A PaaS for Composite Analytics Solutions

Paula Austel, Han Chen, Parijat Dube, Thomas Mikalsen, Isabelle Rouvellou, Upendra Sharma, Ignacio Silva-Lepe, Revathi Subramanian, Wei Tan, Yandong Wang
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Rt 134, P.O. Box 218, Yorktown Heights, NY 10598
{pka, chenhan, pdube, tommi, rouvellou, usharma, isilval, revathi, wtan, yandong}@us.ibm.com

ABSTRACT

In their pursuit of market competitiveness and sustainable top line growth, enterprises are increasingly turning to sophisticated analytics solutions to derive insights and value from the deluge of data that are being generated from all sources. Leading practitioners of Big Data analytics have already moved past the stage of using single analytics modalities on siloed data sources. They are starting to create *composite* analytics solutions that take advantage of multiple analytics programming models and are also integrating them into their existing enterprise IT systems. At the same time, the CIOs have wholeheartedly embraced cloud computing as a means of reducing the capital and operational cost of their IT systems and streamlining their DevOps processes. Platform-as-a-Service (PaaS) as a cloud computing consumption model has seen wide acceptance by developers and IT administrators. Although there are PaaS platforms for individual workload types involved in these advanced composite analytics solutions, the composition aspect is not addressed by any of these individual PaaS platforms. Further, there is no lifecycle management support for the solution as a single logical entity. This paper argues for the need of a true PaaS for composite analytics solutions in order to accelerate their adoption by the industry and foster the creation of a healthy ecosystem. We present the design and prototype implementation of such a platform and our early experience of using it to deploy a Telco Fraud Detection solution.

Keywords

Cloud Computing, Platform-as-a-Service, Big Data Analytics, Solution Composition, Solution Lifecycle

1. INTRODUCTION

Two emerging technologies, Big Data Analytics and Cloud Computing, are increasingly being adopted by enterprises large and small to drive the top-line growth, by deriving value from data while at the same time reducing cost of running their IT.

Big Data Analytics is a term used to describe a large family of analytics techniques and workload types. MapReduce is one of the most well-established patterns for processing large amounts of data. Stream processing, graph analytics, machine learning, and deep learning are some of the other emerging techniques that are gaining popularity in the analytics community. When enterprises first started using analytics, these different techniques were treated as isolated islands. Increasingly, practitioners are creating sophisticated solutions that combine multiple distinct analytics techniques, and further integrating them with traditional n-tier web applications for transaction, visualization, etc.

An example, which we shall describe in greater details in the following sections, involves Telco fraud detection using call data. One scenario of the fraud detection is based on thresholds on apparent velocity of mobile device movement. An existing enterprise IT system produces live call data records, which include the subscriber identity of a mobile device and the current cell tower that it is associated with. These data are fed into a continuous stream processing application that computes the apparent movement velocity of a device based on the ever changing association between the mobile device and the cell tower, and the known positions of these towers. When abnormally high movement velocity is detected, the corresponding call record is flagged as a potential fraud and an alert event is emitted. These events are sent to a Hadoop Distributed File System (HDFS) for further downstream processing using MapReduce. They are also fed into a visualization web application that gives the administrator an at-a-glance view of the flagged events.

Even in this relatively simple scenario of an overall fraud detection solution suite, three different application component types are used to create a single integrated application: web application, MapReduce application/HDFS, and stream processing. Developing, testing, deploying and managing such a composite solution can be a complex task.

To reduce the operational cost of running their IT and to simplify the management of solutions, including analytics solutions, enterprises are embracing cloud computing. The two main cloud computing consumption models that are relevant to our discussion here are Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS). At the risk of grossly simplifying the picture, we can view IaaS as an API (Application Programming Interface) for customers to request compute resources (CPU, memory, disk, network, etc.) on demand in order to execute their workload; whereas PaaS provides an API for a customer to deploy a workload directly without having to worry about the underlying compute resources.

While IaaS maximizes flexibility by allowing customers to configure the compute infrastructure to suit their solution (workload) need, the overhead may be significant. For example, to deploy the aforementioned Telco fraud detection solution, one would need to request multiple properly sized virtual machines, configure the network, install a number of middleware stacks (Tomcat and MySQL for the web application, Apache Hadoop for MapReduce and HDFS,

CF'15, May 18 – 21, 2015, Ischia, Italy
Copyright 2015 ACM 978-1-4503-3358-0/15/05 ...$15.00
http://dx.doi.org/10.1145/2742854.2747281.

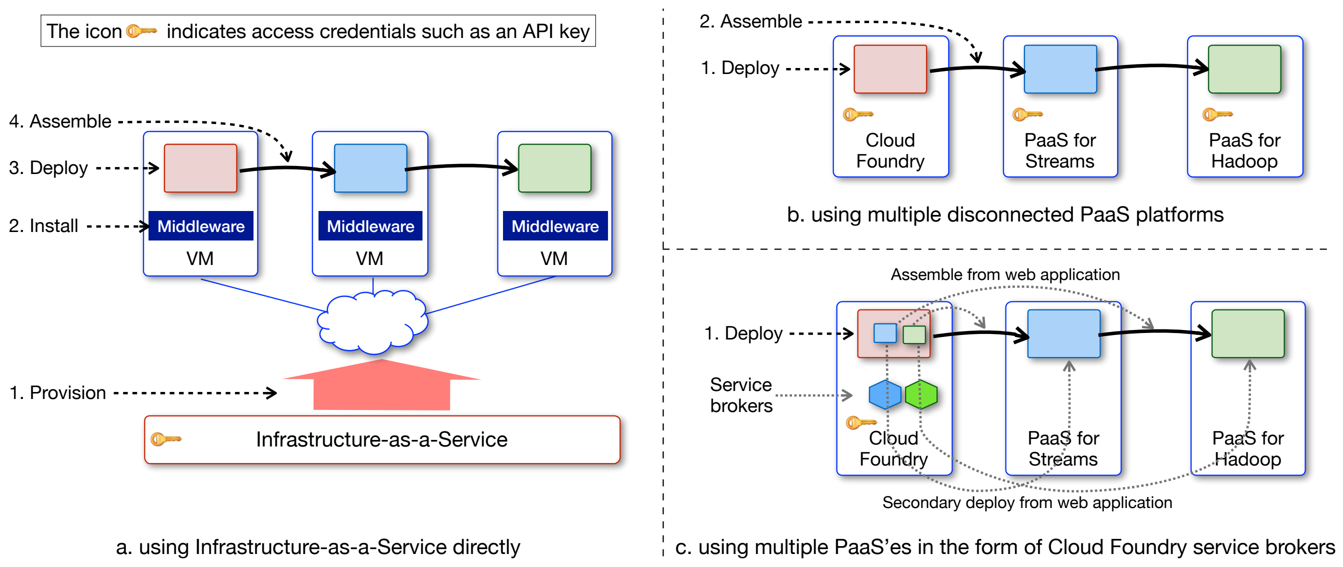
The icon indicates access credentials such as an API key
4. Assemble
3. Deploy
2. Install
Middleware
VM
Middleware
VM
Middleware
VM
1. Provision
Infrastructure-as-a-Service
a. using Infrastructure-as-a-Service directly
2. Assemble
1. Deploy
Cloud Foundry
PaaS for Streams
PaaS for Hadoop
b. using multiple disconnected PaaS platforms
Assemble from web application
1. Deploy
Service brokers
Cloud Foundry
PaaS for Streams
PaaS for Hadoop
Secondary deploy from web application
c. using multiple PaaS'es in the form of Cloud Foundry service brokers

Figure 1: Deploying composite analytics solutions with existing technologies

Apache Storm for Stream processing), deploy the solution components to the respective middleware containers, and configure the components to communicate with each other so that they form a single composite solution (See Figure 1a). With the increasing demand for DevOps integration and continuous delivery being placed on the development organization, simply managing such a complex deployment on IaaS can become a major productivity drain. Although there are several frameworks such as Chef, Puppet, Jenkins, that address the automation issues, non-trivial effort is still required to create the proper plugins/scripts for these tools.

This has motivated the creation of PaaS and the workload-centric view of how Cloud should be consumed. For example, Cloud Foundry is a PaaS that allows customers to deploy and manage web applications on a completely virtualized web container with a simple API[1] and without worrying about the virtual machines and middleware stacks that are required to run these web applications. Similarly, Genie [5] can be viewed as a PaaS for MapReduce workloads, in that it provides a developer with an API to deploy and execute MapReduce jobs without thinking about the underlying Hadoop cluster.

With PaaS support for individual solution component types, a composite solution can be more easily deployed than with an IaaS. However, the solution assembly/composition aspect is still not being adequately addressed. After the constituent components are deployed to the respective PaaS stacks, additional configuration of these application components is still required to connect them so that they are aware of each other and thus form a single composite solution. This configuration is not easily captured as artifacts that can be managed as code to enable seamless DevOps integration. In addition, each PaaS API usually requires a distinct authentication credential, further complicating the picture (see Figure 1b).

As PaaS for web applications, such as Cloud Foundry, continues to gain popularity among developers and IT administrators alike, analytics service providers have also started to expose analytics PaaS APIs as services using CF service brokers. This allows a web application to consume analytics PaaS services using a well-established convention (VCAP_SERVICES environment variable at runtime) without having to manage multiple sets of credentials separately. However, in this model, the web application is forced to become the center of the composite solution, additional analytics components are relegated to being second class objects and the assembly/composition becomes logic inside the web application (see Figure 1c). As such, this is also a less than desirable approach.

[1]We use API to include both REST API and command line interfaces.

We argue that, to truly facilitate the management of composite analytics solutions, there needs to be a PaaS for composite analytics solutions. By this, we mean the following

1. All solution components shall be treated as first class objects: web applications, MapReduce jobs, Streams processing graphs, etc.

2. The assembly/composition of these components shall be captured as first class objects as well, so that they can be easily understood. Further, this enables the separation of duty between component developer and solution assembler and fosters the creation of an ecosystem of analytics solutions

3. There shall be a high level API that allows a solution deployer to deploy and manage the composite as a single entity. This, by our generalized definition of PaaS, means the creation of a PaaS for composite analytics solutions.

To this end, this paper describes a research prototype that realizes this vision. This system is part of a much larger effort to create a software defined environment for composite solutions called Solution Foundry. For more details about Solution Foundry, please refer to [6]. The rest of the paper is organized as follows. Section 2 describes the high-level architecture and system design of Solution Foundry that are relevant to PaaS support for composite analytics solutions. Section 3 describes our initial prototype implementation. Section 4 uses an example solution to illustrate our experiences with the system. Related work is discussed in Section 5, and Section 6 concludes the paper.

2. SYSTEM ARCHITECTURE

The key components of the Solution Foundry include i) a specific language for describing solutions as composite applications, ii) a command line interface, iii) Maestro: a deployment engine that directs the deployment of composite applications and iv) a set of

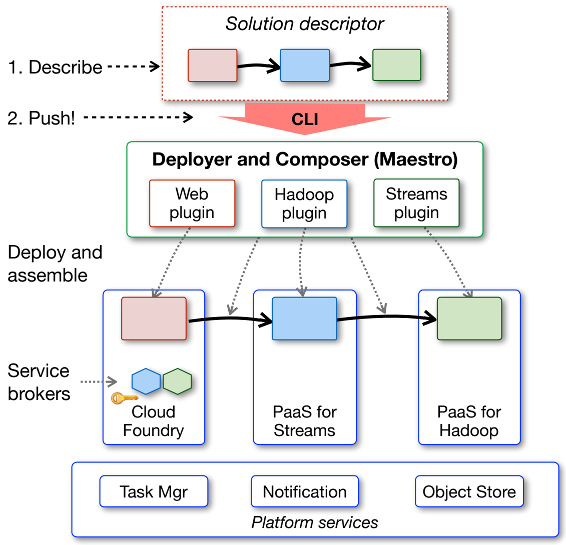

Figure 2: High level system architecture

platform-provided services that may be leveraged by the solution (composite application).

The high level architecture of Solution Foundry (SF) is shown in Figure 2. SF allows the developers to describe a solution in a declarative manner. At the core of SF lies *Maestro* that deploys various solution components on their respective underlying platforms and also binds them together to work together as a single large distributed application. Maestro leverages plugins to deploy on individual platforms and can be easily extended by adding new plugins.

In this section we will briefly describe the architecture of SF and discuss some of the important design choices that we have made.

2.1 Solution Descriptor

SF allows developers or solution administrators to describe the complete solution in a declarative manner as a configuration file; we call this configuration file the *Solution Descriptor*. The primary intention behind adopting a declarative approach is to aid the developer to easily describe the individual components of the solution and also their dependencies without worrying about the exact control flow of the complete solution deployment.

We have modeled various solution component types, namely *webapps, mapreduce, streams, services and tasks*, and captured their resource and configuration requirements along with inter-component dependencies in a solution descriptor as shown in Figure 3.

2.2 Command line interface (CLI)

CLI is a tool that would be locally present on deployer's development machine. CLI is designed to communicate with Maestro using REST APIs (Maestro itself runs as a service) for performing all the solution deployment and management tasks on the platform remotely. For instance for deploying solutions, deployer needs to provide the solution descriptor; CLI will parse the solution descriptor to identify all the solution components (located either locally or some remote repository like github or some object-store) and will upload the contents to the Maestro.

2.3 Maestro

Maestro is the core of our platform that interacts with *CLI* for various user operations and with the underlying component platforms for solution deployment. Maestro interacts with *CLI* over the REST APIs to allow the platform users to perform basic operations namely login, logout, solution-deployment , solution-status etc and various advanced operations like snapshotting application, rolling upgrade etc. On the other hand it parses the solution descriptor and deploys various component workloads on corresponding platforms; for instance web component-workloads are deployed on PaaS for web applications, Hadoop type of analytics workloads are deployed over PaaS for Hadoop and so on and so forth.

Maestro interacts with various platforms on behalf of the user/developer and performs the task of deployment of corresponding component workloads over underlying platforms and finally binds them together as a single distributed application. Maestro has a pluggable architecture, as shown in Figure 2, wherein a plugin communicates with the underlying platform for deployment, monitoring and management of a component-workload of the solution. Pluggable architecture makes extending the Maestro, and thereby our platform, very easy as for any new type of workload one has to just write a small plugin. For this to work we have modeled each of the operations of Maestro as a set of abstract operations and each plugin would implement it for their respective platforms; for instance the deployment of a solution has been modeled as a three phase operation, namely i) *create*, ii) *configure* and iii) *start* and each of plugin would implement these phases for their specific underling platform.

Create phase: in this phase the Maestro creates necessary resources on the respective platforms; for instance if the Maestro has to deploy a web application, it will create a domain for the solution component, a space for uploading the web-application and corresponding buildpack[2].

Configure phase: Maestro interprets the declarative specification of the solution and executes them by co-ordinating and managing the relationships and dependencies of solution components. It includes connecting/binding the related components by informing the contact endpoints of related components to each other or moving the data to and from analytics applications. It is assumed that contact end-points of components can be obtained before they can be started and also that they are reachable across different underlying platforms. This is not a limitation as most of the underlying platforms treat the end-points as resources, for instance one could reserve floating IP-addresses before creating VMs.

Start phase: Finally once each component workload of the solution is properly configured individual components are started. Deployers could also specify a sequence in which the solution components have to started and Maestro will ensure that sequence.

2.4 Platform services

SF also allows solution deployers to include existing platform services as a solution component under the category of *services*, as shown in Figure 3. This allows service deployers to use third party API-consumable services on which their solution depends; for instance if a solution depends on object-store service from an IaaS provider, one could easily leverage it by including it in the solution descriptor, which then provides the service end-point and access credentials to other solution components.

3. PROTOTYPE

We have implemented a prototype of the proposed PaaS system (as part of the larger Solution Foundry) on SoftLayer®, an IBM

[2]Buildpacks provide framework and runtime support for applications that are to be deployed on PaaS. Buildpacks examine application artifacts to determine dependencies and configuration to communicate with bound services.

```
 1  ---
 2  properties:
 3    container: demo
 4    inputDir: /input
 5    outputDir: /output
 6
 7  webapps:
 8  - name: front-end
 9    package: bin/FrontEnd.war
10    instances: 1
11    memory: 256
12    environment:
13      runHadoopJobUrl: "#{run-hadoop-job.url}"
14
15  mapreduce:
16  - name: hadoop-job
17    package: bin/hadoop-job.zip
18    target: icas-biginsights
19    cluster:
20      type: dedicated
21      size: 2
22
23  streams:
24  - name: stream-app
25    package: bin/stream-app.zip
26    composite: com.acme.solution::StreamMain
27    parameters:
28      param1: foo
29      param2: bar
30    target: icas-streams
31    cluster:
32      type: dedicated
33      size: 1
34
35  services:
36  - name: object-store
37    kind: SoftLayer-Object-Storage
38
39  tasks:
40  - name: deleteInput
41    type: hdfs
42    parameters:
43      cluster: hadoop-job
44      action: rmdir
45      dirname: "#{inputDir}"
46
47  - name: import
48    type: data_transfer
49    parameters:
50      objectStore: object-store
51      cluster: hadoop-job
52      source: "swift://#{container}.softlayer/${file}.txt"
53      target: "#{inputDir}"
54
55  - name: run
56    type: run_hadoop
57    parameters:
58      application: hadoop-job
59      parameters:
60        inputDir: "#{inputDir}"
61        outputDir: "#{outputDir}"
62
63  - name: run-hadoop-job
64    type: workflow
65    parameters:
66      definition: sequence[deleteInput, import, run]
67    trigger: http
```

Figure 3: Describing components and composition in a YAML-based descriptor

IaaS cloud. This prototype system supports three different component types that are directly relevant to analytics solutions: web application, MapReduce on IBM® InfoSphere® BigInsights™ and Stream processing on IBM® InfoSphere® Streams. Web application components are targeted to a private Cloud Foundry installation that we have created. We use the *IBM Cloud Analytics Application Services* offering as our analytics platform. We have created two experimental CF service brokers for the BigInsights and Streams services and registered them in our CF environment. The BigInsights and Streams deployer plugins are configured to recognize and exploit these services.

3.1 Analytics plugins

As we have described earlier, the system uses a plugin mechanism to extend both the descriptor language and the platform, so that analytics service providers can grow the platform and provide support for additional component types and target runtimes. Here we describe two analytics plugins that we have implemented.

3.1.1 MapReduce on InfoSphere BigInsights

This plugin allows solution architect to include MapReduce/Hadoop components in composite solutions and deploy them onto IBM InfoSphere BigInsights.

The plugin processes the `mapreduce` section of the solution description (see lines 15 to 21 in Figure 3). This section specifies an array of MapReduce applications that the solution has. Each application has the following attributes. The attribute `name` specifies the name of the component. It must be unique among all components in the solution. The attribute `package` specifies the path to locate the binary package for the MapReduce application. In our design, the deployer introspects the package format to determine the compatible runtime target, much like the way Cloud Foundry handles different language runtimes. Solution developer may override the runtime selection mechanism with the `target` attribute. Currently, because we have only support for InfoSphere BigInsights, the introspection is omitted and the target always defaults to `icas-biginsights`.

The workload placement strategy is declared with the `cluster` attribute tree.The attribute `type` specifies if a `dedicated` cluster shall be created or if the application shall be `collocated` with another application. In the former case, cluster `size` shall be specified; in the latter case, the name of target `application` shall be specified.

During deployment time, the plugin does the following. First, it tries to acquire a target Hadoop cluster for the given MapReduce application. The acquisition is determined by the placement directive. If collocation with another application is specified, the cluster for that application is used. If a dedicated cluster is requested, the plugin instantiates a service instance of the IBM Cloud Analytics Services BigInsights service (hence the `icas-biginsights` target) in the main Cloud Foundry environment. The service details are obtained from Cloud Foundry and they include the cluster management REST API endpoint and the access credentials. The plugin then uses this REST API to create a cluster of the specified size and waits for the cluster to become ready. Next, with the cluster available, the plugin uses BigInsights REST API to upload, install and deploy the application package.

At this point, the MapReduce application is ready to be invoked. Because, unlike web applications which are long-running request-response type, MapReduce applications are batch processing in nature, the actual invocation and runtime parameters to the invocation are specified as platform-managed tasks, which we will describe in subsequent sections.

After a MapReduce application is deployed, the plugin publishes a few assembly-time parameters to the solution namespace for other components to use. These include the public and private IP addresses of the master node (`.publicIp` and `.privateIp`), the username and password for accessing the cluster (`.username` and `.password`) and the application ID (`.appId`).

3.1.2 Stream processing on InfoSphere Streams

This plugin allows solution architect to include continuous stream processing components in composite solutions and deploy them onto IBM InfoSphere Streams.

The plugin processes the `streams` section of the solution description (see lines 23 to 33 in Figure 3). The syntax and semantics of this section closely mirror those of the `mapreduce` section with one major difference. Because streams applications are long running in nature, the declaration must include the entry point to the application and its invocation parameters. The attribute `composite` specifies the main Streams class when the deployment target is `icas-streams`. The attribute `parameters` is a map from parameter names to values.

During deployment time, the plugin first acquires a target InfoSphere Streams cluster based on the placement directive using the same algorithm as the MapReduce on InfoSphere BigInsights plugin. InfoSphere Streams currently lacks a REST API for job management. Therefore the plugin uses `scp` to upload the application package and then uses `ssh` to remotely install, compile and submit the streams application.

Also similar to the MapReduce on InfoSphere BigInsights plugin, these assembly-time parameters are published by the plugin once the deployment of a streams application component is complete: the public and private IP addresses of the master node and the username and password for accessing the cluster.

3.2 Platform services

In addition to the analytics plugins described above, a number of platform services support the deployment and runtime execution of composite solutions. Here we describe three services that are directly related to analytics solutions.

3.2.1 Task manager

The asynchronous and batch nature of Hadoop/MapReduce jobs motivate us to include declarative specification of tasks at solution level and provide support for runtime execution and integration with other component types. Our simple definition of task is a parameterizable activity that can be invoked or scheduled asynchronously.

A task management plugin processes the `tasks` section of the solution descriptor at solution deployment time (see lines 39 to 67 in Figure 3). This section declares an array of individual tasks. Each task definition contains four attributes. The attribute `name` specifies a name of the task, which has to be unique among the names of all components and tasks. The attribute `type` determines the activity that the task will perform and the attribute tree `parameters` is a map of type-specific configuration parameters of a given task. The prototype system currently supports four types.

`hdfs` tasks (see lines 40 to 45) manipulate the HDFS filesystem. The `cluster` parameter points to a declared `mapreduce` component. The operation is performed on the cluster that this component is deployed to. The `action` parameter specifies the operation, for example, `rmdir` for deleting a directory and `mkdir` for creating a directory.

`data_transfer` tasks (see lines 47 to 53) move data between an Object Storage and a HDFS. The `objectStore` parameter points to a service instance of support Object Storage service. The `cluster` parameter points to a `mapreduce` component. The `source` and `target` parameters specify either a local path on the HDFS of the cluster or a URL for the object storage. Currently OpenStack SWIFT is the only supported protocol.

`run_hadoop` tasks (see lines 55 to 61) start a Hadoop job that is declared in the `mapreduce` section. The `application` parameter points to the Hadoop component. The parameters in the nested `parameters` attribute are passed to the Hadoop job when it is invoked.

`workflow` tasks (see lines 63 to 67) allow a solution developer to perform simple orchestration. The `definition` parameter declares a simple workflow composed from other tasks. In the current prototype, only `sequence` is implemented, which executes a list of tasks sequentially.

For all the types above, when specifying parameter values, a solution developer can use assembly-time variables (`#{var}` notation) which are resolved at deployment time and run-time variables (`${var}` notation) which will obtain they actual values during invocations at runtime.

Finally, the attribute `trigger` specifies how a task is invoked at runtime.

`http` tasks are triggered by an HTTP POST invocation to a URL. The URL value is made available at assembly time to other components as `#{task_name.url}` (see line 8 for an example). The client can optionally pass a JSON map object in the POST body, which the task manager uses to extract run-time parameters.

`cron` tasks run autonomously according to a schedule, which is specified using the same format as UNIX `crontab`.

The task manager platform service receives the task definitions from the deployer plugin. For all `http`-triggered tasks, it listens at the URL and starts the task when HTTP POST is received. For all `cron`-triggered tasks, it schedules appropriate timers and initiates the task when the timer expires.

3.2.2 Notification

Notification service is used by a number of Solution Foundry components to communicate status messages asynchronously. It is essentially a wrapper around Apache Zookeeper and provides a REST API for updating and retrieving statuses. Within the context of analytics solutions, the Task Manager uses Notification service to communicate task progress status back to clients.

When an `http`-triggered task is invoked (by HTTP POST) the task manager creates a key in the Notification service and returns the Notification service REST URL endpoint for the key in the HTTP response. The client can then invoke the Notification service API using this endpoint to query the current status of the task, including its current progress, error state, etc.

3.2.3 Object storage

Object storage is commonly used in analytics solutions as a permanent storage of data while HDFS acting as a temporary storage when analyses are performed on the data. Object storage is also a natural bridging mechanism to connect disparate application components together. For example, a transactional web application may produce records and store them in an object storage. Periodically or

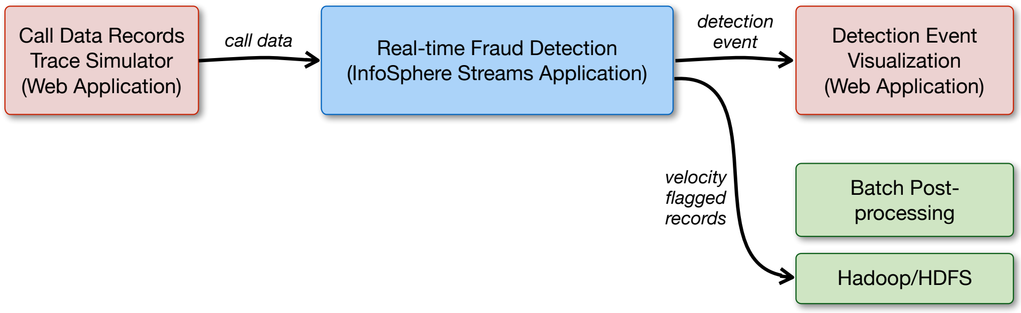
Call Data Records Trace Simulator (Web Application)
call data
Real-time Fraud Detection (InfoSphere Streams Application)
detection event
Detection Event Visualization (Web Application)
velocity flagged records
Batch Post-processing
Hadoop/HDFS

Figure 4: High level solution structure of Telco Fraud Detection

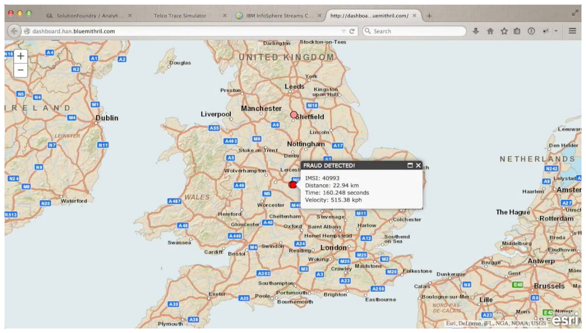

Figure 5: Visualization UI of the Telco Fraud Detection solution

on an on-demand basis, these records are transferred to the HDFS of a Hadoop cluster so that analytics can be perform on them.

We have implemented a Cloud Foundry service broker that provides OpenStack SWIFT object storage as a service (see lines 30 to 32 in Figure 3). The service broker obtains the actual object storages from SoftLayer. It manages multi-tenancy and enforces isolation through a proxy that implements the initial SWIFT authentication protocol. The clients to the object storage service are given randomly generated access API keys. The proxy authenticates the client and returns the temporary authentication token during the initial logon phase, while the service broker manages the actual SoftLayer API key for accessing the object storage. This allows any SWIFT compatible clients to run as-is and be isolated from each other.

As part of the platform services, this SoftLayer Object Storage service is understood and used by the Task Manager service implicitly for `data_transfer` tasks.

4. EARLY EXPERIENCE

We have validated the design of our proposed system with a Telco fraud detection solution that we introduced briefly in Section 1. Here we explain the solution in more details and present our early experience in deploying the solution on our prototype system.

4.1 Telco fraud detection solution

We use the velocity based alert generation scenario from a Telco Fraud detection solution suite as a test case for our PaaS for composite analytics solution. Because we do not have access to real live call data records (CDR) from the actual Telco provider, we use a simulator to replay a recorded and annonymized CDR stream as the input.

Since this paper is concerned with the assembly and deployment of composite analytics solutions, but not the actual logic inside these components, we will not delve into the details of the fraud detection solution itself, but rather focus on the components' input/output behavior that is relevant to solution assembly. Figure 4 shows the high level structure of this solution. There are four main solution components, each comes with a deployable solution artifact (binary).

CDR trace generator is a Java-based web application that runs on Apache Tomcat web container. It's packaged as a standard WAR (Web Archive) file. At runtime, it has a simple user interface that allows us to set simulation parameters, such as, start time of the trace, the speed of replay. A trace file contains pre-recorded and anonymized call data records. Each record contains, among several other fields, the mobile subscriber identifier number, the current timestamp, and the ID of the associated cell tower. The application sends the records to a TCP listener running on a remote host. The host IP and port are two integration parameters that the trace generator accepts. The host is a required parameter while the port number is optional and defaults to a pre-set value.

Real-time fraud detection is a continuous stream processing application that runs on IBM InfoSphere Streams. The main solution artifact is a program in SPLMM format. The program and additional dependency libraries are packaged into a zip file.

The streaming processing application listens on a TCP port for incoming CDR stream. The apparent movement velocity of a mobile device is computed from the changing association of the device and the stationary cell towers whose geographic positions are known a priori. A call record is flagged as suspicious if the calculated velocity is above certain threshold. These flagged records are stored into HDFS of a Hadoop cluster where secondary batch post-processing will happen. The required integration parameters are the URI and username for the HDFS and the output filename in HDFS. More concise version of these flagged records are also sent as detection events via HTTP POST method to a visualization web application, whose HTTP URL must be known to the streams processing application as a parameter.

Real-time fraud detection is a MapReduce application packaged into InfoSphere BigInsights application zip format. The MapReduce job shall be executed periodically.

Real-time fraud detection is another Java-based web application packaged in WAR format. It has a servlet that responses to HTTP POST method to receive alert events. At runtime, it uses the ESRI GIS library to present a map and renders to most recent alert events on the map with relevant information such as the IMSI, time of detection (see Figure 5).

We capture the above solution composition using the proposed solution descriptor YAML format (see Figure 6). With the description of the analytics-related features of the descriptor language in Section 3, most content of this example is self-explanatory. Here

```
 1  ---
 2  name: telco-fraud-detection
 3  domain: telco.bluemithril.com
 4
 5  endpoints:
 6  - host: dashboard
 7    target: fraud-viz
 8  - host: simulator
 9    target: cdr-sim
10
11  webapps:
12  - name: fraud-viz
13    package: bin/FraudVisualizer.war
14    instances: 1
15    memory: 256
16
17  - name: cdr-sim
18    package: bin/CdrSimulator.war
19    instances: 1
20    memory: 256
21    environment:
22      listenerHost: "#{fraud-analytics.ip}"
23
24  mapreduce:
25  - name: post-processing
26    package: bin/post-processing.zip
27    buildpack: mapreduce
28    cluster:
29      type: dedicated
30      size: 2
31
32  streams:
33  - name: fraud-analytics
34    package: bin/fraud-analytics.zip
35    composite: com.ibm.research.st.cdr::Main
36    parameters:
37      hdfsUri: "hdfs://#{post-processing.privateIp}:9000"
38      hdfsUser: "#{post-processing.username}"
39      out: velocityFlaggedRecords.txt
40      hist: histogram.txt
41      vizHost: "#{fraud-viz.url}"
42    cluster:
43      type: dedicated
44      size: 1
```

Figure 6: Deployment descriptor for the Telco Fraud Detection solution

is a brief explanation of the features that we have not explicitly introduced.

- The `name` attribute (line 2) specifies the solution name, which is used to identify the solution during its lifecycle management.

- The `domain` attribute (line 3) specifies the domain name of the externally accessible web components.

- Each `endpoint` section (line 5) specifies an externally accessible endpoint of the solution. The `host` attribute (lines 6 and 8) specifies the hostname that is used in conjunction with the domain name to form the URL. The `target` attribute (lines 7 and 9) points to the web application that this URL maps to. For example, when deployed, the visualization dashboard (Figure 5) is reachable at the URL `http://dashboard.telco.bluemithril.com`.

4.2 Experience and qualitative evaluation

As we are still in the process of building the complete platform and there is really not a similar system available that has similar functionality, we defer a more thorough and quantitative evaluation to later publications. Here we present our first hand experience with our early prototype system and give a qualitative evaluation of its benefits.

1. The first thing we notice is that with the solution descriptor, all solution components are made equal and first class in the composition. We no longer have to embed analytics as secondary binaries that a master web application has to manage. This separates the assembly/composition aspect of the solution from the internal logic of any of the individual components.

2. The assembly/composition of the components are fully captured as code (the YAML descriptor) and can thus be easily source controlled and versioned, thus enabling DevOps integration for the entire solution.

3. The high level solution descriptor enables solution architect to declare *what* to do but without having to worry about or even understand *how* to do it. Case in point is the deployment of application packages to the target runtimes: BigInsights has a REST API whereas Streams does not. This level of details is completely hidden by the plugins. When the underlying products/services evolve with new features, the service providers just need to update the plugins and the high level solution descriptor will remain stable.

4. When we deploy the solution, we notice the ease and speed of deployment. On our prototype system, it takes about 10 minutes for the aforementioned solution to be deployed from scratch and become available to use by end users. This would be impossible to achieve if we were to use IaaS directly or with individual PaaS'es for the component types. For solution architects who want to rapidly prototype a complex analytics solution, this represents a significant productivity improvement.

5. With Solution Foundry, we can manage the entire solution as a single logical entity. We can start, stop, query status of and/or update a solution atomically. This reduces the operational cost of running analytics solutions.

5. RELATED WORK

Cloud Foundry [2] is an open source cloud computing platform as a service (PaaS). It allows developers to build, deploy, test and scale web applications without the burden of installing and operating the required software stack. It also allows applications to connect, or bind, to support services, such as databases, messaging middleware, etc. Cloud Foundry applications can be easily deployed via a manifest file [3] that includes details such as how many instances to create, how much memory to allocate and what services the application needs.

In this paper we introduce a PaaS for composite applications in which Cloud Foundry is a key element. It provides the foundation for one of the application component types that can comprise a composite application. And it is driven by the composite application deployment orchestrator to deploy web app components. On the other hand, one the main points of our paper is to extend the reach of Cloud Foundry-type of PaaS to include composite applications that comprise other types of components, such as analytics applications.

Genie [4, 5] provides a Job and Resource management ecosystem in the Cloud. From the perspective of the end-user, Genie abstracts away the physical details of various (potentially transient) resources in the Cloud (like YARN, Spark, Mesos clusters etc.), and provides a REST-ful Execution Service to submit and monitor jobs to these clusters, without having to install any clients themselves. And from the perspective of an administrator, Genie provides a set of Configuration Services, which serve as a registry for these various clusters, and the configurations of the various commands that can be run on these clusters. Genie is a set of REST-ful services for job and resource management in the Hadoop ecosystem. Two key services are the Execution Service, which provides a RESTful API to submit and manage Hadoop, Hive and Pig jobs, and the Configuration Service, which is a repository of available Hadoop resources, along with the metadata required to connect to and run jobs on these resources.

From our point of view, Genie plays a similar role for analytics jobs as Cloud Foundry does for web applications. And so it is conceivable that our PaaS for composite applications could also build on Genie the way it does on Cloud Foundry. However, in the particular case of analytics application components, we have also provided a simple way data source, processing and sink flows that blend well with other application component types, such as web applications.

Apache Spark [1, 10] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Other related work includes [9], which describes SaaS applications as being composites and requiring heterogeneous runtimes. Although the authors do not give examples of a SaaS application descriptor, they mention that their multi-cloud infrastructure uses SCA both for the definition of the services provided by the federated PaaS layer and for the services of the SaaS applications that run on top of this PaaS, and talk about SCA terms such as components, services, references, properties, links, etc. Thus their Solution description is essentially an SCA descriptor, and things like cross-configuration, etc. could be expressed just as well in their descriptor. On the other hand, the solution description provided in our paper is not based on SCA and exposes certain SaaS/PaaS level attributes that are different from what's covered in this paper. Also, this paper does not concern itself with issues pertaining to DevOps.

Also, in [7], the authors describe deployment and management of distributed applications spanning multiple PaaS. Deployment performs an analysis of the Cloud based application's requirements to build a specific application deployment descriptor. Since no examples given the reader can only guess that the types of information collected in the descriptor could be different. In addition to deployment, this paper talks about matchmaking to find the most suitable PaaS offerings for running an application, semantics of PaaS selection, automatically generating an application deployment descriptor based on the application's requirements and preferences, monitoring and SLA management.

Finally, in [8], the authors present a new solution to the Big Data problem, that is, how to handle the explosion of heterogeneous information that has to be stored and processed to handle functions such as monitoring across clouds. The paper does not deal with solutions such as ours which is more than just a composition of data. There is also no mention of deployment or dev ops. Thus, although related, this paper is distinct from our work.

6. CONCLUSIONS

Enterprises are increasingly turning to sophisticated Big Data analytics solutions running on the cloud to derive insights from the vast amount of data that are available today. Leading adopters of these technologies are taking advantage of multiple analytics programming models and creating composite analytics solutions and integrating them with existing transitional enterprise systems.

We argue that, in order to effectively and efficiently manage the lifecycle of these composite analytics solutions including the DevOps aspect, there needs to be a high-level declarative language for describing the composition and a Platform-as-a-Service cloud that treats the entire composition as a single logic entity.

To this end, we present the design and a prototype system which includes a YAML-base declarative solution descriptor that extends the Cloud Foundry manifest and an extensible deployer and composer. We have implemented two plugins for the language and deployer which support MapReduce on IBM InfoSphere BigInsights and stream processing on IBM InfoSphere Streams. We have successfully deployed a Telco Fraud Detection solution that integrates web application, stream processing and MapReduce on the prototype system. Our early experience indicates that the system achieves the goals that we have set.

Acknowledgement

The authors would like to thank Kailash Marthi, James van Oosten, Raghu Ganti, Mudhakar Srivatsa and Mark Hulber for their support throughout the project.

7. REFERENCES

[1] Apache Spark. `https://spark.apache.org/`.

[2] Cloud Foundry Community. `http://www.cloudfoundry.org/index.html`.

[3] Deploying with Application Manifests. `http://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html`.

[4] Genie wiki. `https://github.com/Netflix/genie/wiki`.

[5] Hadoop Platform as a Service in the Cloud . `http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html`.

[6] P. Austel, H. Chen, T. Mikalsen, I. Rouvellou, U. Sharma, I. Silva-Lepe, and R. Subramanian. Solution Foundry: A Platform for Composite Applications. IBM Research Technical Paper. `http://domino.watson.ibm.com/library/CyberDig.nsf/home`.

[7] F. D'Andria, S. Bocconi, J. G. Cruz, J. Ahtes, and D. Zeginis. Cloud4soa: Multi-cloud application management across paas offerings. In *SYNASC*, pages 407–414. IEEE Computer Society, 2012.

[8] M. Fazio, A. Celesti, M. Villari, and A. Puliafito". The need of a hybrid storage approach for iot in paas cloud federation. In *28th International Conference on Advanced Information Networking and Applications Workshops*, pages 779–784, 2014.

[9] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier. A federated multi-cloud paas infrastructure. In R. Chang, editor, *IEEE CLOUD*, pages 392–399. IEEE, 2012.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.