploited; (b) the version of that API; (c) the endpoint of the API; (d) the URL from which the component code can be downloaded.

Drawing again from the PeopleFlow use-case, we supply an imaginery PaaS configuration for the Hive component based on the following fragment:

```
paas configuration HiveConfig{

    api 'CloudFoundry'  the API to be exploited for the PaaS-based deployment

    version '1.0'  the API's version

    endpoint 'http://192.34.45.56/cloudFoundry'  the API's endpoint

    download URL 'https://www-us.apache.org/dist/hive/hive-3.1.0/apache-hive-3.1.0-bin.tar.gz'  the URL from which the component code can be downloaded

}
```

As it can be seen, this imaginery PaaS configuration refers to three main elements of the API to be exploited for the component deployment, mainly the API type, version and endpoint while it also points to the URL from which the component code can be downloaded.

Finally, a serverless configuration denotes a configuration for the deployment of a component in a serverless platform. For this configuration kind, the modeller needs to specify the URL of the component's binary code (optional), whether the component will be continuously deployed (i.e., changes in its code are immediately reflected in the component deployment in the respective serverless platform) (optional – expected that by default this does not hold), a build configuration and an event configuration.

The build configuration explicates how the component can be built from its source code. This can actually happen if the following pieces of information are specified: (a) the artifact id of the component; (b) the serverless framework to be exploited for the built; (c) the URL where the source code of the component resides. Optionally, the modeller can indicate whether particular directories in the source code can be included or excluded from the built.

The event configuration explicates which kinds of events can be used to trigger the execution of the serverless application component. Currently, CAMEL 2.0 supports the configuration of HTTP trigger events that can trigger the execution of the serverless component which is usually wrapped in the form of a REST service in the platforms in which it is deployed. This configuration can be enabled through the specification of: (a) the name of the method to be used for the triggering; (b) the type of the method (e.g., GET or POST), (c) the schedule of the triggering, i.e., how often the component will be triggered; (d) the input parameters to be used for the triggering which take the form of attributes (key-value) pairs within the context of a certain feature specification. Please note that this event configuration serves actually two purposes currently: (i) the specification of how the respective serverless component should be wrapped in the form of a REST service; (ii) the specification of how this component execution can be triggered in a periodic manner according to a certain time interval. Such a periodic triggering could enable, for instance, to overcome the well-known cold start problem by keeping the container, in which the serverless component is deployed, warm.

An example of an imaginery serverless configuration can be found in the camel/camel/examples/serverless directory of the CAMEL repository (branch oxygen-new) with model name Example.camel (https://bitbucket.7bulls.eu/projects/MEL/repos/camel/browse/camel/examples/serverless/Example.camel?at=refs%2Fheads%2Foxygen-new). The following fragment denotes the way such a configuration for a certain serverless component, in that example, can be specified.

```
serverless configuration ServerlessCompConfig{
        continuous-deployment  here we denote that the component should be continuously deployed
        build configuration BuildConfig{
                artifactId "MyComp"  the id of the artifact to be generated
                framework "maven"  the name of the framework to be used for the build
                source code URL ""  the URL from which the component source can be downloaded
        }
        event configuration EventConfig{
                method name "myComp"  the name of the HTTP method used for triggering the serverless component
                method type get  the type of the HTTP method
                schedule MetrModel.TenMinutes  reference to the trigger's schedule
                config feature ExecParams{
                        attribute param1 : int 12  first input parameter value for the triggering
                        attribute param2 : string "Hello World"  second input parameter value for the triggering
                }
        }
}
```

The above fragment denotes that we need to continuously deploy the serverless component with the respective build being performed according to the BuildConfig configuration. For that build configuration, we specify the id of the artifact to be produced, the framework for performing the build and the URL from which the component source code can be downloaded. The serverless configuation ends with the configuration of the events that will be used for triggering the serverless component. There we specify the name and type of the HTTP method used for triggering the component as well as how often the platform will trigger it (via a reference to a schedule of 10 minutes frequency period). In addition, we model the two input parameter values to be used for invoking the serverless component during its triggering in the form of two attributes (key-value pairs) that are grouped within the same feature.

Couplings

A coupling, as its name denotes, is a kind of coupling or grouping of application components. Such a coupling is location-based. It denotes that two or more application components should be grouped/coupled together within the same location. Such a location can be the same host (i.e., VM), the same zone, the same region or the same cloud. Each kind of location coupling serves its own purpose. For instance, hosting-based coupling can enable the deployment of two application components within the same host to cater for reducing their communication overhead to the minimum. While the coupling of components in the same zone could map to a moderate communication between the components as well as the need to individually place them in hosting components to avoid the respective overlap/interference of VM-based co-hosting. Thus, such a coupling could be used to minimise the communication delay, especially when the data exchanged between the components is not minimal, as the components will be normally placed in the same datacentre. The coupling can be relaxed or strict. Strict means that the platform should enforce by all means. While relaxed means that the platform can investigate whether the coupling should be enforced or not. Potentially, it might not be enforced if the respective utility of this solution, in the context of deployment reasoning, is higher with respect to the utility of the coupling. It should be noted here that the Melodic platform will mainly support strict coupling of components at the host level. The other kinds of location couplings could be supported though in the near future.

An example of a coupling has been drawn from the PeopleFlow use case and the following fragment shows how it is specified.

```
coupling HiveHDFSCoupling{
        type same-host  here we denote that the coupling of the components should be performed in the same host
        software [ HDFSComponent, Hive ]  here we refer to the components to be coupled
}
```

In the above fragment, we specify a same-host location coupling for two application components, namely HDFS and Hive, which are referred by it. These components should be coupled in the same host in order to reduce the respective communication overhead and cost to the minimum.

Communications

A communication indicates the exchange of data or information between a pair of application components. It is actually represented through the

coupling of the provided and required communication ports of these components. It can be also assorted with a script configuration of each communication side (to enable configuring the respective component once it needs to be (re-)connected with the other one).

An example of a communication has been drawn from the PeopleFlow use case. The following fragment denotes the communication between a specific pair of PeopleFlow application components.

```
communication SparkOnDemandToHive from SparkOnDemand.HiveRequiredPort to Hive.HiveProvidedPort
```

The above fragment denotes the communication-oriented coupling of the SparkOnDemand and Hive components where the direction of communication is from the first to the second. Indeed, the coupling is being specified through referring to the respective required communication port of the SparkOnDemand component and the corresponding provided communication port of the Hive component. As it can be seen, no configuration is needed at both sides of this communication.

This concludes the documentation of the deployment aspect in CAMEL 2.0. For the interested reader, more concrete examples of whole deployment type models in CAMEL can be found in the CAMEL repository (https://bitbucket.7bulls.eu/projects/MEL/repos/camel/browse/camel/examples?at=oxygen-new) (oxygen-new brach, camel/camel/examples directory) where the whole CAMEL models of use-case applications have been specified.