Kritikos *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2019) 8:18
https://doi.org/10.1186/s13677-019-0143-x

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Multi-cloud provisioning of business processes

Kyriakos Kritikos[1*], Chrysostomos Zeginis[1], Joaquin Iranzo[2], Roman Sosa Gonzalez[2], Daniel Seybold[3], Frank Griesinger[3] and Jörg Domaschka[3]

Abstract
The Cloud offers enhanced flexibility in the management of resources for any kind of application while it promises the reduction of its cost as well as its infinite scalability. In this way, due to these advantages, there is a recent move towards migrating business processes (BPs) in the Cloud. Such a move is currently performed in a manual manner and only in the context of one Cloud. However, a multi- & cross-Cloud configuration of a BP can be beneficial as it can allow exploiting the best possible offers from multiple Clouds and enable to avoid the lock-in effect by also having the ability to deploy different instances of the BP in different Clouds close to the locations of BP customers. In this respect, this article presents a novel architecture of an environment which realises the vision of multi-Cloud BP provisioning. This environment involves innovative components which support the cross-level orchestration of cloud services as well as the cross-level monitoring and adaptation of BPs. It also relies on a certain language called CAMEL which has been extended to support the adaptive provisioning of multi-Cloud BPs.

Keywords: Business process, Workflow, PaaS, SaaS, IaaS, BPaaS, Multi-cloud, Provisioning, Orchestration, Monitoring, Adaptation

Introduction

Cloud computing promises the infinite leasing of cheap commodity resources with a very flexible, pay-as-you-go pricing model. As such, it has been adopted as a very appealing model for application configuration, especially as also via outsourcing the management at the resource level, both operational costs and management effort are reduced. In particular, numerous applications have already been migrated to the Cloud and benefit from its main advantages.

Such a flexible and appealing configuration model seems also to raise the interest of business organisations with a portfolio of business processes (BPs), which need to be better optimised and have their costs reduced. Such organisations have already started the BP migration process to the Cloud. However, they are facing various problems including: (a) the lack of expertise; (b) vendor lock-in. Such problems could be solved by utilising multi-cloud service orchestration frameworks that have been recently introduced in the market, such as Google Anthos[1]. With the advent of these tools, better customer proximity and BP performance could be achieved by operating BP instances near the customers in data centres of potentially different cloud providers. However, most of these frameworks are either cloud-specific or require great expertise from the user to properly utilise them. Further, they do not always cover advanced deployment and adaptation scenarios which could be evident in the context of BPs. Finally, to also support true optimality, BPs need to be concretised with cloud services that might come from different cloud providers, thus giving rise to the need for cross-cloud service orchestration. The latter is also essential in adaptation scenarios, where whole cloud data centres might become unavailable such that the switch to another cloud provider is unavoidable. However, if we check the current market, we cannot yet see tools really offering cross-cloud service orchestration.

The above problems are hardened by the fact that a BP concretisation can involve multiple abstraction levels. In particular, the BP activities can be mapped to

*Correspondence: kritikos@ics.forth.gr
[1]ICS-FORTH, Heraklion, Crete, Greece
Full list of author information is available at the end of the article

[1]https://cloud.google.com/anthos/

(external) SaaS services realising their functionality. Furthermore, internal BP (SaaS) components need to be deployed on suitable IaaS services so as to properly operate. In this sense, not only cross-cloud but also cross-level cloud service orchestration must be supported, something definitely not applicable in the current market.

Even if the latter orchestration type is possible, it is insufficient for proper BP provisioning. This is due to the fact that various kinds of (functional or non-functional) faults can occur within the whole BP hierarchy. As such, there is a need to appropriately monitor and adapt cloud-based BPs across multiple levels and clouds [65]. Such monitoring kind needs a precise and flexible manner via which metrics can be computed via suitable aggregation formulas spanning different levels and clouds so as to cover all possible measurability gaps. Further, the cross-level BP adaptation needs to be performed in a coordinated manner to avoid the vicious adaptation cycle's occurrence [65], where the effects of an adaptation mechanism in one level are diminished by executing an adaptation mechanism at a different (adjacent) level. By looking at the literature, only specific cross-level monitoring and adaptation approaches have been proposed which either take into account fewer levels than needed [6, 31, 46] or are not specialised to support adaptation in the Cloud [27, 49, 66].

The above gap was closed by introducing a BP execution environment that supports the cross-level and cloud adaptive BP provisioning. This environment was developed in the context of the CloudSocket project[2] [59], providing support to all lifecycle phases of a BP [33], which needs or was moved to the Cloud, by introducing phase-specific environments. The proposed environment covers the execution phase by including innovative components which support: (a) the cross-cloud and -level orchestration of cloud services for BP execution; (b) the cross-cloud and cross-level BP monitoring and coordinated adaptation.

It also relies on the existence of CAMEL[3] [1, 52], a multi-domain-specific language (multi-DSL) which supports the management of multi-cloud applications. CAMEL has been extended in the CloudSocket project to support the specification of both the cross-level allocation of a BP as well as the cross-level BP adaptation via the use of sophisticated adaptation rules. Such rules map event patterns to an adaptation workflow which orchestrates adaptation actions that can be performed in multiple abstraction levels.

The proposed BP execution environment has been validated through its application on a CloudSocket's use-case (see "Validation" section). The validation shows the suitability of the environment and its main benefits, which include the automated and adaptive provisioning of a BP driven by a rule-based approach.

We should highlight that this article focuses on BPs that can be moved to the cloud. These can correspond to any kind of BP, whose functionality can be realised by SaaS services and whose internal software components can be deployed via using PaaS or IaaS services. Such BPs can be designed by using BP Design Environments like the one offered in CloudSocket [22, 28] and might be examined in terms of their cloud readiness via the use of assessment tools, like CloudSocket's Cloud Transformation Framework [40]. The latter tools can be used by business experts to evaluate whether a whole BP or some of its parts can be moved to the cloud along with the level of effort needed to support this migration. However, we do not enter into further details on these subjects as both BP assessment and design is out of scope of this article.

The rest of the article is structured as follows. "Related work" section analyses the related work. The extended CAMEL language is detailed in "CAMEL" section. "Business process execution environment architecture" section analyses the BP execution environment's architecture. The application of that environment on a CloudSocket's use case is elaborated in "Validation" section. Finally, the last section concludes the article and draws directions for further research.

Related work

Our work is related to three main research areas: (a) cloud service orchestration; (b) (cloud) service monitoring; (c) (cloud) service adaptation. In the following three subsections, we analyse the related work in these areas by discussing the *benefits* of the respective concepts, existing *tools* as well as *limitations* with respect to their adoption for Cloud-hosted BPs.

Cloud service orchestration

Through the evolution of Cloud computing, the need for Cloud service orchestration has been raised due to the enhanced needs of complex applications. Therefore the design time requirements of application modelling with the associated cloud resources [25] is extended with the need to select the cloud resources at runtime, deploy the resources and the application, as well as monitor and adapt both at runtime [51]. In the first place, the Cloud Orchestration Tools (COTs) [9] focused on service orchestration at the IaaS level. With emerging PaaS offerings, such as Heroku[4] or CloudFoundry[5], the need for COTs focusing on PaaS orchestration has been also raised by both the industry and academia. Consequently, to facilitate cloud-based BP provisioning, IaaS and PaaS

[2]www.cloudsocket.eu
[3]www.camel-dsl.org
[4]https://www.heroku.com/
[5]https://www.cloudfoundry.org/

services should be combined to support a holistic, cross-level, cloud service orchestration by also exploiting the full potential of the cloud while avoiding the vendor lock-in. Yet, the amount of existing cross-level orchestration tools is minimal as this research area is quite novel. In the following, existing COTs are presented with respect to their cloud service orchestration capabilities, usually restrained within one Cloud level.

Container orchestration (with tools like Kubernetes[6]) can be seen as an enabling technology for PaaS or even FaaS (Function as a Service) (see also serverless frameworks like Kubeless[7] relying on Kubernetes). For the latter, we are in-line with the definition in [32] and consider FaaS as the core of serverless computing. FaaS is an enhancement of PaaS, in the sense of exempting the user from strict PaaS offer definitions, such as available libraries or run-time environments, while leaving provisioning tasks (like scaling and monitoring) to the provider. As the orchestration on this end is part of the respective Cloud provider, we do not consider this abstraction layer in our analysis. Please also note that in contrast to our own work, Kubernetes cannot support the orchestration of different kinds of cloud services at different abstraction layers.

IaaS service orchestration

Benefits Following the need for flexible and dynamic provisioning of computing facilities, the advent of Clouds led to a hype resulting in a multitude of public and private cloud offerings. These on-demand resource offerings emphasise the *automated* application deployment and orchestration on cloud resources. Yet, as cloud provider APIs are rather heterogeneous and existing cloud standards poorly adopted by them, it is tremendously important to have an *abstraction* layer enabling the provider-independent management of Cloud offerings, so as to realize multi-cloud service orchestration in an automated manner. The approaches to enable automation and abstraction are mainly: *(i)* libraries, *(ii)* standards and models, and *(iii)* Cloud Orchestration Tools.

Tools: Libraries Common representatives for Cloud abstraction libraries are Apache jclouds[8], Apache Libcloud[9] and Fog[10]. All these libraries provide a single programmatic interface to users that abstracts provider-specific characteristics across a multitude of providers. In some cases (e.g., jclouds), also storage APIs from a subset of the providers is supported. By using the offered abstraction layer, not only the provision and deployment of IaaS resources but also the deployment of applications across different clouds providers, i.e., multi-cloud deployment, is facilitated.

Concerning jclouds, we spotted several inconsistencies / pitfalls in the abstractions. For example, there was inconsistent behaviour when listing available locations. While jclouds lists all availability zones for Amazon EC2 and Google Compute Cloud, it shows a different behaviour for OpenStack's Compute API (Nova). Further, jclouds uses Template Options to pass provider-specific features when starting VMs. For OpenStack, Template Options are required to assign an availability zone or SSH key. Contrarily, the EC2 implementation implicitly supports availability zones and automatically generates and assigns SSH keys. Other issues include missing Windows support, incomplete location hierarchies, and inconsistent behaviour when managing security groups. Apache Libcloud and Fog exhibit similar inconsistencies.

Tools: Standards and Models Standards and model specifications try to tackle cloud abstraction on the level of application and resource definition. The Open Cloud Computing Interface (OCCI)[11] targets the definition of an API for Cloud resources, primarily on the IaaS level. There have been some attempts to implement OCCI on private Clouds (e.g., OpenStack) and libraries (e.g., jclouds), but wide adoption and commercial usage are still missing.

The Cloud Infrastructure Management Interface (CIMI)[12] provides a model and protocol for managing interactions between an IaaS provider and service consumer. Efforts to implement CIMI in OpenStack and Apache Deltaclouds are already inactive; therefore, wide adoption is also missing here.

OASIS TOSCA[13] provides an open standard to describe applications and services in the Cloud. The communication with the cloud provider, and therefore the technical abstraction, needs to be provided into TOSCA implementations. OpenTOSCA[14] supplies an infrastructure of tools to run TOSCA blueprints. ARIA TOSCA[15] is a Cloud orchestration tool implementing TOSCA.

Another cloud application modelling language is CloudML [24]. Based on a CloudML application description, the CloudMF tool supports the application's deployment and adaptation across multiple cloud providers by applying the models@runtime paradigm [10].

The main concerns about TOSCA are that it offers a very abstract application model not specific to the Cloud domain, thus requiring an additional lower abstraction leve. Further, it does not capture the instance level to cater

[6]https://kubernetes.io
[7]https://kubeless.io
[8]https://jclouds.apache.org/
[9]https://libcloud.apache.org/
[10]http://fog.io/
[11]http://occi-wg.org
[12]https://www.dmtf.org/standards/cmwg
[13]https://www.oasis-open.org/committees/tosca/
[14]https://github.com/OpenTOSCA
[15]http://ariatosca.incubator.apache.org/

for the models@runtime paradigm. OCCI and CIMI suffer from a poor adoption. Anyway, adopting a language not widely used comes with the risk that the orchestration tool built on top of it might also become not very appealing. As such, it is better to not explicitly base a Cloud orchestration tool on a certain language. An internal representation model can be adopted which can be targeted by a model transformation approach focusing on supporting not only one but possibly multiple standardised languages.

Tools: Cloud Orchestration Tools Cloud orchestration tools need to not only rely on Cloud abstraction APIs but to also handle both the application deployment based on a certain topology as well as the whole lifecycle management of the resources involved [9]. The latter two aspects are usually covered by a DSL not always corresponding to one of the mentioned standards. Besides the application deployment, orchestration tools may also exhibit monitoring and adaptation features.

During the evolution of cloud computing a variety of scientific and industry driven COTs have been established [63]. A taxonomy and comprehensive analysis framework for these COTs is presented in [63].

Apache Brooklyn[16] is a framework for application modelling, monitoring, and management via blueprints that define an application using a declarative YAML syntax, complying with the CAMP[17] standard and exposing many of the CAMP REST API endpoints. An non-official extension has also been developed to manage TOSCA blueprints.

Cloudify[18] is an orchestration tool using a TOSCA-aligned modelling language for describing the topology of the application which is then deployed to allocated VMs in the Cloud. As in TOSCA, Cloudify splits the blueprint into a type and a template definition. Types define abstract reusable entities that can be referenced by templates. The types, therefore, define the template structure, by, e.g., defining the properties a template can have. The template then provides the concrete property values. This mechanism is used for nodes as well as for relationships between nodes with an application topology.

A comparison of the industry-driven approaches Cloudify, Openstack Heat[19] and the scientific approach TORCH [12] is provided in [13].

Further commercial Cloud orchestration tools, such as Scalr[20], make use of the aforementioned libraries but pursue individual rather closed ways of accessing the functionalities and extending them. This limits their use in the *BP Execution Environment*, as most of them are meant to be used as complete, holistic platforms. Current approaches in this respect split the functionalities of Cloud orchestration tools and focus on partial issues. The Cloud Native Foundation provides a landscape[21] of such tools online.

A requirement analysis of COTS is detailed and applied for Apache Brooklyn, Apache Stratos[22] and Cloudify in [9]. Apache Brooklyn uses jclouds and is, therefore, stuck to the aforementioned limitations. Cloudify comes with plugins supporting a majority of private and public clouds but lacks an abstraction layer as each model must explicitly reference cloud provider-specific features. Therefore, this work tackles those shortcomings by applying an advanced Cloud orchestration tool, described in detail in "Cloud provider engine" section.

Scientic approaches, such as the orchestrator conversation [53], propose a novel approach to orchestrate cloud applications based on a hierarchical collection of independent software agents that collectively manage them. Hereby, each agent can handle independent cloud models of even different cloud modelling DSLs and manages the specific parts of a cloud application.

The Roboconf framework [48] provides an own DSL that enables application description and execution based on a hierarchical manner. The framework supports private, public and hybrid IaaS clouds; it has been validated against the commercial tools Cloudify and Scalr.

Limitations All presented libraries, standards/models and COTs ease the automated application operation and the abstraction from cloud provider specifics. Yet, there are significant differences in their supported automation and abstraction features [9]. Moreover, tools address different abstraction perspectives: libraries focus on the pure technical API abstraction while models and standards focus on the high-level, application topology abstraction. COTs combine both perspectives by building upon established libraries and modelling concepts. Yet, the presented approaches only focus on the IaaS level; this limits their adoption for BPs which require multiple cloud level support [19]

PaaS service orchestration

Benefits While IaaS level orchestration focuses on services running on VMs, the PaaS level offers more application-centric resources based on predefined environments like run-time ones. Further, common services, such as database management systems or load balancers, can be added to theseenvironments.

[16] https://brooklyn.apache.org/
[17] https://www.oasis-open.org/committees/camp
[18] http://cloudify.co/
[19] https://wiki.openstack.org/wiki/Heat
[20] https://www.scalr.com/
[21] https://github.com/cncf/landscape
[22] Retired and therefore no further details provided: https://stratos.apache.org/

Tools The heterogeneity of existing PaaS providers is analysed in [35] and a standard profile for common PaaS offering capabilities is presented. In this scope, a model is derived, which represents the three main PaaS offering aspects: infrastructure, platform and management. Moreover, based on that, PaaS can be categorized in IaaS-centric, generic and SaaS-centric PaaS, depending on the level of provided management and potential platform control. Hence, such a classification needs to be considered during application placement reasoning since an application might demand hard requirements towards the infrastructure, which cannot be guaranteed by all PaaS providers.

An PaaS API abstracting layer is presented in [54], supporting CloudFoundry and OpenShift[23]. This API comprises a unified description model allowing a PaaS provider independent representation of an application and a generic PaaS deployment API named as COAPS. In particular, it allows the specification of a manifest for the application and its environment in a PaaS provider-independent way. Further, it provides a REST-ful API for the application's life-cycle management (e.g., *createApplication*, *destroyApplication*, etc.) that internally maps the calls to the APIs of the chosen PaaS providers. Therefore, this approach provides a generic PaaS life-cycle for PaaS deployment. Yet, the COAPS API only focuses on the PaaS level and does not consider the IaaS level. Further, it provides only the abstraction of multiple PaaS providers but not a COT's full feature set [9].

Another PaaS provider agnostic API is presented in [34],supporting the PaaS providers cloudControl[24], CloudFoundry, Heroku and OpenShift. The focus of this PaaS abstraction API relies on the separation between platform- and application-centric API interactions. Similar to [54], [34] only provides the abstraction layer for PaaS providers and not a COT's full feature set.

While [54] focuses on a uniform PaaS API and PaaS provider agnostic application description, [61] is presenting a middleware called PaaSHopper for application orchestration across multiple PaaS providers. Hereby, the supported PaaS providers are Google App Engine[25] and OpenShift. Similar to [54], PaaSHopper builds upon a uniform PaaS API and PaaS provider agnostic application description with the extension of a policy-driven orchestration layer. Hence, the PaaSHopper middleware enables composing multiple application components running at different PaaS providers into one application. As [54], PaaSHopper only focuses on the PaaS level orchestration and does not support cross-level orchestration or BP integration.

[23] https://www.openshift.com/
[24] cloudControl has been shutdown due to bankruptcy end of February 2016
[25] https://cloud.google.com/appengine/?hl=en

Limitations As PaaS APIs are even more heterogeneous than IaaS APIs, this increases the complexity of application orchestration across multiple PaaS providers. Compared to IaaS tools, PaaS abstraction tools need to increase the abstraction by reducing provider specific features to achieve a common subset of supported PaaS features. While these features can be sufficient to enable specific BPs, the presented tools only support orchestration across the PaaS level; this limits their adoption for generic BPs [19, 47].

Cross-Level orchestration

Benefits While recent research in cloud orchestration mainly focuses on either the IaaS or PaaS level, cross-level orchestration is an emerging topic in cloud research [51]. Especially, as the heterogeneity of IaaS and PaaS offerings is still evolving and consequently, the probability that different kinds of cloud services will be utilised to realise a cloud application based on its (evolving) requirements is increased. Hereby, application deployments might move from IaaS to PaaS or vice versa due to changing business requirements or cloud provider offerings [15]. For instance, if the management overhead needs to be reduced, an application deployed on IaaS can be moved to PaaS to avoid the VM-level management. In return, an application can be moved from PaaS to IaaS if dedicated control over the VM is required to apply additional security mechanisms or integrate legacy applications. Changes in the pricing of the IaaS or PaaS providers can also initiate the movement of application from one service level to another to save costs.

Tools The work in [25] identified the challenges in cross-level orchestration as well as approaches that can be used to address them. Further, a preliminary COT is presented, which exploits CloudML models to enable cross-level orchestration. Cross-level adaptation and the coverage of all suitable abstraction levels for cloud-based BPs are not within the scope of this prototype.

The cross-level orchestration approach in [14] extended Apache Brooklyn with the capability to additionally orchestrate applications over PaaS services. While [14] only focuses on the deployment aspect of cross-level orchestration, an extended version of this tool focuses also on the adaptation aspect [15]. In this context, [15] introduces an algorithm for migrating applications between IaaS and PaaS providers. Yet, cross-level monitoring and extra level-specific adaptation actions in different abstraction levels are not supported. For instance, even at the IaaS level, component scaling support is missing.

TOSCAMP [2] combines the TOSCA and CAMP models to enable cross-level orchestration over the IaaS and PaaS levels. Hereby, TOSCA models are converted into

CAMP execution plans, supporting deployment and runtime adaptations. While the approach is validated based on the deployment of a basic three tier Web application across the IaaS providers Rackspace and IBM Softlayer, the cross-level orchestration capability is only validated on a conceptual level. Further, TOSCAMP only supports a subset of TOSCA which limits its adoption for generic cloud application deployments.

JUGO [29] is the first approach to cover not only the IaaS and PaaS but also the SaaS level. It provides a generic architecture for cloud composition as well as service negotiation and matchmaking. While this work is partially overlapping with the BPaaS approach, it only provides a matchmaking framework while a cross-level orchestration framework is actually missing.

Limitations Cross-level orchestration tools provide the highest flexibility for cloud-hosted BPs as multiple service levels are supported. Yet, the existing cross-level COTs only support a subset of the desired COTs features [9] due to either prototypical implementations or reduced features to achieve a common feature set. While the presented approaches are conceptually suitable to orchestrate BPs in the cloud, in their current state they are not providing dedicated integration support for higher-level BP management frameworks [19] and tools, while not covering all possible abstraction levels.

Monitoring

This section analyses work first on service and then on cloud monitoring. While "Adaptation" section covers the analysis of approaches focusing both on service monitoring and adaptation. We acknowledge that comprehensive surveys on cloud monitoring already exist (e.g., [62]). However, apart from reviewing classical service monitoring work relevant for BPaaS monitoring, we take into account two additional dimensions related to cross-cloud and level monitoring, thus playing a complementary role to these surveys and their results. In our view, BPaaSes usually comprise multiple levels of abstraction and thus require from the monitoring process to: (a) take into account the dependencies between these levels; (b) have the ability to monitor information not in just one but multiple clouds as well as collect and potentially aggregate such information.

Service monitoring

Benefits The monitoring of services is tremendously important to Cloud-based BP management, as it enables to propagate measurement information from the infrastructure to the service up to the BP level, thus providing support to both BP deployment reasoning as well as adaptation. Where both activities then enable to meet the BP requirements even at the highest abstraction level.

Tools and Methods The event-based, non-intrusive monitoring approach in [4], developed in the Astro project[26], extends ActiveBPEL and defines RTML, an executable language targeting the monitoring of service-based application (SBA) properties. Event composition is accomplished by using past-time temporal logics and statistical functions. Monitoring is performed in parallel to BPEL process execution by focusing on the mapping of input and output messages sent or received by that process to monitoring terms (e.g., metrics).

The approach in [44] extends WS-Agreement to support the non-intrusive monitoring of both functional and non-functional properties. The EC-Assertion language is introduced, based on Event Calculus, to complement WS-Agreement with the specification of service guarantees in terms of different event types.

The SBA management platform Colombo [17] incorporates tools for monitoring, evaluating and enforcing service requirements expressed in WS-Policy. Concerning policy enforcement, it supports the execution of certain actions, such as those related to the approval or rejection of a certain message delivery.

Limitations Compared to our work, the state-of-the-art service monitoring work does not enable the monitoring system reconfiguration when respective changes occur in the BP, while it does not also cover all appropriate BP levels.

Cloud monitoring

Benefits Due to the shift of applications and BPs towards the Cloud, cloud monitoring tools and methods require to handle any kind of cloud specificity, including the heterogeneity of provider-specific monitoring APIs, as well as to consider multiple abstraction levels and the possibility of cross-cloud deployment. Further, the infrastructure maintained by them needs to be dynamically adapted so as to be aligned with the adaptive BP provisioning. The adaptive, cross level and cloud monitoring, thus, promises to provide the right basis for evaluating business and technical requirements across any abstraction level and to take the necessary BPaaS management actions in accordance to these requirements so as to evolve the BPaaS system when the respective need arises.

Tools and Methods Various tools provided by cloud providers, such as Amazon's CloudWatch[27] or CloudMonix[28], suffer from vendor lock-in as they are restricted to a single cloud. Well-established open source monitoring tools (e.g., Ganglia or Nagios) can monitor large distributed systems but cannot handle cloud environment

[26] http://www.astroproject.org/
[27] https://aws.amazon.com/cloudwatch/
[28] www.cloudmonix.com

dynamicity. This means that such tools cannot be self-adapted in the case of cloud application reconfiguration or node failures.

Cloud-aware monitoring systems like DARGOS [50] offer a scalable architecture while also focus on OS and customisable application-specific metrics. However, they do not cover additional service levels, like PaaS or SaaS. The PCMONS [16] framework focuses mainly on monitoring private cloud infrastructures. In [36] the monitoring of all cloud service levels is targeted by combining level-specific solutions in an integrated monitoring system. The system architecture is peer-to-peer thus catering for scalability while offering a set of aggregation levels for the gathered monitoring data. However, such an aggregation cannot be changed dynamically at run-time.

A scalable and elastic cloud monitoring system also depends on the applied storage backend. With the evolution of NoSQL databases to offer a suitable performance and scalability level [37], various monitoring solutions have started to rely on them [11]. Yet, the selection of a concrete NoSQL database needs to be carefully evaluated as significant differences in their scalability, elasticity and availability levels exist [56–58]. Due to this pattern of adoption, a more monitoring-centric database type has recently evolved, called a time-series database (TSDB), over NoSQL databases. This type expands NoSQL databases with further monitoring capabilities including the evaluation of statistics-based queries, the measurement aggregation and a monitoring optimised data structure [26]. KairosDB[29], InfluxDB[30] and Druid[31] are well-adopted open source TSDBs.

Tower4Clouds[32], developed in the MODAClouds project[33], is a monitoring platform for multi-cloud applications. This platform relies on a rule-based approach where cloud-provider independent monitoring rules are specified and considered by the centralised *Data Analyzer* which takes care of measurement aggregation and rule evaluation based on the measurements collected by *Data Collectors*. An interesting advantage of this platform is that it can monitor the response time, throughput and availability (another extension introduced by the SeaClouds project) of PaaS applications.

[41] developed a monitoring data distribution architecture enabling cross-site compatibility by employing semantic annotations for lifting the measurements drawn from different monitoring sources. This architecture was realised in form of a distributed semantic repository providing a SPARQL endpoint enabling to pose queries over the semantically lifted monitoring data. Such data are also published to potential subscribers via a distribution hub, which conforms to data policies focusing on the type of the data to be published.

The CASVID monitoring architecture [21] focuses on both infrastructural and application-level monitoring. However, only level-specific and not cross-level monitoring is supported, thus not actually covering all possible measurability gaps. An interesting CASVID feature is that it can automatically detect the most suitable monitoring schedule for metrics by applying a novel optimisation algorithm that selects the sampling interval with the highest utility.

The combined push and pull model for cloud monitoring in [30] intelligently switches from one model to the other based on user requirements and the monitored resources status. This model is claimed to lead to better monitoring performance and cater for different virtual resource privileges and access styles.

The window-based state monitoring framework for cloud applications in [45] is quite robust to value bursts and outliers. It follows a distributed architecture that applies decentralised tuning by enabling the monitoring system to scale to multiple monitoring nodes and allowing these nodes to rely on the local information to tune their parameters. It can exploit two optimisation techniques to reduce communication cost between a coordinator and its monitoring nodes.

The centralised framework for application-level measurement in [42] exploits the Complex Event Processing (CEP) paradigm. In this framework, metrics are mapped to event streams which can be correlated to support the computation of aggregated measurements mapping to complex metrics. An interesting event hierarchy is also employed by enabling the correlation to be achieved at the levels of the host, resource pool and metric.

The runtime model for cloud monitoring in [60] focuses on common monitoring concerns. Monitoring data are then collected based on this model which, via using various techniques, are exploited to construct a cloud's performance profile. A distributed monitoring framework realising this model was also developed with centralised collection/aggregation capabilities able to address the trade-off between monitoring accuracy and cost via adaptively managing the cloud facilities. This framework seems to be able to cover different levels in the cloud abstraction stack reaching even the application level.

Limitations As it is evident from the above analysis, there is no single approach or framework able to support the adaptive, cross level and cloud BP monitoring, making the monitoring component in our proposed BP execution solution as one of its kind. In particular, most approaches cover 1–2 levels and cannot adapt the BP monitoring infrastructure. In some cases, the approaches

[29] https://kairosdb.github.io/
[30] https://www.influxdata.com
[31] http://druid.io
[32] http://deib-polimi.github.io/tower4clouds/
[33] www.modaclouds.eu

or tools seem also to be either cloud-specific or capable to monitor applications and BPs only in the context of one cloud.

We should also note that the monitoring of system metrics in the context of PaaS providers can only be performed via the proprietary APIs offered by those providers. This is also the current way such metrics are monitored in our own work. An alternative solution, also supported via our BPaaS Execution Environment, is to install monitoring probes along with the packaged BPaaS components and rely on the capabilities of the execution environment, e.g., the Java Virtual Machine, so as to query system metrics. This latter solution, though, increases the user effort, which should be avoided. As also outlined in the last section, this issue could be ideally addressed by enhancing the current PaaS abstraction APIs with the capability to integrate PaaS-specific monitoring APIs or to produce monitoring API abstraction libraries. In this case, the monitoring API integration bares the abstraction library utilised by the orchestration engine. Further, this enables the orchestration engine to uniformly monitor applications and BPs across the different abstraction layers.

Adaptation

Benefits The need to monitor both functional and non-functional requirements, and address their violation in a proactive or reactive manner has been widely recognised by the industry and academia, as the way forward to improve and evolve service-based applications and BPs according to their requirements. To this end, multiple approaches focusing on both SBA monitoring and adaptation have been already proposed. This subsection aims at analysing these approaches and especially those featuring cross-level and cloud capabilities.

Tools and Methods: Level-Specific Adaptation The self-healing BPEL process approach in [6] relies on the Dynamo monitoring framework [5]. This approach also employs an AOP extension of ActiveBPEL[34] plus a monitoring and recovery subsystem using Drools Event-Condition-Action (ECA) rules. It also allows defining assertions over invoke, receiving and picking activities of a BP via the use of WSCoL and WSReL DSLs. This approach does not allow the dynamic selection of alternative services while it neglects the interface mismatch problem when replacing a service with a new one. Furthermore, the recovery rules cannot be changed dynamically.

The VieDAME environment [46] extended ActiveBPEL with the capability to support BPEL process monitoring and partner service substitution according to various strategies. Service substitution is supported by using service selectors which operate over a service repository. VieDAME does not explicitly address fault handling and does not provide any other adaptation mechanism.

In [31] an architecture and the MONINA DSL are introduced allowing to integrate the functionality of different components and define suitable monitoring and adaptation capabilities. Monitoring is carried out via CEP queries, while adaptation is conducted via condition-action rules. However, this architecture was not validated while it lacks cross-level and multi-cloud features.

Tools and Methods: Cross-level Adaptation In [49] a methodology for the dynamic and flexible adaptation of multi-level applications is proposed using adaptation templates and adaptation mismatches taxonomies. Templates are exposed as BPEL processes, encapsulating adaptation techniques, which are manually mapped to adaptation mismatches based on the mismatch types they can address. The taxonomies introduced are specified for each level and contain either generic or domain-specific mismatches. Cross-level adaptation is supported via the direct or indirect linking of level-specific adaptation templates. In the first case, a BPEL adaptation template invokes the WSDL interface of another template. In the second case, one adaptation template raises an event that can be caught by another template.

The integrated approach in [27] for monitoring and adapting multi-level SBAs relies on a variant of MAPE control loops where all the steps acknowledge the multi-faceted nature of the system, thus being able to reason holistically and adapt the system in a cross-level and coordinated way. The proposed methodology comprises four steps: (i) monitoring and correlation; (ii) analysis of adaptation needs; (iii) identification of multi-level adaptation strategies; (iv) adaptation enactment. Its main drawback is that it does not also explicate in detail how cross-level monitoring is performed and, in particular, the exact way the various events are synchronized.

The CLAM holistic SBA management framework [66] can deal with cross-level and multi-level adaptation issues. CLAM identifies both the application capabilities affected by the adaptation actions and an adaptation strategy solving the adaptation problem by properly coordinating an adaptation capability set. The tree-based approach proposed to define adaptation paths seems interesting but can be time-consuming. During the ranking of adaptation branches, the cost is also not considered. Finally, it does not handle functional faults.

Limitations Our work advances the above adaptation approaches by handling all possible abstraction levels, being applicable in the cloud, via the capability to dynamically concretise an adaptation workflow at runtime plus

[34] https://sourceforge.net/projects/activebpel502/

its ability to store the adaptation history for analysis purposes, which can enable adjusting the adaptation rules and subsequently optimising the cloud-based BP adaptive behaviour. It also supports the pro-active BP adaptation by employing warning events, which notify about the potential violation of BP requirements.

CAMEL

As highlighted in the previous sections, there is a need to support the adaptive, cross cloud and level BPaaS provisioning. This must be achieved in accordance to BPaaS requirements and should require the least possible input and cloud-specific expertise from the BPaaS modeller. Model-driven engineering promises to automate the tasks involved in such provisioning. It embraces models as its core citizens which encapsulate all required information to support the respective BPaaS lifecycle tasks. Models can be specified via a language which supplies the right syntax and structure for them. As advocated in "IaaS service orchestration" section, most existing cloud modelling DSLs are insufficient to cover the whole BPaaS lifecycle in terms of the information that they convey. However, there is one language, called CAMEL [1, 52], which is almost rich enough and remains at an appropriate, high abstraction level which is cloud (provider) independent. As such, this language has been selected to enable the modelling of the way BPaaSes can be adaptively provisioned in our work. To this end, CAMEL was extended to cover the whole BPaaS hierarchy plus more advanced deployment and adaptation scenarios that consider all possible cloud service types.

CAMEL is a multi-DSL, well designed to cover all appropriate information required for supporting the deployment and adaptive provisioning of multi-cloud applications. CAMEL includes multiple sub-DSLs which capture the information required for a certain aspect. The aspects covered include deployment, requirement, metric, scalability, security, organisation and execution.

CAMEL relies on EMF[35] Ecore for its abstract syntax specification, OCL[36] for modelling domain and cross-model validation rules and Xtext[37] for its concrete syntax specification. It is also supported by certain editors enabling the validation of the edited CAMEL models: (a) the default tree-based Eclipse editor; (b) the textual CAMEL editor catering for DevOps users. In CloudSocket a web-based editor[38] was also developed which enables experts to specify CAMEL models without requiring to have a knowledge of this multi-DSL.

Due to the focus of the article on adaptive BP provisioning, we now concentrate on the analysis only of those aspects of CAMEL that are most relevant and which have been extended to address the whole BP hierarchy as well as more advanced deployment and adaptation scenarios.

Deployment aspect

The deployment meta-model/sub-DSL attempts to cover a multi-cloud application's topology. It originally focused on modelling the mapping of application components on virtual machines (VMs) plus the description of their configuration and communication. This was performed at both the type and instance level, thus catering for supporting the models@runtime paradigm [10]. However, due to the need to cover additional levels, e.g., the PaaS, as well as the whole BPaaS hierarchy, the CAMEL's deployment meta-model has been extended. Part of this extension is shown in Fig. 1.

PaaS Extension The modelling of PaaS requirements and capabilities relied also on the fact that a PaaS-based component deployment is normally faster than an IaaS-based deployment due to the suitable pre-configuration of the deployment environment in the first place. The PaaS extension of CAMEL involved generating a new node type in the topology model which maps to a PaaS. This node encompasses indirectly a VM node due to the fact that PaaS is mainly used in CloudSocket for the BPaaS component provisioning. In this respect, a PaaS node is associated with requirements which concern both the way the most suitable PaaS service can be selected but also the features of the VM to be encompassed. The PaaS requirements restrain the environment in which an application can be hosted and involve constraints on features like the programming framework, runtime, scaling type, and pricing type.

The modelling of the PaaS service capabilities did not require any CAMEL extension as it relies on CAMEL's quite generic and flexible feature meta-model/sub-DSL. Further, the way application components can be configured was extended with the ability to have a PaaS-based configuration via the use of an appropriate PaaS API.

SaaS Extension CAMEL's SaaS level extension moved towards covering the lightweight modelling of SaaS services, i.e., the essential details for their exploitation, like their endpoint and type (SOAP/REST), plus their association to the BP workflow tasks that they realise. Further, two types of SaaS can be modelled: (a) external SaaS services provided by external cloud providers. This service kind is thus considered as an external component, like in the case of VMs and PaaS services, which is not directly controlled by the proposed BPaaS Execution Environment; (b) internal SaaS services, developed or purchased by the organisation owning the BP, which can be deployed in the Cloud. Such internal SaaS services were mapped to

[35] https://www.eclipse.org/modeling/emf/
[36] https://wiki.eclipse.org/OCL
[37] https://www.eclipse.org/Xtext/
[38] https://site.cloudsocket.eu/allocation-tool

a special kind of an (internal) BPaaS component. This distinction is essential from the point of level of control and integration between these two SaaS types. External SaaS services are just directly integrated at the workflow level. On the other hand, internal SaaS services are deployed first as micro-services before they can be fully bound to the workflow of the BP being managed.

Monitoring & adaptation aspects

Monitoring

The monitoring aspect is covered in CAMEL by the metric meta-model/sub-DSL [38] (see Fig. 2). This meta-model attempts to provide all necessary details for metric measurement. Metrics can be simple (e.g, *raw response time*), with measurements produced via sensors, and composite (e.g., *mean response time*) with measurements produced via aggregation formulas over other metrics (e.g., mean over *raw response time* measurements). Any metric kind is mapped to a measurement unit and to the property being actually measured.

Metrics can be associated with conditions which impose thresholds over their values. Such conditions are exploited in a twofold manner in CAMEL: (a) to specify Service Level Objective (SLO) hard requirements; (b) to specify single events that can trigger scalability/adaptation rules. These conditions also determine their evaluation context, explicating what is the object being measured (e.g., a BP component or a VM) plus what is the measurement schedule and window. In case of raw metric conditions, the sensor exploited for producing the metric measurements is also specified. In case of composite metric conditions, a set of composing metric contexts is specified to explicate also the way the component metrics of the composite metric can be computed.

Adaptation

Originally, CAMEL focused on specifying scalability rules [38] which explicate the way an application can scale. Such rules were mappings between events and scaling actions. Both horizontal and scaling actions were supported.

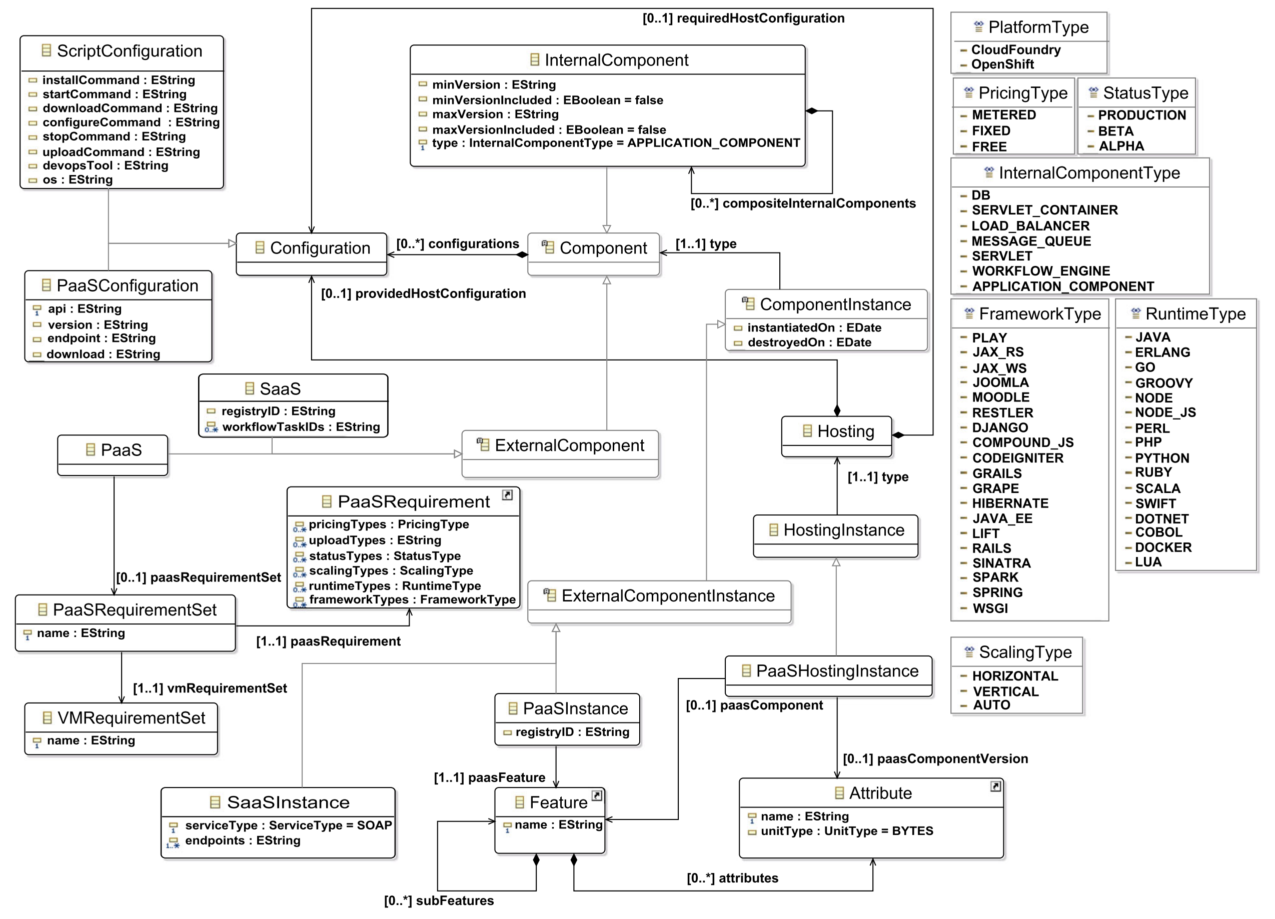


Fig. 1 Deployment sub-DSL of CAMEL

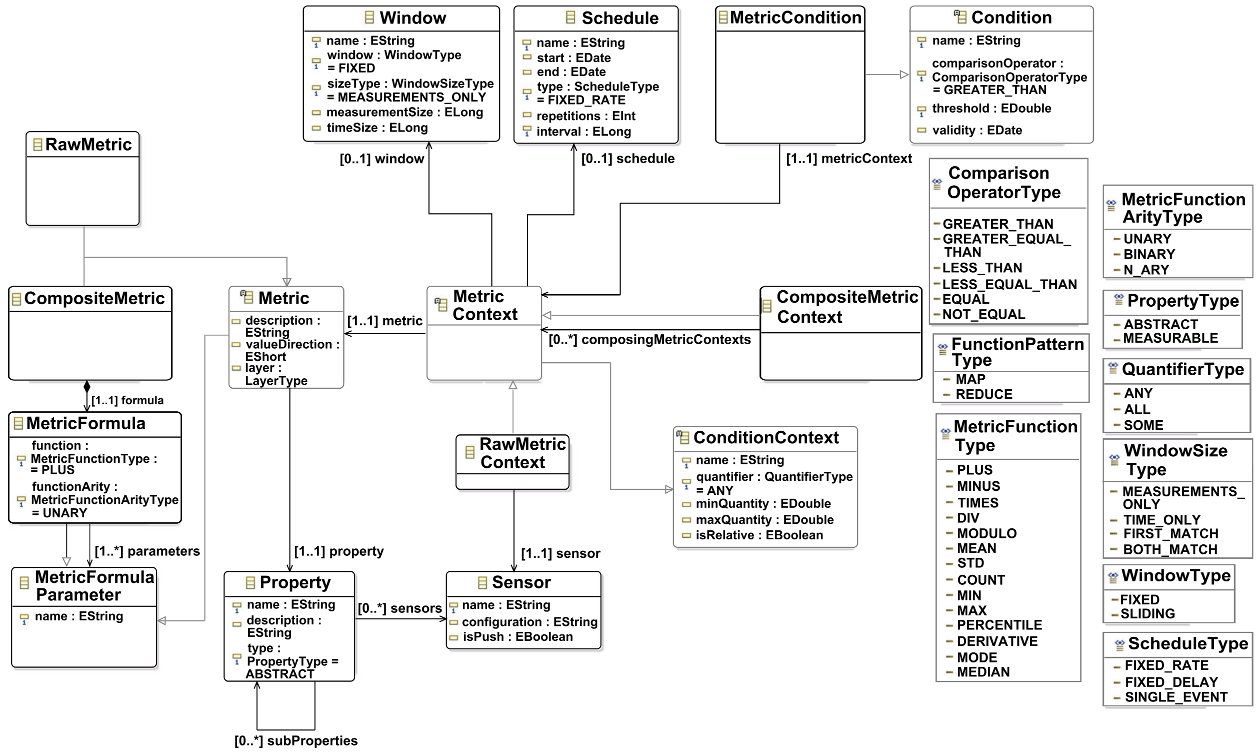


Fig. 2 Metric sub-DSL of CAMEL

Events can be single or composite. Single events map directly to metric condition violations, while composite events (named as event patterns) represent the aggregation of an event set on which a certain time or logic-based operator applies. The modelling of event patterns has been inspired by CEP languages like EPL from Esper[39]. Both unary (like *NOT* and *REPEAT*) and binary (like *AND* and *PRECEDES*) event pattern operators are supported.

The adaptation modelling, however, was too restrictive by focusing only on one abstraction level, the infrastructure one. As such, a CAMEL extension, depicted in Fig. 3, was developed replacing the scalability sub-DSL [38] with an adaptation one. In this extension, an adaptation rule maps an event to an adaptation workflow, which can involve executing different adaptation actions combined with well-known control flow constructs (sequence, parallel, choice, switch). An adaptation workflow is represented by the abstraction of an *AdaptationTask*, which can be simple or composite. A composite adaptation task applies a control-flow operator over other adaptation tasks.

On the other hand, a single adaptation task maps to a level-specific adaptation action. Thus, we have focused on the suitable modelling of different adaptation actions at different abstraction levels. At the IaaS level, we incorporated the possibility to migrate a BP component apart from scaling it. At the SaaS level, we support a SaaS service replacement in the BP workflow. At the workflow level, we support BP workflow adaptation via either this workflow re-composition or the modification of its tasks (e.g., addition, replacement).

In overall, while CAMEL was suitable to cover various multi-cloud application modelling aspects, it was extended in a minimalistic but sufficient way to better cover the peculiarities of a cloud-based BP and all the levels that it incorporates plus more optimally drive its adaptation behaviour at runtime.

Business process execution environment architecture

Our BP Execution Environment relies on a modular architecture comprising components with well-defined functionalities and responsibilities. The orchestration functionality is shared between two components while monitoring and adaptation are mainly mapped to individual components. The components communicate via well-designed interfaces following a service-oriented architecture. There is no real co-location dependency of any of the environment components. Thus, they can be flexibly distributed based on our requirements.

[39] www.espertech.com/esper/

The input to this environment is a so-called BPaaS Bundle. The bundle contains the required CAMEL model plus additional information spanning: (a) the SLA between the BPaaS provider and requester; (b) pricing; (c) business-oriented information related to the BPaaS categorisation according to certain classification schemes. This bundle, initially, is published in a *Marketplace* so as to be purchased by BPaaS customers. While originally in template form, due to missing customer-specific information, once bought, this bundle becomes concretised and is then forwarded to our environment for execution.

Our environment does not provide a concrete output in the form of a file that can be consumed by another component or environment. However, it does supply REST APIs and databases out of which suitable information can be retrieved and exploited. The information is mainly of a deployment and monitoring nature and is consumed by the BP evaluation environment in the CloudSocket prototype platform to facilitate the evaluation of Key Performance Indicator (KPIs) and other types of analysis (e.g., best BP deployment).

In the following, we analyse first the overall architecture of the BP Execution Environment and then we move to the analysis of each of the major components that provide support for the cross- level and cloud adaptive BP provisioning. More technical details about this architecture and our environment's components can be found in two CloudSocket deliverables [64], [22].

Overall architecture

Figure 4 depicts the logical architecture of the BP Execution environment. As it can be seen, it comprises 5 main components. The *Cloud Provider Engine* and the *Workflow Engine* take care of the cross-cloud service orchestration in a cooperative manner. The *Workflow Engine*, as being an execution environment for service-based BP workflows, handles the orchestration at the SaaS level while also contributing to the functionality of both monitoring and service replacement. The SaaS orchestration is

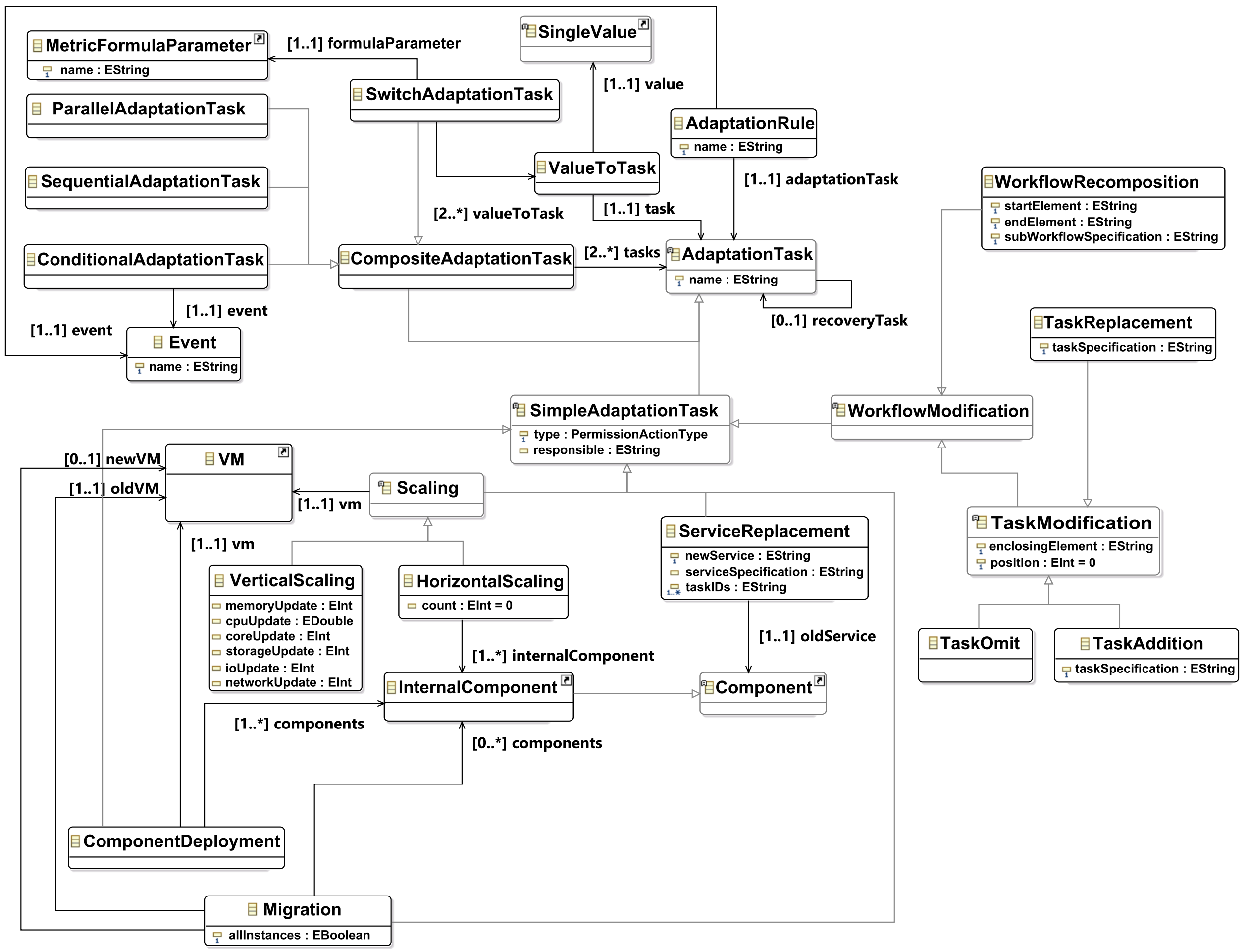


Fig. 3 Adaptation sub-DSL of CAMEL

supported by the capability to manage instances of the BP workflows deployed and execute them in the cloud. The monitoring applies mainly for the workflow and service level as the *Workflow Engine*, while executing a workflow, has the opportunity to also collect this kind of monitoring information. The contribution to service replacement mainly concerns the ability to replace the endpoint of the affected service (e.g., endpoint) mapping to the respective task with that of a new service.

On the other hand, in case the BP workflow comprises internal SaaS components, the *Cloud Provider Engine* handles their deployment to make them ready for execution by exploiting IaaS and/or PaaS services. Apart from deploying normal SaaS components, this component also deploys the monitoring infrastructure. The latter includes monitoring sensors, which provide the needed raw measurements, and measurement aggregators that aggregate these measurements in form of high-level metrics which can then be used to evaluate the respective SLOs of the agreed SLA. The *Cloud Provider Engine* also handles the deployment of the BPaaS bundle's SLA to the *SLA Engine* to support its management. Further, it communicates with the *Workflow Engine* for two main reasons: (a) to deploy the BP workflows of the input BPaaS bundles; (b) to inform the real endpoints of internal SaaS components of these workflows once these components are successfully deployed in the cloud. Thus, the *Cloud Provider Engine* plays the role of the environment's entry point as it receives the input BPaaS bundle and then orchestrates all appropriate actions and interactions in the environment to guarantee its successful deployment.

The *Monitoring Engine* handles the BP monitoring across different levels and clouds. It is the main component always requiring some kind of distribution, which is supported by exploiting the *Cloud Provider Engine* facilities. This engine comprises level-specific components supporting the sensing and aggregation of monitoring information. These components interact with a publish-subscribe mechanism. This mechanism also supports the retrieval of monitoring information for SLO assessment purposes plus the publishing of SLO evaluation information to the *SLA Engine* and *Adaptation Engine*.

The *SLA Engine* handles the SLA management. In particular, it observes the SLA state and visualises it for both BP providers and requesters. This enables the follow up of the SLAs by their main signatory parties, which can communicate with each other to handle any kind of SLO violation that might occur. The SLA state is maintained via the interaction with the *Monitoring Engine* where the *SLA Engine* subscribes to the SLO evaluations produced.

Finally, the *Adaptation Engine* handles the cross- level and cloud adaptation of the deployed BP workflows. It communicates with the *Monitoring Engine* to obtain the events that will trigger the adaptation workflows execution. The actions in these workflows map to a service-based library providing methods to realise them. These methods interact in some cases with other BP execution environment components. For instance, when performing SaaS replacement, once the new SaaS service

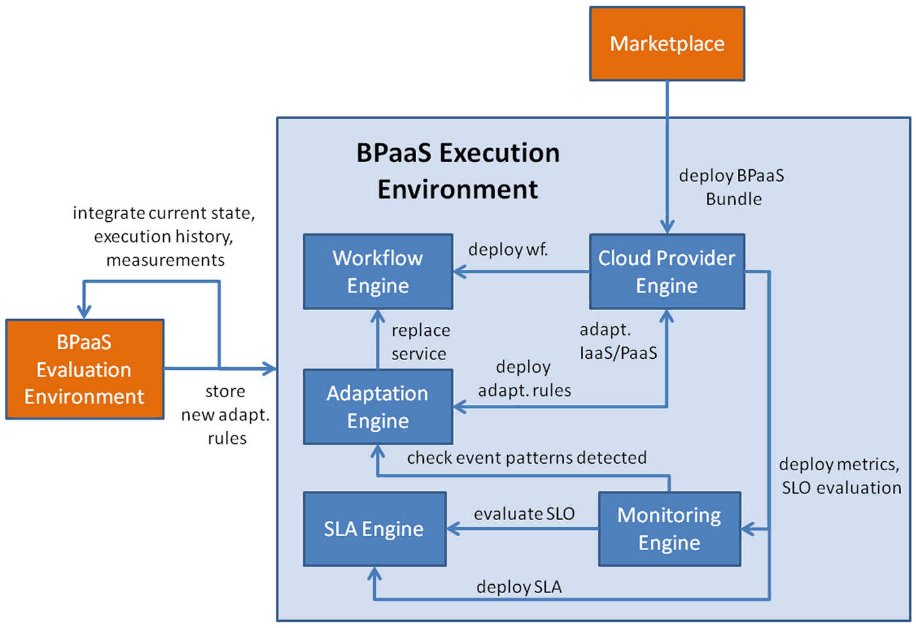


Fig. 4 The logical architecture of the BP execution environment

is identified, the *Workflow Engine* is invoked to replace the old SaaS service endpoint with that of the new one.

The above architecture is at the logical level. At the physical level, the architecture can be quite distributed as each component can operate individually without explicit technical dependencies which might require a tight coupling or co-location of two or more components. By considering the load that can incur to our environment, Fig. 5 depicts a possible physical architecture. In this architecture, the components that have most of the load and need to be distributed are the *Workflow Engine*, *Cloud Provider Engine* and *Monitoring Engine*. The *Workflow Engine* is exploited for manipulating, executing and monitoring multiple instances even of the same BP workflow. The *Cloud Provider Engine* is the environment's entry point while it handles the BPaaS bundles' deployment plus the continuous management of their components. The *Monitoring Engine* can be heavily loaded with the monitoring of multiple raw and aggregate metrics spanning the instances of multiple BP workflows.

The distribution is performed mainly at the BP level. This is accomplished by introducing a *Load Balancer* which: (a) distributes the BPaaS bundle requests among the different environment instances; (b) distributes internally the requests to internal components by having a complete picture of the BP execution environment's distributed topology. This *Load Balancer* along with the centralised components (*Adaptation Engine* and *SLA Engine*) form the central management component agglomeration. On the other hand, each environment instance comprises instances of the aforementioned three distributed components. An alternative deployment topology would position also the centralised components in each environment instance to generate a dedicated environment instance that handles internally any interactions that need to take place in the context of the deployed BPaaS bundles there. In that deployment topology, the *Load Balancer* would then be just the entry point in the overall BP execution environment with the strict load balancing responsibility only to support the distribution of that environment across all of its instances.

At a second level, as there is a need to scale a component even in the case of an environment instance, there can be a local load balancer per environment component distributing the load among all the instances of that component. The scaling could be supported by using the *Cloud Provider Engine*, which has the capability to deploying any component kind in the Cloud. As such, our physical architecture contribution comes with a two-level distribution of the proposed logical architecture. In our opinion, this two-level distribution will handle any kind of load situation that can occur in the BP execution environment. It could also enable the environment to become more robust as it would not be jeopardised by the failure of any of its component instances.

In the following, we analyse in more detail the internal architecture of all main components of the BP Execution Environment.

Cloud provider engine

The *Cloud Provider Engine* enables the provisioning of internal SaaS components for BPaaS by orchestrating them across various IaaS and PaaS providers. To this end, the *Cloud Provider Engine* builds upon the multi-tenant

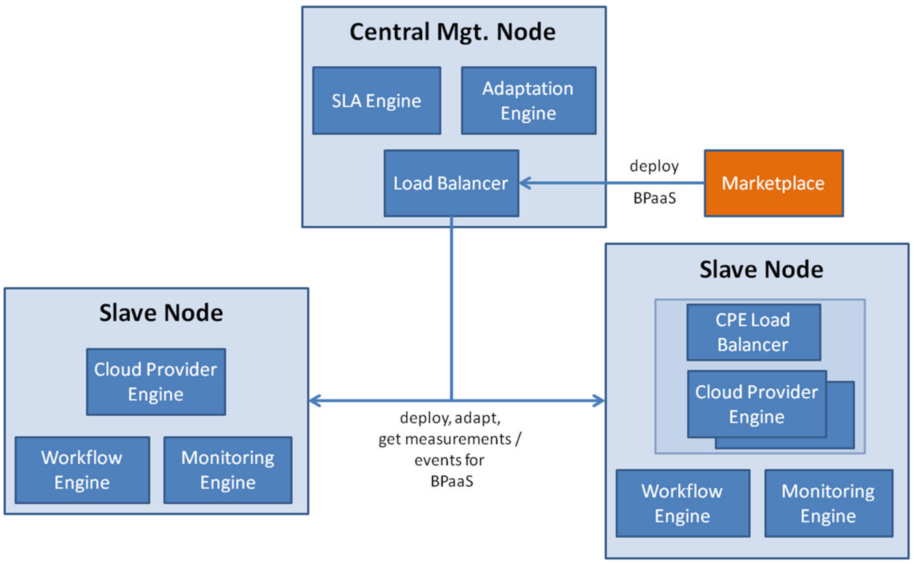


Fig. 5 A distributed architecture of the BP execution environment

COT *Cloudiator*[40] [8, 18] which enables the deployment, monitoring and adaptation of distributed applications in a multi-cloud environment [20].

Figure 6 shows a high-level overview of the *Cloud Provider Engine*, depicting the exploited components of *Cloudiator* for the internal SaaS provisioning. The *Entry point* component provides a RESTful interface to the *Marketplace*, enabling to register BP bundles for their deployment. It also supplies additional meta-data for the BP bundle and its deployment process. The BP bundle, which as already stated includes a CAMEL model, is passed to *Cloudiator*'s *Shield* component. Shield triggers the deployment of internal SaaS components by translating the CAMEL model into the required calls against *Colosseum*. The *Entry point*'s modular architecture enables supporting additional DSLs, such as TOSCA, by adding the respective *Shield* components for them.

Colosseum is the central component of *Cloudiator* which exposes a comprehensive *REST API* unifying the access to cloud resources across different service levels (IaaS/PaaS), such as the IaaS providers Amazon EC2, Google Compute, OpenStack as well as PaaS providers Heroku, OpenShift, CloudFoundry. These supported IaaS and PaaS providers go along with the commonly supported IaaS and PaaS of existing orchestration frameworks (cf. "Cloud service orchestration" section). Recent extensions add as well the orchestration support for Function-as-a-Service providers (AWS Lambda and Azure Functions) and Big Data processing frameworks (Apache Spark) [55]. To enable the abstraction of these cloud providers, two kinds of unified APIs are exploited in Cloudiator:

(1) *Sword* unifies IaaS resources in a uniform way by building and extending jclouds [7]. This extended abstraction layer supports automated cloud resource discovery in terms of hardware flavours, image, and locations for public IaaS providers. More detailed resource information can also be acquired for private clouds, such as hypervisor details or storage details. Further, *Sword's* extension of jclouds offers dedicated support for the physical layer, e.g., realising the notion of physical nodes (hypervisors) in OpenStack. Besides only listing and discovering them, *Sword* supports the selection of a physical node as a deployment location bypassing Openstack's scheduler. Apart from resource orchestration, *Sword* enables installing the *Cloudiator* internal agents *Lance* and *Visor* via a unified remote access interface for Unix and Windows to the provisioned VMs. *Lance* is the *Cloudiator* life cycle agent, installed on each provisioned VM and handling the installation and configuration of the internal SaaS components on respective VMs. *Visor* is the monitoring agent for collecting raw system- and application-specific metrics.

(2) the publicly available *PaaS Unified Library (PUL)*[41] is integrated into *Cloudiator* to enable the unified access to PaaS resources [23]. PUL enables managing resources and applications across multiple PaaS providers via a RESTful interface or as a library. It supports a comprehensive set of operations, such as deploy, undeploy, start, stop, scale and bind.

Based on the IaaS unification of *Sword* and the PaaS unification of *PUL*, Cloudiator constitutes a unified cross-level orchestration tool for web-based BPs in a multi-cloud environment.

Further, Cloudiator enables collecting metrics across IaaS and PaaS via the monitoring agent *Visor*, which provides an extensible interface for custom sensors and the registration of new sensors at runtime. Visor stores the collected measurements in a TSDB where, currently, KairosDB and InfluxDB are supported. Cloudiator supports the cloud-based measurement aggregation and the automated TSDB distribution across multiple clouds to reduce network traffic and query latencies [20]. For advanced measurement processing, *Cloudiator* provides an interface to the *Monitoring Engine* (cf. "Monitoring engine" section).

To adapt specific BPaaS services at runtime, *Cloudiator* offers a comprehensive set of cross-level adaptation actions via its REST API to the *Adaptation Engine* (cf. "Adaptation engine" section), such as scaling a service or migrating it from IaaS to PaaS or vice versa.

The additional *Colosseum* components *Registries*, *Workers* and *Application Repository* are used for the internal storage and processing of the BPaaS deployment state as well as additional meta-data.

After the successful deployment of the internal SaaS services, the deployment details are returned from *Colosseum* to the *Entry point*. The latter extracts the required information, such as the actual endpoints of the deployed cloud services and registers them at the *Workflow Engine* to update the BP workflow's service endpoints. An SLA template is also registered at the *SLA Engine*, triggering the observation of the defined SLAs in the BPaaS bundle.

Hence, *Cloudiator* enables the holistic cross-level service orchestration for BPaaS by providing: (1) a unified abstraction over IaaS and PaaS services, (2) a scalable measurement infrastructure for cross-level/cloud metrics and (3) an extensive API for executing cross-level service adaptations.

Workflow engine

The *Workflow Engine* is a multi-tenant component enabling the management of the BPaaS workflows

[40] http://cloudiator.org/

[41] https://omi-gitlab.e-technik.uni-ulm.de/archive/cloudsocket/unified-paas

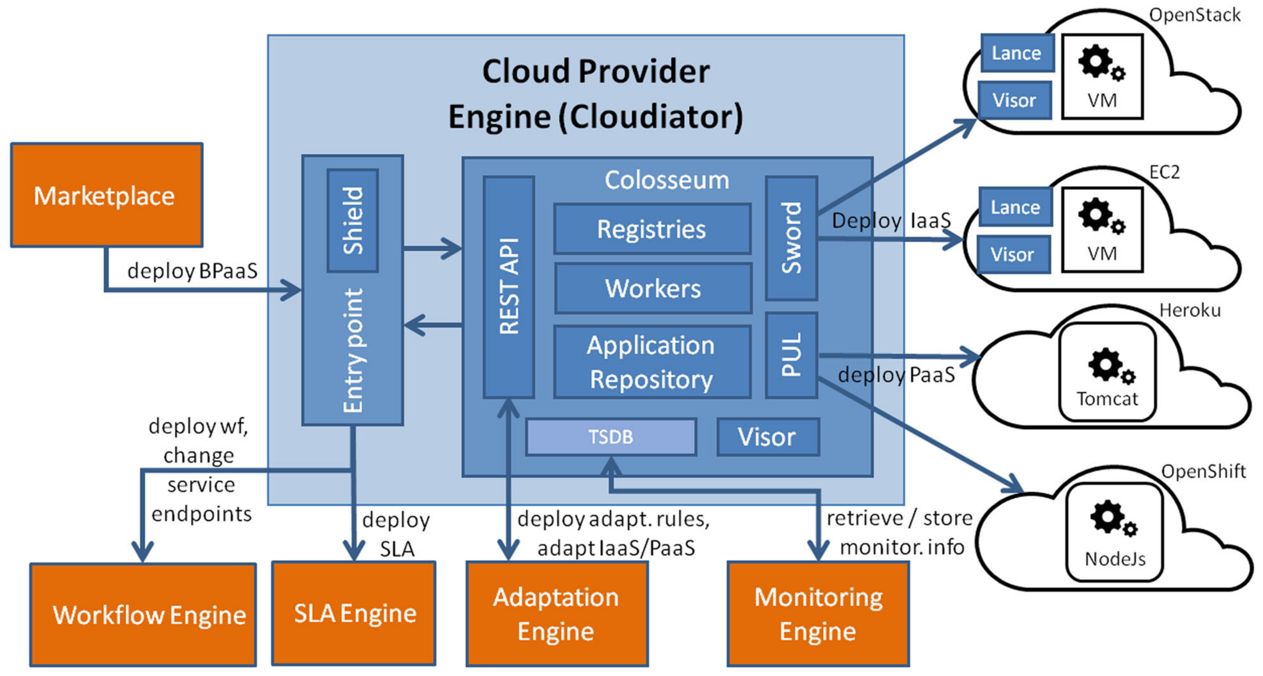


Fig. 6 The architecture of *Cloud Provider Engine*

deployed. Multi-tenancy is supported at the level of the customer organisation. In particular, the *Workflow Engine* enables different customers to only manipulate and exploit their own purchased BPaaS workflows. This open-source component is not only deployable in the Cloud but also able to handle cloud-based workflows and cater for their peculiarities, such as the frequent need for their adaptation which usually maps to modifying respective service endpoints at the workflow level.

Its input is a service-based workflow of a BPaaS to be deployed. There is no direct output produced. However, as this component produces monitoring information and stores workflow execution history information, its respective interface can be exploited to draw such information for, e.g., BPaaS analysis purposes in CloudSocket's BP evaluation environment.

This composite component follows a three-tier architecture, as depicted in Fig. 7, involving the UI, the backend and the database layer. Some of its subcomponents extend functionalities exhibited by the BPMN engine actually exploited, Activiti[42], such as the APIs offered, the multi-tenancy, the impersonalisation of integration services and the UI usability.

At the UI level, three main sub-components are provided. The *REST Workflow* is a RESTful service component exposing an API for the programmatic management of workflows and their instances as well as the retrieval of workflow execution history information. This API also extends the basic functionalities of Activiti with capabilities to deploy workflows with dynamic endpoints.

The *Workflow Explorer* sub-component exposes almost the same functionality with respect to *REST Workflow* but this functionality is supplied for human-oriented consumption only. As such, this sub-component offers an appropriate interface, a dashboard, via which the lifecycle of BPaaS workflows can be managed in a multi-tenant way by the roles involved in a BPaaS customer organisation. The *Editor Workflow* is a sub-component which does not belong to the BP execution but the design environment. Its main responsibility is to provide a suitable graphical UI via which workflows can be edited. Such an editing is restricted at the design phase with the rationale that the structure of BPaaS workflows cannot be altered once they are purchased[43]. So, only the BPaaS provider can edit them at the design phase before they reach the form of a BPaaS bundle published in CloudSocket's *Marketplace*.

The backend layer involves three main sub-components. The *Core Workflow Engine* manages the whole functionality exposed by the *Workflow Engine* which can be invoked by using a certain interface. This sub-component also handles the storage of workflows, their instances plus of other entities at the underlying database. The *Workflow Parser* sub-component can parse BPMN workflows and update their endpoints. It is can also generate service task tags for invoking both SOAP and RESTful SaaS services. The

[42]https://www.activiti.org/

[43]However, the structure of these workflow instances can be adapted at runtime

Bind Proxy, finally, enables interacting with the underlying database in the context of the storage and retrieval of service task to SaaS bindings/mappings.

At the database layer lies the *Workflow Database*, which is a relational database, is responsible for the storage and retrieval of any information related to the workflows, such as roles, tenants, and workflow (instances).

Due to the high-correlation of most sub-components of the *Workflow Engine* component, any kind of individual component distribution is not relevant. On the contrary, this component must be considered as a whole for distribution and scaling purposes. In this way, the actual distribution/load-balancing can be achieved either at the level of the BPaaS workflow (such that BPaaS workflows can be split among the different instances of this component) or the level of tenants (such that the split is performed based on the number of tenants to be handled by each instance of this component).

SLA engine

The *SLA Engine* is the component responsible for managing SLAs that conform to WS-Agreement (WSAG) [3]. In particular, it exhibits the following functionality: (a) generation of SLA templates to be used in BP offerings from CAMEL models. Such SLA templates represent the service level guarantees promised by the BP providers (i.e., the brokers) to their customers; (b) generation of an SLA out of the respective template that should be respected during the execution of the purchased BP; (c) follow-up of the actual WSAG SLAs for both BP brokers/providers and customers. Such a follow-up is possible via the subscription to monitoring events coming from the *Monitoring Engine*. It includes the evaluation of the SLOs within the WSAG SLA and the visualisation of their evaluation result. Via its capability to serve both BP providers and customers, it is obvious that this component is also multi-tenant.

The *SLA Engine*'s architecture can be seen in Fig. 8. This engine includes 6 main sub-components and follows a three-tier architecture. The UI component is the *SLA Dashboard*, responsible for visualising the status of the WSAG SLAs. This visualisation enables the customers to observe the status of all SLAs of their purchased BPs. Similarly, the BP providers/brokers can also observe the status of all SLAs signed with all their customers. The *REST Service* enables the SLA information retrieval while it encapsulates all *SLA Engine*'s core functionality offered and realised by the *WSAG Generator*.

The follow-up of SLAs becomes possible by involving the *Monitoring Engine Adapter*. This component, once the BP has been purchased and the deployment of both the BP and its monitoring infrastructure has taken place, it registers to the publish/subscribe mechanism of the *Monitoring Engine* to receive measurements for the metrics involved in the SLOs of the respective SLA. These measurements are then transformed into an appropriate format and passed to the *Assessment* component to perform the actual SLO assessment. The SLO assessment result is then imprinted back to the component of the last tier, the *Repository*, to enable the follow-up by means of

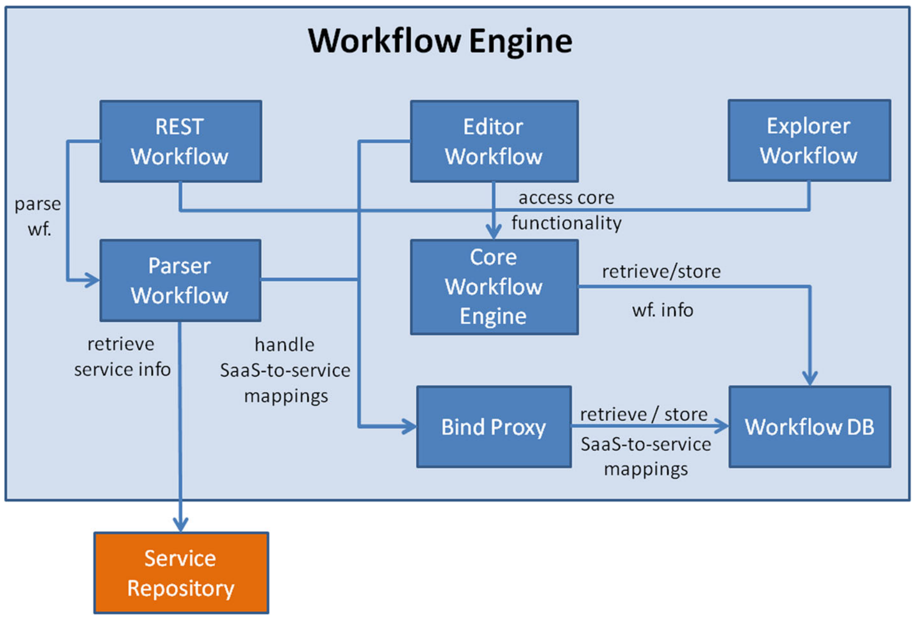


Fig. 7 The architecture of the *Workflow Engine*

the *SLA Dashboard*. The *Repository* is deposits all SLA-related information that is managed by the *SLA Engine*, thus providing appropriate support to its operation.

Please note that the registration to measurement information is initiated by the *Cloud Provider Engine*, which first checks that both the BP and its monitoring infrastructure has been deployed and then invokes the *REST Service* with the CAMEL application model as input. The *REST Service* in turn forwards the registration invocation to the *Monitoring Engine Adapter*.

The *SLA Engine* was developed with extensibility and reusability in mind. In this sense, the *Monitoring Adapter* and the *WSAG Generator* could be replaced to allow using other monitoring platforms or other SLA languages.

As indicated in "Overall architecture" section, this is a lightweight component that does not need to scale when the BP Execution Environment becomes overloaded. The only sub-component that could be scaled in very extreme cases would be the *Monitoring Engine Adapter*, which might have some additional processing load for the reception and transformation of the measurement information. However, as we foresee that only SLOs over aggregated metrics will be involved in the signed SLAs, we do not expect that the load of this component will become very high to justify its distribution.

Monitoring engine

The *Monitoring Engine* is a distributed component advancing the state-of-the-art via its capabilities: (a) to cover the measurement at any abstraction level and thus be able to fill-in any measurability gap; (b) to flexibly cover any kind of metric that might need to be monitored, thus not mapping to a fixed metric list, which would by far constraint the monitoring capabilities of the BP execution environment. In addition, by using a publish-subscribe mechanism, this component is not tightly coupled with any other component providing a complementary functionality to the environment. In fact, it enables the latter components to scale as needed such that each component instance can handle a different share from the information published by the *Monitoring Engine*. For instance, the *Adaptation Engine* could become distributed and based on the distribution logic employ instances which subscribe to certain partitions of all events that can be generated by the *Monitoring Engine*.

The flexible and dynamic metric manipulation comes with the adoption of CAMEL. CAMEL enables the complete specification of any metric which is then exploited by the *Monitoring Engine* to guarantee the deployment of suitable monitoring components that measure the right information utilised for the computation of that metric. Due to the generic way via which metrics are specified, CAMEL's adoption enables covering different metric types: (a) cross-cloud metrics, i.e., metrics computed from measurements (i.e., other metrics) derived from different clouds; (b) cross-level metrics, i.e., metrics computed from measurements derived in different abstraction levels.

The input to this engine is the actual CAMEL model and especially the metric, adaptation and requirement sub-models. The metric sub-model explicates how metrics can be computed, the adaptation what events need to be generated while the requirement what SLOs need to be evaluated. The output is the events generated which can be consumed by other components (i.e., the adaptation and SLA engines). Indirectly, the measurement database(s) enables any component to, e.g., perform additional analysis on its content. This indeed takes place in the CloudSocket's BP evaluation environment, as indicated in the beginning of this section, for supporting KPI evaluation.

The *Monitoring Engine*'s internal architecture is depicted in Fig. 9. It comprises level-specific agglomerations of monitoring components, called level-specific monitoring mechanisms, plus an evaluation component. At each level, three component kinds are employed: (a) the *Measurement Database* (MDB) for the storage and retrieval of monitoring information; (b) the sensors deployed for measuring and storing such information in the MDB; (c) the *Aggregator* for aggregating measurement information according to the CAMEL-based specification of respective high-level metrics. The latter component kind also offers a publish-subscribe mechanism via which measurements can be propagated to the evaluation component. The same mechanism is also exploited to propagate measurements to other levels to assist in computing respective metrics. For instance, the *Aggregator* at the workflow level can subscribe to the measurements provided by the *Aggregator* at the SaaS level to compute, e.g., execution time metrics for the workflow by relying on the response time of the respective SaaS services executed.

The level-specific mechanisms rely on exploiting different monitoring frameworks/components (e.g., SaaS – [65], IaaS/PaaS – [20]) developed by the three main partners involved in the BP execution environment. All these frameworks were structured to comply with the expected interfaces of the monitoring components in each level so as to be integrated into the *Monitoring Engine*.

The evaluation component (see more details in [39]), subscribes only to measurements/metrics mapping to SLO or single event conditions. As such, it just focuses on evaluating these conditions to generate the respective SLO assessments or events which can then be consumed by the SLA and adaptation engines via the use of the publish-subscribe mechanism offered. Internally, it employs a CEP engine (Esper) to handle the further processing of events so as to generate event patterns, which are mainly

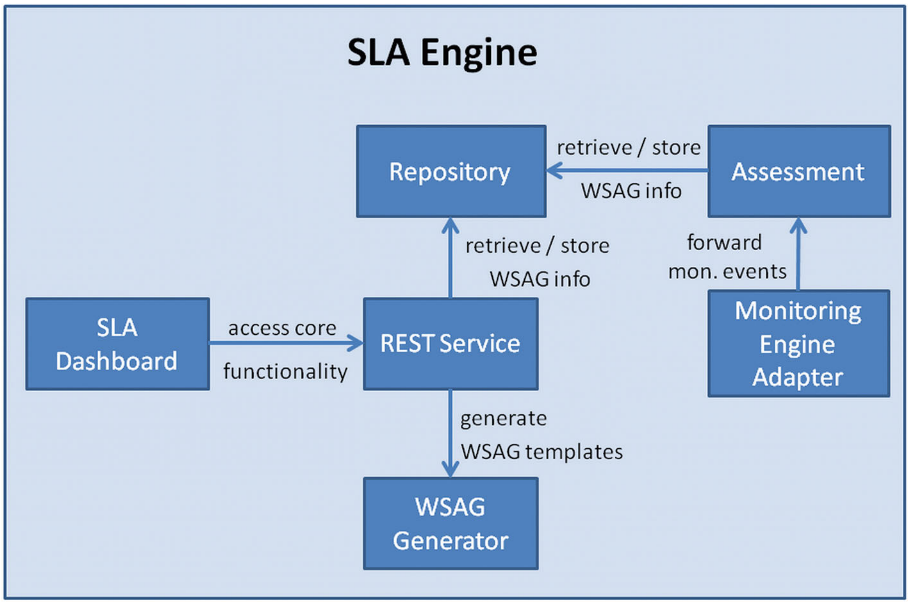


Fig. 8 The architecture of the *SLA Engine*

consumed by the *Adaptation Engine* in the context of adaptation rules triggering.

The evaluation component also offers a REST interface via which the event patterns to be detected are managed. This interface can be handy in case we must manipulate new adaptation rules, which require detecting new event patterns or modifying/deleting existing ones. The handling of metrics, mapping to these event patterns, is addressed by the *Cloud Provider Engine* which takes care of, e.g., in case that a new metric needs to be measured, deploying the needed sensor plus reconfiguring the corresponding *Aggregator* at the appropriate abstraction level (in case we need to compute a high-level metric).

Each level in the *Monitoring Engine*'s internal architecture maps to a different distribution logic. The workflow level is realised within the respective *Worflow Engine*. As such, it distributes equivalently with the latter component. The SaaS level is handled in two complementary ways: for internal SaaS components, the respective mechanisms are employed within the VMs hosting these components, while for external ones, the mechanisms are employed within the *Monitoring Engine*'s centralised part which also involves the CEP component. A similar situation occurs for the PaaS/IaaS level where the cloud-specific monitoring occurs in the internal SaaS component VMs, while the cross-cloud monitoring occurs at *Monitoring Engine*'s central part. This physical architecture is depicted in Fig. 10. For reasons of resource economy, the mechanisms deployed in the same VM share the same physical MDB.

This distribution logic is suitable to handle well the respective load by considering that most measurements generated at a high frequency concern mostly the infrastructure and service level. When moving to a higher abstraction level, the measurement frequency becomes more coarse-grained, thus reducing the load incurring to the respective monitoring components. Further, this distribution logic also conforms well to the one mapping to the whole *Monitoring Engine* as explicated in "Overall architecture" section. In particular, when scaling the *Monitoring Engine* to generate its new instances, we actually scale the components of the central part. The rest of the components are already distributed and quite fit to their purpose, so they do not need further distribution.

Adaptation engine

The *Adaptation Engine* is a novel component, supporting the cross-level and cross-cloud BP workflow adaptation, which well advances the state-of-the-art by exhibiting the following advantages: (a) covers all possible levels involved in the BP hierarchy; (b) it is cloud-based and handles cloud-based BPs instead of traditional ones; (c) dynamically concretises the adaptation workflow based on the current adaptation mechanisms available and by considering preferences and requirements over adaptation time and cost; (d) allows the dynamic adaptation rules modification during BP execution time; (e) stores the adaptation history to allow its further analysis. This can enable, for instance, to derive some knowledge about the successability of certain adaptation rules that can lead to their improvement via their manual or automatic modification.

This component requires as input a CAMEL model incorporating the adaptation rules to be triggered plus

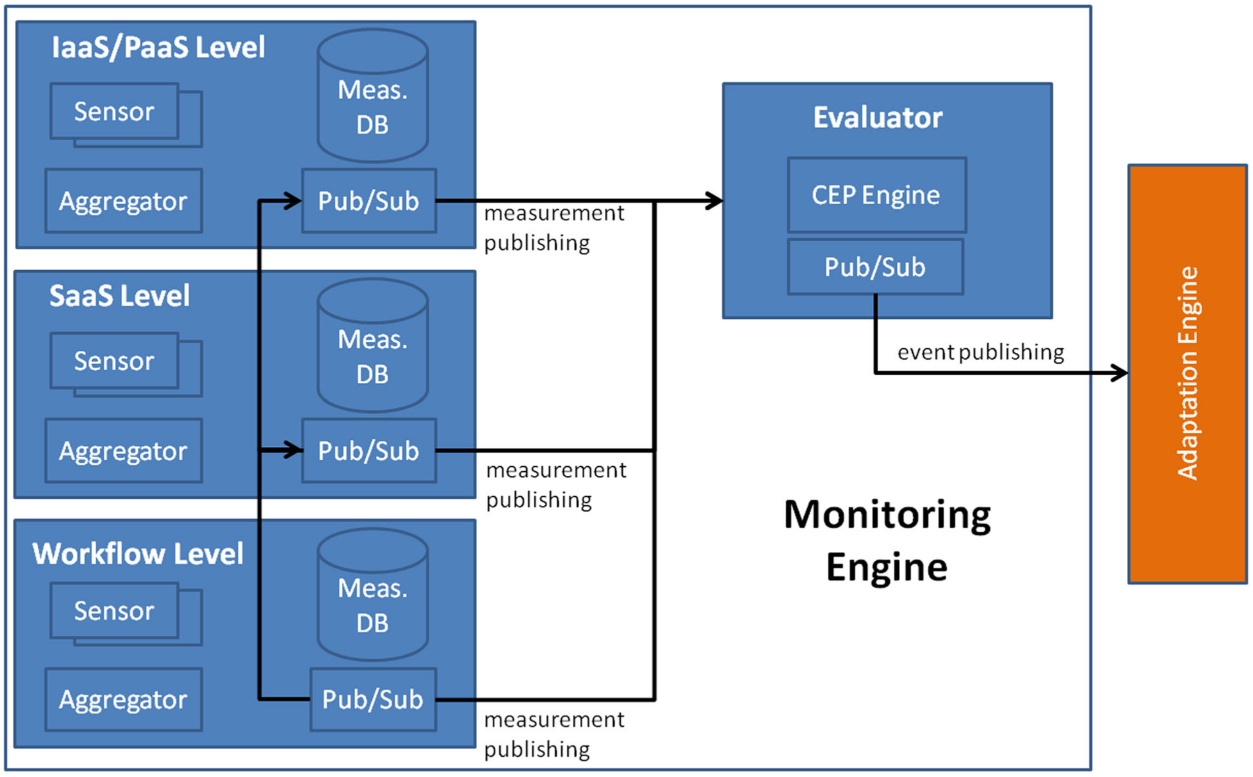


Fig. 9 The architecture of the Monitoring Engine

the event patterns detected by the *Monitoring Engine* which can enable this triggering. There is no direct output produced as the main engine's duty is BP adaptation. However, as indicated in above point (e), the adaptation history storage can support the analysis and improvement of the adaptation rules modelled for a certain cloud-based BP.

Figure 11 depicts the *Adaptation Engine*'s architecture. This architecture is quite complex involving multiple components. The *Knowledge Base* (KB) is responsible for storing the adaptation rules as well as their triggering. The adaptation workflow part of the triggered rules passes via the *Transformer* which translates it into a workflow language specification (BPMN[44]). This specification is the concretised by the *Concretiser*, which accesses the service registry to support the needed service discovery and selection. The completed service-based workflow specification is then deployed in the *Adaptation Workflow Engine*, a normal workflow engine responsible for executing (adaptation) workflows. While executing these workflows, this engine also handles the storage of their execution information in the underlying *Adaptation Database*.

An adaptation workflow involves the execution of service-based adaptation actions, which map to services offered via an adaptation library. These services are organised into different levels and rely on interfaces offered by other components in the execution environment. The workflow level involves services that modify the currently running BP workflow and thus cooperate with the *Workflow Engine*. The service level involves a service replacement service, which keeps a dynamic list of functionally-equivalent services, based on their availability, and finds the right service from this list for the replacement whose endpoint is then used to modify the respective BP workflow by cooperating with the *Workflow Engine*. The PaaS/SaaS level is covered by migration and scaling services exploiting the *Cloud Provider Engine*'s facilities.

The *Adaptation Engine* incluses a *UI* enabling experts to view the adaptation rules currently specified for a BPaaS and adjust them as needed. This UI also enables experts to define new adaptation rules to cover new problematic situations that might be observed. It also allows experts to define directly some adaptation workflows in CAMEL which can then be directly executed by passing them via the *Transformer* to the *Adaptation Workflow Engine*. This

[44]www.bpmn.org

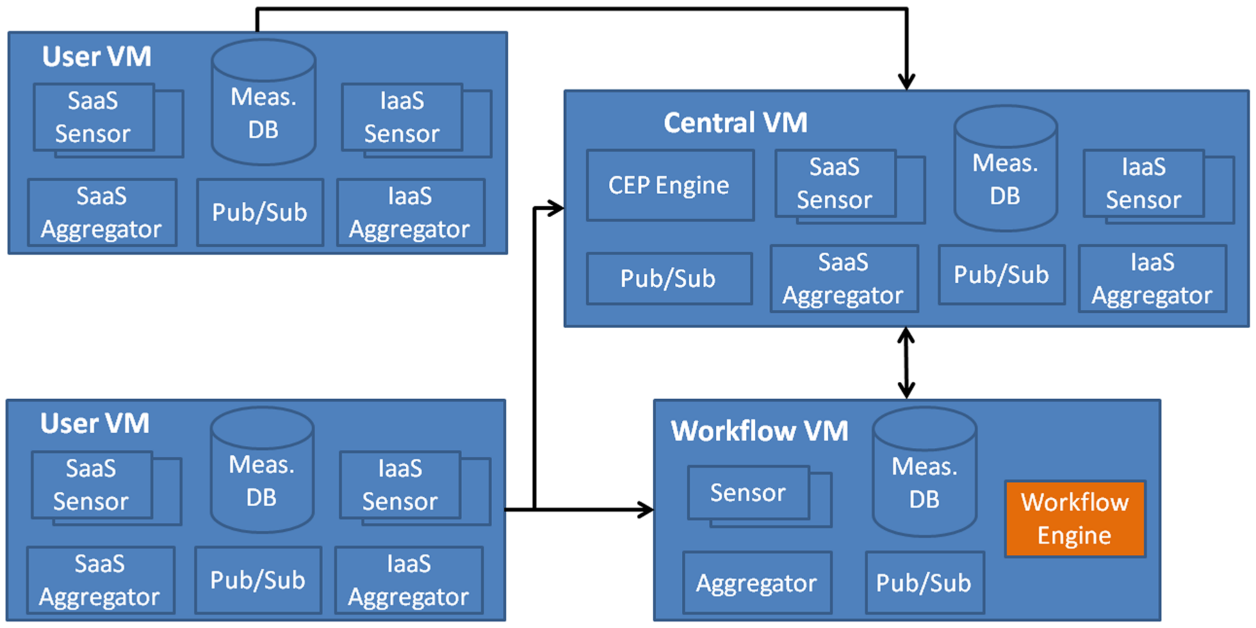


Fig. 10 The distributed architecture of the Monitoring Engine

can be useful for a rapid reaction to an unforeseen situation before this situation can be permanently handled by incorporating an adaptation rule.

The generation of new rules is facilitated by exploiting a service offered in the CloudSocket's BP evaluation environment. That service can detect event patterns resulting in SLO violations by following a logic-based mining approach [65] and (semi-)automatically generate respective adaptation rules [65]. These rules are then given as input to the *Adaptation Engine* which first passes them to the *Transformer* to translate them into the KB's internal format. These rules become active only when consulted and possibly adjusted by the expert via the *UI*. In fact, it is in the expert's discretion which rules from the current set mapping to a certain BP can become activated or deactivated.

Most core components of the *Adaptation Engine* can be distributed. The only components which can be mostly centralised are the *UI* plus the two types of databases, the KB and the *Adaptation Database*. This is due to the fact that we do not expect much load to be incurred to the latter components. In any case, we do not expect that the *Adaptation Engine* will be much overloaded thus requiring its appropriate distribution as the respective adaptation-triggering events will be not so frequent per each BP workflow.

Validation

The BP execution environment and the whole CloudSocket prototype platform were validated via a number of use cases which involved the management of respective cloud-based BPs [43]. In this section, we explicate the application of the proposed environment over one of these use cases concerning the management of a supporting BP for invoice management called "Invoice BPaaS".

This BP handles the production of invoices based on client information collected from an external Customer Relationship Management (CRM) SaaS called "YMENS CRM" provided by CloudSocket's YMENS partner, plus their sending via email to clients. The invoice manipulation is supported by the use of a well-known invoice management SaaS called "Invoice Ninja". This SaaS is internal to the BP which means that it has been purchased and needs to be deployed in the Cloud. Both SaaS services are operated in Europe due to data privacy conformance reasons mapping to respective EU legislation by considering the fact that European customers are mainly targeted.

During the first two BP lifecycle phases, the initially designed BP will be mapped to a service-based workflow which will then be concretised and transformed into a deployable workflow. This will end up in producing a BPaaS bundle which will then be published in the *Marketplace*. The BPaaS bundle will comprise a CAMEL model involving the topology depicted in Fig. 12.

In this topology, it is well clarified that the "Invoice Ninja" component will be deployed on the Amazon AWS' "m1.medium" VM within the European region `eu-central-1`. As shown in the CAMEL excerpt in

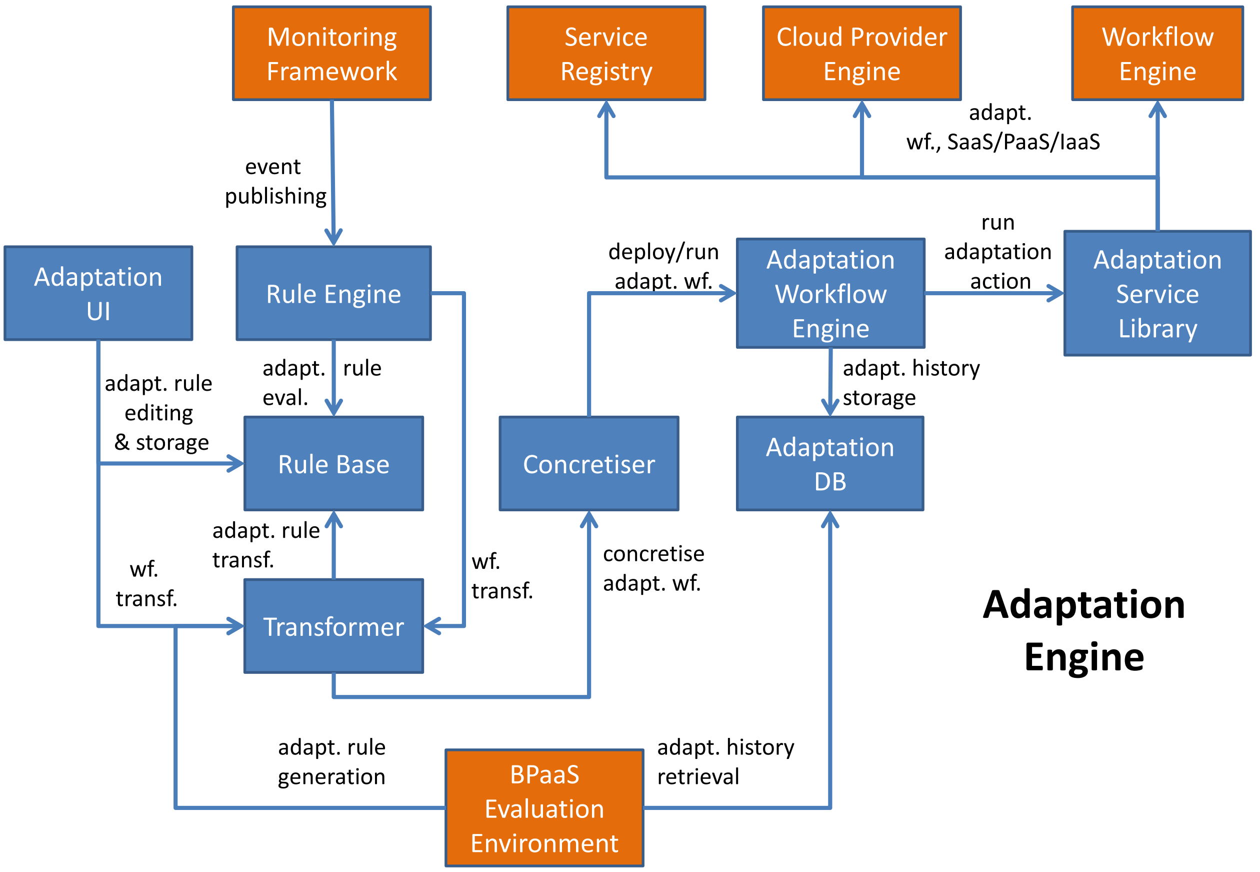


Fig. 11 The architecture of the Adaptation Engine

Fig. 12, both SaaS will be allocated to certain BP workflow tasks. We have omitted from the analysis configuration details for "Invoice Ninja" as they are quite technical. These can be inspected in the respective CAMEL model available online[45].

In this CAMEL model, there exist two main adaptation rules which will drive the BPaaS adaptation behaviour. These rules are depicted as follows.

$$R_1 : e_1 \Rightarrow restart\left(i\_ninja\right) \tag{1}$$

$$R_2 : e_3 \Rightarrow sequence(paasDeploy\left(i\_ninja\right), servReplace\left(i\_ninja\right), destroyIaaSDep\left(i\_ninja\right)) \tag{2}$$

$$e_3 : e_1 \wedge e_2 \tag{3}$$

$$e_1 : meanAvail\left(i_{ninja}\right) < 20\% \tag{4}$$

$$e_2 : fail\left(R_1\right) == 1 \tag{5}$$

The first rule attempts to re-start the "Invoice Ninja" component if it is unavailable. Its main rationale is that many software bugs are transient and can be easily fixed by restarting the software. However, in case that the bug insists, the second rule will compensate for this by migrating the component in another VM. This migration has the rationale that possibly either the component has become corrupted and/or reached an inconsistent state such that it needs to be redeployed from scratch. To support this migration, a PaaS service is exploited for the redeployment to enable the component's rapid recovery and thus the respective minimisation of its downtime. However, as the component re-configuration leads to its endpoint's modification, a restricted service replacement form can take place to evolve the BPaaS workflow in a new version which includes just endpoint replacement and not the replacement of all service information in the workflow. In this sense, we actually deal with an adaptation workflow for the 2nd rule involving three actions: (a) PaaS deployment; (b) service replacement; (c) component (instance) undeployment. The latter action is executed lastly to first guarantee that the PaaS redeployment comes into effect before we can undeploy the initial instance of the "Invoice

[45] https://drive.google.com/file/d/0B1oLQgQCVlqramYwa1hDZmtnSGc/view?usp=sharing

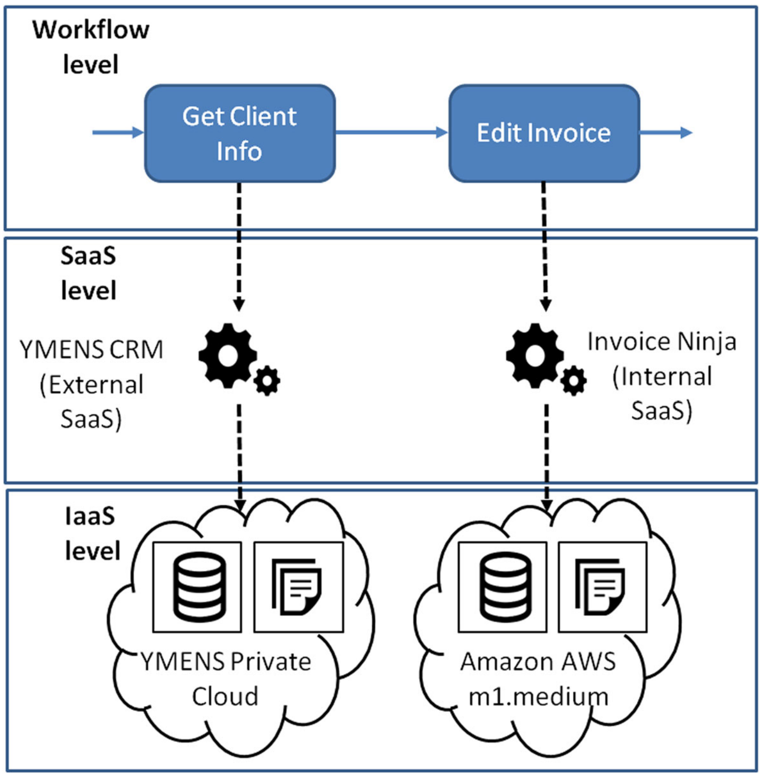


Fig. 12 Excerpt of CAMEL topology model

Ninja" component. In case that this redeployment fails, we can still somehow use the initial deployment. For instance, a manual adaptation action can take place by the administrator who must detect and correct the respective component fault by connecting via appropriate means (e.g., SSH) to the VM hosting this component. Please note here that as indicated in "Adaptation" section none of the existing, proprietary platforms support the latter form of adaptation workflow while the respective research prototypes do not handle the infrastructure level or were not extended to become cloud-enabled.

The detection of the component fault is accomplished by introducing a single event e_1 that evaluates whether the component mean availability has overpassed a certain

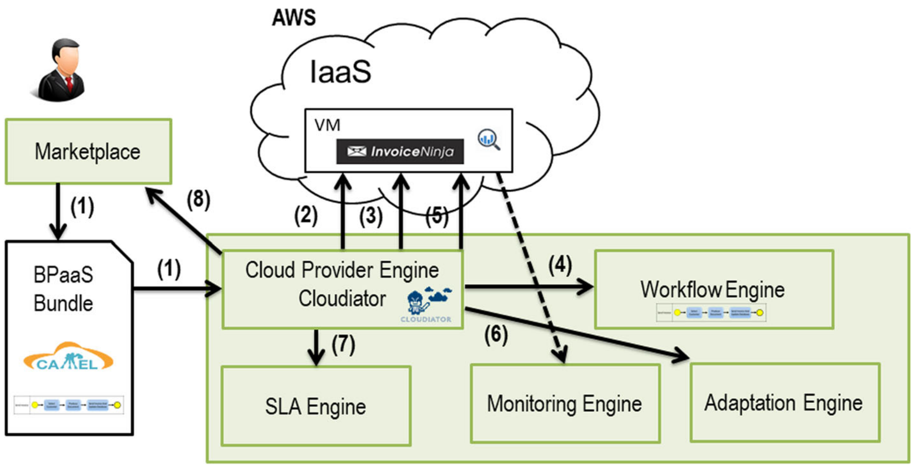
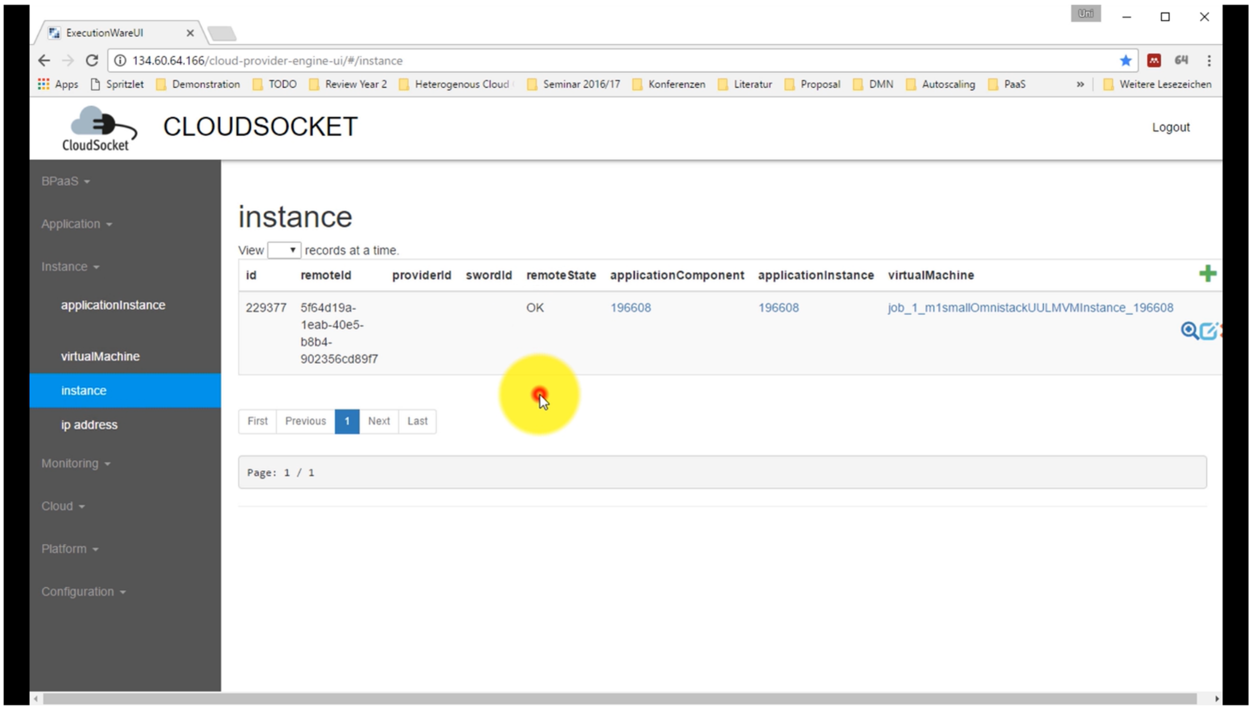


Fig. 13 Workflow of the initial deployment



instance

View records at a time.

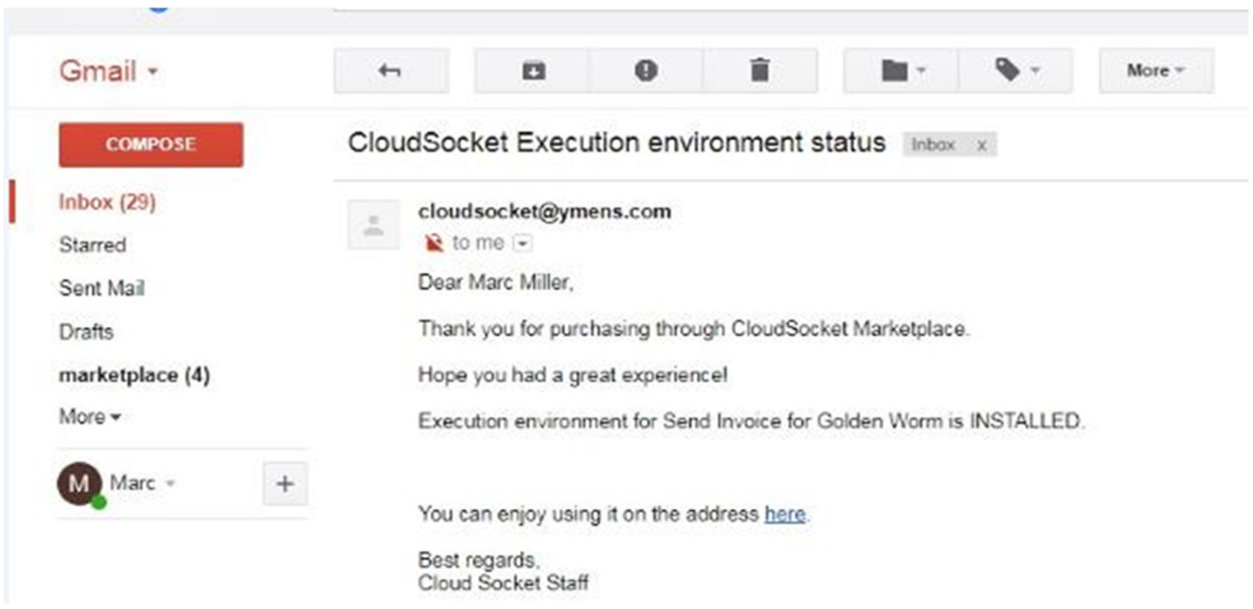
id	remoteId	providerId	swordId	remoteState	applicationComponent	applicationInstance	virtualMachine
229377	5f64d19a-1eab-40e5-b8b4-902356cd89f7			OK	196608	196608	job_1_m1smallOmnistackUULMVMInstance_196608

First Previous 1 Next Last

Page: 1 / 1

Fig. 14 *Cloud Provider Engine*'s UI showing the deployment of the "Invoice Ninja" BPaaS component

threshold. The component's *mean availability* is a composite metric computed every minute from the component's *raw availability*, another composite metric computed every 10 seconds. The latter metric is then computed in turn from *raw uptime* metric information within the 10 second period. That event is the only one for Rule R_1. However, for the second rule, R_2, we need to consider the fact that the first rule failed. This is a special event kind mapping to a system metric that indicates whether the adaptation rule has failed. In this sense, it is enough



CloudSocket Execution environment status

cloudsocket@ymens.com
to me

Dear Marc Miller,

Thank you for purchasing through CloudSocket Marketplace.

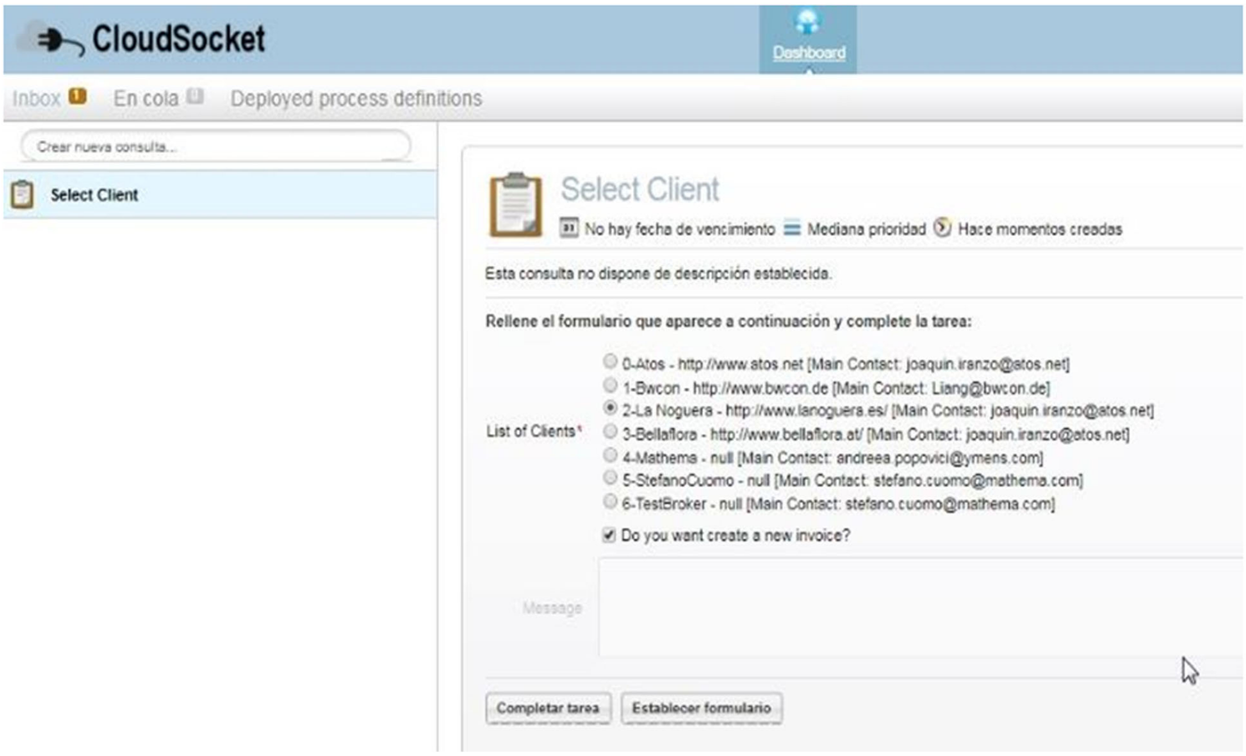
Hope you had a great experience!

Execution environment for Send Invoice for Golden Worm is INSTALLED.

You can enjoy using it on the address here.

Best regards,
Cloud Socket Staff

Fig. 15 Sent email content upon successful BPaaS deployment


CloudSocket

Dashboard

Inbox 1 En cola 0 Deployed process definitions

Crear nueva consulta...

Select Client

Select Client

No hay fecha de vencimiento Mediana prioridad Hace momentos creadas

Esta consulta no dispone de descripción establecida.

Rellene el formulario que aparece a continuación y complete la tarea:

List of Clients*	0-Atos - http://www.atos.net [Main Contact: joaquin.iranzo@atos.net]
	1-Bwcon - http://www.bwcon.de [Main Contact: Liang@bwcon.de]
	[✓ 2-La Noguera - http://www.lanoguera.es/ [Main Contact: joaquin.iranzo@atos.net]]
	3-Bellaflora - http://www.bellaflora.at/ [Main Contact: joaquin.iranzo@atos.net]
	4-Mathema - null [Main Contact: andreea.popovici@ymens.com]
	5-StefanoCuomo - null [Main Contact: stefano.cuomo@mathema.com]
	6-TestBroker - null [Main Contact: stefano.cuomo@mathema.com]
	[✓] Do you want create a new invoice?
Message	

Completar tarea Establecer formulario

Fig. 16 Execution of one BPaaS workflow instance

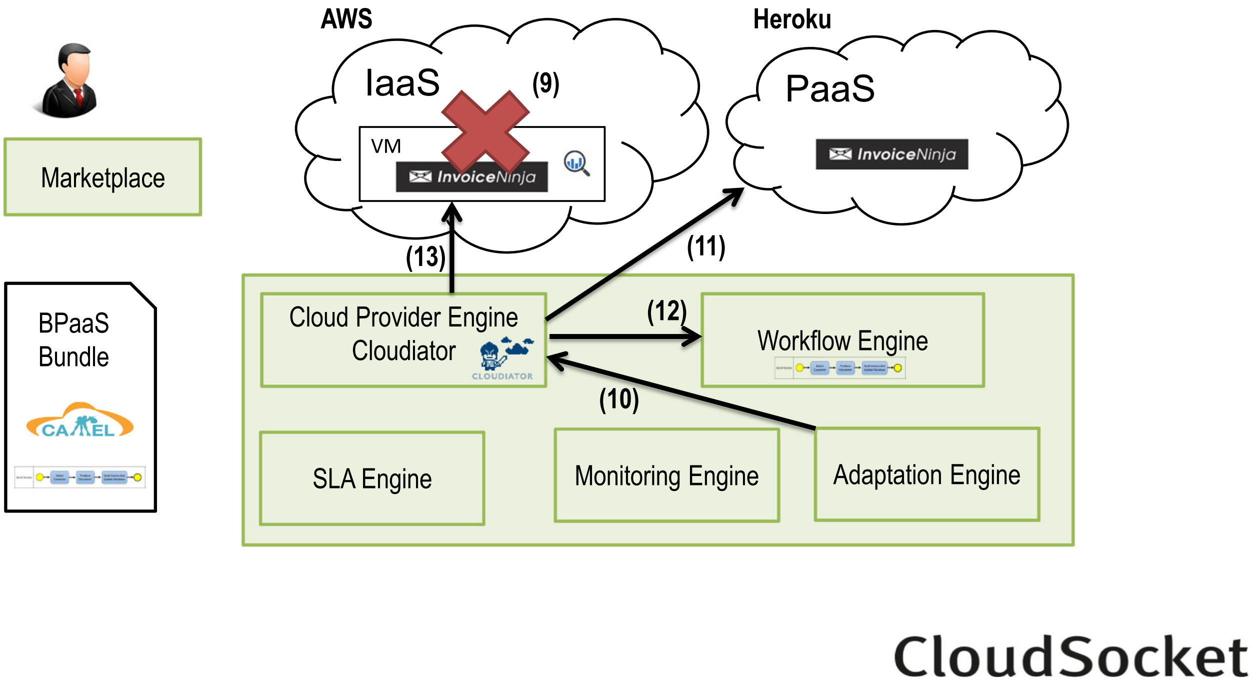
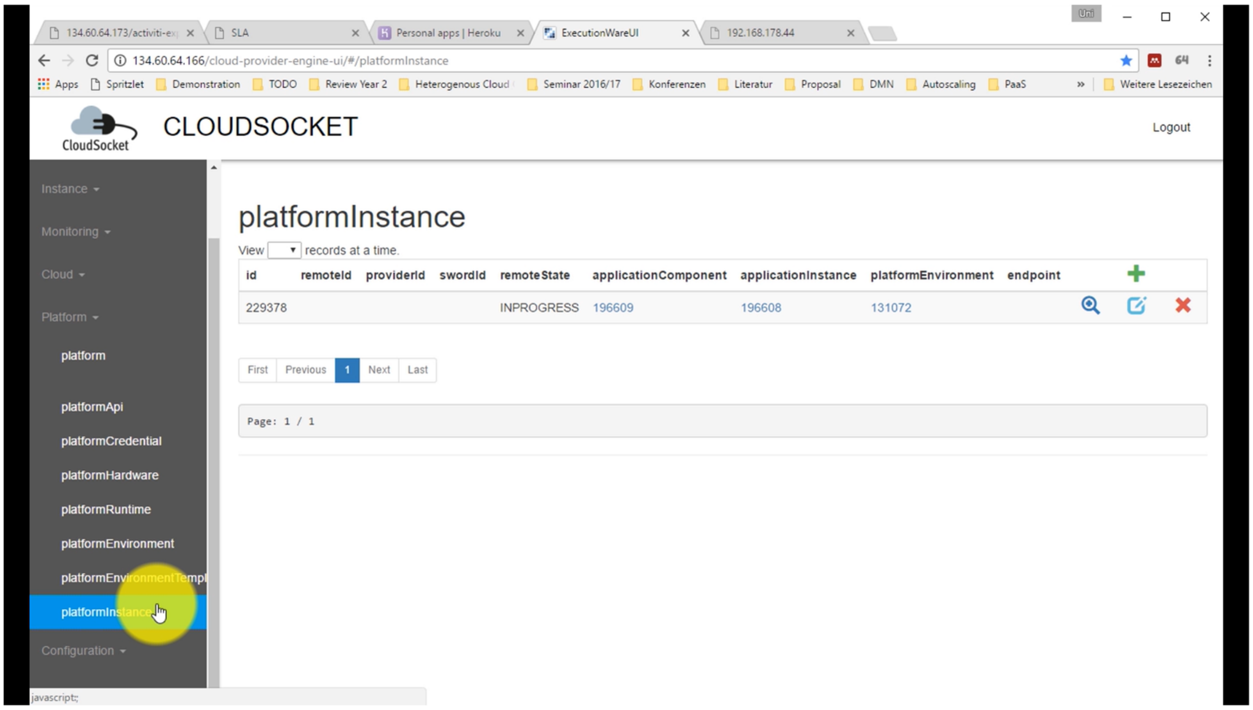


Fig. 17 Workflow of the environment components execution upon the second adaptation rule's triggering



id	remoteId	providerId	swordId	remoteState	applicationComponent	applicationInstance	platformEnvironment	endpoint
229378				INPROGRESS	196609	196608	131072	

Fig. 18 *Cloud Provider Engine*'s UI showing the PaaS-based deployment of the "Invoice Ninja" BPaaS component

to insert a condition here that checks whether that metric is equal to 1 (signifying the rule failure). This condition is mapped to event e_2. Both events e_1 and e_2 are then associated with another event e_3 which is composite and maps to their conjunction.

Figure 13 shows the workflow of an initial deployment. Once the BPaaS is purchased by a customer (1), its bundle will be given as input to *Cloud Provider Engine*. This component will then handle the bundle deployment. It will first deploy the "Invoice Ninja" component in Amazon AWS by creating a suitable VM (2) and installing this component (3). Then, it will send the BPaaS BPMN workflow for deployment to the *Workflow Engine* along with the "Invoice Ninja" component's endpoint. (4) Once this deployment is successful, it will attempt to set up the monitoring infrastructure at the cloud level by deploying the respective monitoring mechanisms at the VM of "Invoice Ninja". (5) Finally, it will deploy the two adaptation rules to the *Adaptation Engine*(6) and also deploy the BPaaS bundle's SLA in the *SLA Engine*(7).

Upon successful deployment (see Fig. 14), an email is sent to the BPaaS customer identifying the *Workflow Engine*'s endpoint (8) (see Fig. 15). Via this endpoint, the customer connects to the latter engine's UI via which he/she can create instances of the deployed workflow. Upon execution of such instances (see Fig. 16), the orchestration of the SaaS services is achieved. In parallel, monitoring information is collected from the different levels. In fact, IaaS/PaaS/SaaS measurements are computed once the bundle deployment is successful. On the other hand, measurements at the workflow level are computed only when the respective BPaaS workflow instances are executed. All these measurements generate respective events which can trigger adaptation rules in the *Adaptation Engine* like those aforementioned in this section.

Figure 17 shows the adaptation workflow in our environment for the BPaaS at hand. For instance, once the "Invoice Ninja" component is down (9), it will be attempted to be restarted by using the respective script-based command in its CAMEL-based configuration specification (see rule R_1). If this fails, the *Adaptation Engine* will attempt to run "Invoice Ninja" on a PaaS provider (see rule R_2). In this case, the *Adaptation Engine* will first invoke a respective method of the *Cloud Provider Engine*'s REST interface to trigger the deployment on PaaS (10). The *Cloud Provider Engine* will then instantiate "Invoice Ninja" (11) (see respective result in Fig. 18), and update the respective endpoint at the *Workflow Engine)* (12). Finally, the *Cloud Provider Engine* will trigger the undeployment of "Invoice Ninja" from its original hosting place by invoking a suitable method of the *Cloud Provider Engine* API (13). This adaptation workflow is covered by a certain demonstration video of CloudSocket at[46].

[46] https://www.youtube.com/watch?v=aGtQ210wih8&authuser=0

The use case at hand has been executed and checked multiple times during CloudSocket lifetime. Upon the final CloudSocket platform version, all these times resulted in the successful handling of the use case which witnesses the suitability and reliability of that platform and its encompassed environments.

Conclusions

This article has presented a BP execution framework in the cloud. This framework can support the cross-cloud and cross-level adaptive provisioning of cloud-based BPs called BPaaSes. It involves components enabling the cross-cloud and cross-level orchestration of cloud services that provide functional and infrastructural support to the BP. It also involves components supporting the cross-cloud and cross-level monitoring and adaptation of the BPaaSes.

Apart from presenting the overall architecture of the environment as well as the internal architecture of its main components, this article has also explicated how this environment can be distributed and scaled in order to handle the additional workload as well as to become more robust.

The environment has been applied over various BPaaSes which have been derived from corresponding use-cases of the CloudSocket project. In this article, we showcased the application over one use-case as a validation means which also explicates some of the main benefits of the proposed environment.

The following future work directions are envisioned. First, the implementation and assessment of the proposed distributed physical environment architecture. Second, the better coverage of the monitoring at the PaaS level. Third, the dynamic incorporation of extra adaptation mechanisms in the environment's *Adaptation Engine* to enhance its adaptation capabilities. Fourth, the capability to dynamically modify the adaptation rules modelled or derived for BPs based on their execution history to support the optimisation of the adaptation behaviour of these BPs at runtime. Finally, we will pursue the possibility to automate the transition from old to new provider API versions to keep pace with the frequency of change of these APIs. Our intended contribution could be quite beneficial to cloud abstraction APIs which struggle to be up-to-date with the provider API changes.

Abbreviations

AWS: Amazon web services; BP: Business process; BPaaS: Business-process-as-a-service; CEP: Complex event processing; CIMI: Cloud infrastructure management interface; COT: Cloud orchestration tool; CRM: Customer relationship management; EC: Event calculus; ECA: Event-condition-action; IaaS: Infrastructure-as-a-service; KB: Knowledge base; KPI: Key performance indicator; MDB: Measurement database; MONINA: (MONitoring, INtegration, Adaptation); multi-DSL: Multi-domain-specific language; OCCI: Open cloud computing interface; PaaS: Platform-as-a-service; PUL: PaaS unified library; SaaS: Software-as-a-service; SBA: Service-based application; SLA: Service level agreement; SLO: Service level objective; TSDB: Time-series database; UULM: University of ULM; VM: Virtual machine; WSAG: WS-agreement

Acknowledgements

The authors would like to thank the partners of the CloudSocket project (http://www.cloudsocket.eu) for their valuable advice and comments.

Authors' Contributions

KK is the corresponding author and contributed in all of the manuscript sections. Dr. CZ contributed to "Monitoring" and "Adaptation" sections as well as "Monitoring engine" and "Adaptation engine" sections. JI and RSG contributed to "Workflow engine" and "SLA engine" sections as well as the overall architecture of the proposed BP execution environment. DS, FG and JD contributed to "Cloud service orchestration", "Monitoring", "Cloud provider engine", "Monitoring engine", and "Adaptation engine" sections as well as well as the overall architecture of the proposed BP execution environment. All authors have read and approved the manuscript.

Funding

The research leading to these results has received funding from the EC's Framework Programme HORIZON 2020 (ICT-07-2014) under grant agreement number 644690 (CloudSocket).

Availability of data and materials

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]ICS-FORTH, Heraklion, Crete, Greece. [2]ATOS, Barcelona, Spain. [3]University of Ulm, Ulm, Germany.

Received: 28 January 2019 Accepted: 29 October 2019
Published online: 27 November 2019

References

1. Achilleos AP, Kritikos K, Rossini A, Kapitsaki GM, Domaschka J, Orzechowski M, Seybold D, Griesinger F, Nikolov N, Romero D, Papadopoulos GA (2019) The Cloud Application Modelling and Execution Language. J Cloud Comput. Accepted
2. Alexander K, Lee C, Kim E, Helal S (2017) Enabling end-to-end orchestration of multi-cloud applications. IEEE Access 5:8,862?18,875
3. Andrieux A, Czajkowski K, Dan A, Keahey K, Ludwig H, Nakata T, Pruyne J, Rofrano J, Tuecke S, Xu M (2007) Web Services Agreement Specification (WS-Agreement). Tech Rep. Open Grid Forum
4. Barbon F, Traverso P, Pistore M, Trainotti M (2006) Run-time monitoring of instances and classes of web service compositions. In: Proceedings of the IEEE International Conference on Web Services, ICWS '06. IEEE Computer Society, Washington. pp 63–71. https://doi.org/10.1109/ICWS.2006.113. http://dx.doi.org/10.1109/ICWS.2006.113
5. Baresi L, Guinea S (2005) Dynamo: Dynamic monitoring of ws-bpel processes. In: Proceedings of the Third International Conference on Service-Oriented Computing, ICSOC'05. Springer-Verlag, Amsterdam. pp 478–483. https://doi.org/10.1007/11596141_36. http://dx.doi.org/10.1007/11596141_36
6. Baresi L, Guinea S, Pasquale L (2007) Self-healing bpel processes with dynamo and the jboss rule engine. In: International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting, ESSPE '07. ACM, Dubrovnik. pp 11–20. https://doi.org/10.1145/1294904.1294906
7. Baur D, Domaschka J (2016) Experiences from building a cross-cloud orchestration tool. In: Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud '16. ACM, New York. pp 4:1–4:6. https://doi.org/10.1145/2904111.2904116
8. Baur D, Seybold D, Griesinger F, Masata H, Domaschka J (2018) A provider-agnostic approach to multi-cloud orchestration using a constraint language. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Press. pp 173–182. https://doi.org/10.1109/ccgrid.2018.00032
9. Baur D, Seybold D, Griesinger F, Tsitsipas A, Hauser CB, Domaschka J (2015) Cloud orchestration features: Are tools fit for purpose? In: Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on. IEEE. pp 95–101

10. Blair G, Bencomo N, France RB (2009) Models@run.time. Computer 42(10):22–27. https://doi.org/10.1109/MC.2009.326
11. Brinkmann A, Fiehe C, Litvina A, Lück I, Nagel L, Narayanan K, Ostermair F, Thronicke W (2013) Scalable monitoring system for clouds. In: Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC '13. IEEE Computer Society, Washington, DC. pp 351–356. https://doi.org/10.1109/UCC.2013.103
12. Calcaterra D, Cartelli V, Di Modica G, Tomarchio O (2018) Exploiting bpmn features to design a fault-aware tosca orchestrator. In: CLOSER. pp 533–540. https://doi.org/10.5220/0006775605330540
13. Calcaterra D, Cartelli V, Di Modica G, Tomarchio O (2019) A comparison of multi-cloud provisioning platforms. In: CLOSER. pp 507–514. https://doi.org/10.5220/0007765005070514
14. Carrasco J, Cubo J, Durán F, Pimentel E (2016) Bidimensional cross-cloud management with TOSCA and brooklyn. In: 9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco. pp 951–955. June 27 - July 2. https://doi.org/10.1109/CLOUD.2016.0143
15. Carrasco J, Durán F, Pimentel E (2017) Component-wise application migration in bidimensional cross-cloud environments. In: CLOSER. https://doi.org/10.5220/0006372302870297
16. de Chaves SA, Uriarte RB, Westphall CB (2011) Toward an architecture for monitoring private clouds. IEEE Commun Mag 49(12):130–137
17. Curbera F, Duftler MJ, Khalaf R, Nagy WA, Mukhi N, Weerawarana S (2005) Colombo: Lightweight middleware for service-oriented computing. IBM Syst J 44(4):799–820. https://doi.org/10.1147/sj.444.0799
18. Domaschka J, Baur D, Seybold D, Griesinger F (2015) Cloudiator: a cross-cloud, multi-tenant deployment and runtime engine. In: 9th Symposium and Summer School on Service-Oriented Computing. IBM, Armonk
19. Domaschka J, Griesinger F, Seybold D, Wesner S (2017) A cloud-driven view on business process as a service. In: CLOSER. pp 739–746. https://doi.org/10.5220/0006393107670774
20. Domaschka J, Seybold D, Griesinger F, Baur D (2015) Axe: A novel approach for generic, flexible, and comprehensive monitoring and adaptation of cross-cloud applications. In: European Conference on Service-Oriented and Cloud Computing. Springer. pp 184–196. https://doi.org/10.1007/978-3-319-33313-7_14
21. Emeakaroha VC, Ferreto TC, Netto MAS, Brandic I, De Rose CAF (2012) Casvid: Application level monitoring for sla violation detection in clouds. In: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, COMPSAC '12. IEEE Computer Society, Washington, DC. pp 499–508. https://doi.org/10.1109/COMPSAC.2012.68
22. Falcioni D, Utz W, Woitsch R, Iranzo J, Sosa R, Gallo A, Cacciatore S, Davidescu R, Ganga A, Tuguran CV, Popovici A, Seybold D, Griesinger F, Kritikos K, Hinkelmann K, Laurenzi E D4.6/D4.7/D4.8 – Final BPaaS Prototype. https://site.cloudsocket.eu/documents/251273/350509/CloudSocket_D4.6_D4.7_D4_8-v1.0.pdf/16944349-1a92-4ae6-a6d3-2601a8faacaa?download=true
23. Ferrer AJ, Pérez DG, González RS (2016) Multi-cloud platform-as-a-service model, functionalities and approaches. Procedia Comput Sci 97:63–72
24. Ferry N, Chauvel F, Rossini A, Morin B, Solberg A (2013) Managing multi-cloud systems with cloudmf. In: Proceedings of the Second Nordic Symposium on Cloud Computing &#38; Internet Technologies, NordiCloud '13. ACM, New York. pp 38–45. https://doi.org/10.1145/2513534.2513542
25. Ferry N, Rossini A, Chauvel F, Morin B, Solberg A (2013) Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: Cloud Computing (CLOUD) 2013 IEEE Sixth International Conference on. IEEE. pp 887–894. https://doi.org/10.1109/cloud.2013.133
26. Goldschmidt T, Jansen A, Koziolek H, Doppelhamer J, Breivold HP (2014) Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In: CLOUD. IEEE Computer Society. pp 602–609. https://doi.org/10.1109/cloud.2014.86
27. Guinea S, Kecskemeti G, Marconi A, Wetzstein B (2011) Multi-layered monitoring and adaptation. In: Proceedings of the 9th International Conference on Service-Oriented Computing, ICSOC'11. Springer-Verlag, Paphos. pp 359–373. https://doi.org/10.1007/978-3-642-25535-9_24
28. Hinkelmann K, Kurjakovic S, Lammel B, Laurenzi E, Woitsch R D3.2 – Modelling Prototypes for BPaaS. https://site.cloudsocket.eu/documents/251273/350509/CloudSocket_D3.2_Modelling_Prototypes_for_BPaaS_Final.pdf/747dc1d5-1e12-4b06-b1c5-2dc6fefae9aa?download=true
29. Hossain M, Khan R, Al Noor S, Hasan R (2016) Jugo: A generic architecture for composite cloud as a service. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). IEEE. pp 806–809. https://doi.org/10.1109/cloud.2016.0112
30. Huang H, Wang L (2010) P&p: A combined push-pull model for resource monitoring in cloud computing environment. In: IEEE CLOUD. IEEE Computer Society. pp 260–267. https://doi.org/10.1109/cloud.2010.85
31. Inzinger C, Hummer W, Satzger B, Leitner P, Dustdar S (2014) Generic Event-Based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems. Softw Pract Experience. http://dsg.tuwien.ac.at/staff/inzinger/dl/SPE_2014_monina.pdf
32. Jonas E, Schleier-Smith J, Sreekanti V, Tsai CC, Khandelwal A, Pu Q, Shankar V, Menezes Carreira J, Krauth K, Yadwadkar N, Gonzalez J, Popa RA, Stoica I, Patterson DA (2019) Cloud programming simplified: A berkeley view on serverless computing. In: Tech. Rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html
33. Karagiannis D (1995) Bpms: Business process management systems. SIGOIS Bull 16(1):10–13. https://doi.org/10.1145/209891.209894
34. Kolb S, Röck C (2016) Unified cloud application management. In: Services (SERVICES) 2016 IEEE World Congress on. IEEE. pp 1–8. https://doi.org/10.1109/services.2016.7
35. Kolb S, Wirtz G (2014) Towards application portability in platform as a service. In: Service Oriented System Engineering (SOSE) 2014 IEEE 8th International Symposium on. IEEE. pp 218–229. https://doi.org/10.1109/sose.2014.26
36. König B, Calero JA, Kirschnick J (2012) Elastic monitoring framework for cloud infrastructures. IET Commun 6(10):1306–1315
37. Konstantinou I, Angelou E, Boumpouka C, Tsoumakos D, Koziris N (2011) On the elasticity of nosql databases over cloud management platforms. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11. ACM, Glasgow. pp 2385–2388. https://doi.org/10.1145/2063576.2063973
38. Kritikos K, Domaschka J, Rossini A (2014) SRL: A scalability rule language for multi-cloud environments. In: CloudCom. IEEE Computer Society. pp 1–9. https://doi.org/10.1109/cloudcom.2014.170
39. Kritikos K, Zeginis C, Paravoliasis A, Plexousakis D (2017) CEP-Based SLO Evaluation. In: BPM@Cloud Workshop in ESOCC. Springer. https://doi.org/10.1007/978-3-319-79090-9_2
40. Lammel B, Kurjakovic S, Grivas SG, Hinkelmann K, Giovanoli C, Utz W D2.3 – Cloud Transformation Framework. https://site.cloudsocket.eu/documents/251273/350509/CloudSocket_D2.3_Transformation_Framework_v1.0.pdf/b1f0b3ea-2830-47e3-ade0-40f386a41958?download=true
41. Larsson L, Henriksson D, Elmroth E (2011) Scheduling and monitoring of internally structured services in cloud federations. In: ISCC. IEEE Computer Society. pp 173–178. https://doi.org/10.1109/iscc.2011.5984012
42. Leitner P, Inzinger C, Hummer W, Satzger B, Dustdar S (2012) Application-level performance monitoring of cloud services based on the complex event processing paradigm. In: SOCA. IEEE Computer Society. pp 1–8. https://doi.org/10.1109/soca.2012.6449437
43. Liang Y, Jähnert J, Woitsch R, Falcioni D, Yuste JI, Cuomo S, Naldini S, Kritikos K D5.6 – Demonstration Run Report. https://site.cloudsocket.eu/documents/251273/350509/CloudSocket-D5.6_Demonstration_Run_Report_v1.0_FINAL.pdf/1042c849-af11-49c3-9f19-a4cdfb461974
44. Mahbub K, Spanoudakis G (2007) Monitoring WS-Agreements: An Event Calculus–Based Approach. Springer, Berlin. pp 265–306. https://doi.org/10.1007/978-3-540-72912-9_10
45. Meng S, Liu L, Wang T (2011) State monitoring in cloud datacenters. IEEE Trans Knowl Data Eng 23(9):1328–1344. https://doi.org/10.1109/TKDE.2011.70
46. Moser O, Rosenberg F, Dustdar S (2008) Non-intrusive monitoring and service adaptation for ws-bpel. In: Proceedings of the 17th International Conference on World Wide Web, WWW '08. ACM, Beijing. pp 815–824. https://doi.org/10.1145/1367497.1367607
47. Papazoglou MP (2012) Cloud blueprints for integrating and managing cloud federations. In: Software service and application engineering. Springer. pp 102–119. https://doi.org/10.1007/978-3-642-30835-2_8
48. Pham LM, Tchana A, Donsez D, De Palma N, Zurczak V, Gibello PY (2015) Roboconf: a hybrid cloud orchestrator to deploy complex applications. In:

2015 IEEE 8th International Conference on Cloud Computing. IEEE. pp 365–372. https://doi.org/10.1109/cloud.2015.56
49. Popescu R, Staikopoulos A, Brogi A, Liu P, Clarke S (2012) A formalized, taxonomy-driven approach to cross-layer application adaptation. In: ACM Trans Auton Adapt Syst Vol. 7. pp 7:1–7:30. https://doi.org/10.1145/2168260.2168267
50. Povedano-Molina J, Lopez-Vega JM, Lopez-Soler JM, Corradi A, Foschini L (2013) Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds. Future Gener Comput Syst 29(8):2041–2056. https://doi.org/10.1016/j.future.2013.04.022
51. Ranjan R, Benatallah B, Dustdar S, Papazoglou MP (2015) Cloud resource orchestration programming: overview, issues, and directions. IEEE Internet Comput 19(5):46–56
52. Rossini A, Kritikos K, Nikolov N, Domaschka J, Griesinger F, Seybold D, Romero D (2015) D2.1.3 —- CAMEL Documentation. https://paasage.ercim.eu/images/documents/docs/D2.1.3_CAMEL_Documentation.pdf
53. Sebrechts M, Van Seghbroeck G, Wauters T, Volckaert B, De Turck F (2018) Orchestrator conversation: Distributed management of cloud applications. Int J Netw Manag 28(6):e2036
54. Sellami M, Yangui S, Mohamed M, Tata S (2013) Paas-independent provisioning and management of applications in the cloud. In: Cloud Computing (CLOUD) 2013 IEEE Sixth International Conference on. IEEE. pp 693–700. https://doi.org/10.1109/cloud.2013.105
55. Seybold D, Baur D, Held F, Skrzypek P D4.5 data processing layer prototype. http://www.melodic.cloud/deliverables/D4.5DataProcessingLayerPrototype.pdf
56. Seybold D, Hauser CB, Volpert S, Domaschka J (2017) Gibbon: An availability evaluation framework for distributed databases. In: OTM Confederated International Conferences On the Move to Meaningful Internet Systems. Springer. pp 31–49. https://doi.org/10.1007/978-3-319-69459-7_3
57. Seybold D, Keppler M, Gründler D, Domaschka J (2019) Mowgli: Finding your way in the dbms jungle. In: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering. ACM. pp 321–332. https://doi.org/10.1145/3297663.3310303
58. Seybold D, Wagner N, Erb B, Domaschka J (2016) Is elasticity of scalable databases a myth? In: Big Data (Big Data) 2016 IEEE International Conference on. IEEE. pp 2827–2836. https://doi.org/10.1109/bigdata.2016.7840931
59. Seybold D, Woitsch R, Domaschka J, Wesner S (2018) Bpaas execution in cloudsocket. Universität Ulm, Ulm. pp 292–293
60. Shao J, Wei H, Wang Q, Mei H (2010) A Runtime Model Based Monitoring Approach for Cloud. In: CLOUD. IEEE Computer Society. pp 313–320. https://doi.org/10.1109/cloud.2010.31
61. Walraven S, Van Landuyt D, Rafique A, Lagaisse B, Joosen W (2015) Paashopper: Policy-driven middleware for multi-paas environments. J Internet Serv Appl 6(1):2
62. Ward JS, Barker A (2014) Observing the clouds: a survey and taxonomy of cloud monitoring. J Cloud Comput 3:1–30
63. Weerasiri D, Barukh MC, Benatallah B, Sheng QZ, Ranjan R (2017) A taxonomy and survey of cloud resource orchestration techniques. ACM Comput Surv (CSUR) 50(2):26
64. Woitsch R, Falcioni D, Utz W, Sosa R, Iranzo J, Pavelescu M, Cacciatore S, Gallo A, Griesinger F, Seybold D, Kritikos K, Laurenzi E, Lammel B, Hinkelmann K D4.5 – final cloudsocket architecture. https://site.cloudsocket.eu/documents/251273/350509/D4.5/3fd5e0de-63c0-47eb-b8e7-a7942e517928?download=true
65. Zeginis C, Kritikos K, Plexousakis D (2015) Event pattern discovery in multi-cloud service-based applications. IJSSOE 5(4):78–103
66. Zengin A, Kazhamiakin R, Pistore M (2011) CLAM: cross-layer management of adaptation decisions for service-based applications. In: ICWS. IEEE Computer Society. pp 698–699. https://doi.org/10.1109/icws.2011.76

Publisher's Note