Received: 8 December 2019 | Revised: 24 February 2020 | Accepted: 3 April 2020
DOI: 10.1002/cpe.5802

SPECIAL ISSUE PAPER

WILEY

Toward a domain-specific language for scientific workflow-based applications on multicloud system

Gennaro Cordasco[1] | Matteo D'Auria[2] | Alberto Negro[2] | Vittorio Scarano[2] | Carmine Spagnuolo[2]

[1]Dipartimento di Psicologia, Università degli Studi della Campania "Luigi Vanvitelli", Caserta, Italy

[2]Dipartimento di Informatica, Università degli Studi di Salerno, Salerno, Italy

Correspondence
Carmine Spagnuolo, Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, Italy.
Email: cspagnuolo@unisa.it

Summary
The cloud computing paradigm has emerged as the backbone of modern price-aware scalable computing systems. Many cloud service models are competing to become the leading doorway to access the computational power of cloud providers. Recently, a novel service model, called function-as-a-service (FaaS), has been proposed, which enables users to exploit the cloud computational scalability, left out the configuration and management of huge computing infrastructures. This article discloses Fly, a domain-specific language, which aims at reconciling cloud and high-performance computing paradigms adopting a multicloud strategy by providing a powerful, effective, and pricing-efficient tool for developing scalable workflow-based scientific applications by exploiting different and at the same time FaaS cloud providers as computational backends in a transparent fashion. We present several improvements of the Fly language, as well as a new enhanced version of a source-to-source compiler, which currently supports Symmetric Multiprocessing, Amazon AWS, and Microsoft Azure backends and translation of functions in Java, JavaScript, and Python programming languages. Furthermore, we discuss a performance evaluation of Fly on a popular benchmark for distributed computing frameworks, along with a collection of case studies with an analysis of their performance results and costs.

KEYWORDS
distributed computing, domain-specific languages, functions as a service, parallel computing, scientific computing, serverless computing, workflow-based application

1 | INTRODUCTION

The cloud computing paradigm[1] has remodeled the computing systems horizon during the past decade. It has emerged as the backbone of modern computing systems by offering subscription-based services, following a pay-as-you-go model. Many service models are competing to become the leading model of cloud computing architectures.

This field is maturing, with a wealth of well-understood service-models: *infrastructure-as-a-service* (IaaS), *platform-as-a-service* (PaaS), and *software-as-a-service (SaaS)*. A novel service-model, named *serverless computing model* or *function-as-service* (FaaS), is providing a shift in the way to access to the cloud computing computational power. Serverless computing architecture is the natural evolution of *microservices architecture*,[2] in which code snippets are executed over the cloud infrastructure in a transparent manner, that is, without any knowledge about when and where the code is executed. This novel service model was first introduced and made available to the world by hook.io in late 2014 and was shortly followed by AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and many others.

FaaS can be seen as a fine grain computing partitioning of cloud applications, which enables to scale according to the provider capacity. FaaS has been designed to easily build and deploy scalable business-oriented applications such as mobile, Internet of Things (IoT), real-time file/stream processing, and web applications as well as service-oriented applications.

Since the very beginning of cloud, it was clear how the paradigm represented an opportunity to easily develop and execute extreme-scale applications maintaining their costs extremely low compared with high-performance computing solutions, as shown by the experiments.[3] Scientific computing applications are commonly developed using general-purposes languages or parallel languages/frameworks such as C, Java, Python, Fortran, Chapel, MPI Paradigm, OpenMP, Swift, and many others (see Section 3.1 for more details). Moreover, scientific computing problems are typically computing-intensive and require the computational power of a distributed system (clusters or HPC). On the other hand, cloud infrastructure usually provides general purposes services in an accessible fashion through web endpoints, and/or APIs while scientific applications typically do not require such services (for instance access to database or providing web pages), but they require ad hoc coding that implements algorithms, which solve specific problems. Although many cloud computing companies are recently providing MapReduce[4] programming paradigm as a cluster of machines running MapReduce compliant framework such as Apache Hadoop (eg, AWS's *Elastic Map Reduce*), many computing-intensive problems do not fit well the MapReduce paradigm. Moreover, although the cloud providers offer solutions with a high level of scalability, very often the migration of a scientific application on IaaS or PaaS represents a humongous and complex task, which can conceal serious cost considerations, thereby often preclude scientific application developers to fully exploit the scalability and cost-effectiveness of cloud computing in their own application domain.

Nowadays, the need for several simple requirements like availability, cost reductions, or special features has resulted in a move from single-user private cloud to multitenant multiple clouds.[5] Multicloud strategies provides many advantages, but it undoubtedly adds an extra layer of management complexity, which requires specific skills.

The aim of this work is reconciling cloud and high-performance computing by providing a powerful, effective, and pricing-efficient tool for the development of scalable workflow-based scientific computing applications,[6-8] on different FaaS platforms, eventually, at the same time (as a multicloud system), through the design and implementation of the Fly *domain-specific language* (DSL). Fly is *powerful* because it enables to exploit the computing capabilities of different cloud providers at once, in a single application, and, then, the most efficient solutions can be merged together. Fly is *effective* because it consists of a user-friendly programming language that frees the programmer from the management and configuration of several complex computation systems. Finally, Fly is *pricing-efficient* because the programmer becomes conscious of the maximum computing costs, based on the prices provided by various cloud providers. In this way, the programmer also has the possibility to choose the service that provides the best value for money, based on the characteristics of the computation that is going to perform.

As far as we know, no previous research has investigated the application of a multicloud system by exploiting FaaS providers to design and develop scientific workflow-based applications. This idea has been presented in Reference 9, a preliminary work where is described the Fly language programming model and the initial language definition Alpha 1.0. In this article, we also provide an improvement on the Fly language definition and a detailed description of the Fly compiler Alpha 1.5, that was only sketched in the preliminary version of the article. Moreover, we evaluated the performance of Fly running on several backends. Overall, the contributions of this article are:

1. The design of the Fly programming model, a novel domain-specific language for computing-intensive workflow-based scientific applications running on top of a multicloud system by exploiting FaaS service-model.
2. The Alpha 1.5 compiler version developed using the Xtext[10] framework and released under MIT License on a public repository code GitHub. This compiler version supports three computing backends: Symmetric Multiprocessing (SMP), Amazon AWS Lamba,[11] Microsoft Azure Functions[12] and provides the functions translation in three programming languages: Java, JavaScript, and Python.
3. A performance evaluation of Fly on a popular benchmark for distributed computing frameworks.
4. A collection of three algorithm use cases developed in Fly with an analysis of their performance results on the Amazon AWS Lamba[11] FaaS cloud provider.

1.1 Article organization

The remainder of this article is organized as follows. Sections 3 and 2 recall the preliminaries of the cloud computing service-models, describes the novel service model FaaS, summarizes the cloud computing providers that offer FaaS, and analyzes the related works. The Fly design is presented in Section 4, while in Section 5, the language definition is discussed. In Section 6, the Alpha 1.5 compiler version is described. Section 7 provides an analysis of Fly capabilities in terms of performance and cost-effectiveness. Section 8 illustrates the implementation of two classical computing-intensive applications for scientific computing, showing also the performance running exploiting computing backends. Finally, we conclude the article in Section 9.

2 | PRELIMINARIES

2.1 | Cloud computing service-models

Cloud computing enables companies to use computing resources as a service (like electricity) rather than having to buy, set up, and maintain computing infrastructures in house. Several cloud computing service-models[13] well known as *software-as-a-service (SaaS)*, *platform-as-a-service(PaaS)*, and *infrastructure-as-a-service (IaaS)* have been proposed during the last two decades. At a first sight, cloud service models look promising for the scientific computing community, as they may take advantage of the adoption of cloud computing, in their compute-intensive applications and workflows, in each of the service models described above. However, the scenario does not come without effort and costs, as, for example, the developers still need to manage (complex) virtual machines (IaaS), or configure the services (PaaS and SaaS).

2.2 | Domain-specific languages

Domain-specific languages (DSLs) are designed to provide a notation tailored toward an application domain that is based only on the concepts and features that are relevant for the domain. DSLs enable solutions to be expressed at the same level of abstraction of the problem domain and can be of significant help in shifting the development of business information systems from software developers to a larger group of domain-experts who, despite having less technical expertise, have deeper knowledge of the domain and, therefore, if an easy-to-use, tailored tool is provided, can be much more effective. Furthermore, DSLs are much easier to learn, given their limited scope. It must be said that DSLs have specific design goals that contrast with those of general-purpose languages: DSLs are much more expressive in their domain and should exhibit minimal redundancy.

Realizing a DSL means developing a compiler that is able to read text written in that DSL, parse it, process it and eventually interpreter it, and/or generate/translate the code. With more details a DSL development requires the following phases:

- *lexical analysis*, in which the code is partitioned in *token*, where each token is a single atomic element of language;
- *syntactic analysis*, which checks that the identified tokens form a valid statement in the language. This phase produces the abstract syntax tree (AST), which is a representation of the syntactic structure of the programs;
- *semantic analysis*, which checks that the assignment of values is between compatible data types (type checking). This phase keeps track of identifiers, their types and expressions, and checks whether identifiers are declared before their use;
- *code generation*, which generates the machine code or code in another language.

3 | RELATED WORK

This section presents and discusses the research and state-of-the-art for the cloud computing service-models domain as well as an introduction to domain specific languages.

3.1 | Parallel and distributed languages for scientific computing

Here we describe several languages and frameworks that are suitable for developing scalable applications in the scientific computing (SC) research area.

Fortran is a programming language designed for numeric computation and scientific computing. *Python*[14] and *Julia* are interpreted high-level programming languages for general-purpose programming. *Limbo*[15] is a programming language intended for applications running distributed systems on small computers. *Chapel*[16] is a portable programming language designed for productive parallel computing on large-scale systems. *Cilk*[17] is a C/C++ extension designed for multithreaded parallel computing.

Apache Hadoop[18] is a framework that enables the distributed processing of large datasets across clusters of computers using a simple programming model. *Apache Spark*[19] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R.

MPI[20] is a standardized message passing API useful for the development of distributed applications. Open Computing Language (*OpenCL*) is the open, royalty-free standard for crossplatform, parallel programming of diverse processors available in personal computers, servers, mobile devices, and embedded platforms. *OpenMP*[22] is an application programming interface (API) that supports multi-platform-shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows.

Swift[23] is a featured data-flow oriented coarse grained scripting language, which is designed for scientists, engineers, and statisticians that need to execute domain-specific application programs many times on large collections of file-based data.

Swift/T[24] is the high-performance computing version of Swift languages, in which the Swift programs are translated in MPI-based programs to be executed on HPC systems. Swift and Swift/T provide setup on cloud IaaS*. *OpenMole*[25] offers tools to run, explore, diagnose, and optimize numerical models, taking advantage of distributed computing environments.

3.2 | Cloud computing service-models

Serverless computing service-model (or function-as-a-service, FaaS)[26-28] answers to the needs of new scalable price-effective cloud applications, by providing an easy framework for deploying extremely scalable, functionally partitioned applications. FaaS enables developers to run their back-end applications on complex computing systems, without a thorough knowledge of the management and configuration of such systems. Indeed, using FaaS, the user is able to execute independent piece of code (functions), written in different languages, over the cloud infrastructure, without taking care about which is and what kind of configuration has the server running the code. FaaS service-model architecture is events-triggered, which means that developers must deploy the functions on the cloud infrastructure, and those functions are executed in response to events generated on the cloud infrastructure (eg, insert a new record in a database, send a message on a queue). Our proposal is guided by the vision to adopt this service-model in a different context that is for computing-intensive applications. In this context, scaling among the computing infrastructure is a design requirements, which enable to solve scientific problems, defined by a heavy computational load, that cannot be solved using sequential computing.

3.3 | Domain-specific languages

DSL developers may benefit by using a DSL development framework, which helps them to develop each implementation phase. One of these is *JetBrains MPS*,[29] which is a tool for designing domain-specific languages, based on language-oriented programming. It uses projectional editing, which enables overcoming the limits of language parsers and building DSL editors.

Another popular tool is also *Xtext*[10] is a framework for the development of programming languages and domain-specific languages. With Xtext, the DSL developers need only to provide a grammar specification for the novel language. Xtext, starting from this definition, provides a full infrastructure, including lexer, parser, the AST model, linker, type-checker, compiler as well as editing support for Eclipse and/or any editor that supports the language server protocol as well as web browsers. However, every single aspect can be customized by the DSL developer. For the above reasons, we have used Xtext to develop our source-to-source compiler for Fly.

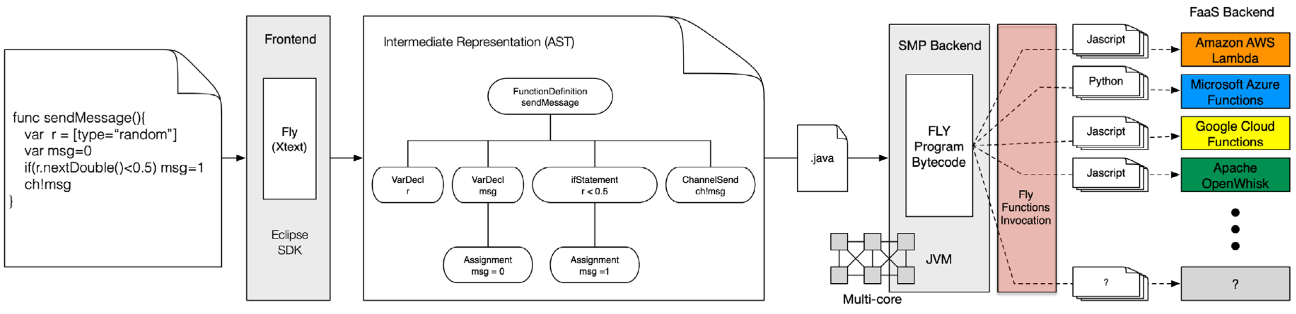
4 | FLY DESIGN

Fly has been designed in order to enable the domain developers (ie, domain experts with limited knowledge about complex parallel and distributed systems) to develop their applications exploiting data and task parallelism on a FaaS architecture. This is achieved by a rich language that provides domain-specific constructs that allow the developers to easily interact with different FaaS computing backends, using an abstraction of the cloud providers services. The Fly principal design goals are:

- *expressiveness*, in deploying large-scale scientific workflow-based applications;
- *programming usability,* writing programs for Fly should be straightforward for domain experts, while the interaction with the cloud environment should be completely transparent; the users do not need to know the cloud providers services;
- *scalability,* either on SMP architectures or cloud computing infrastructures that support FaaS.

Fly provides implicit support for parallel and distributed computing paradigms and memory locality, enabling the users to manage and elaborate data on a cloud environment without the effort of knowing all the details behind cloud providers API. A Fly program is executable either on an SMP or a cloud infrastructure (supporting FaaS) without a deep knowledge of the underlying computing resources. Fly is translated in

* swift-lang.org/tutorials/cloud/tutorial.html



```
func sendMessage(){
  var  r = [type="random"]
  var msg=0
  if(r.nextDouble()<0.5) msg=1
  ch!msg
}
```

FIGURE 1 Fly compilation workflow

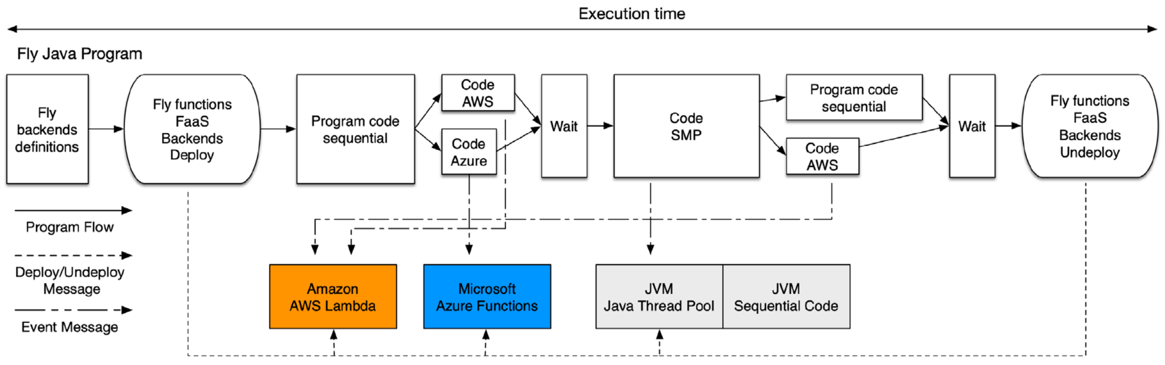


FIGURE 2 Fly execution workflow

Java code and is able to automatically exploit the computing resources available that better fit its computation requirements. The main innovative aspect of Fly is represented by the concept of Fly *function*. A Fly function can be seen as an independent block of code that can be executed concurrently. Fly functions can be executed in sequential mode, in parallel on an SMP or on a FaaS backend. The language provides programming constructs for functions definition, execution, synchronization, and communication. Communication among different environments/backends is obtained through some virtual communication path named *channels*. Along these lines, Fly has been designed as an enhanced scripting language and is composed of a sequence of standard instructions integrated with a number of Fly functions invocation, which communicates via channels.

Figure 1 depicts the Fly compilation workflow. On the left side, the Fly program is given in input to the compiler (written using XText). The intermediate AST representation is translated into a Java program. Each Fly function is translated into different executable codes (one for each backend). Therefore, Fly provides compiled functions code that can be executed on each cloud infrastructure backend (see the right side of Figure 1).

In detail, in Figure 2, we show a general execution flow of a Fly program along the execution time. First, the program initializes all the backends required by the Fly code and *deploys* the generated code on the corresponding backend. We notice that the Fly functions are already compiled when the main Fly program is executed, thereby avoiding run-time compilation overheads. After these initialization steps, the main program is executed following the Fly code instructions. Each time the **fly** keyword is used, the program generates *events* on the corresponding SMP and/or FaaS backend, in order to execute the Fly functions. Fly supports synchronous and asynchronous execution models. A simple example of a Fly program, which computes a PI estimation through the Monte Carlo Method, is presented in Reference 9.

5 | FLY LANGUAGE IMPROVEMENTS

Fly language definition has been described in Reference 9. Here we are going to present the improvements and changes that have been introduced in the current version of the language design. The principles of the language design have not changed, but we have improved several domain object definitions and we have introduced other basic data structures. Following we discuss, in details, the list of improvements compared with the previous language programming model (notice that, in the following definitions, the question mark is used for optional parameters):

- *Array basic type*. We have introduced the array type. A Fly array is a collection of homogeneous basic types laid on a tabular form having one, two, or three dimensions. The declaration of an array is described in the following:

```
var ID =  Integer|Float|Boolean|String[N][M][K],
```

  where N, M, and K denotes the length (ie, number of elements) of each array dimension.

- *Objects*. Fly objects are declared using braces ({ … }) and the mapping between object IDs and values are made using the assignment (=) symbol. For instance:

```
var ID={?ID=VALUE,?ID=VALUE,...}  (eg, var m={i=0,s="text"})
```

- *Domain objects*. The syntax of the domain objects declaration has changed in order to be compliant with standard types. Now the objects are declared using square brackets ([…]). For instance, in the following, we provide the declaration of the `Environment` object:

```
var ID = [type=("smp|aws|aws-debug|azure"), threads=Integer, language=String,
          memory=Integer(MB), seconds=Integer, region=String, user=String,
         (cloud-provider-parameters)]
```

  where *cloud-provider-parameters* provides several options defined according the considered cloud backend (see Fly documentation on GitHub for additional parameters of Amazon AWS and Microsoft Azure backends). The declarations of other domain types, such as *File*, *Random*, and *Channels*, respect the same syntax and appear as follows:

```
var ID = [type="file|dataframe|image|json", path=String, ?sep=String] ?on ID
var ID = [type="channel"] on ID
var ID = [type="random" seed=Integer]
```

- *Parallel/distributed statement*. We have extended the definition of the Fly execution statement in order to support the new type *Array*, as follows:

```
fly ID in Range|Object|File|Array ?(by row|col|square) on ID then ID thenall ID
```

  Moreover, when the range is specified using the *Array* type, a new option, defined by the keyword **by** has been introduced. This option enables the programmers to define the partitioning method to be used for partitioning the input array onto input subarrays for functions.

- *Native code statement*. The native statement, defined by the keyword **native**, enables the programmers to invoke directly native code in the Fly functions. In the previous version of Fly, only JavaScript code was supported. The novel version of Fly supports also Python language.

6 | FLY COMPILER

Fly source-to-source compiler and its documentation are available on a public GitHub repository[†], released under Alpha 1.5 version. An implementation of the language grammar and code generators for the SMP and Amazon AWS and Microsoft Azure FaaS backends have been developed. Fly has been deployed in order to generate a Java program, which is able to support several backends. We decided to design Fly language compiler using *Xtext*[10] framework, which enables the user to create JVM based DSL. The Fly code is translated in a pure Java program that exploits cloud APIs in

[†]github.com/spagnuolocarmine/FLY-language

TABLE 1 Code generation mapping of Fly types

	Computing environments/backends		
		FaaS	
Name	SMP	JavaScript	Python
Integer	java.lang.Integer	Number	Numbers
Float	java.lang.Double	Number	Numbers
Boolean	java.lang.Boolean	Boolean	Boolens
String	java.lang.String	String	String
Object	java.util.HashMap < String,String>	Object	Dict
Environment	java.util.concurrent.ScheduledThreadPoolExecutor		
	com.amazonaws.auth.*;		
	com.amazonaws.services.s3.*;		
	com.amazonaws.services.lambda.*;		
	com.amazonaws.services.indentitymanagement.*;		
	com.microsoft.azure.AzureEnvironment;		
	com.microsoft.azure.management.*;		
	com.microsoft.azure.credentials.*;		
Channel	java.util.concurrent. LinkedTransferQueue< String>	aws-sqs	aws-boto3-sqs
	com.amazonaws.services.sqs.*;	azure-storage-queue	azure-storage-queue
	com.microsoft.azure.storage.queue.*;		
Random	java.util.Random	Math.random()	Random package
File	java.io.FileInputStream	File I/O	File I/O
Dataframe	tech.tablesaw.api.Table[30]	dataframe-js[31]	Pandas Library
Image	javax.imageio. stream.FileImageInputStream	File I/O	File I/O

order to use FaaS services. Xtext leverages the powerful ANTLR parser, which implements an LL parser. We recognize that LL-parsing has some drawbacks: LR parsers are more general than LL parsers, which do not allow left recursive grammars. Moreover, LR parsers are, in general, more efficient. On the other hand, LL parsers have significant advantages over LR algorithms with respect to readability, debuggability, and error recovery, as well as that is much simpler to understand.

Xtext provides an initial grammar specification, a full infrastructure including lexer, parser, AST model, linker, type-checker, compiler as well as editing support for Eclipse and/or any editor that supports the language server protocol as well as web browsers. We designed an LL grammar for Fly language, which provides the complete language definition, presented in Cordasco et al[9] and discussed in Section 4. Xtext has been also used to develop a code generator that, given the intermediate AST program representation (the output of the first compilation phase), generates a Fly Java program. The Fly Java program execution flow is shown in Figure 2 and has been described in Section 4. The code generation phase is the core of our compiler, it generates different codes according to the backend where the Fly code has to be executed.

1. SMP backend. A Java thread pool is used to implement the backend for the SMP architecture. The Fly main program is executed as Java code on a JVM, which executes also the SMP backend. We assume that the underlying hardware provides at least two physical cores, one to run the main program and one that acts as SMP backend. Therefore, the Fly functions are translated in pure Java code. In details, all Fly types are mapped on a particular Java type, as described in Table 1, while the Fly functionality are provided exploiting the Java language.

2. FaaS backends. The FaaS computing backends have been developed using the API of each cloud provider, as described in Table 1 we exploited a particular cloud service, according to each provider and computing language API, for each type and functionality of Fly. Our Fly compiler translates each Fly function in Java, JavaScript, or Python languages, automatically according to the computing backend where the function will be executed during the workflow. For each backend and each Fly function, the compiler generates a deploying package containing the source code and libraries structured according to the destination backend. The Fly deploying phase on each cloud backends exploits the cloud client command-line interface

(CLI): AmazonAWS CLI[‡] and Microsoft Azure CLI[§]. An important aspect to notice is that we invoke (or trig) the functions by using asynchronous HTTP POST requests, which, according to experimental results, ensures the shortest invoking latency.

Finally, the compiler produces a Java Maven project including all dependencies, the Fly Java class program (named as the Fly source code), and the code of functions. The command `mvn package`, which generates am executable *JAR* file, is used to build the project.

6.1 | Debugging Fly applications

The Fly language is a workflow language that enables to execute functions on remote/distributed backends, and the programmers have no control about the execution and the memory of the remote computing backend. For this reason, debugging Fly applications is very complex, considering that to analyze functions logs the programmers have to access each cloud provider used in the computation using specific API of Web portals. Moreover, a possible incorrect execution has a considerable cost. Indeed, assuming that a Fly application generates errors only in some rare cases (ie, memory requirements or index out of bounds), then according to FaaS execution model all incorrect executed functions must be paid as correct executed ones.

For the reasoning above, Fly provides a particular debugging backend for the Amazon AWS cloud provider. By using this backend the programmer is able to execute and test the code on a virtual local environment that emulates the Amazon AWS ecosystem. The programmer has to define the AWS backend using a particular environment type `aws-debug`. We developed the AWS debugging environment using Local Stack,[32] which provides an easy-to-use test/mocking framework for developing cloud applications. In particular, the AWS debugging environment enables the programmers to test Fly applications onto the local machine using the same functionality and APIs as the real AWS cloud environment. Local Stack framework exploits the containers virtualization technology to emulate the AWS environment and for this reason, in order to exploit the AWS debugging environment, the Docker[¶] application needs to be installed on the local machine.

Listing 1: Fly word count

```
var local = [type="smp",threads=4]
var aws = [type ="aws",user="default", access_key="k1", secret_key="k2", region="us-east-2",
           language="python3.6", threads=1000,memory=128, seconds=300]
var ch = [type="channel"] on aws
var dir = [type="file", path="data"]
for f in dir{
    var file = [type="file", path=f] on aws
}
func count(files){
        var counts = {}
        for f in files{
                var file = [type="file", path=f]
                for strs in file{
                        var words = strs.split(" ")
                        for w in words{
                                var word = w.v as String
                                if(not counts.containsKey(word))
                   counts[word] = 1
        else
                    counts[word]=counts[word]+1
    }}}
    ch!counts on aws }
func reduce(){
        var total_counts = {}
        for i in [0:1000]{
                var res = ch? as Object
                for w in res{
                        var word = w.k as String
                        var tmp = w.v as Integer
                        if(not total_counts.containsKey(word))
                                total_counts[word] = tmp
                                else
```

‡ aws.amazon.com/cli
§ docs.microsoft.com/en-us/cli/azure
¶ www.docker.com

```
33
34
35                                                     total_counts[word] = total_counts[word] + tmp
36                      }
37
38
39
40              }
41              for value in total_counts{
42                      println value.k+ " "+ value.v
43              }
44      }
45      var t_start = time()
46      fly count in dir on aws thenall reduce
47      println time(t_start)
```

7 | LANGUAGE EVALUATION

We evaluated Fly efficiency analyzing the performance of Fly on the word count problem, a popular benchmark for distributed workflow computing framework, which consists of listing the frequencies of different words in a set of text files. Listing 1 shows the Fly code for the word count problem. Initially, the input files are loaded on an AWS S3 bucket (lines 6-8). Lines 10-23 show the code of the `count` function, which takes as a parameter a list of string (files path), which are the AWS endpoint of the input files on the S3 bucket, and iteratively counts the number of occurrences of each word. Then, a message containing the Fly object that maps each word with the corresponding frequency is sent on the channel `ch` (line 23). Lines 24-40 provide a `reduce` function that will be executed on the local machine. The `reduce` function reads from the channel `ch` the words frequencies, computed by the Fly functions, and generates a single occurrences list. Finally, line 41 shows the Fly construct to execute the process on the backend AWS, which executes at most 1000 concurrent `count` functions on AWS. Each function receives a balanced subset of the files contained in the local directory *data*. When all Fly functions are completed, the `reduce` function will be executed (synchronization is obtained thanks to the synchronous Fly construct, line 42).

We performed several benchmarks of the word count program, in order to explore Fly capabilities in terms of performance and cost-effectiveness. Table 2 shows the results obtained running nine different input size (rows) and varying the execution backends (columns). Specifically, we tested three files size: 125, 250, and 500 MB, with three cardinalities of files: 500, 1500, and 2000. Therefore, we analyzed, in the worst-case, approximately 1 TB of data.

We compared the performance of the sequential executions (Fly Java) against two parallel executions, using 4 (SMP-4) and 64 (SMP-64) cores, and a distributed execution using 1000 concurrent Fly functions at time (AWS-64). Furthermore, we evaluated the performances of the previous configuration of Fly compared with the execution of the same task on Apache Hadoop[#], a well-known big data analytic framework (H-16). We used the Amazon AWS cloud as computing environment, in particular AW-64 exploits a *m4.16xlarge* EC2 instance[**], which provides 64 virtual cores. We measured the total computation time (in hours), without considering the time required to load the file on the computing environment.

We performed also the same experiment using an *m4.xlarge* EC2 instance, which provides four virtual cores, obtaining similar results in terms of computational time and cost. There are several possible explanations for this outcome. First, the computing time for Fly functions is the same in both the configurations. Moreover, the greatest number of cores of AWS-64 are exploited only for the *reduce* functions, which is not a computationally intensive task for this task.

The H-16 configuration refers to the same task executed on an Apache Hadoop cluster machine, running exploiting the Amazon AWS EC2 and Elastic Map Reduce (EMR) services. We used a cluster of 16 computing nodes and one master node, for a total of 128 virtual cores. Each node is a *m4.xlarge* EC2 instance, which costs 0.80$ hourly plus the cost of EMR running on it that is 0.24$ hourly, hence the total cost of the cluster is 16.64$ hourly. We notice that this cluster configuration requires, on average, about 20 minutes of cold starting time. For this reason, we added to the cost of each experiment the price to start the cluster, but we present, in Table 2, only the time for running the experiment, also without considering the time to load the file on the HDFS.

The table shows the obtained speed-up with respect to the sequential execution, and the total cost (in dollars) considering the cost of the Fly functions and the cost of the EC2 instance machine running the Fly main program. As highlighted, in the table, the best performance has been obtained by the backend that exploit FaaS (AWS-64). On the other hand, H-16 is the most expensive experiment, while SMP-4 results as the cheaper execution configuration. As shown, the performance of FaaS backend is the most effective, compared with the parallel backend and the Apache Hadoop, also considering their cost for running on the Amazon AWS. It is important to notice that we adjust the number of nodes composing the Hadoop cluster machine to be comparable, in terms of cost, to the configuration execution using FaaS.

[#] hadoop.apache.org
[**] aws.amazon.com/ec2/instance-types

TABLE 2 Fly words count backends comparison: T (time, h) hours vs S (speed-up with respect to Fly Java) vs C (costs, $)

Configurations			Fly Java		SMP-4			SMP-64			AWS-64			H-16		
MB	N	Tot. GB	T (h)	C ($)	T (h)	S	C ($)	T (h)	S	C ($)	T(h)	S	C ($)	T (h)	S	C ($)
125	500	~ 62	0.751	0.2	0.324	2.3	**0.1**	0.016	44.9	0.1	**0.015**	**47.7**	0.2	0.061	12.3	6.6
125	1500	~ 187	3.036	0.6	1.042	2.9	**0.2**	0.089	34.1	0.3	**0.018**	**166.6**	0.5	0.103	29.5	7.3
125	2000	~ 250	3.711	0.7	1.351	2.7	**0.3**	0.124	30.0	0.4	**0.018**	**196.1**	1.0	0.106	35.0	7.3
250	500	~ 125	1.518	0.3	0.670	2.3	**0.1**	0.062	24.3	0.2	**0.018**	**82.9**	0.5	0.560	27.1	6.5
250	1500	~ 375	4.659	0.9	2.069	2.3	**0.4**	0.177	26.3	0.6	**0.023**	**198.6**	1.5	0.146	29.9	8.1
250	2000	~ 500	6.060	1.2	2.738	2.2	**0.5**	0.234	26.0	0.8	**0.024**	**250.6**	2.0	0.205	29.6	8.9
500	500	~ 250	2.911	0.6	1.323	2.2	**0.3**	0.116	25.0	0.4	**0.023**	**121.8**	1.0	0.106	27.5	7.3
500	1500	~ 750	9.017	1.8	3.981	2.3	**0.8**	0.359	25.1	1.2	**0.034**	**261.3**	2.5	0.302	29.9	10.6
500	2000	~ 1000	10.918	2.2	5.065	2.2	**1.1**	0.437	25.0	1.4	**0.035**	**307.6**	3.8	0.400	27.3	12.2

7.1 | Fly as a distributed computing framework

It is worth mentioning that our experiments on H-16 are consistently with the complete analysis presented in Reference 33. The authors detail the performance of several benchmarks (such as word count, K-means, etc), on several cloud-based cluster machines. In particular, they exploited an Amazon EC2 cluster composed by 8 *r5.24xlarge* instance type, where each machine is equipped with 96 cores and 768 GB of memory. The described results provide a good comparison for our purposes. As described by the authors, Hadoop spends about 0.3 hours (18.3 minutes) for counting 1.6TB of data. Analyzing the cost for this experiment, we have that each EC2 instance cost 6.048$ per hour, while the *Elastic MapReduce* service, running on this kind of instance, costs 0.26$ per hour; hence, the cluster total cost for 0.3 hours is about 14.60$. We can compare this data with our results on Fly, which computes the word count on 1 TB of data with a cost of 3.8$. Moreover, our system is a clear advance in terms of performance by considering also the opportunity to exploit multicloud, which enables the programmer to easily increase the amount of computational resources by exploiting different cloud providers in the same application. Moreover, the paid price for our solution is extremely cheaper (about four times less). We plan to investigate a more detailed comparison by running this benchmark varying the computing backends on different cloud providers, like Microsoft Azure, and comparing the cost and performance of Apache Hadoop and Apache Spark running on the same cloud provider.

8 | USE CASES

In the following, we provide two well-known scientific computing use cases developed in Fly: *supervised machine learning classification* and *sequence alignment*. For both the problems, we identified a well-known algorithm that has been developed and tested using Fly. With more details, we developed a Fly implementation of the linear *k*-nearest neighbors (*k*-NN) algorithm, which is one of the most used learning algorithms as well as the Smith-Waterman algorithm, a dynamic programming algorithm for determining the optimal local alignment between two strings according to a gap-scoring scheme.

We tested the performance of *k*-NN algorithm using the *20 Newsgroups (20NG)*[34] dataset, which comprises a collection of approximately 20 000 newsgroup documents. The data are organized into 20 different newsgroups, each corresponding to a different topic. The dataset is composed of 19 300 instances, described by 1006 attributes. We used 9000 instances for the training set, while a combination of instances in the dataset plus some synthetic instances has been used for the test sets. For the Smith-Waterman algorithm, we performed the alignment of several synthetic protein sequences using the BLOSUM50 matrix as a gap-scoring scheme.

The experimental case studies evaluation has been performed using six computing environments, as described in Section 7. Five environments exploit Fly: Java, SMP-4, SMP-64, AWS-4, and AWS-64. Moreover, we considered also a baseline environment represented by a pure Java implementation, running on a single thread machine. In order to perform a cost comparison of each environment, we executed all the tests on AWS using AWS EC2 instances. In particular, we used the r5.large (2 vCPU, 16GB RAM), r5.xlarge (4 vCPU, 32GB RAM) , and m4.16xlarge (64 vCPU, 256GB RAM) instances to run respectively the tests that requires 1, 4, and 64 threads. Fly functions have been deployed on AWS Lambda, configured with 3 GB of memory. It is worth to mention that tests performed on different environments always provide the same result (this was expected since in both the case we did not modify the algorithms).

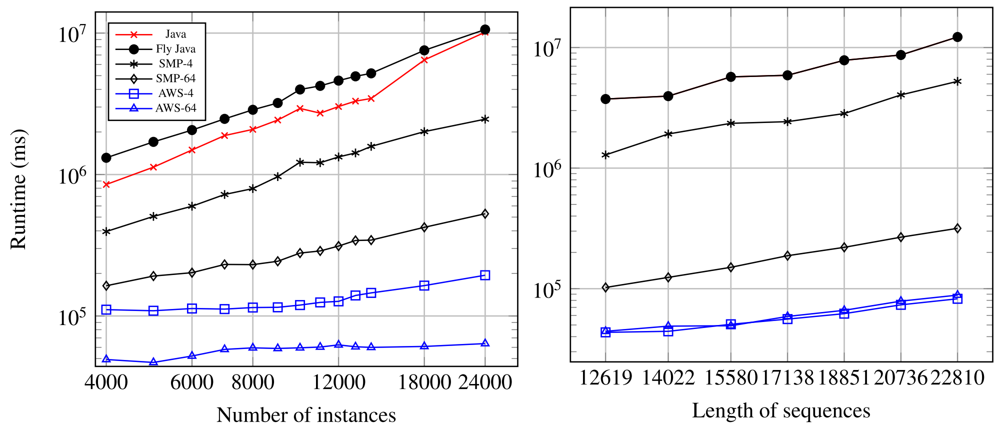


FIGURE 3 Fly scalability on *k*-NN (left) and Smith-Waterman (right)

TABLE 3 $\frac{\text{Speed-up}}{\text{Cost}}$ comparison

	k-NN				Smith-Waterman			
	4k	**8k**	**12k**	**24k**	**≈ 12k**	**≈ 16k**	**≈ 19k**	**≈ 23k**
SMP-4	1.76	2.61	1.96	6.46	**22.21**	15.49	20.12	14.30
SMP-64	1.06	3.22	3.70	14.55	138.29	149.23	131.19	**154.47**
AWS-4	**0.34**	1.91	3.48	21.85	60.85	76.46	83.28	92.46
WS-64	0.72	3.46	6.66	**63.58**	56.26	78.26	76.26	94.88

k-NN algorithm

We developed several experiments varying the number of testing instances and the computing environment. For each test, we compute the total computation time and its cost. Figure 3 (left) presents the performance in terms of strong scalability on the 20NG dataset for each computing environment (series). It is worth to mention that the implementation of the algorithm in Pure Java and Fly provide the same accuracy. Therefore in the following, we compare the different configurations only in terms of computation time. The total computing times are shown in milliseconds (*y*-axis, log scale) varying the number of instances in the test set (*x*-axis, log scale) from 4000 to 24 000. As shown in the figure, the best performance is always achieved by the configurations that use the FaaS backend. The pure Java code is slightly better than the SMP using one thread (this slow-down represents the overhead due to Fly), but it is outperformed by the SMP configuration with four threads. The overall billing cost for the larger test set ranges from ≈ 0.35$ for the SMP-4 configuration (ie, the cost of the r5.xlarge instance) to ≈ 0.85 $ for the AWS-64 configuration (ie, the cost for the m4.16xlarge instance plus the cost for the AWS Lambda functions). We notice also that the pure Java implementation is more expensive (≈ 0.37$) than the SMP-4 configuration.

Smith-Waterman algorithm

Similarly to the *k*-NN evaluation, we developed several experiments using the Smith-Waterman algorithm, varying the length of the sequences to be aligned. Figure 3 (right) presents the performance in terms of scalability on the synthetic protein sequences for each computing environment (series). As for the previous case, the best performance is always achieved by the configurations that exploit the FaaS backend but in this experiment, the capability of the machine that executes the Fly main program does not affect the performance. Moreover, the overhead of Fly with respect pure Java code is negligible. The overall billing cost for the larger test set ranges from ≈ 0.18$ for the SMP-4 configuration to ≈ 1.56$ AWS-64 one. The Java pure implementation cost is ≈ 1.12$. Overall the FaaS backend always provides a speed up to the performance that ranges from 17× (*k*-NN with 4000 instances) to 160× (*k*-NN with 24 000 instances).

Speed-up vs cost comparison

Table 3 provides a comparison in terms of the ratio between the speed-up obtained and its extra cost, respect to the cost of the environment used for the pure Java implementation. The table shows that, except for a small number of instances, the ratio is always greater than 1, meaning that the benefit of using Fly in terms of speed-up is always greater than their extra costs.

9 | CONCLUSION

This article discloses Fly, a domain-specific language, which aims at reconciling cloud and high-performance computing paradigms by providing a powerful, effective, and pricing-efficient tool for the development of scalable workflow-based scientific computing applications on any FaaS environment enabling, at the same time, the users to exploit different cloud systems with the same application (multicloud systems). Besides presenting the motivations, the language programming model, and the source-to-source compiler, in this article, we provide (i) an improved version of the language, which now enables to exploit the capabilities of different cloud providers as well as the functions translation in Python, and the support to Python native code; (ii) a performance evaluation of Fly on a popular benchmark for distributed workflow computing frameworks; and (iii) a collection of three algorithm examples developed in Fly with an analysis of their performance results on the Amazon AWS Lamba. Future works and studies are already planned to improve Fly language definition including library and namespaces definitions, compiler optimizations (according to the FaaS execution model), derived data types (as Java class), and data visualizations, in addition to a performance comparison of different clouds providers. Furthermore, Fly will provide specific libraries/integration of algorithms (optimized for FaaS), such as machine learning, graph/hypergraphs algorithms, and data mining, and will integrate functionalities for developing parallel and/or distributed discrete-event simulations.[35]

ACKNOWLEDGEMENT

An preliminary version of this article was presented at the 1st International Workshop on Parallel Programming Models in High-Performance Cloud, ParaMo 2019.[9]

ORCID

Gennaro Cordasco https://orcid.org/0000-0001-9148-9769

1. Shawish A, Salama M. Cloud computing: Paradigms and technologies. *Inter-cooperative collective intelligence: Techniques and applications*; 2014.
2. Nadareishvili I, Mitra R, McLarty M, Amundsen M. *Microservice Architecture: Aligning Principles, Practices, and Culture*; 2016.
3. Cordasco G, Scarano V, Spagnuolo C. Distributed MASON: a scalable distributed multi-agent simulation environment. *Simulat Modell Pract Theory*. 2018;89:15-34.
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107-113.
5. Ferrer AJ, Pérez DG, González RS. Multi-cloud platform-as-a-service model, functionalities and approaches. *Proc Comput Sci*. 2016;97:63-72.
6. Brugere I, Gallagher B, Berger-Wolf TY. Network structure inference, a survey: motivations, methods, and applications. *ACM Comput Surv (CSUR)*. 2018;51(2):1-39.
7. Cordasco G, De Chiara R, Raia F, Scarano V, Spagnuolo C, Vicidomini L. Designing computational steering facilities for distributed agent based simulations. Proceedings of the 2013 ACM SIGSIM Principles of Advanced Discrete Simulation; 2013.
8. Carillo M, Cordasco G, Serrapica F, Scarano V, Spagnuolo C, Szufel P. Distributed simulation optimization and parameter exploration framework for the cloud. *Simulat Modell Pract Theory*. 2018;83:108-123.
9. Cordasco G, D'Auria M, Negro A, Scarano V, Spagnuolo C. Towards a Domain-specific language for scientific computing on multicloud systems. Paper presented at: Proceedings of the 1st I. Workshop on Parallel Programming Models in High-Performance Cloud; 2019.
10. Project E. *Xtext, language engineering for everyone!*; 2019. Accessed December 2019.
11. Inc. A. *Amazon web services - AWS Lambda*; 2019. Accessed December 2019.
12. Corporation M. Microsoft Azure functions; 2019. Accessed December 2019.
13. Hwang H, Fox GC, Dongarra JJ. *Distributed and cloud computing from parallel processing to the Internet of Things*; 2013.
14. Rossum G. Python reference manual. Technical report; 1995.
15. DM Ritchie. The Limbo programming language. *Inferno Programmer's Manual 2*; 2018.
16. Chamberlain B, Callahan D, Zima HP. Parallel programmability and the chapel language. Int J High Perform Comput Appl 2007.
17. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multi threaded runtime system. PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, SIGPLAN. 1995; 7-216.
18. White T. *Hadoop: The Definitive Guide*. Sebastopol, California: O'Reilly Media, Inc; 2009.
19. Zaharia M, Xin RS, Wendell P, et al. Apache spark: a unified engine for big data processing. *Commun ACM*. 2016;59(11):56-65.
20. MPI F. MPI: a message-passing interface. Technical report; 1994.
21. Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. *Search Results Web results IEEE Design & Test of Computers*. 2010;12(3):66-73.
22. Dagum L, Menon R. OpenMP: an industry-standard API for shared-memory programming. *IEEE Comput Sci Eng*. 1998;5(1):46-55.
23. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, Foster I. Swift: a language for distributed parallel Scriptin. *Parallel Comput*. 2011;37(9):633-652.
24. Wozniak J, Armstrong T, Wilde M, Katz DS, Lusk E, Foster I. Swift/T: large-scale application composition via distributed-memory dataflow processing. Paper presented at: Proceedings of the 13th IEEE/ACM International Symposium Cluster, Cloud and Grid Computing; 2013.
25. Reuillon R, Leclaire M, Rey-Coyrehourcq S. OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Gener Comput Syst*. 2013;29(8):1981-1990.
26. Baldini I, Castro PC, Shih-Ping CK, et al. Serverless computing: current trends and open problems. *CoRR abs/1706.03178 (arXiv)*. 2017.
27. McGrath G, Brenner PR. Serverless computing: design, implementation, and performance. Paper presented at: Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW); 2017.

28. Stigler M. *Beginning serverless computing: developing with Amazon web services, microsoft azure, and Google cloud*; 2017.
29. JetBrains. MPS: domain-specific language creator by JetBrains; 2019. Accessed December 2019.
30. Tablesaw. *JTablesaw is a Java dataframe similar to Pandas in Python, and the R data frame*; 2019. Accessed December 2019.
31. Dataframe JS. Dataframe-js provides an immutable data structure for javascript and datascience; 2019. Accessed December 2019.
32. LocalStack. LocalStack 2019. Accessed December 2019.
33. Lee H, Fox G. Big data benchmarks of high-performance storage systems on commercial bare metal clouds. Paper presented at: Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD); 2019.
34. The 20 newsgroup dataset. 2019. Accessed December 2019.
35. Cordasco G, Spagnuolo C, Scarano V. Toward the new version of D-MASON: Efficiency, effectiveness and correctness in parallel and distributed agent-based simulations. Paper presented at: Proceedings of the IEEE 30th International Parallel and Distributed Processing Symposium; 2016; IEEE.

How to cite this article: Cordasco G, D'Auria M, Negro A, Scarano V, Spagnuolo C. Toward a domain-specific language for scientific workflow-based applications on multicloud system. *Concurrency Computat Pract Exper*. 2020;e5802. https://doi.org/10.1002/cpe.5802