CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015;
Published online in Wiley InterScience (www.interscience.wiley.com).
Pre-print version

JS4Cloud: Script-based Workflow Programming for Scalable Data Analysis on Cloud Platforms

Fabrizio Marozzo[§,*], Domenico Talia[§], Paolo Trunfio[§]

[§]*DIMES, University of Calabria, Rende, Cosenza, Italy*

SUMMARY

Workflows are an effective paradigm to model complex data analysis processes, e.g. Knowledge Discovery in Databases (KDD) applications, which can be efficiently executed on distributed computing systems such as a Cloud platform. Data analysis workflows can be designed through visual programming, which is a convenient design approach for high-level users. On the other hand, script-based workflows are a useful alternative to visual workflows, because they allow expert users to program complex applications more effectively. In order to provide Cloud users with an effective script-based data analysis workflow formalism, we designed the JS4Cloud language. The main benefits of JS4Cloud are: i) it extends the well-known JavaScript language while using only its basic functions (arrays, functions, loops); ii) it implements both a data-driven task parallelism that automatically spawns ready-to-run tasks to the Cloud resources and data parallelism through an array based formalism; iii) these two types of parallelism are exploited implicitly so that workflows can be programmed in a fully sequential way, which frees users from duties like work partitioning, synchronization and communication. We describe how JS4Cloud has been integrated within the Data Mining Cloud Framework (DMCF), a system supporting the scalable execution of data analysis workflows on Cloud platforms. In particular, we describe how data analysis workflows modelled as JS4Cloud scripts are processed by DMCF by exploiting parallelism to enable their scalable execution on Clouds. Finally, we present some data analysis workflows developed with JS4Cloud and the performance results obtained by executing such workflows on DMCF.
Copyright © 2015 John Wiley & Sons, Ltd.

KEY WORDS: JS4Cloud; Data analysis; Workflows; Cloud computing; Scalability

1. INTRODUCTION

Data analysis applications often are complex processes in which multiple data processing tools are executed in a coordinated way to analyze large datasets. The term Knowledge Discovery in Databases (KDD) is used to describe the data analysis process of extracting useful information from large datasets. Data analysis applications that use large datasets are often composed by many concurrent and compute-intensive tasks that can be efficiently executed only on scalable computing infrastructures, such as HPC systems, Grids and Cloud platforms. Among the different paradigms used to implement data analysis applications, workflows are very effective particularly when data analysis applications must be executed on such platforms. In fact, workflows provide an effective way for specifying the high-level logic of a complex application while hiding the low-level details that are not fundamental for application design, including platform-dependent execution details [22][23].

[§]E-mail: {fmarozzo,talia,trunfio}@dimes.unical.it
[*]Correspondence to: Fabrizio Marozzo, DIMES, University of Calabria, Via P. Bucci 41C, 87036 Rende, Cosenza, Italy

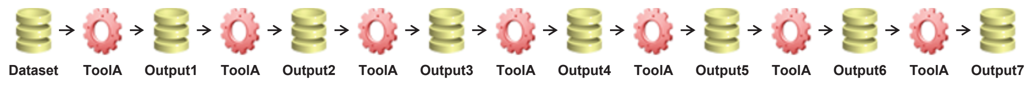
Data analysis workflows can be designed through visual programming, which is a convenient design approach for high-level users, e.g. domain-expert analysts having a limited understanding of programming. In addition, a graphical representation of workflows intrinsically captures parallelism at the task level, without the need to make parallelism explicit through control structures [13]. On the other hand, visual programming may not be very flexible since it usually offers a fixed set of visual patterns and limited ways to configure them. To address this issue, popular visual languages give users the opportunity to customize the behaviour of a pattern by adding a block of procedural code which may specify, for instance, the actions performed by that pattern when an event occurs. Another issue of complex workflows designed through visual programming is their size, which is often solved by visual workflow management systems splitting a big workflow into sub-workflows.

Script-based workflows are a useful alternative to visual workflows, because they allow users to program complex applications more rapidly, in a more concise way, and with greater flexibility. Figure 1 shows two workflow patterns that are defined in a more compact and easier way using a script-based formalism rather than a visual one. The pattern in Figure 1a can be found in workflows where a data processing tool must be applied multiple times to a given data source. For example, `ToolA` may be an image processing software including `N` different filters (e.g., to automatically adjust brightness, contrast, etc.), that can be applied independently of each other to an input image to produce a filtered output image. In this scenario, the workflow represents seven different filters that are applied sequentially to the original data (`Dataset`) to produce a final result `Output7`, with all the intermediate results (`Output1` ... `Output6`) remaining available for inspection or further use. The pattern in Figure 1b can be found in workflows where a data processing tool must be executed multiple times to analyze growing amounts of data. For example, `Dataset0` ... `Dataset9` may be partitions of a larger dataset, and `ToolB` may be a data mining algorithm producing a classification model (e.g., a decision tree) starting from one or more datasets. In this scenario, `Model0` is the classification model produced from the smallest amount of data, while `Model9` is produced using all the partitions. The models could be compared to find a good trade-off between input size and model accuracy. By comparing the visual and script-based versions of the two workflows, it is clear that in many cases visual workflows can be less compact and require a bigger design effort, in particular, when the number of tools/data is higher.

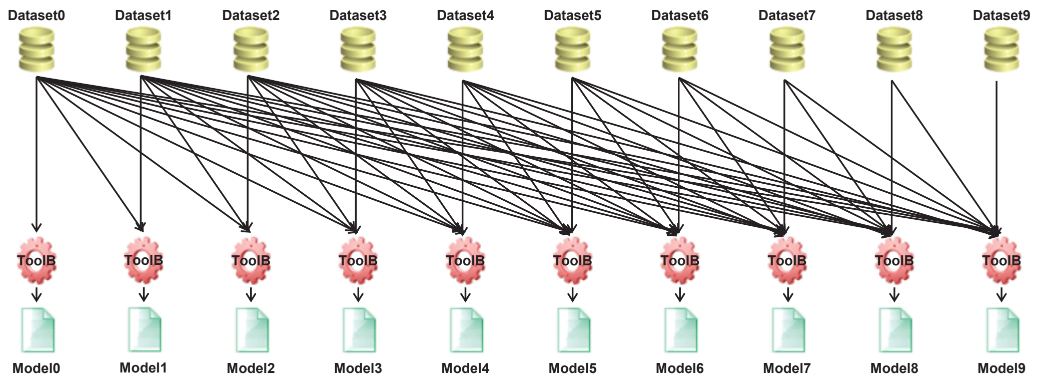
For providing Cloud users with an effective script-based data analysis workflow formalism without developing a new scripting language from scratch, we looked for: i) a scripting language that is easy to learn and use, and includes typical functionalities of high-level programming languages (control structures, loops, etc.); ii) a language that can be easily extended with additional functionalities; and iii) a language ready to be integrated into a Web-based application. JavaScript meets all those requirements. In fact, it is a lightweight programming language easy to learn and use and is natively integrated into browsing environments. In addition, despite having a syntax relatively similar to high-level languages like C, C++ and Java, it is an efficient scripting language.

Starting from JavaScript we designed JS4Cloud, a language for programming and executing data analysis workflows on the Cloud. The main benefits of JS4Cloud are: i) it extends the well-known JavaScript language while using only its basic functions (arrays, functions, loops); ii) it implements both a data-driven task parallelism that automatically spawns ready-to-run tasks to the Cloud resources and data parallelism through an array based formalism; iii) these two types of parallelism are exploited implicitly so that workflows can be programmed in a totally sequential way, which frees users from duties like work partitioning, synchronization and communication.

We present the language features and describe how JS4Cloud has been integrated within the *Data Mining Cloud Framework* (DMCF), a system supporting the scalable execution of data analysis workflows on Cloud platforms [17]. In DMCF, data analysis workflows can be designed through visual programming, using an ad hoc workflow language. Therefore, by integrating JS4Cloud within DMCF, we extended the latter to support also script-based data analysis workflows, as an additional and more flexible interface for users who prefer script-based programming. We discuss how data analysis workflows modelled as JS4Cloud scripts are processed by DMCF to make parallelism explicit and to enable their scalable execution on Clouds. In addition, we present some data analysis



a1)



b1)

```
var DRef = Data.get("Dataset");
var ftype = ["f1", "f2", ..., "f7"];
var ORef = Data.define("Output", ftype.length+1);
ORef[0] = DRef;
for(var i=0; i<ftype.length; i++)
 ToolA({input:ORef[i], filter:ftype[i],
    output:ORef[i+1]});
```

a2)

```
var DRef = Data.get(new RegExp("^Dataset"));
var M = DRef.length; // equal to 10
var MRef = Data.define("Model", M);
var tmp = [];
for(var i=0; i<M; i++){
 tmp.push(DRef[i]);
 ToolB({input:tmp, model:MRef[i]});}
```

b2)

Figure 1. Visual composition vs script-based programming: a) Pipeline workflow; b) Workflow where the input dataset size increases at each step.

workflows developed with JS4Cloud, and the performance results obtained by executing such workflows with DMCF on the Microsoft Azure platform.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 shortly presents the Data Mining Cloud Framework and its visual workflow formalism. Section 4 presents JS4Cloud and discusses how workflows programmed through this language are executed by DMCF. Section 5 describes data analysis applications developed with JS4Cloud and presents performance results obtained executing such applications with DMCF. Finally, Section 6 concludes the paper.

2. RELATED WORK

Several systems have been proposed to design workflows using script-based or visual formalisms [22], but only some of them currently work on the Cloud. In the following, we discuss the most representative workflow management systems that support either script-based or visual workflow design, which can be used in Cloud environments.

Pegasus [4], developed at the University of Southern California, includes a set of technologies to execute workflow-based applications in a number of different environments, including desktops, clusters and Grids. It has been used in several scientific areas including bioinformatics, astronomy, earthquake science, gravitational wave physics, and ocean science. The Pegasus workflow management system can manage the execution of an application formalized as a visual workflow by mapping it onto available resources and executing the workflow tasks in the order of their dependencies. Recent research activities carried out on Pegasus investigated the system implementation on Cloud platforms and how to manage computational workflows in the Cloud for developing scalable scientific applications [8].

Taverna [25] is a workflow management system developed at the University of Manchester. Its primary goal is supporting the life sciences community (biology, chemistry, and medicine) to design and execute scientific workflows and support in silico experimentation, where research is performed through computer simulations with models closely reflecting the real world. Even though most Taverna applications lie in the bioinformatics domain, it can be applied to a wide range of fields since it can invoke any REST or SOAP-based Web services. This feature is very important for allowing users of Taverna to reuse code (represented as a service) that is available on the Internet. Taverna can orchestrate Web Services and these may be running in the Cloud, but this is transparent for Taverna, as demonstrated in the BioVel project [25].

Kepler [12] is a graphical workflow management system that has been used in several projects to manage, process, and analyze scientific data. Kepler provides a graphical user interface (GUI) for designing scientific workflows, which are a structured set of tasks linked together that implement a computational solution to a scientific problem. Data is encapsulated in messages or tokens, and transferred between tasks through input and output ports. Kepler provides an assortment of built-in components with a major focus on statistical analysis and supports task parallel execution of workflows using multiple threads on a single machine.

WS-PGRADE [9] is a general purpose workflow management system that allows users to create and run workflows on distributed computing systems such as a Grids and Cloud platforms. The system allows users to define workflows through a graphical interface and to execute them on different distributed computing infrastructures (DCIs), including popular Cloud systems like Amazon EC2 and Google App Engine. The visual formalism expresses parallelism through parallel paths or through parametric input nodes. A parametric input node will be executed in as many instances as many files arrive on its port. End-users may use the system through a simplified interface where they can download a workflow from a repository, configure its parameter, and launch and monitor its execution on the underlying DCI.

ClowdFlows [10] is a Cloud-based platform for the composition, execution, and sharing of interactive data mining workflows. According with the Software-as-a-Service approach, ClowdFlows provides a user interface that allows programming visual workflows in any Web browser. In addition, its service-oriented architecture allows using third party services (e.g., Web services wrapping open-source or custom data mining algorithms). The server side consists of methods for the client side workflow editor to compose and execute workflows, and a relational database of workflows and data.

E-Science Central (e-SC) [7] is a Cloud-based system that allows scientists to store, analyze and share data in the Cloud. Like ClowdFlows, e-Sc provides a user interface that allows programming visual workflows in any Web browser. Its in-browser workflow editor allows users to design a workflow by connecting services, either uploaded by themselves or shared by other users of the system. One of the most common use cases for e-Sc is to provide a data analysis back end to a standalone desktop or Web application. To this end, the e-SC API provides a set of workflow control methods and data structures. In the current implementation, all the workflow services within a single invocation of a workflow execute on the same Cloud node.

Differently from the systems above, which support visual workflow design, the Data Mining Cloud Framework provides both visual and script-based workflow programming, so as to meet the needs of both high-level users and who prefer to program. Moreover, the DMCF differs from Kepler and Taverna because it natively supports the execution of workflow's tasks on distributed environments composed by multiple machines. In addition, the DMCF's runtime differs from that of ClowdFlows and E-Science Central because it is able to parallelize the execution of the tasks of each workflow, an important feature to ensure scalable data analysis workflows execution on the Cloud.

COMPSs [15] is a programming model and an execution runtime, whose main objective is to ease the development of workflows for distributed environments, including private and public Clouds. With COMPSs, users create a sequential application and specify which methods of the application code will be executed remotely. This selection is done by providing an annotated interface where these methods are declared with some metadata about them and their parameters.

The runtime intercepts any call to a selected method creating a representative task and finding the data dependencies with all the previous ones that must be considered along the application run. This COMPSs strategy is similar to that exploited in DMCF for parallelizing JS4Cloud workflows. However, while in COMPSs users must provide explicit annotations to specify which methods will be executed remotely, JS4Cloud is a pure implicit parallel language, since no special directives or annotations are needed to enable parallel execution.

Swift [24] is a parallel scripting language that runs workflows across several distributed systems, like clusters, Clouds, grids, and supercomputers. It provides a functional language in which workflows are modelled as a set of program invocations with their associated command-line arguments, input and output files. Swift uses a C-like syntax consisting of function definitions and expressions that provide an implicit data-driven task parallelism. The runtime comprises a set of services that implement the parallel execution of Swift scripts exploiting the maximal concurrency permitted by data dependencies within a script and by external resource availability. Users can use Galaxy [5] to provide a visual interface for Swift [14].

Swift and JS4Cloud provide the same type of parallelism (i.e., implicit data-driven task parallelism and data parallelism). However, this is obtained in different ways: by defining a new C-like language from scratch in Swift, versus by extending the JavaScript functionalities in JS4Cloud. Since a JS4Cloud code is a valid JavaScript code, it can be executed by any JavaScript intepreter, and therefore the JS4Cloud programming environment can be embedded into any HTML browser. In Section 4 we describe how JS4Cloud users can exploit a HTML5 Web editor to compose, check and run data analysis workflows. Like Swift, the DMCF that is used to run JS4Cloud workflows on the Cloud, hides the complexity of the underlying infrastructure, thus freeing users from resource configuration and management duties.

We finally mention two script-based workflow languages, Gscript [13] and JOLIE [19], which are related to JS4Cloud, even if they are not explicitly designed for Cloud-based systems.

Gscript is a script-based workflow language, designed to be semantically equivalent to GWENDIA, a visual language to express scientific workflows involving complex data flow patters [18]. A Gscript program is composed of a series of statements, blocks, scalar or array expressions. Each statement defines a processor, its inputs and outputs, and the iteration strategies in a single statement. JOLIE allows programmers to compose statements in a workflow by making sequences, parallelism and non-deterministic choices. Using its communication primitives and its compositional operators, JOLIE can compose other services by exploiting their input operations. JOLIE provides also statements for user input/output console interaction.

Both Gscript and JOLIE are custom languages with a given syntax to write a workflow and to invoke services from it, while JS4Cloud relies on the widely-known JavaScript language. Furthermore, Gscript and JOLIE require the user to explicitly deal with parallelism (through specific control structures in the case of Gscript; by specifying operators between statements in the case of JOLIE), whereas JS4Cloud relies on sequential JavaScript programming and leaves to the runtime the task of exploiting workflow parallelism.

Table I summarizes the features of related work in comparison with the system proposed in this paper (last row in the table). We take into account only systems that provide a textual programming language to define workflows, and consequently we do not consider systems that allow workflow design through a visual formalism or a data representation language (e.g., XML and JSON). For each system, the table indicates: (i) on which programming language is based; (ii) what type of parallelism it provides; (iii) the level of programming skills required to write a script-based workflow; (iv) whether or not an end-user must configure the distributed execution environment; and (v) whether or not the system has been designed for the Cloud. As shown in the table, JS4Cloud and Swift are the only Cloud-oriented languages for data analysis applications that needs low programming skills and does not require specific configurations of the distributed execution environment.

Table I. Comparison with main related systems.

System	Language	Type of parallelism	Level of programming skills required	Configuration of the distributed system	Cloud
COMPSs	Java	Data-driven task parallelism driven by Java annotations	Medium	No	Yes
Swift	C-like syntax	Implicit data-driven task parallelism and data parallelism	Low	No	Yes
Gscript	Custom language	Defined by specific control structures	High	Yes	No
JOLIE	C-like syntax	Specified by operators between statements	High	Yes	No
JS4Cloud	*JavaScript*	*Implicit data-driven task parallelism and data parallelism*	*Low*	*No*	*Yes*

3. DATA MINING CLOUD FRAMEWORK

The *Data Mining Cloud Framework* (DMCF) is a software framework for designing and executing data analysis workflows on the Cloud. DMCF supports a large variety of processing patterns that can be used in data mining, including single-task applications, parameter sweeping applications, and workflow-based applications. Following the approach proposed in [2], DMCF represents knowledge discovery workflows as graphs whose nodes denote resources (datasets, data analysis tools, mining models) and whose edges denote dependencies among resources. A Web-based user interface allows users to compose their applications and to submit them for execution to the Cloud platform, following a Software-as-a-Service (SaaS) approach.

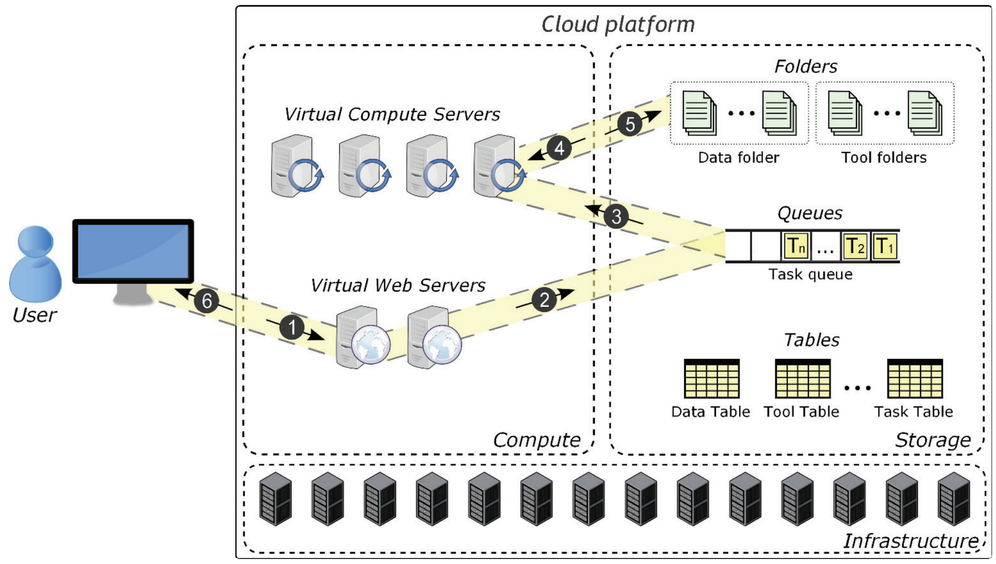


Figure 2. Architecture of the Data Mining Cloud Framework.

The architecture of DMCF includes different kinds of components that can be grouped into storage and compute components (see Figure 2).

The storage components include:

- A *Data Folder* that contains data sources and the results of knowledge discovery processes. Similarly, a *Tool Folder* contains libraries and executable files for data selection, pre-processing, transformation, data mining, and results evaluation.

- *Data Table*, *Tool Table* and *Task Table* contain metadata information associated with data, tools, and tasks.
- The *Task Queue* contains the tasks ready for execution.

The compute components are:

- A pool of *Virtual Compute Servers*, which are in charge of executing the data analysis tasks.
- A pool of *Virtual Web Servers* host the Web-based user interface.

The user interface provides access to three functionalities: $i)$ *App submission*, which allows users to submit single-task, parameter sweeping, or workflow-based applications; $ii)$ *App monitoring*, which is used to monitor the status and access results of the submitted applications; $iii)$ *Data/Tool management*, which allows users to manage input/output data and tools.

The DMCF architecture has been designed in a sufficiently abstract and generic way to be implemented on top of different Cloud systems. The current implementation discussed here is based on Microsoft Azure[†].

Although DMCF is a SaaS solution, in the future it could evolve to a Platform-as-a-Service (PaaS) solution for developing custom SaaS systems. To this end, DMCF should provide APIs and services to access its core functionalities (workflow design, workflow execution, workflow monitoring, etc.), so that developers could directly exploit these functionalities to implement their SaaS solutions.

3.1. *Execution mechanisms*

A user interacts with the system to perform the following steps for designing and executing a knowledge discovery application:

1. The user accesses the Website and designs the application (either single-task, parameter sweeping, or workflow-based) through a Web-based interface.
2. After application submission, the system creates a set of tasks and inserts them into the Task Queue on the basis of the application.
3. Each idle Virtual Compute Server picks a task from the Task Queue, and concurrently executes it.
4. Each Virtual Compute Server gets the input dataset from the location specified by the application. To this end, a file transfer is performed from the Data Folder where the dataset is located, to the local storage of the Virtual Compute Server.
5. After task completion, each Virtual Compute Server puts the result on the Data Folder.
6. The Website notifies the user as soon as her/his task(s) have completed, and allows her/him to access the results.

The set of tasks created on the second step depends on the type of application submitted by the user. In the case of a single-task application, just one data analysis task is inserted into the Task Queue. If the user submits a parameter sweeping application, the set of tasks corresponding to the combinations of the input parameters values are executed in parallel[‡]. In the case of a workflow-based application, the set of tasks created depends on how many data analysis tools are invoked within the workflow; initially, only the workflow tasks without dependencies are inserted into the Task Queue. All the potential parallelism of the workflow is exploited by using the needed Virtual Compute Servers. In addition, multi-threaded tasks exploit all the cores available on the Virtual Compute Servers they are assigned to.

To reduce the overhead of data transfers between Data Folder and the local storage of Virtual Compute Servers, it is important that data are kept physically close to the virtual servers where processing takes place. In the Microsoft Azure implementation, this is achieved by exploiting the

[†] http://azure.microsoft.com

[‡] In general, the number of tasks is given by $\prod_{i=1}^{n} v_i$, where n is the number of input parameters and v_i is the number of values assumed by the i^{th} parameter

Azure's Affinity Group feature, which allows Data Folder and Virtual Compute Servers to be located near to each other in the same data center for optimal performance.

In DMCF, at least one Virtual Web Server runs continuously in the Cloud, as it serves as user front-end. Moreover, users can specify the minimum and maximum number of Virtual Compute Servers. Since storage is managed by the Cloud platform, the number of storage servers is transparent to the user. DMCF can exploit the auto-scaling features of Microsoft Azure that allows spinning up or shutting down Virtual Compute Servers, based on the number of tasks ready for execution in the DMCFs Task Queue. The average sign-up and shut-down times in Azure are 4 minutes and 2 minutes, respectively.

3.2. *Visual workflow formalism*

Visual workflows can be programmed in DMCF through a language called *VL4Cloud* (Visual Language for Cloud). VL4Cloud workflows are directed acyclic graphs whose nodes represent resources and whose edges represent the dependencies among the resources. Workflows includes two types of nodes:

- *Data* node, which represents an input or output data element. Two subtypes exist: *Dataset*, which represents a data collection, and *Model*, which represents a model generated by a data analysis tool (e.g., a decision tree).
- *Tool* node, which represents a tool performing any kind of operation that can be applied to a data node (filtering, splitting, data mining, etc.).

The nodes can be connected through direct edges, establishing specific dependency relationships among them. When an edge is being created between two nodes, a label is automatically attached to it representing the kind of relationship between the two nodes.

Data and Tool nodes can be added to the workflow singularly or in array form. A *data array* is an ordered collection of input/output data elements, while a *tool array* represents multiple instances of the same tool.

Figure 3 shows an example of data analysis workflow developed using the visual workflow formalism of DMCF. In this example, the Census dataset is split into a training set and a test set using a partitioning tool. Then the training set is analyzed in parallel by ten instances of the J48 classification tool (a Java implementation of the C4.5 algorithm [20] provided by the Weka toolkit [6]), which are represented as a single tool array node in the workflow. The J48 instances differ each other only for the value of one input parameter (the confidence factor). The ten models generated by the J48 instances, represented as a data array, are then evaluated against the test set by a ModelSelector to identify the best model, which is the final output of the workflow.

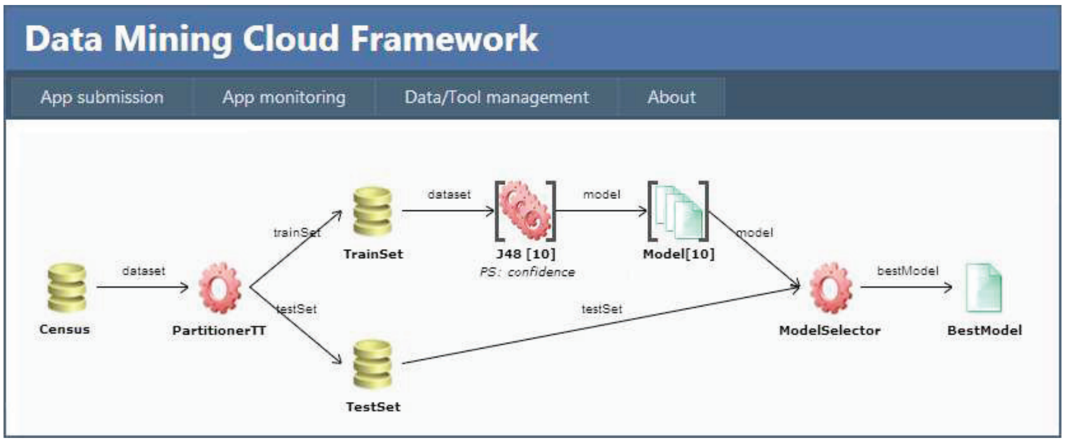


Figure 3. Example of data analysis application designed using the DMCF visual formalism.

4. THE JS4CLOUD LANGUAGE

JS4Cloud (JavaScript for Cloud) is a JavaScript-based language for programming data analysis workflows. It has been introduced as the script-based language for the Data Mining Cloud Framework (DMCF) [16]. The Web interface of DMCF allows to design and execute workflows programmed by the JS4Cloud language, by providing an environment similar to that used to develop visual workflows in the same framework. In particular, the App submission section of the DMCF's Web interface has been extended for allowing users to write, compile and execute JS4Cloud workflows through an integrated development environment including the following functionalities: i) *Code Assist*, which helps users to write JS4Cloud code faster, by providing them with a list of code fragments usable in a specific context; ii) *Syntax Checker*, which detects and reports to the user syntactic errors present in a JS4Cloud program; iii) *Logger*, which provides detailed execution information for debugging purpose, during or after program execution; iv) *Interpreter*, which translates a JS4Cloud workflow into a set of concurrent tasks that can be executed on the Cloud.

In particular, the Interpreter generates a JSON descriptor of the workflow, specifying which are the tasks to be executed and the dependency relationships among them. The Interpreter is a standard JavaScript interpreter that invokes an ad hoc library, called *JS4Cloud.js*, to interpret the specific JS4Cloud functions that will be introduced in Section 4.2. This is shown in Figure 4, where the JavaScript interpreter takes the JS4Cloud workflow as input to generate the corresponding JSON representation. Whenever the JavaScript interpreter encounters one of the JS4Cloud functions for data access, data definition and tool execution, it delegates interpretation to JS4Cloud.js, which knows how those functions must be managed.

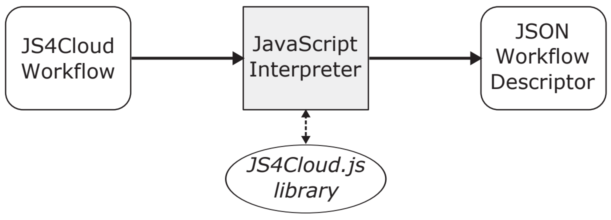


Figure 4. Relationship between JavaScript interpreter and JS4Cloud.js library.

The main benefits of JS4Cloud are: i) it extends the well-known JavaScript language while using only its basic functions (arrays, functions, loops); ii) it implements both a data-driven task parallelism that automatically spawns ready-to-run tasks to the Cloud resources and data parallelism through an array based formalism; iii) these two types of parallelism are exploited implicitly so that workflows can be programmed in a totally sequential way, which frees users from duties like work partitioning, synchronization and communication.

Two strength points of JavaScript motivated its adoption as the basis for JS4Cloud: i) JavaScript natively provides support to arrays and calls to external functions, which are fundamental to implement parallelism and remote task execution in DMCF; ii) JavaScript code can be executed using the standard interpreters available in any modern Web browser, a key feature to write and execute script-based workflows using the Web interface of DMCF.

4.1. Key programming concepts

Two key programming abstractions in JS4Cloud are *Data* and *Tool*:

- *Data* elements denote input files or storage elements (e.g., a dataset to be analyzed) or output files or stored elements (e.g., a data mining model).
- *Tool* elements denote algorithms, software tools or complex applications performing any kind of operation that can be applied to a data element (data mining, filtering, partitioning, etc.).

For each Data and Tool element included in a JS4Cloud workflow, an associated descriptor, expressed in JSON format, is included in the environment of the user who is developing the workflow.

A Tool descriptor includes a reference to its executable, the required libraries, and the list of input and output parameters. Each parameter is characterized by name, description, type, and can be mandatory or optional. An example of descriptor for a data classification tool is presented in Figure 5.

```
"J48": {
  "libraryList": ["java.exe","weka.jar"],
  "executable": "java.exe -cp weka.jar weka.classifiers.trees.J48",
  "parameterList":[{
      "name": "dataset", "flag": "-t",
      "mandatory": true, "parType": "IN",
      "type": "file", "array": false,
      "description": "Input dataset"
    },{
      "name": "confidence", "flag": "-C",
      "mandatory": false, "parType": "OP",
      "type": "real", "array": false,
      "description": "Confidence value",
      "value": "0.25"
    },{
      "name": "model", "flag": "-d",
      "mandatory": true, "parType": "OUT",
      "type": "file", "array": false,
      "description": "Output model"}]}
```

Figure 5. Example of Tool descriptor in JSON format.

The JSON descriptor of a new tool is created automatically through a guided procedure provided by DMCF, which allows users to specify all the needed information for invoking the tool (executable, input and output parameters, etc.). A DMCF module, called *Tool Manager*, supports the deployment of new tools in the system (see Figure 6), which allows their subsequent use in JS4Cloud workflows. To this end, the Tool Manager performs the following tasks: 1) uploads libraries and executable files in Tool Folder; 2) creates a tool descriptor in JSON format; 3) publishes the JSON descriptor in Tool Table.

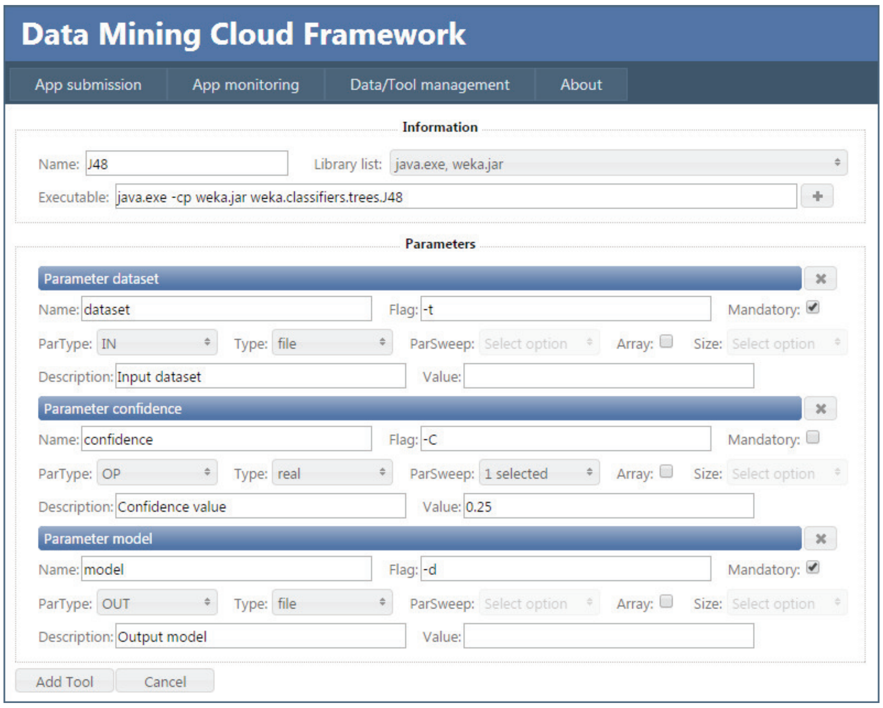
Similarly, a Data descriptor contains information to access an input or output file, including its identifier, location, and format. Differently from Tool descriptors, Data descriptors can also be created dynamically as a result of a task operation during the execution of a JS4Cloud script. For example, if a workflow W reads a dataset D_i and creates (writes) a new dataset D_j, only D_i's descriptor will be present in the environment before W's execution, whereas D_j's descriptor will be created at runtime.

Another key element in JS4Cloud is the *task* concept, which represents the unit of parallelism in our model. A task is a Tool, invoked from the script code, which is intended to run in parallel with other tasks on a set of Cloud resources.

According to this approach, JS4Cloud implements *data-driven task parallelism*. This means that, as soon as a task does not depend on any other task in the same workflow, the runtime asynchronously spawns it to the first available virtual machine. A task T_j does not depend on a task T_i belonging to the same workflow (with $i \neq j$), if T_j during its execution does not read any data element created by T_i.

4.2. *JS4Cloud Functions*

JS4Cloud extends JavaScript with three additional functionalities, implemented by the set of functions listed in Table II:



Information

Name: J48 Library list: java.exe, weka.jar

Executable: java.exe -cp weka.jar weka.classifiers.trees.J48

Parameters

Parameter dataset

Name: dataset Flag: -t Mandatory: ✓

ParType: IN Type: file ParSweep: Select option Array: ☐ Size: Select option

Description: Input dataset Value:

Parameter confidence

Name: confidence Flag: -C Mandatory: ☐

ParType: OP Type: real ParSweep: 1 selected Array: ☐ Size: Select option

Description: Confidence value Value: 0.25

Parameter model

Name: model Flag: -d Mandatory: ✓

ParType: OUT Type: file ParSweep: Select option Array: ☐ Size: Select option

Description: Output model Value:

Figure 6. Adding a new tool to the system.

Table II. JS4Cloud functions.

Functionality	*Function*	*Description*
Data Access	Data.get(<*dataName*>);	Returns a reference to the data element with the provided name.
	Data.get(new RegExp(<*regular expression*>));	Returns an array of references to the data elements whose name match the regular expression.
Data Definition	Data.define(<*dataName*>);	Defines a new data element that will be created at runtime.
	Data.define(<*arrayName*>,<*dim*>);	Define an array of data elements.
	Data.define(<*arrayName*>,[<dim_1>,...,<dim_n>]);	Define a multi-dimensional array of data elements.
Tool Execution	<*toolName*>({<par_1>:<val_1>,...,<par_n>:<val_n>});	Invokes an existing tool with associated parameter values.

- *Data Access*, for accessing a data element stored in the Cloud;
- *Data Definition*: to define a new data element that will be created at runtime as a result of a tool execution;
- *Tool Execution*: to invoke the execution of a tool available in the Cloud.

Data Access is implemented by the `Data.get` function, which is available in two versions: the first one receives the name of a data element, and returns a reference to it; the second one returns an array of references to the data elements whose name match the provided regular expression. For example, the following statement:

```
var ref = Data.get("Census");
```

assigns to variable `ref` a reference to the dataset named `Census`, while the following statement:

```
var ref = Data.get(new RegExp("^CensusPart"));
```

assigns to `ref` an array of references (`ref[0]...ref[n-1]`) to all the datasets whose name begins with `CensusPart`.

Data Definition is done through the `Data.define` function, available in three versions: the first one defines a single data element; the second one defines a one-dimensional array of data elements; the third one defines a multi-dimensional array of data elements. For instance, the following piece of code:

```
var ref = Data.define("CensusModel");
```

defines a new data element named `CensusModel` and assigns its reference to variable `ref`, while the following statement:

```
var ref = Data.define("CensusModel", 16);
```

defines an array of data elements of size 16 (`ref[0]...ref[15]`).

The following is an example statement defining a two-dimensional array of data elements of size 4 times 16:

```
var ref = Data.define("ClassDataset", [4,16]);
```

In all cases, the data elements defined using `Data.define` will be created at runtime as result of a tool execution.

Differently from Data Access and Data Definition, there is not a named function for Tool Execution. In fact, the invocation of a tool T is made by calling a function with the same name of T. The DMCF makes the tools available to the users by loading their descriptions into the integrated development environment (i.e., Code Assist, Syntax Checker, Interpreter). For example, the J48 tool defined in Figure 5 can be invoked as in the following statement:

```
J48({dataset:DRef, confidence:0.05, model:MRef});
```

where `DRef` is a reference to the dataset to be analyzed, previously introduced using the `Data.get` function, and `MRef` is a reference to the model to be generated, previously introduced using `Data.define`.

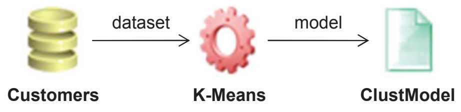
From an implementation perspective, the `Data.get` primitive returns a reference to a data element stored in *Data Folder*, which is a persistent storage independent from the local storage of each Virtual Compute Server. Whenever a data element referenced by `Data.get` must be processed, it is transparently copied to the local storage of the virtual server onto which processing will take place. Similarly, the `Data.define` primitive defines a new data element that will be created at runtime in the local storage of a virtual server, as a consequence of a tool execution. The data elements so created are then transparently copied to the *Data Folder*.

4.3. Basic patterns

In the following we describe how the basic control flow patterns can be programmed with JS4Cloud. We focus on basic patterns [1] such as *single task*, *pipeline*, *data partitioning* and *data aggregation*, and on three additional patterns provided by the visual workflow formalism of DMCF, namely *parameter sweeping*, *input sweeping* and *tool sweeping*. For each pattern, we first introduce an example as a visual DMCF workflow, and then we show how the same example can be coded using JS4Cloud.

Single task

An example of single-task pattern is shown in the following figure:



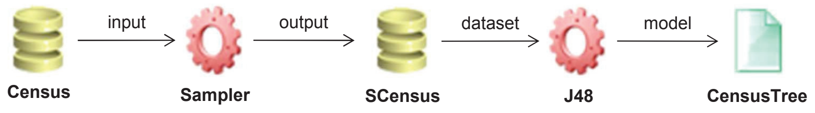
This example represents a K-Means tool that produces a clustering model from a dataset. Each workflow node hides some configuration parameters that have been set by the user, e.g., the number of clusters for the K-Means tool. The following JS4Cloud script is equivalent to the visual workflow shown above:

```
var DRef = Data.get("Customers");
var nc = 5;
var MRef = Data.define("ClustModel");
K-Means({dataset:DRef, numClusters:nc, model:MRef});
```

The script accesses the dataset to be analyzed (`Customers`), sets to `5` the number of clusters, and defines the name of data element that will contain the clustering model (`ClustModel`). Then, the `K-Means` tool is invoked along with the parameters indicated in its JSON descriptor (input dataset, number of clusters, output model).

Pipeline

In the pipeline pattern, the output of a task is the input for the subsequent task, as in the following example:



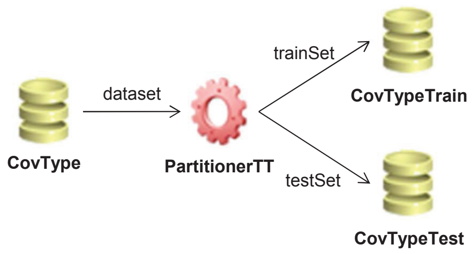
The first part of the shown example extracts a sample from an input dataset using a tool named Sampler. The second part creates a classification model from the sample using the J48 tool. This pattern example can be implemented in JS4Cloud as follows:

```
var DRef = Data.get("Census");
var SDRef = Data.define("SCensus");
Sampler({input:DRef, percent:0.25, output:SDRef});
var MRef = Data.define("CensusTree");
J48({dataset:SDRef, confidence:0.1, model:MRef});
```

In this case, since `J48` receives as input the output of `Sampler`, the former will be executed only after the end of the latter.

Data partitioning

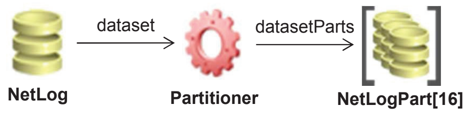
The data partitioning pattern produces two or more output data from an input data element, as in the following example:



In this example a training set and a test set are extracted from a dataset, using a tool named `PartitionerTT`. With JS4Cloud, this can be written as follows:

```
var DRef = Data.get("CovType");
var TrRef = Data.define("CovTypeTrain");
var TeRef = Data.define("CovTypeTest");
PartitionerTT({dataset:DRef, percTrain:0.70, trainSet:TrRef, testSet:TeRef});
```

If data partitioning is used to divide a dataset into a number of splits, the DMCF's data array formalism can be conveniently used as in the following example:



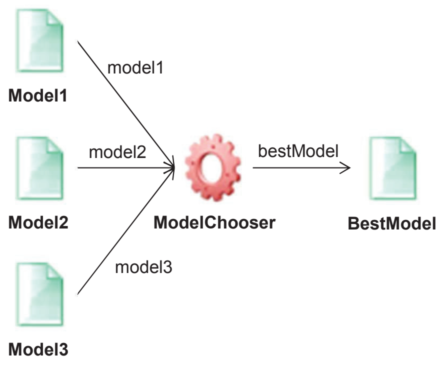
In this case, a Partitioner tool splits a dataset into 16 parts. The corresponding JS4Cloud code is:

```
var DRef = Data.get("NetLog");
var PRef = Data.define("NetLogParts", 16);
Partitioner({dataset:DRef, datasetParts:PRef});
```

Note that an array of `16` data elements is first defined and then created by the `Partitioner` tool.

Data aggregation

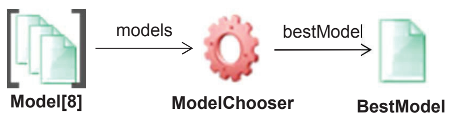
The data aggregation pattern generates one output data from multiple input data, as in the following example:



In this example, a `ModelChooser` tool takes as input three data mining models and chooses the best one based on some evaluation criteria. The corresponding JS4Cloud script is:

```
var M1Ref = Data.get("Model1");
var M2Ref = Data.get("Model2");
var M3Ref = Data.get("Model3");
var BMRef = Data.define("BestModel");
ModelChooser({model1:M1Ref, model2:M2Ref, model3:M3Ref, bestModel:BMRef});
```

DMCF's data arrays may be used for a more compact visual representation. For example, the following pattern example chooses the best one among 8 models:



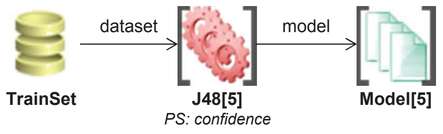
The same task can be coded as follows using JS4Cloud:

```
var BMRef = Data.define("BestModel");
ModelChooser({models:MsRef, bestModel:BMRef});
```

In this script, it is assumed that `MsRef` is a reference to an array of models created on a previous step.

Parameter sweeping

Parameter sweeping is a data analysis pattern in which a dataset is analyzed in parallel by multiple instances of the same tool with different parameters, as in the following example:



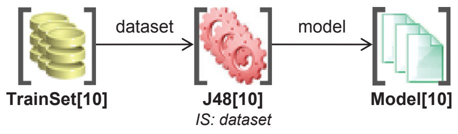
In this example, a training set is processed in parallel by 5 instances of J48 to produce the same number of data mining models. The DMCF's tool array formalism is used to represent the 5 tools in a compact form. The J48 instances differ each other by the value of a single parameter, the *confidence* factor, which has been configured (through the visual interface) to range from 0.1 to 0.5 with a step of 0.1. The equivalent JS4Cloud script is:

```
var TRef = Data.get("TrainSet");
var nMod = 5;
var MRef = Data.define("Model", nMod);
var min = 0.1;
var max = 0.5;
for(var i=0; i<nMod; i++)
   J48({dataset:TRef, model:MRef[i], confidence:(min+i*(max-min)/(nMod-1))});
```

In this case, the `for` construct is used to create `5` instances of `J48`, where the `i`-th instance takes as input the same training set (`TRef`), and produces a different model (`MRef[i]`), using a specific value for the confidence parameter (`0.1` for `J48[0]`, `0.2` for `J48[1]`, and so on). It is worth noticing that the tools are independent of each other, and so the runtime can execute them in parallel.

Input sweeping

Input sweeping is a pattern in which a set of input data is analyzed independently to produce the same number of output data. It is similar to the parameter sweeping pattern, with the difference that in this case the sweeping is done on the input data rather than on a tool parameter. An example of input sweeping pattern is represented in the following figure:

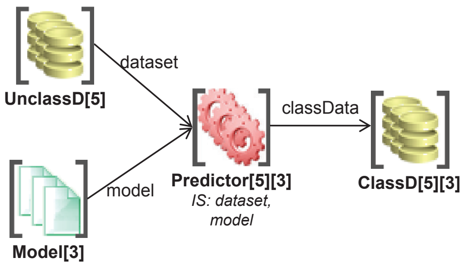


In this example, 10 training sets are processed in parallel by 10 instances of J48, to produce the same number of data mining models. Data arrays are used to represent both input data and output models, while a tool array is used to represent the J48 tools. The following JS4Cloud script corresponds to the example shown above:

```
var nMod = 10;
var MRef = Data.define("Model", nMod);
for(var i=0; i<nMod; i++)
   J48({dataset:TsRef[i], model:MRef[i], confidence:0.1});
```

It is assumed that `TsRef` is a reference to an array of training sets created on a previous step. The `for` loop creates `10` instances of `J48`, where the `i`-th instance takes as input `TsRef[i]` to produce `MRef[i]`.

Another example of input sweeping pattern is represented in the following figure:



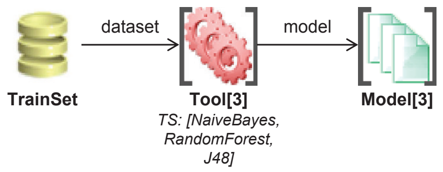
In this case there are 15 instances of a Predictor. Each Predictor takes in input one unclassified dataset and one model, and generates concurrently one classified dataset. The following JS4Cloud script corresponds to this example:

```
var nData = 5, nMod = 3;
var CRef = Data.define("ClassD", [nData, nMod]);
for(var i=0; i<nData; i++)
   for(var j=0; j<nMod; j++)
      Predictor({dataset:DRef[i], model:MRef[j], classDataset:CRef[i][j]});
```

Here is assumed that `DRef` is a reference to an array of unlabeled datasets, and `MRef` is a reference to an array of models, created on previous steps. The double `for` loop creates a two-dimensional array of classified datasets, denoted `CRef`, where `CRef[i][j]` is the classified dataset generated by a Predictor instance on `DRef[i]` using `MRef[j]`. Also in this case, since the tools are independent each other, they can be executed in parallel by the runtime.

Tool sweeping

Tool sweeping is a pattern in which a dataset is analyzed in parallel by different tools, as in the following example:



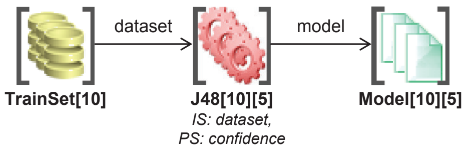
In this case, each of the three classification tools (`NaiveBayes`, `RandomForest`, `J48`) analyzes the same training set to produce three classification models. This corresponds to the following JS4Cloud script:

```
var TRef = Data.get("TrainSet");
var MRef = Data.define("Model", 3);
NaiveBayes({dataset:TRef, model:MRef[0], kernelDensity:true});
RandomForest({dataset:TRef, model:MRef[1], numberOfTrees:500});
J48({dataset:TRef, model:MRef[2], confidence:0.1});
```

Combination of sweeping patterns

In JS4Cloud, it is possible to combine parameter, input and tool sweeping patterns. In the following we present two examples of sweeping patterns combinations.

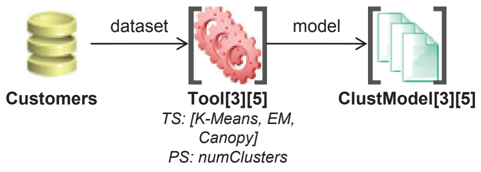
As a first example we show an input/parameter sweeping, i.e. the combination of input and parameter sweeping. With this pattern, each input data is analyzed in parallel by multiple instances of the same tool with different parameters, as in the following figure:



In this example, each of the 10 training sets is processed by 5 instances of J48 to produce 5 data mining models. Thus, there are in total 50 instances of J48, represented by a two-dimensional array of size 10 times 5, that generate the same number of models. The following JS4Cloud script corresponds to this example:

```
var nTr = 10;
var conf = [0.1, 0.2, 0.3, 0.4, 0.5];
var MRef = Data.define("Model", [nTr, conf.length]);
for(var i=0; i<nTr; i++)
   for(var j=0; j<conf.length; j++)
      J48({dataset:TsRef[i], model:MRef[i][j], confidence:conf[j]});
```

The second example is a tool/parameter sweeping, i.e. the combination of tool and parameter sweeping. With this pattern, a dataset is analyzed in parallel by a set of tools, each of them configured with different parameters, as in the following figure:



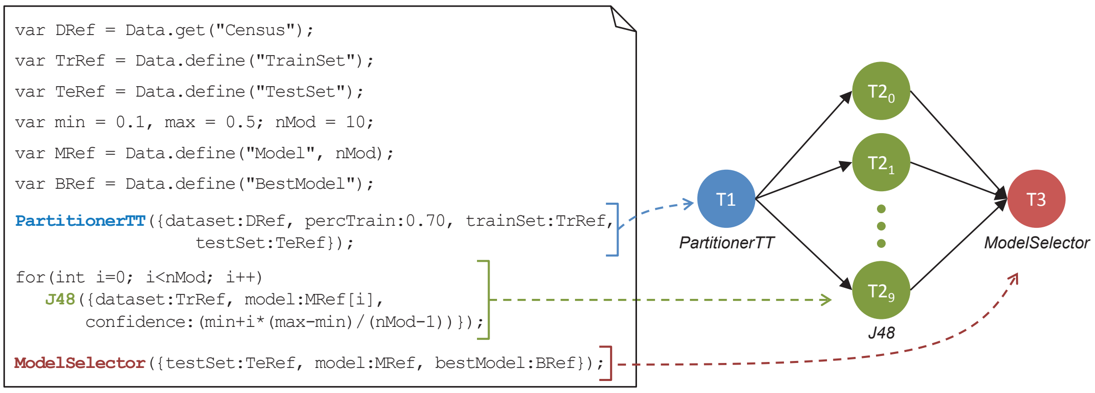
In this workflow, three clustering tools, `K-Means`, `Em` and `Canopy`, analyze in parallel the same dataset. Each clustering algorithm is executed five times varying an algorithm parameter (the number of clusters). Thus, for each of the three clustering tool there five instances, represented by a two-dimensional array of size 3 times 5. This is the equivalent JS4Cloud script:

```
var DRef = Data.get("Customers");
var nt = 3;
var nc = [3,4,5,6];
var MRef = Data.define("ClustModel", [nt, nc.length]);
for(var i=0; i<nc.length; i++){
   K-Means({dataset:DRef, numClusters:nc[i], model:MRef[0,i]});
   EM({dataset:DRef, numClusters:nc[i], model:MRef[1,i]});
   Canopy({dataset:DRef, numClusters:nc[i], model:MRef[2,i]});}
```

4.4. Parallelism exploitation

As explained above, as soon as a task in a JS4Cloud workflow does not depend on any other task, the DMCF runtime asynchronously spawns it to the first available virtual machine. To better explain how parallelism is exploited with this approach, let us consider again the visual workflow shown in Figure 3, which performs a data classification with parameter sweeping. The equivalent JS4Cloud workflow is shown in the left part of Figure 7.

The workflow can be seen as composed of three steps. In the first step, `PartitionerTT` splits the input dataset into training set and test set (task $T1$). The second step consists in the concurrent



```
var DRef = Data.get("Census");
var TrRef = Data.define("TrainSet");
var TeRef = Data.define("TestSet");
var min = 0.1, max = 0.5; nMod = 10;
var MRef = Data.define("Model", nMod);
var BRef = Data.define("BestModel");
PartitionerTT({dataset:DRef, percTrain:0.70, trainSet:TrRef,
                  testSet:TeRef});
for(int i=0; i<nMod; i++)
   J48({dataset:TrRef, model:MRef[i],
       confidence:(min+i*(max-min)/(nMod-1))});
ModelSelector({testSet:TeRef, model:MRef, bestModel:BRef});
```

Figure 7. JS4Cloud script equivalent to the workflow in Figure 3, with associated task dependency graph.

execution of 10 instances of J48 (tasks $T2_0...T2_9$). During the third step, `ModelSelector` chooses the best model (task $T3$).

Overall, the workflow generates 12 tasks that are related each other as specified by the dependency graph shown in the right part of Figure 7. The graph shows that, as soon as $T1$ completes, tasks $T2_0...T2_9$ can be executed. After completion of all such tasks, $T3$ can be finally executed. The parallelism exhibited by the graph is fully exploited by executing the dependency-free tasks on the available virtual machines. In this case, tasks $T2_0...T2_9$ will run in parallel, thus resulting in a significant execution speedup.

5. EXPERIMENTAL EVALUATION

In this section we present some experimental performance results obtained executing two JS4Cloud workflows with the Data Mining Cloud Framework. The first workflow represents an ensemble learning application, while the second workflow represents a parallel classification application. The main goal of the first workflow is to illustrate the JS4Cloud capability of expressing a complex data analysis process, while the second workflow shows the high scalability that can be achieved by executing JS4Cloud workflows. The Cloud environment used for the experimental evaluation was composed by up to 64 virtual servers, each one equipped with a single-core 1.66 GHz CPU, 1.75 GB of memory, and 225 GB of disk space with a cost of $0.08/hr.

5.1. Ensemble learning workflow

This workflow is the implementation of a multi-class cancer classifier based on the analysis of genes, using an ensemble learning approach [11]. The input dataset is the Global Cancer Map (GCM)[§], which contains the gene expression profiles of 280 samples representing 14 common human cancer classes. For each sample is reported the status of 16,063 genes and the type of tumor (class label). The GCM dataset is available as a training set containing 144 instances and as a test set containing 46 instances. The goal is to classify an unlabeled dataset (*UnclassGCM*) composed by 20,000 samples, divided in four parts.

The workflow begins by analyzing the training set using n instances of the J48 classification tool and m instances of the `JRip` classification tool (Weka's implementation of the Ripper [3] algorithm). The n J48 instances are obtained by sweeping the *confidence* and the *minNumObj* (minimum number of instances per leaf) parameters, while the m `JRip` instances are obtained by sweeping the *numFolds* (number of folders) and *seed* parameters. The resulting $n + m$ classification models (classifiers) are passes as input to $n + m$ evaluators, which produce an evaluation of each

[§]http://eps.upo.es/bigs/datasets.html

```
 1: var TrRef = Data.get("GCM-train");
 2: var conf = [0.1, 0.25, 0.5], mno = [2, 5, 10], nfol = [3, 5, 10],
       snum = [1487, 5741, 7699];
 3: var n = conf.length*mno.length, m = nfol.length*snum.length;
 4: var M1Ref = Data.define("Model1", n), M2Ref = Data.define("Model2", m);
 5: for(var i=0; i<conf.length; i++)
 6:    for(var j=0; j<mno.length; j++)
 7:       J48({dataset:TrRef, model:M1Ref[i*mno.length+j], confidence:conf[i],
             minNumObj:mno[j]});
 8: for(var i=0; i<nfol.length; i++)
 9:    for(var j=0; j<snum.length; j++)
10:       JRip({dataset:TrRef, model:M2Ref[i*snum.length+j], numFolds:nfol[i],
             seed:snum[j]});
11: var TeRef = Data.get("GCM-test"), EvM1Ref = Data.define("EvModel1", n),
       EvM2Ref = Data.define("EvModel2", m);
12: for(var i=0; i<n; i++)
13:    Evaluator({dataset:TeRef, model:M1Ref[i], evalModel:EvM1Ref[i]});
14: for(var i=0; i<m; i++)
15:    Evaluator({dataset:TeRef, model:M2Ref[i], evalModel:EvM2Ref[i]});
16: var k = 4;
17: var DRef = Data.get("UnlabGCM", k), CRef = Data.define("ClassGCM",[k,n+m]);
18: for(var i=0; i<k; i++){
19:    for(var j=0; j<n; j++)
20:       Predictor({dataset:DRef[i], model:M1Ref[j], classDataset:CRef[i][j]});
21:    for(var j=0; j<m; j++)
22:       Predictor({dataset:DRef[i], model:M2Ref[j], classDataset:CRef[i][n+j]});
23: }
24: var FRef = Data.define("FinalClassGCM", k), EvMRef = EvM1Ref.concat(EvM2Ref);
25: for(var i=0; i<k; i++)
26:    WeightedVoter({classDataset:CRef[i], evalModel:EvMRef,
          finalClassDataset:FRef[i]});
```

Figure 8. Ensemble learning JS4Cloud workflow.

model against the test set. Then, k unclassified datasets are classified using the $n + m$ models by $k * (n + m)$ predictors. Finally, k voters take in input $n + m$ model evaluations and the $k * (n + m)$ classified datasets, producing k classified datasets through weighted voting.

Figure 8 shows the JS4Cloud code of the workflow. At the beginning, the training set is specified (*line 1*). Then, arrays *conf* and *mno* specify, respectively, the *confidence* and *minNumObj* values for J48, while arrays *nfol* and *snum* specify, respectively, the *numFolds* and *seed* values for `JRip` (*line 2*). Given the size of the above arrays, variables n and m as defined on *line 3* are both equal to 9. Afterwards, n instances of J48 and m instances of `JRip` are executed, where each instance uses a different combination of its parameters to analyze the training set (*lines 5-10*). *Line 11* specifies the test set, which is used to evaluate the two arrays of models generated by the n J48 instances and the m `JRip` instances (*lines 12-15*). Then, k unlabeled datasets are specified as input, with $k = 4$ (*line 17*). Each of the k input datasets is classified by n predictors using the n models generated by J48, and by m predictors using the m models generated by `JRip`; therefore, for each of the k input datasets, $n + m$ classified datasets are generated (*lines 18-22*). As a final step, k weighted voters are executed; the i-th voter receives the $n + m$ classified datasets generated from the i-th input and the $n + m$ models, and returns the final classified dataset for the i-th input (*lines 25-26*). In general, the workflow is composed of $k + (k + 2)(n + m)$ tasks, which are related each other as specified by the dependency graph shown in Figure 9. In the specific example (with $n = 9$, $m = 9$, $k = 4$) the number of tasks is 112.

This example demonstrates the flexibility of JS4Cloud. In fact, thanks to its integration with native JavaScript functionalities, we could use it to express a quite complex ensemble learning application.

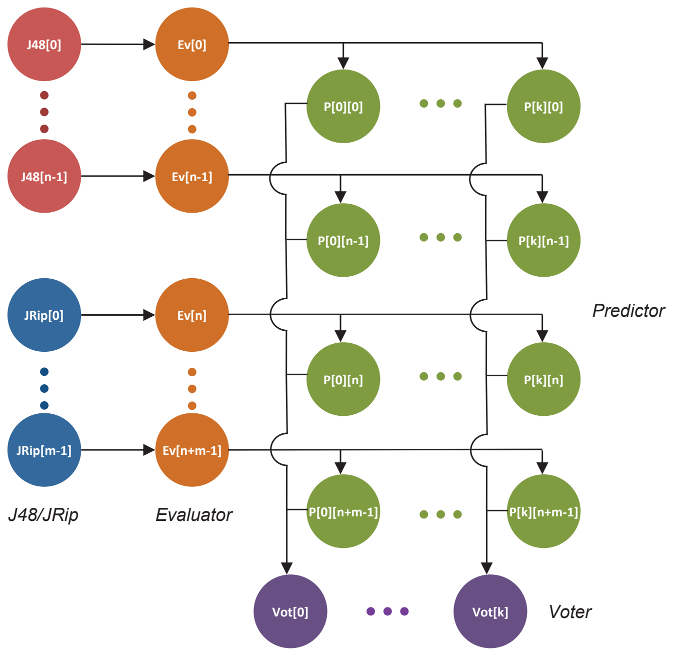


Figure 9. Task dependency graph associated with the ensemble learning workflow in Figure 8.

In addition, the experimental evaluation of the workflow, conducted using 19 virtual servers, showed a significant reduction of turnaround time compared to that achieved by the sequential execution. In particular, the turnaround time passed from about 162 minutes using a single server, to about 11 minutes using 19 servers, which results in a speedup of about 14.7.

5.2. Parallel classification workflow

This workflow analyzes a dataset using n instances of the J48 algorithm that work on n partitions of the training set and generate n classification models. By using the n models and the test set, n predictors produce in parallel n classified datasets. In the final step of the workflow, a voter generates the final classification (in the file `FinalClassTestSet`) by assigning a class to each data item. This is done by choosing the class predicted by the majority of the models [27].

The input dataset, containing about 46 million tuples and with a size of 5 GB, was generated from the *KDD Cup 1999*'s dataset¶, which contains a wide variety of simulated intrusion records in a military network environment.

Figure 10 shows the JS4Cloud code of the workflow, where $n = 64$. At the beginning, the input dataset is split into training set and test set by a partitioning tool (*line 3*). Then, the training set is partitioned into 64 parts using another partitioning tool (*line 5*). As third step, the training sets are analyzed in parallel by 64 instances of the J48 algorithm, to produce the same number of classification models (*lines 7-8*). The fourth step classifies the test set using the 64 models generated on the previous step (*lines 10-11*). The classification is performed by 64 predictors that run in parallel to produce 64 classified test sets. As the last operation, the 64 classified test sets are passed to a voter that produces the final classified test set. The workflow is composed of $3 + 2n$ tasks, whose dependency graph is shown in Figure 11. In the specific example in which $n = 64$, the total number of tasks is 131.

Figure 12 shows a snapshot of the parallel classification workflow taken during its execution in the DMCF's user interface. Beside each code line number, a colored circle indicates the status of execution. The green circles at lines 3 and 5 indicate that the two partitioners have completed

¶http://kdd.ics.uci.edu/databases/kddcup99/kddcup99

```
 1: var n = 64;
 2: var DRef = Data.get("KDDCup99_5GB"), TrRef = Data.define("TrainSet"),
       TeRef = Data.define("TestSet");
 3: PartitionerTT({dataset:DRef, percTrain:0.7, trainSet:TrRef,
 testSet:TeRef});
 4: var PRef = Data.define("TrainsetPart", n);
 5: Partitioner({dataset:TrRef, datasetPart:PRef});
 6: var MRef = Data.define("Model", n);
 7: for(var i=0; i<n; i++)
 8:    J48({dataset:PRef[i], model:MRef[i], confidence:0.1});
 9: var CRef = Data.define("ClassTestSet", n);
10: for(var i=0; i<n; i++)
11:    Predictor({dataset:TeRef, model:MRef[i], classDataset:CRef[i]});
12: var FRef = Data.define("FinalClassTestSet");
13: Voter({classDataset:CRef, finalClassDataset:FRef});
```

Figure 10. Parallel classification JS4Cloud workflow.

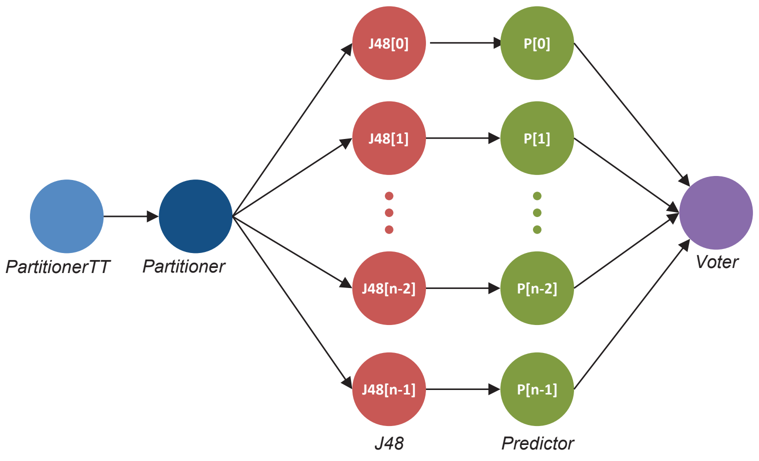
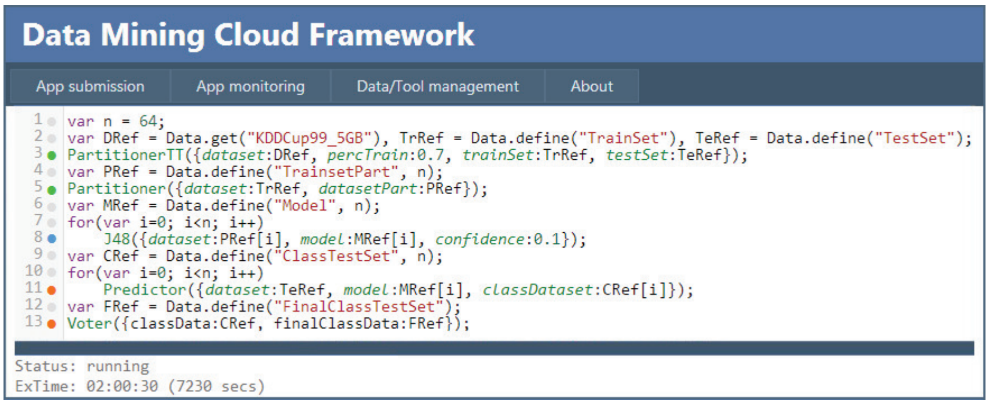


Figure 11. Task dependency graph associated with the parallel classification workflow in Figure 10.

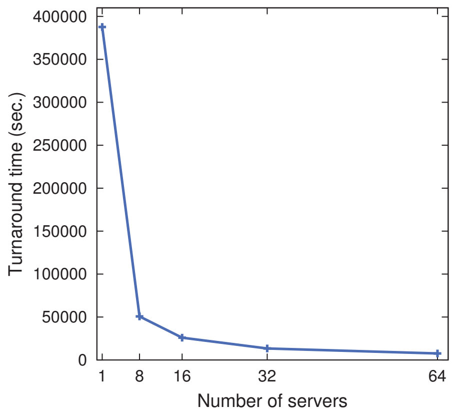


```
 1   var n = 64;
 2   var DRef = Data.get("KDDCup99_5GB"), TrRef = Data.define("TrainSet"), TeRef = Data.define("TestSet");
 3 ● PartitionerTT({dataset:DRef, percTrain:0.7, trainSet:TrRef, testSet:TeRef});
 4   var PRef = Data.define("TrainsetPart", n);
 5 ● Partitioner({dataset:TrRef, datasetPart:PRef});
 6   var MRef = Data.define("Model", n);
 7   for(var i=0; i<n; i++)
 8 ●     J48({dataset:PRef[i], model:MRef[i], confidence:0.1});
 9   var CRef = Data.define("ClassTestSet", n);
10   for(var i=0; i<n; i++)
11 ●     Predictor({dataset:TeRef, model:MRef[i], classDataset:CRef[i]});
12   var FRef = Data.define("FinalClassTestSet");
13 ● Voter({classData:CRef, finalClassData:FRef});
```

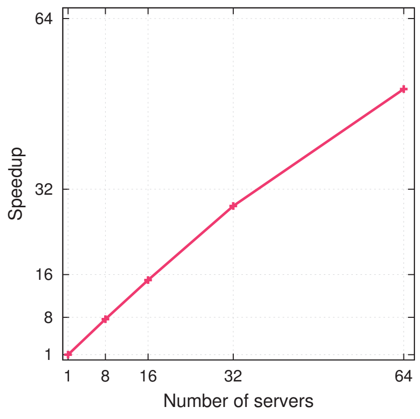
```
Status: running
ExTime: 02:00:30 (7230 secs)
```

Figure 12. Snapshot of the JS4Cloud workflow in Figure 10 running in the DMCF's user interface.

their execution; the blue circle at line 8 indicates that J48 tasks are still running; the orange circles indicates that the corresponding tasks are waiting to be executed. A text area at the bottom of the user interface shows the overall program's status and execution time.



a)



b)

Figure 13. Parallel classification workflow: a) Turnaround time vs. number of available servers; b) Speedup vs. number of available servers; c) Efficiency vs. number of available servers.

Figure 13a shows the turnaround times of the workflow, obtained varying the number of virtual servers used to run it on the Cloud from 1 (sequential execution) to 64 (maximum parallelism). As shown in the figure, the turnaround time decreases from more than 107 hours (4.5 days) by using a single server, to about 2 hours on 64 servers. This is an evident and significant reduction of time, which proves the system scalability.

The scalability achieved by the system can be further evaluated through Figure 13b, which illustrates the relative speedup achieved by using up to 64 servers. As shown in the figure, the speedup increases from 7.64 using 8 servers to 50.78 using 64 servers. This is a very positive result, taking into account that some sequential parts of the implemented application (namely, partitioning and voting) do not run in parallel.

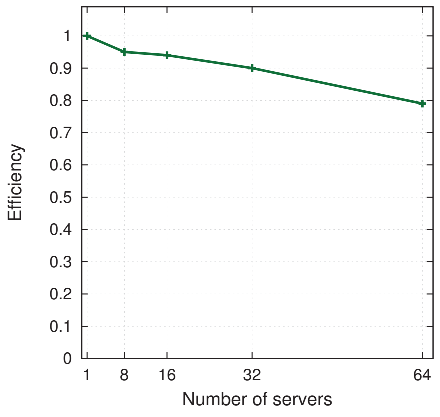


Figure 14. Parallel classification workflow: Efficiency vs. number of available servers.

Figure 14 shows the application efficiency, calculated as the speedup divided by the number of servers used. As shown in the figure, efficiency on 32 servers is equal to 0.9 whereas on 64 servers it is equal to 0.8. Thus in this case, 80% of the computing power of each used server is exploited.

The experiments described above have been executed with all the Virtual Compute Servers continuously running. To test the auto-scaling features of the Cloud platform, we re-executed the workflow with 64 servers, by using the auto-scaling feature of Azure turned on. To this end, we set the minimum and maximum number of Virtual Compute Servers equal to 1 and 64, respectively, to evaluate the impact of server sign-up on turnaround time and cost paid by the user. The results are as follows. With all the Virtual Compute Servers continuously running, the turnaround time was 2.11 hours with a total cost of $10.82. With the auto-scaling enabled, the turnaround time was 2.23 hours with a total cost of $8.88. The increase in the turnaround time registered in the auto-scaling case (about 5 minutes), is due to the time needed to switch-on the servers used to run the 64 instances of J48, after completion of the partitioning task.

6. CONCLUSION

Data analysis applications are often composed by many concurrent and compute-intensive tasks that can be efficiently executed only on scalable computing infrastructures, such as Cloud platforms. Workflows are an effective paradigm for modelling complex data analysis applications, particularly when they must executed on such platforms. In this paper, we presented the JS4Cloud language designed to provide Cloud users with an effective script-based data analysis workflow programming paradigm.

The main benefits of JS4Cloud are: i) it extends the well-known JavaScript language while using only its basic functions (arrays, functions, loops); ii) it implements both a data-driven task parallelism that automatically spawns ready-to-run tasks to the Cloud resources and data parallelism through an array based formalism; iii) these two types of parallelism are exploited implicitly so that workflows can be programmed in a fully sequential way, which frees users from duties like work partitioning, synchronization and communication.

Experimental performance results, obtained designing and executing JS4Cloud workflows in DMCF, have proven the effectiveness of the language for programming data analysis workflows, as well as the scalability that can be achieved by executing such workflows on a public Cloud infrastructure. Cloud environments like DMCF and its visual and script-based programming interfaces are important components for supporting researchers and developers in the implementation of Big data analysis applications [21]. A full implementation of the DMCF system is in use in our lab. We are currently planning to make the system available to the research community in the near future.

REFERENCES

1. S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, 2008.
2. E. Cesario, M. Lackovic, D. Talia, and P. Trunfio. Programming knowledge discovery workflows in service-oriented distributed systems. *Concurrency and Computation: Practice and Experience*, 25(10):1482–1504, July 2013.
3. W. W. Cohen. Fast effective rule induction. In *In Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.
4. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
5. B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, W. Miller, W. J. Kent, and A. Nekrutenko. Galaxy: A platform for interactive large-scale genome analysis. *Genome Res*, 15:1451–1455, 2005.
6. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.
7. H. Hiden, S. Woodman, P. Watson, and J. Cala. Developing cloud applications using the e-Science Central platform. *Philosophical Transactions of the Royal Society A: Mathematical,Physical and Engineering Sciences*, 371(1983), January 2013.
8. G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, SC '10, pages 1–9. IEEE, November 2010.
9. P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, and I. Marton. Ws-pgrade/guse generic dci gateway framework for a large variety of user communities. *J. Grid Comput.*, 10(4):601–630, Dec. 2012.
10. J. Kranjc, V. Podpečan, and N. Lavrač. ClowdFlows: A Cloud Based Scientific Workflow Platform. In P. Flach, T. Bie, and N. Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 7524 of *Lecture Notes in Computer Science*, pages 816–819. Springer, Heidelberg, Germany, 2012.
11. L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

12. B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
13. K. Maheshwari and J. Montagnat. Scientific workflow development using both visual and script-based representation. In *Proceedings of the 2010 6th World Congress on Services*, SERVICES '10, pages 328–335, Washington, DC, USA, 2010. IEEE Computer Society.
14. K. Maheshwari, A. Rodriguez, D. Kelly, R. Madduri, J. Wozniak, M. Wilde, and I. Foster. Enabling multi-task computation on galaxy-based gateways using swift. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–3, Sept 2013.
15. F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia. Enabling cloud interoperability with compss. In *Proc. of the 18th International European Conference on Parallel and Distributed (Europar 2012)*, volume 7484, pages 16–27, Rhodes Island, Greece, 27-31 August 2012. Lecture Notes in Computer Science.
16. F. Marozzo, D. Talia, and P. Trunfio. Scalable script-based data analysis workflows on clouds. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, WORKS '13, pages 124–133, New York, NY, USA, 2013. ACM.
17. F. Marozzo, D. Talia, and P. Trunfio. Using clouds for scalable knowledge discovery applications. In *Proceedings of the 18th International Conference on Parallel Processing Workshops*, Euro-Par'12, pages 220–227, 2013.
18. J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. B. Fornarino. A data-driven workflow language for grids based on array programming principles. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM.
19. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. Jolie: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.
20. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
21. D. Talia. Clouds for scalable big data analytics. *IEEE Computer*, 46(5):98–101, 2013.
22. D. Talia. Workflow systems for science: Concepts and tools. *ISRN Software Engineering*, 2013.
23. D. Talia and P. Trunfio. *Service-oriented distributed knowledge discovery*. Chapman and Hall/CRC, October 2012.
24. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, September 2011.
25. K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, July 2013.
26. K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Research*, 2013.
27. Z.-H. Zhou and M. Li. Semi-supervised learning by disagreement. *Knowl. Inf. Syst.*, 24(3):415–439, 2010.