Tasks meet Flows: Merging Two Paradigms in a Cloud Applications Development Platform

Gustavo Alves¹, Everton Cavalcante¹, Frederico Lopes¹, Ernani Azevedo², Ramide Dantas², Thais Batista¹, Stenio Fernandes², Carlos Alberto Kamienski³

¹Federal University of Rio Grande do Norte (UFRN), Natal, Brazil
²Federal University of Pernambuco (UFPE), Recife, Brazil
³Federal University of ABC (UFABC), Santo André, Brazil

{gustavo, evertonrsc}@ppgsc.ufrn.br, fred@imd.ufrn.br, {ernani, ramide}@gprt.ufpe.br, thais@ufrnet.br, stenio@gprt.ufpe.br, cak@ufabc.edu.br

Abstract—In this paper we present a new architecture for the Cloud Integrator platform that extends it in two directions. The first one aimed at improving the way of an application can be defined by a developer. Instead of dealing with the burden of defining applications only by specifying semantic workflows in terms of activities expressed as a tuple <task, object>, a developer can rely on the definition of complete execution flows or partial execution flows. The second extension aimed at the automatic deployment of applications in a cloud platform, thus allowing the simultaneous use of it by several clients over the Internet. We also present a proof-of-concept that illustrates the use of the cloud-based platform in a flight booking application.

Keywords—Cloud Computing; workflows; tasks; service composition;

I. INTRODUCTION

The growing interest in the Cloud Computing paradigm is grounded on its utility model in which computing services are delivered through a pay-per-use model. By exploiting such model, applications can be composed of services provided by distinct third-party cloud providers. In this scenario, a single service offered by a cloud platform may not be enough to meet all the requirements of client applications. To fulfill such requirements, it may be necessary, instead of a single service, a service composition that aggregates services provided by different Cloud Computing platforms. However, current cloud platforms are not implemented using common standards. Each one has its own APIs, development tools and virtualization mechanisms. This is a barrier for composing applications by using services provided by different cloud platforms.

In this context, our previous work introduced Cloud Integrator [1], a service-oriented middleware platform that provides an environment to facilitate the development and execution of applications that use services provided by different Cloud Computing platforms. In such an environment, the user specifies an application in terms of a semantic workflow [2] and the service composition and selection mechanisms choose the cloud services provided by the integrated platforms that fulfill the application’s business goal.

However, Cloud Integrator has some limitations, such as: (i) applications created in the Cloud Integrator environment are locally executed, thus meaning that the application runs on the same computational node that Cloud Integrator and its outputs are only available in such node; (ii) inexperienced users cannot specify the application in terms of its inputs and outputs, so that users need to deal with the complexity for creating a semantic workflow; and; (iii) experienced users cannot directly determine the concrete Web services that will perform the workflow.

In this paper, we propose a new architecture for Cloud Integrator that incorporates new elements and reformulates the existing ones in order to address some of the aforementioned limitations. In a nutshell, we have improved the element responsible for supporting users to create applications, so that they can specify applications not only in terms of abstract workflows, which describes activities as a tuple <task, object>, but now in terms of: (i) complete execution flows, which support the selection of concrete services, or; (ii) partial execution flows, which enable users to select just the inputs and outputs of the application. Furthermore, we have also added elements that enable Cloud Integrator to deploy and manage the created applications in private/public cloud platforms, so that applications can be simultaneously used by several clients (users and/or applications) over the Internet.

This paper is structured as follows. Section II presents the basic concepts needed to fully understand the representation of the application in terms of workflows (Section II.A) and the type of services considered in this work (Section II.B). Section III presents the reformulated architecture of Cloud Integrator. Section IV presents a proof-of-concept application in order to enable a better understanding of the original Cloud Integrator and the changes proposed in this work. Section V discusses related works. Section VI contains the final remarks.

II. BASIC CONCEPTS

This section presents the basic concepts needed to fully understand our approach.

A. Workflows

A workflow is the automation of a business goal (or part of it) of an application [3]. Workflows are specified in terms of a set of steps (or activities) that must be performed by services
in order to meet such a business goal. In this perspective, they are designed to be independent of the concrete services that perform the specified activities, thus decoupling applications from the underlying services [2]. In this work, we consider two types of workflows, namely abstract workflows (hereinafter semantic workflows) and concrete workflows (hereinafter execution plans), as explained in the following subsections. This separation is useful because the user can only specify the activities for a workflow in a high level of abstraction, so that he/she does not need to know the services that will perform the specified activities.

1) Semantic Workflows

A semantic workflow is described in terms of abstract activities as its sequential steps in order to meet the application’s business goal. Each activity is specified by a tuple \(<\text{task}, \text{object}>\) (e.g., \(<\text{store}, \text{file}>\), \(<\text{send}, \text{message}>\), etc.) in which a task represents an operation implemented by one or more services and an object can describe inputs, outputs, preconditions, and effects related to a task. In this perspective, the concrete services that perform each abstract activity are found at runtime based on \(<\text{task}, \text{object}>\) tuples. In our approach, applications can be described as a semantic workflow. For example, considering a flight booking application, the user searches for available flights based on criteria such as departure and arrival airport and dates. If there are available flights, the user selects the desired flights and the system generates a booking regarding the selected flights. Next, the payment via credit card is executed and it is generated and stored a payment receipt containing all data about the booking and the client if the payment is successfully performed. Finally, a confirmation message is sent to the user. Fig. 1 depicts the execution flow with these activities and their tuples.

Fig. 1. Execution flow of the activities of a typical flight booking application.

The task and object concepts are represented in terms of ontologies [4], which provide a common vocabulary for a specific domain with a high degree of formal expressiveness. In our context, we use an ontology called domain ontology, which models specific concepts regarding a given class of applications and is structured in terms of tasks and objects [4]. Such ontology is described by using the Web Ontology Language (OWL) [5]. Fig. 2 shows a partial representation of the FlightBooking domain ontology, which describes task and object concepts related to a flight booking application. The services that perform the specified activities are described as semantic Web services [6] in order to improve the power of service representation. Such services are described by using the Web Ontology Language for Web Services (OWL-S) [7], thus enabling to describe Web services in a machine-interpretable and unambiguous way and increasing the degree of automation of service composition. The semantic description of services enables inference machines to automatically identify the services that perform the specified activities, so that the application developer does not need to directly choose the services to be used at development time.

2) Execution Plans

An execution plan is a concrete workflow that contains a set of orchestrated Web services, thus representing a concrete specification for a semantic workflow. The execution plans are built through an on-the-fly process of service and composition according to the semantic interface of the selected services and the semantic workflow specification. Once a semantic workflow is abstractly specified in terms of the activities of an application, a service composition can be performed based on such a specification, i.e., by searching for concrete services that perform each one of the activities specified in the workflow. Then, execution plans are created by composing these concrete services. However, the user can also directly compose execution plans if he/she knows the concrete services.

Since different services with similar functionality may be available in the environment at a given moment, more than one execution plan can be created for a workflow. Therefore, it is necessary to perform a service selection algorithm that selects an execution plan to be executed among the available ones. To guide this choice, the selection algorithm uses service metadata, such as Quality of Service (QoS) parameters and prices. In our approach, the execution plans of a workflow are represented by an directed acyclic graph (DAG) in which each intermediate graph node represents a service and the directed edges represent the sequence of services execution. Fig. 3 shows an example of a DAG that represents a set of four possible execution plans to a workflow with seven activities (indicated by the strips).

B. Service Types

This work considers two types of services, namely atomic services and composite services. An atomic service, which is the most basic type of service, is an indivisible software component with few business or technical functionalities that cannot be decomposed in smaller components, e.g., sending pictures to a Web site. In turn, composite services are generated...
by the composition of atomic services and other composite services, so that the services collaboratively work in order to achieve a more complex functionality, e.g., a photo album service to which people can send pictures and comment on it. Fig. 4 depicts atomic and composite services.

![PhotoAlbumService](image)

**LEGEND:** Composite Service  Atomic Service

Fig. 4. Atomic and Composite Services.

III. ENHANCING CLOUD INTEGRATOR

Cloud Integrator [1] is a service-oriented middleware platform for composing, executing, and managing services provided by different Cloud Computing platforms, so that it works as a mediator between service providers and applications (clients). Such platform provides an environment to facilitate the development and execution of applications that use the different cloud services by composing semantic Web services [6] in a semantic workflow [2] composed of a sequence of abstract activities that must be performed by concrete cloud services in order to achieve the application’s business goal. After the specification of a semantic workflow in terms of activities, Cloud Integrator performs a service composition based on such specification, i.e., it searches for services that perform each activity specified in the workflow and creates possible execution plans that contain a set of orchestrated concrete Web services that completely fulfill the mission of each activity. Usually, there is more than one execution plan for a workflow, according on the amount of available services with the same functionality available at the moment. Hence, it is necessary to execute a service selection algorithm that selects an execution plan among the available ones. To make this choice, the algorithm uses metadata about the services, such as Quality of Service (QoS) parameters and cost. Finally, Cloud Integrator executes the application, or the selected execution plan, by making requests to each concrete Web service and returns the outputs to the users.

However, applications created in the Cloud Integrator environment are locally executed, thus meaning that the application runs on the same computational node that Cloud Integrator and its outputs are only available in such node. Therefore, Cloud Integrator should include mechanisms to deploy and manage applications in public and private clouds, in order to allow applications to be used by several clients (users and/or applications) over the Internet. In addition, inexperienced users should also be able to specify the application in terms of its inputs and outputs, thus removing the complexity to create a semantic workflow, such as proposed by Dantas et al. [8]. Finally, experienced users cannot directly determine the concrete Web services that will perform the abstract workflow activities in Cloud Integrator, which means that the user cannot choose services that he/she already know and trust because such activities in the semantic workflow can only be specified in a high level of abstraction, thus lacking information about the respective concrete services.

In order to solve some of these limitations, in this paper we propose a reformulated architecture for Cloud Integrator, as depicted in Fig. 5. This new architecture defines three layers, namely: (i) **GUI Layer**, which support users in the applications development; (ii) **Workflow Composition Manager**, which contains components for composing semantic Web services and creating the correspondent execution plans, and; (iii) **Service Management Layer**, which contains components responsible for selecting an execution plan based on QoS metadata, generating an executable code from the selected execution plan, and deploying and managing the executable code in private/public cloud platforms. The Service Management Layer also contains: (i) a *service monitor*, which stores and analyzes monitored data from the applications and cloud platforms; (ii) an *adaptation component* that is responsible for reconfiguring the application in case of service failures or quality degradation, and; (iii) an *elasticity manager* that is responsible for providing elasticity capabilities for the deployed application. The components of each layer are detailed in the following subsections.

![Reformulated architecture for Cloud Integrator](image)

**A. GUI Layer**

The **GUI Layer** is responsible for supporting the user to create applications by: (i) specifying semantic workflows, in which the user defines activities that compose the application in terms of tasks and objects, or; (ii) completely defining the application execution flow, so that the user selects the concrete services of each activity of the flow, or; (iii) using an intermediate approach in which the user defines a partial flow in terms of inputs and outputs and/or some services, and the execution flow is automatically generated. The execution flow is basically an execution plan that is directly specified by the user, i.e., it is not automatically generated from a semantic workflow. As described in Section II.A, the specification of the abstract workflow consists in describing the activities that compose the application in terms of tasks and objects that are in the domain ontology stored in the **Base Repository** component. The graphical user interface shows the list of concepts related to the task and list of concerns related to the objects, and the user chooses the <task, object> tuple to describe an activity of the abstract workflow. For instance, the first activity of the abstract workflow presented in Fig. 1 is described by the <search, flight> tuple. As the workflow activities are abstract, the corresponding concrete services are found in the **Base Repository** by the
Semantic Composer component. In turn, the complete definition of the execution flow consists of directly selecting concrete services to compose an application flow. A list of concrete services is presented to the user for choosing the services that will compose the execution flow. Fig. 6 presents a complete execution flow defined by a user, in which he/she chooses seven services to compose a flight booking application. The user is free to choose services stored in the Base Repository component as well as add new services to such repository and use them.

Finally, the partial definition of the execution flow consists of choosing only the application inputs and outputs and the Semantic Composer component dynamically creates the execution flow by using services of the Base Repository component. Moreover, the user can directly select some services to compose the flow, while the Semantic Composer component is in charge of selecting the additional services to complete the application flow. Fig. 7 presents the partial execution flow defined by the user, in which he/she selected only the inputs (departure and arrival airports and arrival and departure dates) and outputs (confirmation message) and two concrete services of a flight booking application.

B. Workflow Composition Layer

The Base Repository component is responsible for storing the OWL-S semantic description of atomic or composite services, as well as the domain ontologies (such as the FlightBooking domain ontology presented in Error! Fonte de referência não encontrada., Fig. 2).

The Semantic Composer component generates execution plans from the specification of the abstract workflow or the execution flow. When the input is an abstract workflow, this component performs an automatic composition of services from the semantic description by using the algorithm proposed in [9]. This algorithm composes semantic Web services from the specification of an abstract workflow by taking into account inputs, preconditions, and the output of each workflow activity. The services are then organized according to the message flow between the output of a service and the input of the subsequent one. During the composition of the services, when two or more services perform the same activity, two or more execution plans are generated for the workflow that contains such an activity. When the Semantic Composer component receives a complete execution flow, it translates this flow into a single execution plan since all Web services has been already selected by the user. However, if the execution flow is only partially described, the Semantic Composer component searches in the Base Repository component for services that meet the input parameters by concatenating services until matching the output parameters, as proposed by Dantas et al. [8]. In such concatenation process, if it finds two or more services with the same input and output, then the algorithm generates two or more execution plans and continues processing in order to complete the execution flow.

C. Service Management Layer

After the generation of execution plans is completed, the Selection Manager component selects one of them (if more than one plan has been generated), based on QoS metadata (e.g. response time, up time, mean time between failure, etc.) related to services that compose them and some user preferences (e.g. cloud provider, adaptation and elasticity thresholds, virtual machine resources, etc.). QoS metadata are provided by Service Monitor component, which is responsible for retrieving the quality metadata from atomic and composite services. That component also is responsible for receiving notifications about the need for adaptation, which means replacing the execution plan for another one that performs the same features, and for elasticity. We have already presented a candidate selection algorithm in Cavalcante et al. [10]. Once the selection is performed, the selected execution plan is forward to the Code Generator component. Such component is responsible for encapsulating an execution plan in an executable code that can be deployed in virtual machines created in a given cloud platform. Code Generator encapsulates the semantic descriptions of all atomic/composite services included at the execution plan and their parameters (stored in the Base Repository component) to generate the application code in WS-BPEL (Web Service Business Process Execution Language) [11]. Moreover, that component is also responsible for exporting metadata with user preferences. After the generation of executable code, the Scheduler component deploys it onto virtual machines instances on available cloud platforms. This component considers user preferences - if ever specified - to choose the most appropriate server to install the virtual machine in a cloud platform. Note that it is possible to choose the server just if Scheduler is dealing with private clouds, since it has total knowledge and autonomy to allocate/deallocate resources on demand. In cases of public clouds, not every cloud platform enables the allocation/deallocation on demand. Scheduler component is even responsible for executing installation scripts for dealing with application dependencies. Examples of such scripts are compilers, interpreters and libraries, which will make it possible to appropriately deploy and execute an application. Such a component is also responsible for installing application dependencies in virtual machines, such as a WS-BPEL engine. Finally, it is also responsible for deploying a local monitor together with the deployed application. Such local monitor is responsible for monitoring: (i) applications; (ii) requests to services that compose an application, and; (iii) parameters of the used virtual machine resources, in which the application is deployed, such as CPU and memory usage.

The Service Monitor component stores and analyses data received by all monitors deployed with the running applications. According to user-defined time intervals, such compo-
A component receives information about the requests to the services that comprise the application in order to identify service failure and to assess service quality parameters in order to indicate the degradation of the quality of atomic/composite services. In both cases, the Adaptation Manager component processes adaptation events, and decides whether adaptation is worthwhile according to different criteria, also known as the adaptation thresholds, specified by the user. This component considers information about virtual machine resource usage, such as CPU and memory, in order to trigger elasticity events on the virtual machine in which the application is running. Elasticity is handled by the Elasticity Manager component.

The Adaptation Manager component deals with the need of adaptation in case of: (i) failure of atomic/composite services, (ii) quality degradation of atomic/composite services that significantly impacts on the application quality, or (iii) the introduction of new services into the system. An adaptation process has been presented in [12]. In case of adaptation for service failures, new execution plans are generated by Semantic Composer and then, a new execution plan is selected considering the updated quality metadata. In case of quality degradation, the Selection Manager component updates the quality values of the services that compose the application and verifies if the degradation is meaningful to offset the cost of deploying a new application, considering thresholds defined by the user. Finally, in cases where new services are added to the system, the Selection Manager component investigates whether the execution plan quality may be improved with and also uses its thresholds to decide if the adaptation must happen or not.

The Elasticity Manager component is responsible for allocating/deallocating on demand virtual machines so they reflect the need of resources. Elasticity is triggered when the virtual machine is overloaded or idle. These concepts depend on the specification of the user preferences and/or the use of the virtual machine resources, which are monitored by the local monitors deployed with the applications. For instance, it is possible to allocate additional virtual machines when the average use of CPU, of the set of machines used to execute the application, exceeds a given threshold (e.g.: 70% or 80%). Similarly, when the average use of CPU is below a given threshold (e.g.: 10% or 20%) the Elasticity Manager has deal with possible deallocation of one or more virtual machines. An important feature associated to elasticity is load balancing, which must be executed after the allocation of new virtual machines to distribute the load among them whenever a request is received. Complementarily, when a virtual machine is deallocated, it must be removed from the set of virtual machines available to the load balancer. Please notice that in some public cloud platforms users do not have access to dynamically creating new virtual machines, which may harm or even prevent an adequate operation of the load balancer. We adopted the use of thresholds for both creating a new virtual machine and to releasing an existing one. These thresholds are specified by the user through our GUI and they are made available to the Service Monitor component, which is the responsible for verifying the need to perform elasticity. Elasticity events are based on hysteresis to protect performance and cost oscillations in short times scales. In other words, a counter verifies the number of times the elasticity threshold is overcome to avoid the creation/release of virtual machines according to peaks/valleys of resource consumption. Thus, a new machine is created/released when the counter reaches a value defined by the user (owner) of the application.

IV. PROOF-OF-CONCEPT

In order to enable a better understanding of the improved Cloud Integrator, the proof-of-concept of this work addresses the variation of the flight booking application used in the abovementioned example, which is an entirely cloud-based flight ticket purchase application using a credit card. The final application should be a service able to consider destination, arrival and departure times – expected requirements of such application. Using the original Cloud Integrator users can only specify the application in terms of an abstract workflow. The user can also define preferences, such as maximum execution time, or adaptation thresholds. The workflow must be described in terms of abstract activities, for example, in the case of flight ticket scenario, expected activities are search for flight (<search, flight>), make payment (<make, payment>), and generate receipt (<generate, receipt>). The concrete services that implement such activities are not accessible in this abstraction level. From the abstract description, Cloud Integrator generates execution plans, which will in fact comprise the concrete services, able to be executed on an appropriate environment. Fig. 8 depicts the example defined through Cloud Integrator.

Fig. 8. Cloud Integrator using a FlightBooking ontology

However, the new Cloud Integrator allows inexperienced users to define an application in terms of its inputs, outputs and some concrete services. Even if the user is not aware of the services that should be used, modules of the Workflow Composition Layer can give assistance to the user in the full composition of the execution flow. In the ticket purchase example, the user would provide the semantically aided input
types DepartureAirport, DepartureDate, ArrivalAirport, ArrivalDate and the output type ConfirmationMessage, so the composition is supposed to have a payment component. To assist the workflow generation, the user could optionally insert an intermediate service/parameter to the composition, specifying it is necessary to include a specific component (for instance, to accept only a given credit card flag). Fig. 9 illustrates a partial workflow description, using the composition (black) box. After discovering, a set of matching services to compose a concrete composition is ready to be translated to code and deployed.

Finally, the new approach combines the ease of description using tasks with the power of directly modifying the workflows. In the same scenario, a developer describes the tasks of the first example, and then decides to assign a specific payment method to it. Having the concrete workflow (obtained from Semantic Composer), the partial workflow interface is used to change one step of the composition to a desired service. Likewise, an extra SendSMS atomic service can be added (along with its appropriate parameters), to send a confirmation message to the cellphone of the ticket buyer. Once the final composition is defined, the Code Generation module uses the provided data to yield the workflow and its alongside dependencies, so the composition is ready to be deployed, through the Scheduler component, in the prepared virtual machine. Then, the Service Monitor starts monitoring the running service and it is subjected to Adaptation and/or Elasticity, if the monitor assesses this is needed.

V. RELATED WORK

In this section, we briefly discuss some works in terms of application selection and composition strategies for the Cloud Computing applications that are somewhat related to our proposal. Regarding to the composition and selection of Cloud services, Bao and Dou [12] propose a Web service composition modeled as a Finite State Machine (FSM) in which a search is performed to find concrete services according to the composition business process, or target process, using a composition tree, to generate possible execution paths. Despite similarities, the service composition proposed by the authors is modeled as a FSM, so that a composition tree can be built to enable the discovery of concrete services that will perform the activities of the business process. However, in our proposal the service composition process is simpler. Users can specify an application in terms of a semantic workflow, a complete execution flow or a partial execution flow. Except for the complete execution flow specification, when the user directly selects concrete services to compose an application, the platform dynamically discovers the concrete services that will perform the activities. Wu and Khoury [14] propose a cloud Web service composition algorithm that has elements similar to Cloud Integrator. The service composition is modeled as a task sequence specified as an UML Activity Diagram, which defines the order that the services should be composed and executed (and that can be seen as the abstract workflow in Cloud Integrator). Then, according to the available concrete services and the task sequence, a tree is generated in which the tree nodes represent Web services and the edges represent the composition between two Web services. Therefore, each path in the tree from the root node to leaf nodes represents a complete composition and can be seen as an execution plan of our approach. A filtering process is performed over the tree to remove edges that violate user’s QoS thresholds and illegal compositions. Finally, a selection heuristic algorithm is executed to select the path that represents the service composition.

VI. FINAL REMARKS

In this paper we presented a new architecture for the Cloud Integrator platform to give the possibility to developers specify applications by using semantic tasks, complete or partial flows and also to allow the simultaneous use of cloud applications. The new architecture encompasses new modules: (i) Code Generator and Scheduler that enables the generation of executable code and the deployment and management of the newly created application in private/public cloud platforms. We also have improved the element responsible for supporting users to create applications and generate execution plans; (ii) GUI, to enable users to specify applications in terms of abstract workflows, complete execution flows, or partial execution flows, and; (iii) Semantic Composer, to generate execution plans from the specification of abstract workflows or execution flows. Finally, we illustrated the development of a flight booking application using the new Cloud Integrator.

REFERENCES