Model-driven Deployment and Management of Workflows on Analytics Frameworks

Merlijn Sebrechts, Sander Borny, Thomas Vanhove
Gregory Van Seghbroeck, Tim Wauters, Bruno Volckaert and Filip De Turck
Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium
Email: merlijn.sebrechts@ugent.be

***Abstract*—The data science skills shortage means that those who have the knowledge are under constant pressure to do more with less. While the data science tools are improving at a staggering pace, the operational tools around them can not keep up. Even researchers at Google state that the issue of automatic configuration and dependency management of services is still an ”open, hard problem”. This manifests itself in data scientists either constantly having to solve operational challenges or having to be in constant close collaboration with a skilled operations team. This paper addresses the operational challenges behind deploying and managing workflows on top of analytics platforms by starting from three key requirements: data scientists want to model their workflows in a reusable way, this model should be automatically deployed, managed and connected to other services, and this solution should be compatible with existing cloud modeling languages, infrastructure, analytics platforms and tools. The paper explores where the state-of-the-art falls short in meeting these requirements, proposes an architecture to solve the open challenges, and implements and evaluates this architecture.**

I. Introduction

The amount of digital data is growing at an enormous pace. Inside this ever-growing jungle of digital data lies the key to valuable insight that can help both industry players and society as a whole. Albeit turning the raw data into useful insight is still a big challenge, and the acute data skills shortage[1][2] is not helping. 40 percent of companies are struggling to find and retain data analytics talent according to the 2015 MIT Sloan Management Review. Due to this, companies are constantly on the lookout for innovations that empower their data scientists to do more with less manpower.

There is an ever-growing set of tools and techniques that support data scientists, from massively scalable datastores to new programming concepts and analytics frameworks. While the software frameworks are evolving rapidly, the operational tools around them cannot keep up. As a result of this, data science is not only about analytics, there are also big challenges related to the deployment and management of analytics frameworks. These challenges creep up into the daily life of data scientists when they want to test and run their algorithms. Data scientists that have the operational skills necessary to solve these issues lose a lot of time and data scientists without these skills either cannot use some of the cutting-edge analytics frameworks or need constant support of a strong operations team. Even though this challenge is so pressing, the issue of automatic configuration and dependency management of services is still an ”open, hard problem” according to researchers at Google[3]. Solving these operational challenges allows the data scientists to achieve more with less time, and helps combat the skills shortage by empowering people with an analytic skill-set that do not have operational knowledge.

Further in this Section, the *City of Things* project is used to identify three challenges present in current data analytics solutions. Section II takes a deep-dive into where the state-of-the-art falls short. A novel architecture that addresses these shortcomings is proposed in Section III. The concepts of this architecture are mapped to state-of-the-art cloud modeling languages in SectionIV. Section V explains some fundamentals about the preexisting City of Things setup and its challenges. The proposed architecture is implemented to solve those challenges in Section VI, and thoroughly evaluated in Section VII. Finally, Section VIII concludes this paper.

The *City of Things* project at IMEC[1] aims to transform the city of Antwerp, Belgium into a smart city. Sensors all over the city collect data and send it to an analytics platform. This platform exposes the data in a standardized format over a REST API. The sensor data entering the platform is encoded in a number of different ways, depending on the sensor and the transfer technology. This data needs to be decoded and standardized before it is stored. The setup shown in Figure 1 solves this need by using the Apache Storm[2] streaming framework as an ETL tool. WSO2 ESB[3] provides an endpoint API for sensors to send data to and puts that data on the Apache Kafka[4] message broker. Apache Storm then extracts the data from Kafka, decodes it and loads it into MongoDB.

The challenges addressed in this paper manifest themselves around the ETL workflow running on top of Storm. This workflow has to be updated each time the decoding format of a sensor changes, or a new sensor is added, which happens frequently in a project that is constantly

[1]https://www.iminds.be/en/succeed-with-digital-research/go-to-market-testing/city-of-things
[2]https://storm.apache.org/
[3]http://wso2.com/products/enterprise-service-bus/

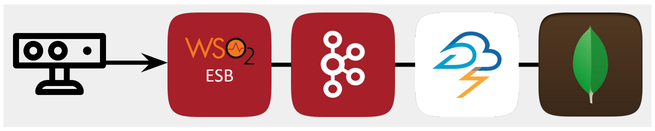

Figure 1. The sensors send data to a WSO2 ESB endpoint that puts the data on Kafka. Storm extracts the data from Kafka, decodes it and loads it into MongoDB.

testing cutting edge sensor devices, network technologies and protocols. Updating a workflow running on top of Storm is no easy task, and connecting that workflow to other services in the setup requires error-prone manual configuration and a lot of knowledge about the operational details of the running setup as further explained in Section V-A. Moreover, the flexibility of the setup is greatly hindered by the need for manual configuration. The challenges of the *City of Things* project are synthesized in the following three general requirements that data analytics setups have but the current state-of-the-art tools cannot fulfill.

a) Data scientists want to easily write workflows that interface with other services. They want to easily reuse parts of the workflow code in other deployments.
b) Data scientists want to deploy these workflows and connect the workflows to other services without the need for operational knowledge and skill-sets. Consequently, when the operational details of the underlying cluster or connected services change, the setup reconfigures automatically.
c) Projects using this solution do not want to lose the existing investment and encapsulated knowledge of their setup. Therefore, the solution has to be compatible with existing cloud modeling languages, their orchestrators and artifacts created with and for these languages.

II. Related work

A number of solutions have been proposed to model and manage workflows running in clouds. One approach has been to create application-specific modeling languages such as StormGen[5] and modeling tools such as StreamFlow[4]. StreamFlow's drag and drop topology builder enables non-developers such as data scientists and analysts to create processing workflows. There are two drawbacks to the application-specific approaches. It is not possible to model a workflow that spans over multiple applications and the integration between the workflow and other services requires manual configuration. StreamFlow for example has the ability to create workflows that interact with a number of different data sources and storage services, but the connection parameters of that storage provider have to be manually configured. This means that StreamFlow users need the help of a system administrator each time they want to use a new data source or a storage system. System administrators also need to update the StreamFlow configuration when the infrastructure changes. The drawback of these tools is addressed in a new generation of high-level dataflow modeling tools such as Apache NiFi[5] and StreamSets[6]. These languages can be used to model workflows that span over a number of different data services such as Apache Hadoop HDFS[7] and MongoDB[8]. However, these tools also have the issue that they require extensive manual configuration for each external data service. Having to contact the operations team for each new datasource and datastore limits the power of the users of these tools.

A second approach builds on existing cloud modeling languages to model and manage both the underlying applications and the workflows on top of them. Qasha et al. [6] propose the cloud modeling language OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)[7] as a way to describe both the internal topology and deployment process of a scientific workflow. This approach shows much potential to improve the portability, automatic deployment and scalability of these workflows. The proposed solution, however, does not address the issue of modeling workflows that run on top of existing analytics frameworks and is highly specific for the TOSCA cloud modeling language. Juju[9] and CloudML[8] are two cloud modeling languages with runtime model orchestrators that tackle the challenge of creating vendor- and cloud-independent models of cloud applications. These languages, however, do not address the challenge of deploying workflows on analytics platforms.

Guerriero et al. [9] propose an architecture supporting model-driven Big Data design that uses the DICE models as an abstraction layer on top of existing modeling languages. That approach allows for deploying and managing both the workflows and the frameworks by translating the DICE models into more technology-specific models such as TOSCA. The extra abstraction layer introduced in their approach, however, requires complex transformations from DICE models to technology-specific models and requires a new metamodel for each data analytics framework. Rather than creating new abstractions and meta-models we propose a solution for the challenges described in the introduction that builds on top of the existing cloud modeling languages.

Going back to the requirements discussed in the introduction, the state of the art has three deficiencies that are addressed in this paper.

a) Existing solutions do not solve the challenge of modeling and deploying workflows that run on top of existing analytics frameworks.

[4]https://github.com/lmco/streamflow/wiki

[5]https://nifi.apache.org/
[6]https://streamsets.com/
[7]https://hadoop.apache.org/
[8]https://www.mongodb.com/
[9]https://jujucharms.com/

b) Application-specific modeling languages and tools require manual configuration of operational parameters. When the operational details of the running setup change, these values have to be updated manually.

c) Current solutions are either language specific, or require development of a whole new set of models and tools. There is a lot of investment in existing cloud modeling languages. A solution that starts from scratch loses the knowledge encapsulated in existing languages, tools, models and artifacts.

III. Architecture

This section proposes an architecture to deploy workflows on analytics frameworks using a cloud modeling language. This architecture bases itself on the concepts introduced in the authors' previous work[10] namely the *service agent* and the concept of *peer relationships*. These two concepts are defined as follows.

Definition 1 *A **service agent** represents and manages a single service.*

Definition 2 *A **peer relationship** connects two service agents and allows them to exchange information.*

A service agent manages a service such as a MySQL database. It reasons about the current state and executes actions to setup, configure and manage that service. Service agents collaborate using peer relationships such as the relationship between a service agent managing a MySQL database and one managing a webserver that uses the MySQL database as backend. The peer relationship is used: to solve service dependencies, such as the dependency that the webserver can only start after the MySQL database is online; to exchange operational details, such as IP addresses and port numbers; and to exchange requests, such as the webserver requesting the creation of a user account. The service agent can use configuration management tools such as Chef[10] and Puppet[11] for the actual fulfillment of these requests and configuration of the service.

This paper introduces two new architectural concepts shown in Figure 2 and defined as follows.

Definition 3 *A **workflow component** represents and manages one step in the workflow.*

Definition 4 *A **deployer** assembles the workflow and deploys it onto the data analytics framework.*

The workflow component contains the actual workflow step and reasons about that workflow step and its relations to other steps and services. Peer relationships between workflow components represent how the data flows from one step to the next. A peer relationship between a workflow component and another service is used to exchange

[10]https://www.chef.io/
[11]https://puppet.com/

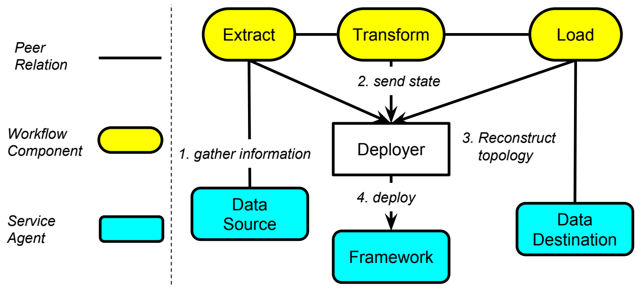

Figure 2. The deployer component contains the logic to assemble the workflow components and deploy the resulting workflow onto the analytics framework. Workflow components connect to other services to get configuration values such as port numbers and IP addresses.

configuration information such as the IP address and the port number of the service.

In cloud modeling languages such as TOSCA and Juju, a service agent does not have access to the full model. Instead, it only has access to the information about itself and its relations. This isolation of scope makes sure that all dependencies between service agents are explicitly defined as a relationship. A service agent cannot have an implicit dependency on the state of another service agent since it has no way to know what that state is. Just as in programming languages, isolation of scope reduces complexity and creates reusable components. In this architecture, however, the deployer has to know about the state of the workflow components in order to deploy them properly. Giving the deployer full access to the model means losing the benefits of scope isolation. To have the best of both worlds, the deployer is connected to each workflow component using a peer relationship. The workflow components send their state, including their relations to each other, to the deployer. This approach has two advantages: the deployer only has knowledge about the scope of the connected workflow components, and the model clearly reflects what components the deployer knows about.

After receiving the state of each individual workflow component, the deployer reconstructs the entire workflow topology by combining the state of each component. At this stage, the deployer can decide whether or not to deploy the workflow by checking if it is valid. The method of deployment is also entirely up to the deployer itself. The deployer can reason about the most optimal way to deploy the workflow, and use that method. Every time the cloud model changes, the changes get propagated from the workflow components to the deployer. The deployer then decides how this change should be propagated to the running topology. Simple deployers might stop and redeploy the workflow each time it changes. More complex deployers might be able to change a running topology on the fly.

Figure 2 shows an example of how this architecture can be used to model an ETL pipeline. The workflow components connect to a datasource and a data destination using a peer relationship. This relationship is used to exchange configuration information between the service and the workflow components. All workflow components connect to a deployer that receives the information about each workflow step, assembles the complete workflow, and deploys it onto the framework.

The workflow components and the deployer created for this architecture successfully address the shortcomings of the state of the art discussed in Section II.

a) The usage of workflow components makes it possible to model the workflow as a set of interconnected steps. Each step is an isolated entity, making it possible to recombine the steps into a new workflow, and interchange one step for another one.
b) Workflow components can have peer relationships to other services, and reason about those relationships. This gives them the ability to gather operational information about the service, and configure the workflow step using that information. The deployer uses its peer relationship to the framework to gather operational information about the framework, and uses that information to change the deployment process accordingly. The deployer also reasons about whether and how to update the running topology when either the workflow or operational details change.
c) All the knowledge about how to deploy and manage a workflow is contained in the deployer. As a consequence, this architecture does not require any changes either in cloud modeling or their orchestrators. Moreover, because the deployer poses as a proxy between the workflow and the framework, the artifact managing the framework also does not require any changes, making the adoption of this architecture easier and limiting the complexity of the artifact managing the framework. The deployer also has the advantage that it hides the operational complexity of the framework from the workflow: the workflow components connect to a single deployer regardless of whether the underlying framework is a single service or a cluster of services working together.

IV. Mapping

This section elaborates on how the concepts of this architecture can be mapped to concepts in popular cloud modeling languages. OASIS TOSCA and Ubuntu Juju are studied more in-depth because they are the state of the art in cloud modeling languages and gather interest both in academia and in the industry.

The two new concepts created for this architecture can be seen as specialized versions of the concepts explained in the authors' past work[10].

- The **workflow component** is a service agent that represents a step in a workflow and manages the configuration of that step. The workflow component reconfigures the workflow step in the same way as a service agent reconfigures a service.
- The **deployer** is a service agent that gathers information from all workflow components, assembles the entire workflow and deploys it onto the framework.

Table I
Mapping of architecture concepts to concepts in popular cloud modeling languages

Term	**TOSCA**	**Juju**
Service agent	Node template	Charm
Peer relation	Dependency	Relation
Workflow component	*Node template*	*Charm*
Deployer	*Node template*	*Charm*

This leads to the conclusion that *this architecture can be implemented in any cloud modeling language that has concepts that map to a service agent and a peer relationship*. Table I shows the mapping of these concepts to concepts in two popular cloud modeling languages, TOSCA and Juju. The following two sections will elaborate on this mapping.

A. OASIS TOSCA

The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) is a cloud modeling language and orchestrator specification. The service agent concept maps well to TOSCA's node templates since a node template represents a single service and contains the logic to deploy and configure that service. The TOSCA specification is highly focused around managing services using lifecycles. Each node template has a lifecycle with operations such as start, stop and configure. Each operation brings the service into a new state. Previous work[10] discussed the pain points of lifecycle-based management of services and proposed that service agents include an event-based service engine to reduce complexity and increase flexibility. It is possible to use TOSCA's lifecycle to drive a service engine by interpreting each lifecycle operation as an event instead of a transition from one state to another.

Node templates can specify which other node templates they need using a *requirement*. A node template specifies what requirements it can fulfill using a *capability*. Requirements and capabilities of the same type can be connected by a dependency. This concept maps to the first part of the peer relationship definition (1): *"A peer relationship connects two service agents"*.

Node templates are able to query information from another node that is connected using a dependency. For example a web server node template can query the IP address of the connected database node template. This information can be properties such as IP addresses or attributes. Attributes are special in that they can be changed at run-time from an operation. An example of this is a database password that is automatically generated during configuration of the database node template. It is possible to run an operation when an attribute changes by using the dependency lifecycle or more specifically, the *target_changed* operation. This means that attributes can be used to implement the second part of the peer relationship definition: ”A peer relationship allows two service agents to exchange information”. Communication between two node templates is made possible by connecting the target_changed operation to the service engine and using attributes as communication medium.

B. Ubuntu Juju

Ubuntu Juju is a cloud modeling language and orchestrator developed by Canonical[12]. The Service Agent concept maps very well to a Juju Charm. Just like with TOSCA, Juju Charms operate using a lifecycle. Juju specifies a number of lifecycle actions called *hooks* that execute code to move the service from one state to another. An event-based service agent can be implemented using a service engine that interprets each hook as an event. Peer relationships map to normal relationships in Juju since Juju relationships can be used to send information from one Charm to another. Each change in this relationship information triggers the *config-changed* hook. The service engine can interpret this hook as an event.

V. Background about City of Things and related technologies

A. Technology background

Apache Storm is a distributed streaming analytics framework. It is focused on high-bandwidth and low-latency analysis on continuous streams of data. Datastreams start in a *Spout*. The Spout pulls the stream from a messaging broker, or generates the stream itself. Data processing is done in Storm *Bolts*. A Bolt takes a number of input streams, processes them, and emits the resulting output streams. A Bolt can have a number of tasks that can process multiple streams in parallel. *Stream Groupings* define which streams a Bolt receives as input by connecting the output stream of a Bolt or Spout to the input of a Bolt. Stream grouping also define how streams are partitioned over a bolt’s tasks. A Bolt can also choose to send a stream to an external service such as a datastore. The combination of Bolts and Spouts connected by Stream Groupings is called a *Storm Topology* as shown in Figure 3.

This Storm topology is deployed onto the Storm Cluster. A Storm Cluster consists of a master called *Nimbus*, a set of workers and an Apache Zookeeper[13] cluster that Storm uses for coordination. New topologies are submitted to the nimbus, who then distributes the Bolts and Spouts over the workers.

Apache Storm is a very powerful tool for stream analysis, making it popular for low-latency real-time processing of

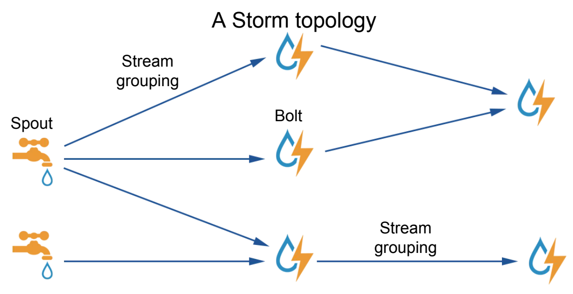

Figure 3. Spouts pulls data from an external datasource and send it to bolts for processing. The bolts at the end of the topology either send the data to an external datastore or discard the data. Stream groupings are used to send the data from spouts and bolts to other bolts.

data streams, but it has some disadvantages that halter its adoption. Deploying and managing Apache Storm is a complex and time-consuming task, especially since a Storm cluster also requires a correctly configured Zookeeper cluster. It is common to use some form of automation to setup and configure such a solution. Cloud modeling languages are one of the appropriate tools used for this since they provide a great way to manage the complexity of such a deployment. Another issue with using Storm is that deploying and updating Storm topologies is not trivial. It requires copying the topology onto the Nimbus server, acquiring CLI access to the server and running a series of commands with a complex syntax. This process is complex, time-consuming, error-prone and requires CLI access to the server.

Storm provides a read-only web-interface that can be used to check the state of a running topology, but this web-interface does not reflect how that topology communicates with external services. As a streaming tool, Storm is just one link in a bigger analytics pipeline, although Storm’s UI only shows the Storm part of that pipeline. Neither the other components in that pipeline, nor Storm’s relations to those components are shown in this UI.

The options for code reuse in Storm are limited. Storm’s bolt and spout paradigm lends itself nicely to code sharing and reuse but there is no way to encapsulate a bolt or spout in a way that it can be reused without having to refactor code.

B. Preexisting City of Things setup and limitations

This section further elaborates on the preexisting setup of the City of Things back-end and its limitations. These limitations are addressed by implementing the architecture proposed in this paper.

The City of Things setup discussed in the introduction is deployed and managed using Juju. Juju was chosen for the reason that it has a battle-tested orchestrator, has a large community around it, and because all of the components required for this setup are readily available as Juju Charms. The Juju community maintains Charms for MongoDB and

[12]http://www.canonical.com/
[13]https://zookeeper.apache.org/

Apache Kafka. The Apache Kafka Charm is split up into two Charms that setup two different components: Apache Kafka itself and Apache Zookeeper. All three Charms implement a number of relationship interfaces that allow other Charms to connect to them by either requiring or providing the relationship. The Apache Zookeeper Charm provides the *zookeeper* interface. This interface allows clients to connect to and use Apache Zookeeper. The Apache Kafka charm is an example of a Charm consuming this interface. This interface is also very useful for the Apache Storm Charm, which was created as part of an earlier project of the authors' research group. Apache Storm also uses Apache Zookeeper as a coordination service and the *zookeeper* interface allowed us to reuse the Apache Zookeeper Charm for Apache Storm, significantly reducing the development effort of the Apache Storm Charm. Similarly, the WSO2 ESB Charm, also created as part of an earlier project, uses the *kafka-client* interface to setup a connection between WSO2 ESB and Apache Kafka. The WSO2 ESB Charm configures the ESB as an API endpoint that sends all messages it receives to Kafka.

As with all cloud modeling languages, Juju falls short in the relationship between Apache Kafka and Apache Storm, and the relationship between Apache Storm and MongoDB. These two relationships are not actually relationships between services. Apache Storm requires no knowledge about both Kafka and MongoDB. These relationships are actually between the workflow running on Storm and Kafka, and between the workflow running on Storm and MongoDB. Creating such a relationship in Juju is not possible since Juju only models the services and not the workflows running on top of those services. This has a couple of disadvantages that stem from the open challenges in the state of the art as described in Section II.

a) The workflow is a monolithic entity that is specific to the setup. Reusing parts of the workflow is not possible without manually re-factoring the code each time.
b) Deployment and configuration of the workflow has to be done manually. This requires CLI access to the Storm cluster, and is a manual, complicated and error-prone process. Because of this, the system cannot reconfigure itself when configuration details such as the IP addresses of Apache Kafka and MongoDB change. This has an adverse effect on the flexibility of all concerned services and the stability of the entire system.
c) Current state of the art solutions for these challenges are not compatible with the existing model, artifacts, and the currently running infrastructure.

Having an easy way to update the workflow is important for this scenario since the workflow has to be changed each time a new sensor is added or the encoding format of an existing sensor changes.

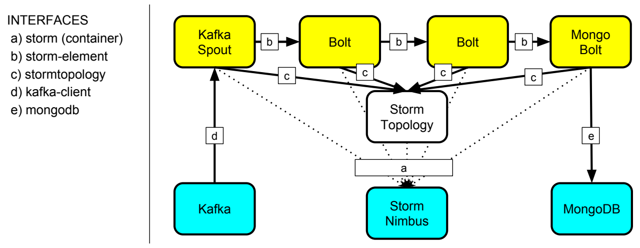

Figure 4. The implementation of the architecture in Juju. The labels on the relationships denote the relationship interface type. Interface types denote a common protocol. Only components that implement the same interface can be connected to each other. This makes sure that components connected to each other actually know how to talk to each other.

This is where the architecture proposed in this paper comes in. The existing model is extended with a workflow and a deployer. This makes it possible to automatically deploy and manage the workflows using the Juju orchestrator. Cloud modeling concepts such as isolation and relationships enable the workflow model to be modular and reusable. The cloud modeling language poses as an abstraction layer between the user and the actual running infrastructure. A user can update the workflow without having to be concerned about the operational details of the setup and without having to come in contact with servers where the cluster is actually running. The actual implementation of this solution is explained further in the next section

VI. Implementation of Architecture

As explained in Section V-B, the preexisting setup is deployed and managed using Juju. To enable continuity and not lose the investment and knowledge contained in the model and artifacts, we chose Juju as the language to implement this architecture in.

Figure 4 shows a model using the final implementation. The storm-topology Charm is an implementation of the deployer. It has a relationship to the Storm Charm. A number of Bolt and Spout Charms connect to both the Storm Topology and the Storm charm. These Bolt and Spout Charms are workflow components representing the workflow running on top of Storm. The Spout Charms can connect to a data source and the Bolt Charms can connect to a data store. The next Subsections go into greater detail about these Charms and their relationships. Figure 5 shows an ETL workflow modeled using this implementation in the Juju GUI.

A. Bolts and Spout Charms

The Bolt and Spout Charms are workflow components. The topology of the workflow is specified by a set of Bolts and Spouts connected with *storm-element* relationships. Each Bolt and Spout specifies the Java code of that specific step in the *class* configuration option, which points to a Java class file that contains all the code for that Bolt or Spout.

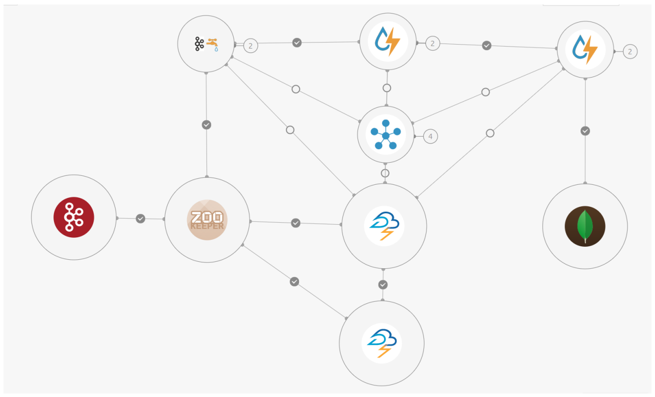

Figure 5. An ETL workflow running on Apache Storm shown in the Juju GUI. The workflow, the Storm cluster and the services are deployed using the cloud modeling framework Juju.

Bolts have a few additional configuration options. The *groupings* configuration option specifies how the incoming data should be distributed over the Bolt's tasks. Technically, this configuration is a property of the relationship of a Bolt, and not the Bolt itself. However, Juju does not support relationship properties at the moment, only Charm properties. Therefore each Bolt specifies the properties of its incoming relationships. Note that a Spout does not specify these properties since a Spout does not have incoming relationships. All Bolt and Spout Charms have a relationship to the Storm Topology Charm. This interface is used to send information about the Bolt's or Spout's relationships and the Java class to the Storm Topology Charm.

The classes in Bolt and Spout Charms can specify certain dependencies.

B. *Storm Topology Charm*

The Storm Topology Charm gathers information about the connected workflow and its code classes. The Charm rebuilds workflow as a Flux[14] topology. The Charm then evaluates whether the topology is deployable. A Storm topology without a Spout cannot be deployed. If the topology is deployable, the Charm compiles the classes it received from the Bolt and Spout Charms, and deploys the topology using the Storm command-line tools.

Because the Storm Topology Charm uses the Storm command-line tools, it has to have access to the Storm Nimbus node. For this reason, the Storm Topology Charm has a *subordinate* relationship with Nimbus. A subordinate relationship is like a *hosted-on* relationship in TOSCA. It denotes that both Charms are co-located.

The deployer gets notified each time one of the connected bolts and spouts change. The topology charm then rebuilds the topology, recompiles if a class has changed, and redeploys the topology. This redeploying of the topology does not present any problems to our setup because Apache Kafka acts as a buffer between the web API and the Storm topology.

Bolt and Spout Charms are also co-located with the Storm Topology Charm to facilitate moving large files from the Bolt and Spout Charms to the Storm Topology Charm. This co-location is done indirectly by making the Bolt and Spout Charms a subordinate of the Nimbus Charm. This indirect approach is due to a limitation in Juju: subordinate Charms cannot host other subordinate Charms.

VII. Evaluation

Based on the challenges this paper addresses, a number of evaluation parameters have been chosen to determine the quality of this solution. This section evaluates the level of code reuse that this architecture enables, the possibility of automatic reconfiguration of the workflow and the advantages of integrating with an existing deployment tool such as Storm Flux.

By separating a monolithic Storm topology jar into a set of self-configuring Bolt and Spout Charms, the possibility of code reuse is greatly increased. Take the use-case of the creation of another Smart Cities project in a new city. This city uses a new set of sensors that need their own decoding algorithms. The creators of these sensors are tasked to write the decoders that run in a low-latency highly distributed fashion on top of Storm. Table II compares two approaches. In the first approach these developers use a monolithic Storm topology JAR that includes both the ingestion from Apache Kafka, the decoding, and the storage into MongoDB. The developers use the implementation of the architecture created in this paper for the second approach, giving them the opportunity to reuse both the Kafka Spout, the MongoDB bolt, and the Generic Bolt Charms from the original City of Things project. For the second approach they write the decoder as a Storm Bolt Class and attach that class to the Generic Bolt Charm using the *class* config option.

Table II clearly shows the benefit of the code reuse that the implementation of the architecture created in this paper gives the developers. Only 59 lines of code have to be setup-specific in the Bolt Charm approach, compared to the 219 lines of code for the JAR approach. The communication to other components happens using Storm tuples, which means that developers writing setup-specific code only have to use the Storm Bolt API, instead of the complete Storm API together with the Apache Kafka and the MongoDB API. As a final note, the initialization and building of the topology is something that is handled by the Charms in the Storm Bolt

Table II
Benefits of code reuse in Lines of Code and APIs used.

	LoC	Files	APIs
Storm topology JAR	219	4	3
Bolt Charm	59	1	1

[14]https://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html

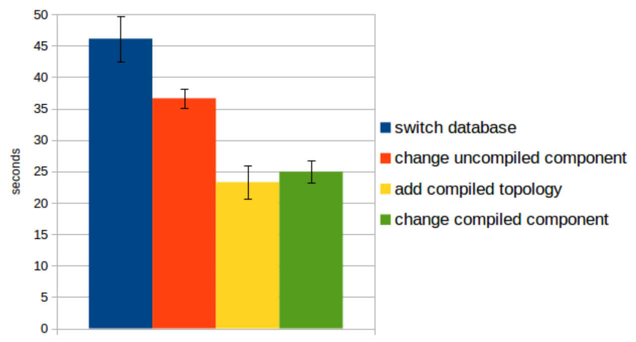

Figure 6. Analysis of time to change the deployed topology.

approach, while this is something that the developers have to do themselves in the Storm Topology JAR approach.

Seamless integration of the architecture with existing cloud modeling languages and tools, a feature that is lacking in the state-of-the-art, is shown to be very advantageous in Section VI since the architecture can seamlessly integrate with the existing cloud models, artifacts and tools used in the City of Things project. Moreover, the advantages of the possibilities of deep integration with analytics frameworks and deployment tools becomes clear when one looks at the time needed to change a running topology as shown in Figure 6. The benefits that the integration with the very useful but very specialized Storm Flux framework offer, are clearly visible. Storm Flux makes it possible to detect whether a Bolt or Spout has already been compiled, and compile only when it is necessary. The time needed to change the class of a Bolt into one that has already been compiled is significantly smaller than the time needed to update the topology if that class is not already compiled. The results also show that the time needed to switch the destination database of the workflow is close to twice the time needed to update the topology with a compiled class. This is because changing the database happens in two steps: first the connection with the old database is severed and then the connection with the new database is created, causing the workflow to be rebuild twice.

VIII. Conclusion

The architecture proposed, implemented and evaluated in this paper fulfills the requirements as stated in the introduction.

a) Code reuse is possible by providing a framework to divide a monolithic workflow into a series of steps connected using standardized interfaces.
b) The evaluation shows that the system can automatically reconfigure when underlying operational details are changed.
c) Having an architecture that allows for seamless integration with existing cloud modeling languages, analytics frameworks and deployment tools provides great benefits by reusing existing infrastructure and encapsulated knowledge, and using highly specialized but highly efficient tools to deploy workflows.

This architecture goes beyond the state of the art with its ability to model, deploy and manage workflows running on analytics frameworks and their connections to other services, and its seamless integration with both existing cloud modeling languages and existing deployment tools.

Acknowledgment

Part of this work has been funded by the iFest project, cofunded by imec and VLAIO.

References

[1] Gartner, *Gartner Survey Highlights Challenges to Hadoop Adoption*. [Online]. Available: https://www.gartner.com/newsroom/id/3051717

[2] TEKsystems, *Lowered Expectations for IT Budgets 2015 TEKsystems*. [Online]. Available: https://www.teksystems.com/en/resources/news-press/2014/teksystems-annual-it-forecast-2015?&year=2014

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," vol. 14, no. 1, pp. 70–93. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2898442.2898444

[4] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a distributed messaging system for log processing."

[5] S. Santurkar, A. Arora, and K. Chandrasekaran, "Stormgen - a domain specific language to create ad-hoc storm topologies," in *2014 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1621–1628.

[6] R. Qasha, J. Cala, and P. Watson, "Towards automated workflow deployment in the cloud using TOSCA," in *2015 IEEE 8th International Conference on Cloud Computing*, pp. 1037–1040.

[7] OASIS. TOSCA simple profile in YAML version 1.0. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.html

[8] N. Ferry, F. Chauvel, H. Song, and A. Solberg, "Continous deployment of multi-cloud systems," in *Proceedings of the 1st International Workshop on Quality-Aware DevOps*, ser. QUDOS 2015. ACM, pp. 27–28. [Online]. Available: http://doi.acm.org/10.1145/2804371.2804377

[9] M. Guerriero, S. Tajfar, D. A. Tamburri, and E. Di Nitto, "Towards a model-driven design tool for big data architectures," in *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*, ser. BIGDSE '16. ACM, pp. 37–43. [Online]. Available: http://doi.acm.org/10.1145/2896825.2896835

[10] M. Sebrechts, T. Vanhove, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Distributed service orchestration: Eventually consistent cloud operation and integration," in *proceedings of the 2016 IEEE International Conference on Mobile Services (MS 2016)*. IEEE.