Towards Evolutionary Machine Learning Comparison, Competition, and Collaboration with a Multi-Cloud Platform

Pasquale Salza
University of Salerno, Italy
psalza@unisa.it

Erik Hemberg
Massachusetts Institute of Technology, USA
hembergerik@csail.mit.edu

Filomena Ferrucci
University of Salerno, Italy
fferrucci@unisa.it

Una-May O'Reilly
Massachusetts Institute of Technology, USA
unamay@csail.mit.edu

ABSTRACT

We present *cCube*, an open source architecture used to automatically create an application of one or more Evolutionary Machine Learning (EML) classification algorithms that can be deployed to the cloud with automatic data factorization, training, result filtering and fusion. *cCube* enables automated EML classification algorithms comparison, competition and multi-party collaboration. It can be used by an algorithm developer, a community working together or a black box user of EML classification. It requires minimal extra code to cloud-scale shared-memory implementations. It employs a microservices architecture and software containers into which user code is integrated allowing to access to the full benefits of cloud computing, e.g., on demand and elastic computing, while not committing (code or patronage) to a specific cloud provider such as Amazon Web Services or OpenStack. We demonstrate *cCube*, straddling our application across two different cloud providers and replicate the collaborative activity at zero cost.

CCS CONCEPTS

•**Computing methodologies** → **Machine learning;** •**Theory of computation** → *Evolutionary algorithms;* •**Software and its engineering** → *Cloud computing;*

KEYWORDS

Evolutionary Machine Learning, Microservices, Cloud Computing

ACM Reference format:
Pasquale Salza, Erik Hemberg, Filomena Ferrucci, and Una-May O'Reilly. 2017. Towards Evolutionary Machine Learning Comparison, Competition, and Collaboration with a Multi-Cloud Platform . In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082474

1 INTRODUCTION

Evolutionary Machine Learning (EML) is a prolific research field and new algorithms are continually being developed. Progress as it occurs in any scientific community involves 3 activities: (1) *comparison*, new methods are compared to existing ones, (2) *competition*, from comparison, a state of art arises, and (3) *collaboration*, because of trade-offs in methods, after comparison it is often advantageous to combine algorithms for a collaborative result, e.g., generate an ensemble model [14]. Elements (1) and (2) are essential whereas (3) is desirable. Indeed, the challenge of collaboration is an impediment to the software design, implementation and accessibility. Comparison requires the coordination of other EML algorithms, not always personally developed, in being set up, in running large number of experiments using many resources, and in lasting a long duration. Large datasets imply that the EML developer often uses a personal shared-memory computer to develop an EML algorithm that, once accurate and fast enough on a subsample of the data, has to next be deployed at scale. Scaling requires complex provisioning on distributed computational units, e.g., creating a commercial cloud account on *Amazon Web Services*[1] (*AWS*), setting up the authorization keys, data flow management, console-based resource administration, distributed experiment setup, results collection, compilation, etc.

An existing solution toward making disciplined and scalable EML comparison and collaboration more effortless is FCUBE[2] [1], which introduced a "Bring Your Own Learner" concept for comparison, competition and collaboration. Despite being noteworthy, FCUBE nonetheless is "locked" into one cloud provider, i.e., AWS, and lacks both automation and robust fault tolerance.

We present *cCube* (compare, compete, collaborate), an open source architecture that helps its user develop an application that deploys EML algorithms to the cloud. The source code is shared at the address https://github.com/ccube-eml under the terms of the *MIT License*[3].

As with FCUBE, *cCube* supports competition and collaboration with filter, factor and fusing. A *cCube* EML application factors data, handles parameter configuration, tasks parallel classifier training with different algorithms, and follows training by filtering and fusing classifier results into a final ensemble model. As such, *cCube* serves 3 types of user: EML algorithm designer, multi-algorithm EML application manager and non-EML literate, i.e., "black box end users". *cCube* also supports crowdfunding, i.e., the sharing of costs by a collaborative group that wishes to execute a large multi-learner, factor, filter, and fuse application.

[1]https://aws.amazon.com
[2]https://flexgp.github.io/FCUBE
[3]https://opensource.org/licenses/MIT

In *cCube* researchers can run different EML algorithms, their own as well as others', developed in different programming languages, without inserting any code into them to accommodate cloud scaling. Instead of being monolithic, *cCube* has a *microservices* architecture [8], i.e., a suite of small services (microservices), each running its own process and communicating with lightweight protocols, e.g., HTTP resource API and message queues. It has one service for each of factorization, scheduling, learning, filtering, and fusion. Each service is independently deployable in a fully automated way making applications easier to scale and more fault tolerant. Collectively *cCube*'s services are minimally centralized and managed by an *orchestrator*. For all of its microservices, *cCube* uses lightweight runtime environments known as "software containers", or simply "containers". While all computation is performed at the hardware level with conventional hypervisor-based virtualization, the containers function at the operating system level. All containers share the same kernel of the host system, instead of each using its own virtual machine. Thus, containers are smaller and lightweight compared to an entire virtualized operating system. Moreover, they are designed to make application packaging and execute microservices easily [3]. *cCube* containers can be automatically deployed using *Docker*.

We developed a *cCube* application and demonstrate its deployment on different clouds, utilizing free resources, describing its employment on two cloud providers, using them both separately and together.

The paper is organized as follows. We start by reviewing the motivations for factoring, filtering and fusion classification and for cloud-scaling and illustrating the relevant related work in Section 2. The *cCube* platform is described in Section 3. Demonstration is in Section 4. Finally, conclusions and future work are in Section 5.

2 RELATED WORK

In the following we review the most relevant related work concerned with collaborative EML and cloud applications architectures.

2.1 Collaborative EML

Only one prior project, FCUBE, has addressed the challenge of the EML community collaboratively developing a compendium solution to a noteworthy "big data" classification problem. The project assumes individuals contribute their algorithms, called "learners", each independently written to solve the problem using a smaller sample of the dataset. The software applies a particular general decomposition called "factor", "filter", and "fuse": once contributed learners are collected, it executes them independently and in parallel by *factoring* the entire data and creating splits of the original data into feasible sizes. During training, the classifiers, i.e., models, resulting from all the learners are collected. Then FCUBE executes a step of classifiers filtering and fusion that reduces the collection before creating an ensemble-based solution. This ensemble classifier is the community's solution to the "big data" problem [1].

FCUBE learners can be completely different EML techniques, implemented in different programming languages. Its "Bring Your Own Learner" paradigm has a plug-and-play style interface that reduces programming burden on the participants. Algorithm developers are required to respect the specifications of the input/output interface, providing the two functions of classifier training (i.e., model building) and evaluation. *cCube* supports the same *Bring Your Own Learner* paradigm as FCUBE.

2.2 Cloud Applications Architectures

Cloud computing exploits a distributed memory resource model. Instead, using a shared memory model, such as GPUs or multi-threaded CPU, requires specific inter-process communication protocols to be embedded within EML software and that force the algorithms to be refactored. Cloud computing removes the inefficiency of owning physical hardware that has be provisioned for infrequent, high workloads and instead offers elastic computation as a service in the form of virtual instances, for whatever time, quantity and quality is required by a specific application.

Significant evolutionary computation work exploits cloud computing while not necessarily solving the explicit collaborative classification challenge. Confining our discussion to those most relevant to *cCube*, besides the aforementioned FCUBE, is one system that uses a synchronous storage service as pool for exchange information among population of solutions [10]. Another, *SPACE* allows the computational resources necessary for running large scale evolutionary experiments to be made available to amateur and professional researchers alike, in a scalable and cost-effective manner, directly from their web browsers [7].

cCube integrates user code into an application that runs on the cloud, in that aspect it is similar to FCUBE. FCUBE runs as a platform on the *Amazon Web Services* (*AWS*) cloud platform, using the *Amazon Elastic Compute Cloud* (*EC2*) service, or can equivalently be described as an architecture that creates an application to run on AWS. It has been impressively demonstrated on an EC2 cluster of 100 instances, using the *Higgs* dataset with 11 000 000 exemplars, running 5 different learning algorithms [1].

However, FCUBE has a number of limitations. For example, FCUBE suffers from a strict dependence on a specific cloud provider, i.e., AWS EC2. Every FCUBE startup process interacts with the Amazon API and its virtual instances are realized and replicated using a technology specifically devised by Amazon, i.e., *Amazon Machine Image* (*AMI*). A goal of *cCube* is to provide software independent cloud vendors, e.g., users can take advantage of market prices.

There are also some shortcomings in FCUBE's *Bring Your Own Learner* implementation:

- FCUBE requires manual intervention whenever a new learner is contributed and each new learner triggers a rebuild of the FCUBE AMI, since the EML algorithms needs to be explicitly declared;
- should a new EML algorithm execute in a currently unsupported programming language, that learner's execution environment on the virtual instance has to be manually configured. FCUBE's maintenance and extension process is fragile, inflexible and labour intensive;
- FCUBE requires manual intervention in the fusion step when outputs of models are collected;
- the current design does not guarantee broad scalability and true fault-tolerance for every component, since the functionalities are not clearly distinguished. The communication protocols are not reliable, e.g., FCUBE uses *Amazon*

S3 as a file-based interchange point and relies upon SSH commands.

We elaborate on how *cCube*'s microservices design addresses the shortcomings of FCUBE in later sections. In general *cCube* offers enhanced automation, robustness, support for integrated development practices and is independent of a cloud provider.

Cloud computing offers a broad spectrum of technologies from which to compose an evolutionary algorithm or system. Merelo Guervós et al. devised *SofEA*, a model for pool-based evolutionary algorithms in the cloud, an evolutionary algorithm mapped to a central *CouchDB* object store [9]. SofEA provides an asynchronous and distributed system for individuals' evaluations and genetic operators application. Later, they defined and implemented the *EvoSpace Model* [6], consisting of two main components: a repository storing the evolving population and some remote workers, which execute the actual evolutionary process. It is the first work to involve technologies on the *Platform-as-a-Service* (*PaaS*) and *Software-as-a-Service* (*SaaS*) level: *Heroku* as PaaS for the population store and *PiCloud* as SaaS for the computing operations. *cCube* takes advantage of *Docker*, *PostgreSQL*, *RabbitMQ*, and *MongoDB*.

Cloud specific development methodologies can ease the development of cloud-scaled evolutionary algorithms but, with the exception of Salza et al. [11], they have not been supported. The authors provide a workflow for a scenario in which the development, deployment and execution of distributed GAs are performed in a DevOps fashion. *cCube* follows suit with the proposal by Salza et al. [11].

Next, in Section 3 we describe *cCube* and motivate its design choices which yield cloud and development practices that are principled, systematic and robust.

3 CCUBE

The aim of *cCube* is to facilitate EML comparison, competition and collaboration. A use case that *cCube* handles is factoring, filtering and fusing. There are three possible users of *cCube*:

(1) EML researcher, using best practices for software methodology;
(2) end user, comparing EML algorithms to gain insight, selecting competing EML algorithms for best performance and collaborating with other end users as a community;
(3) *cCube* engineer, that administers, maintains and intervenes manually for the platform.

To build a *cCube* application, the EML developers copy a template from *cCube*'s repository and customize configuration, e.g., EML algorithm invocation. The developers then become end users and start the *cCube* client on their machine, provide keys for authorization, thus keeping their sensitive information local and secure. The client, through Docker, starts the application after provisioning resources, running resource discovery and set up. The engineer of *cCube* itself expands and maintains the open source code.

EML Researcher. EML developers often use a personal computer to construct an EML algorithm that is accurate and fast enough on a subsample of the available data set. *cCube* is designed to help them integrate best practices into their software development processes.

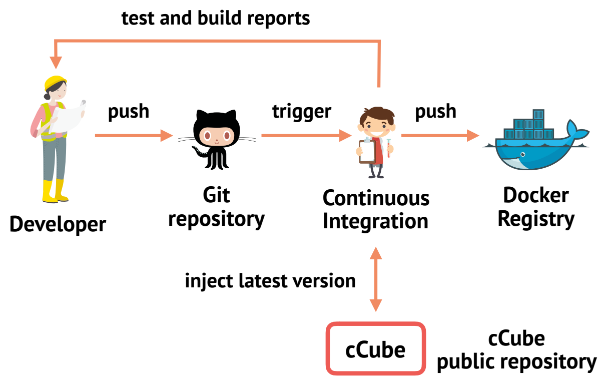


Figure 1: *cCube* is consistent with software development best practices, e.g., Continuous Integration.

Figure 1, for example, depicts an ideal and effective production work flow.

By practicing *Continuous Integration* [5], the learner developers can maintain their source code in a single repository, and access to automated testing, building and deployment processes. Then, after creating *cCube* as an extension, the developer can execute the learner in the cloud. Using Docker, we extended the development capability to allow the developer to include source code and/or to define the algorithm's execution environment, i.e., every component required for learner execution in any programming language or technology, without requiring a manual development intervention. The interaction interface is kept flexible by defining a wrapping interface. Therefore, the only information required is the path and instruction on how to execute the two phases of EML computation, (1) learning and prediction (2) filtering and fusing the trained models output path . The developer does not need to make the source code aware of *cCube*'s functionality or parallel computation. Therefore, *any* algorithm can be executed in *cCube*. Once the container is defined, the developer only needs to build and distribute it on a *Docker Registry* repository, to be downloaded, executed and replicated on demand.

cCube End User. The end user is interested in executing large-scale EML algorithms, and therefore treats *cCube* as a black box. *cCube* does provisioning and distribution of computational units using cloud accounts. The end user view of *cCube* is shown in Figure 2. As we can observe:

- the end user provides configurations for *cCube* tasks, EML algorithms, data set and compute duration;
- the *cCube* client submits requests to cloud providers and a cluster of the required number of nodes is allocated;
- *cCube* is ready to orchestrate containers and start the EML algorithms for factoring, fusion and filtering;
- when a job is submitted a *cCube* cluster pulls the Docker images for the services from the Docker registry and enqueues the tasks for the services.

In another scenario, *cCube* supports end users who collaborate by each contributing some machines they have commissioned from their own account on their cloud provider, in a sort of crowdfunding

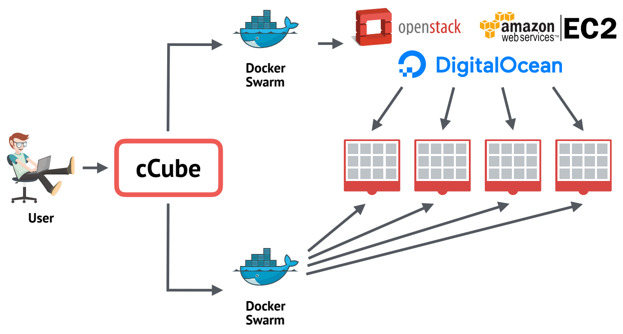


Figure 2: *cCube* for the end user perspective, a convenient black box that performs EML at scale using Docker across multiple clouds for compute.

model. In this case, *cCube* allows each user to keep their cloud credentials private, since the invocation of cloud instances happens local to each of them, i.e., on their own computer. This enables *cCube* to execute securely. The user can leave the cluster at any time.

cCube Engineer. The role of the *cCube* engineer is to extend the capabilities for comparison, competition and collaboration provided by *cCube* as well as maintain and administer the architecture. Publicly available source code and licensing are essential for these responsibilities as are minimal manual or labor intensive steps when handling administration and security tasks. A *cCube* engineer last but not least requires, e.g., expertly designed interfaces and loosely coupled code. For these reason, we took a microservices design approach with *cCube*'s architecture.

3.1 cCube Architecture

The demands of the EML researcher, end user and *cCube* engineer are met with microservices and Docker containers with event queues and REST communication [13]. The design goals for *cCube* have been identified as:

- *portability*, in terms of cross-platform and cloud execution;
- *extendability* with open source code;
- *scalability* in distributing the computation;
- *robustness*, i.e., tolerant of failures;
- *maintainability*, e.g., loosely coupled components and explicit assumptions;
- *deployability*, exploiting standard interfaces for deployment;
- *runability*, run anywhere without recompilation as in plug-and-play insertion;
- *support* best practices in development, maintenance and performance.

Instead of being monolithic, *cCube*'s architecture is based on *microservices*. For each functional component of *cCube*, we identified and implemented a single microservice. These components could be individually developed, using different programming languages (though we wrote them entirely in Python) and technologies, and the interaction between them occurs through simple communication protocols, i.e., REST API with HTTP and AMQP[4] message queues. This allowed *cCube* to have separate provisioning, distribution, communication and EML system execution. An overview of *cCube* is given in Figure 3.

For the EML development, microservices and the software containers approach, we enhanced the *Bring Your Own Learner* model of FCUBE [1]. Instead of inserting the learner within the source code and environment of *cCube*, we injected *cCube* inside the container of the EML algorithm, running it as daemon instructed to manage the communication with the other microservices of the system.

3.2 cCube Implementation

We also focused on the open source properties of *cCube* in order to make the microservices fully accessible to the community, e.g., others could learn and extend *cCube*'s code to develop other EML architectures. To facilitate the deployment, as well as development, we abandoned the traditional use of the hypervisor-based virtualization in favor of the container-based one using Docker[5]. With the hypervisor-based virtualization every hardware component needs to be emulated, and on top of these newly created virtual hardware an operating system is installed. Instead, with Docker is possible to execute more the one container on the same cloud instance by sharing the common resources (e.g., CPUs, RAM) and Linux kernel, without impacting performance. Thanks to the Docker API, it was relatively easy to develop containers that themselves were easy to build, share and quickly execute in a cloud environment. Moreover, once an image of the container is ready, thus completed the build process, it can be stored into a convenient public registry that Docker provides, i.e., *Docker Hub*.

cCube achieves independence from cloud providers by employing *Docker Swarm*. This is a native technology that composes a cluster of Docker platforms between machines running the Docker engine. In addition, by using Docker, we implicitly made *cCube* flexible against the limitations of the quantity of machines the providers usually impose upon their users, through an infrastructure definable as "multi-cloud", i.e., based on the allocation of instances by different cloud providers but participating in the same system [4].

3.2.1 *Microservices.*

An `Orchestrator` creates and provisions the compute units, initiating and directing the symphony of microservices. It uses a bridge pattern [12] to interface with different cloud providers, e.g., *OpenStack* and *Amazon EC2*.

We designed the `Learner`, `Filter` and `Fuser` microservices as part of one single container, i.e., the `Worker`, thus stored by the developer as a single image on a Docker registry. The EML developer needs to inject the *cCube* `Worker` component into the target EML algorithm execution environment, i.e., a Docker container. This component will act as daemon inside the container and be in charge of communication with the *cCube* cluster. Thus, it will execute the EML algorithm for learning or prediction, using environment variables defined by the EML developer. By the means of a parameter given during the orchestration, the container is able to detect which "role" to play.

[4]https://www.amqp.org
[5]https://www.docker.com/

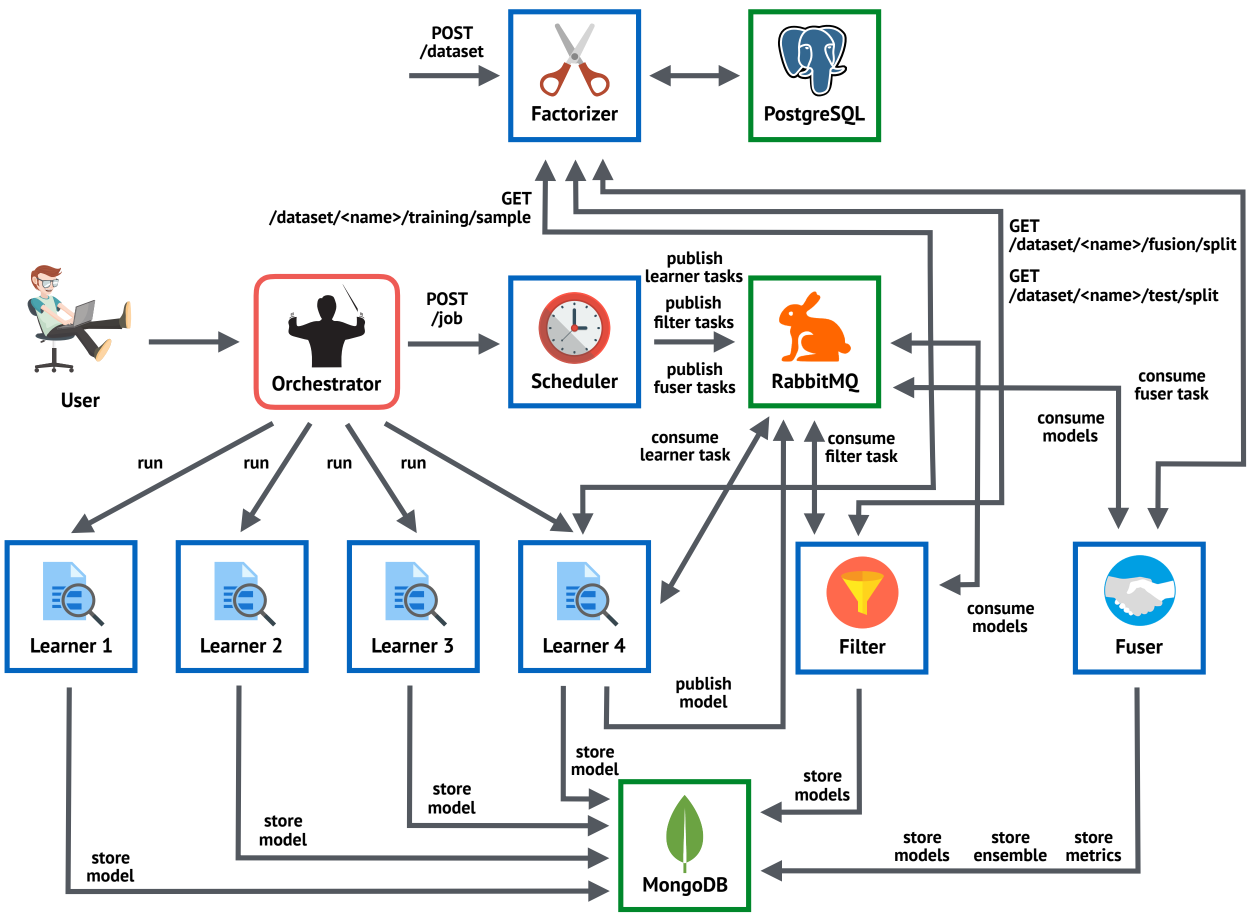


Figure 3: Under the hood of the *cCube* black box there is a microservices and containers architecture using REST API and message queues for extensible, automated, reliable and scalable EML.

We designed and implemented the following microservices, whose overview is shown in Figure 3:

`Factorizer`: uses the bridge pattern [12] to interface with the storage. The storage is currently *PostgreSQL*[6], but it is possible to add other technologies by means of the definition of other driver classes. The `Factorizer` exposes its services through a REST interface thus the communication happens using HTTP. This allows the data set upload via a "*POST /dataset*" request and then the data set can be split into separate parts on demand and following the components needs, e.g., to use for training, fusion, and testing.

`Scheduler`: it accepts jobs through a REST interface. A job is considered as a chain of processes executed on a *cCube* cluster, involving all the components, i.e., microservices. Once a new job is requested, the `Scheduler` creates the tasks for the `Learners`, `Filter` and `Fuser` and publish them on different message queues in `JSON` format. First of all, a task represents a placeholder to let other microservices carry out their duties by consuming them. Exploiting a convenient feature of AMQP, in case of a microservice failed, e.g., cloud provider hardware fault, the task would be put again into the queue and that computation run again by another container. If the EML algorithm concludes the computation, then the task is acknowledged, i.e., definitively removed from the queue. However, in the case the fault is due to an EML algorithm failure, *cCube* avoids the repetition of the same task since, with the same configuration, it would fail again. Thus, the daemon recognizes it, acknowledges the task in the queue and sends just a failure placeholder message to the `Filter`. Moreover, the tasks contain all the relevant information needed for running an activity, e.g., the target dataset name, the parameters of separation for training, fusion and testing, features to include/exclude, duration. In the case of the `Learners`, the tasks would be in the number equal to the degree of parallelization expected by the user. A producer/consumer pattern is controlled by a "queue manager", implemented by using the *RabbitMQ*[7] message broker service. It is worth noting that multiple EML algorithms can participate in the same job and, also, multiple jobs can run on the same cluster.

[6]https://www.postgresql.org

[7]http://www.rabbitmq.com

`Learner`: each of the multiple learners consumes a task, training data sample and parameters, which are possibly generated at random when the task is generated by the `Scheduler`, allowing also a parameters factorization. First, a data sample for training is given by the `Factorizer`, using the "*GET /dataset/<name>/training/sample*" request. Once received the data, the `Learner` executes the EML algorithm according to a given duration, i.e., one of the parameters included in the task. Then, the computed model is compressed and stored in a message and sent to a queue for outputs. Moreover, a copy of each produced model is stored into a *MongoDB* [8] database.

`Filter`: EML algorithm output is filtered according to some "filter policy". When the `Filter` task has been consumed, it knows exactly the number of expected outputs. Therefore, it consumes outputs until the last expected message arrives. Then, the models are executed on a split of the data that has been set aside, as a result of the "*GET /dataset/<name>/fusion*" request, according to a filter policy. It is worth noting that the split is named "fusion" since it is also used during the the fusion phase to produce the ensemble. In our demonstration, the EML algorithm predictive performance is compared to a majority class classifier and if it is above some threshold the model is accepted. After the set of filtered models is done, it is published as a single message to another queue for fusing. The result of the filtering phase is also stored into a MongoDB database.

`Fuser`: filtered models are eventually fused together to collaborate in an ensemble model. A fusion task is consumed and the message provides the needed information. Data for fusion comes from the `Factorizer` through a "*GET /dataset/<name>/fusion*" request, i.e., the same split used during the filter phase. Then, another data split, as a result of the "*GET /dataset/<name>/test*" request, is used to test the ensemble and computing some predictive performance metrics. The "fusion policy" may require the models to be executed both for the fusion and testing phases. In our demonstration, we composed an ensemble based on majority of votes, thus we executed the models only for the test phase. Finally, the ensemble, computed metrics and all the models are stored using the MongoDB service.

3.2.2 *cCube Injection.*

As mentioned above, we let the EML researchers injecting the *cCube* code in their containers. This piece of code is maintained by the *cCube* engineers and publicly distributed in the form of a executable script, that could be for instance make downloadable from a commodity URL (i.e., https://raw.githubusercontent.com/ccube-eml/worker/master/install.sh).

The EML developer is asked to use a *cCube* configuration file that, in the current version, corresponds to a `Dockerfile`, i.e., a Docker container environment definition file. The user is allowed to define the environment using all the features given by Docker, e.g., the inheritance from other public container images already providing the set of tools required for the execution. The only requirement for the connection with *cCube* is the definition of some predefined environment variables, e.g., `$CCUBE_LEARN_COMMAND`, `$CCUBE_PREDICT_COMMAND`, and the inclusion of few download and execution lines, e.g., `ENTRYPOINT ["/ccube"]`.

In our demonstration, we encapsulated the *GP Function* EML algorithm executable [1] into a container without changing the source code. The environment variables define how the *cCube* daemon can interact with the EML algorithm and its outcome for both the learning and prediction phases.

4 DEMONSTRATION

In this section we demonstrate how to use *cCube* to run the *GP Function* EML algorithm with a split of the *Higgs* dataset [2]. We used a private *OpenStack* cloud and then a commercial provider, i.e., *Amazon EC2*. It is worth noting that we used only instances eligible for the *Amazon AWS Free Tier*, thus all the computation was free.

4.1 Environment Preparation

We first had to wrap the current implementation of GP Function in a `Dockerfile` showed in Listing 1. Even though we are not the original developers, it was enough to know only a few input parameters for GP Function to run it in *cCube*:

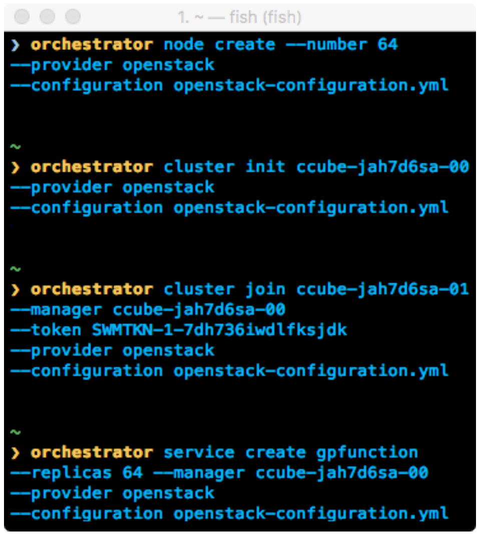
- where to find the data used for learning;
- where to find the algorithm output, i.e., the models;
- how to execute the model to predict data;
- where to collect the ensemble model.

Listing 1: The `.ccube` file for the GP Function algorithm.

```
1  FROM openjdk
2  LABEL maintainer "John Doe john.doe@ccube-eml"
3
4  # cCube injection.
5  RUN curl -sSL https://raw.githubusercontent.com/ccube-eml/worker
         ↪ /master/install.sh | sh
6  WORKDIR /ccube
7  ENTRYPOINT ["python3", "-m", "worker"]
8
9  # Environment preparation.
10 COPY gpfunction.jar /gpfunction/gpfunction.jar
11
12 # cCube configuration.
13 ENV CCUBE_LEARN_COMMAND "java -jar /gpfunction/gpfunction.jar -
         ↪ train \${CCUBE_LEARN_DATASET_FILE} -minutes \${
         ↪ CCUBE_LEARN_DURATION_MINUTES} -properties \${
         ↪ CCUBE_LEARN_PARAMETERS_PROPERTIES_FILE}"
14 ENV CCUBE_LEARN_WORKING_DIRECTORY "/gpfunction"
15 ENV CCUBE_LEARN_OUTPUT_FILES "\${CCUBE_LEARN_WORKING_DIRECTORY}/
         ↪ mostAccurate.txt"
16 ENV CCUBE_PREDICT_COMMAND "java -jar /gpfunction/gpfunction.jar
         ↪ -predict \${CCUBE_PREDICT_DATASET_FILE} -model \${
         ↪ CCUBE_PREDICT_INPUT_FILES}/mostAccurate.txt -o
         ↪ predictions.csv"
17 ENV CCUBE_PREDICT_WORKING_DIRECTORY "/gpfunction"
18 ENV CCUBE_PREDICT_PREDICTIONS_FILE "\${
         ↪ CCUBE_PREDICT_WORKING_DIRECTORY}/predictions.csv"
```

The first section in Listing 1 is used to prepare and set up the environment for the execution. With help of the inheritance capacity of Docker, we were able to start with an environment that had Java already installed. We did this on line (1) by picking the `openjdk` container that is publicly available on *Docker Hub*, i.e., the official and public Docker registry. Lines (5–7) were used to download the *cCube* code (5), install the dependencies, and define the starting point for the container (6–7), i.e., the *cCube* daemon. With line (10) we copied the GP Function `JAR` executable into the environment. The interface between the EML developer and the *cCube* system are on lines (13–18). We define the environment variables, and gave only the relevant information *cCube* would take advantage of during the execution. Moreover, *cCube* provides some predefined environment variables that it will fill in at run time, e.g.,

[8] https://www.mongodb.com



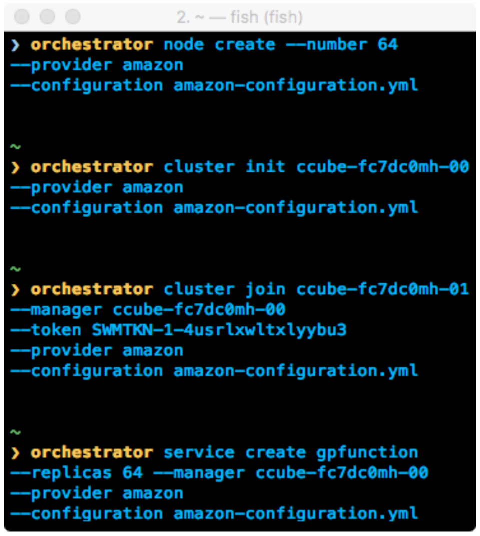
```
> orchestrator node create --number 64
--provider openstack
--configuration openstack-configuration.yml


~
> orchestrator cluster init ccube-jah7d6sa-00
--provider openstack
--configuration openstack-configuration.yml


~
> orchestrator cluster join ccube-jah7d6sa-01
--manager ccube-jah7d6sa-00
--token SWMTKN-1-7dh736iwdlfksjdk
--provider openstack
--configuration openstack-configuration.yml


~
> orchestrator service create gpfunction
--replicas 64 --manager ccube-jah7d6sa-00
--provider openstack
--configuration openstack-configuration.yml
```

(a) OpenStack



```
> orchestrator node create --number 64
--provider amazon
--configuration amazon-configuration.yml


~
> orchestrator cluster init ccube-fc7dc0mh-00
--provider amazon
--configuration amazon-configuration.yml


~
> orchestrator cluster join ccube-fc7dc0mh-01
--manager ccube-fc7dc0mh-00
--token SWMTKN-1-4usrlxwltxlyybu3
--provider amazon
--configuration amazon-configuration.yml


~
> orchestrator service create gpfunction
--replicas 64 --manager ccube-fc7dc0mh-00
--provider amazon
--configuration amazon-configuration.yml
```

(b) Amazon

Figure 4: The terminal execution of the orchestrator for the OpenStack and Amazon providers.

the `$CCUBE_LEARN_DATASET_FILE` will contain the dynamic path *cCube* assigns to the training data samples.

Finally, we started the Docker build process and pushed the resulting image to the public Docker Registry.

4.2 Cloud Swarm Setup

We composed the cluster for *cCube* in three steps:

(1) we ran it on an OpenStack private cloud and created a cluster with 64 nodes using the *cCube* orchestrator;
(2) we composed a separate cluster on Amazon EC2 with 64 other nodes on which we tried the system again;
(3) finally, we just disassembled the two clusters and joined them into a single hybrid cluster with a total of 128 nodes.

We launched only small and cheap instance types: (1) Amazon EC2, "t2.micro" with 1 CPU and 1 GB RAM; (2) OpenStack private cloud, instances with 1 CPU and 1 GB RAM.

To do this, we extended the *cCube* `Orchestrator` to interface with both the cloud providers. The *cCube* `Orchestrator` runs a number of threads equal to the cluster size, in order to ask for new allocations in parallel. To avoid stressing the API interface of the providers we delayed the start of the threads of 1 s each.

We identified two different times to measure:

- "creation", i.e., the necessary time to acquire the virtual machine and be able to communicate with it through an SSH connection;
- "provisioning", i.e., the time required to install Docker on the machine and add it to the Docker Swarm.

Figure 4 shows the commands we executed in the terminal to run the `Orchestrator`. We intentionally hide most of the output, since it was mostly debugging logs. It is worth noting that, even if the cloud providers are different, the commands are quite similar. Given the cloud provider and a configuration file containing the account credentials, the *cCube* `Orchestrator` is able to:

(1) create the 64 nodes on the provider, and provision them with the latest version of Docker (`node create`);
(2) initialize the cluster, i.e., the Docker Swarm, by electing one of the nodes (e.g., `ccube-jah7d6sa-00`) as a manager (`cluster init`);
(3) let all other nodes join the cluster as workers, by using the secret token given by the previous command and the name of the manager (`cluster join`) .

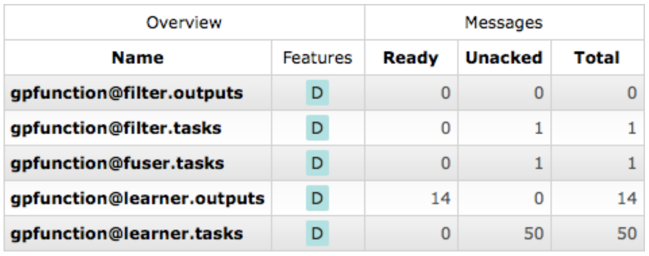
From this point, the infrastructure is ready to run any Docker service, as *cCube* in our case. We measured the setup time, by running each of command 10 times, and we got 64 nodes in 15 min on average for OpenStack and 9 min for Amazon. It took only a couple of minutes to provision the two clusters and let the nodes join a single Docker Swarm.

4.3 cCube Execution

Once the cluster was ready, we executed *cCube* with the GP Function classifier on a split of the `Higgs` dataset [2]. This was done by running the `service create` command for each of the microservices. The template of the command is shown in Figure 4 where the execution of the `Learner` microservices is specified. By specifying the `--replicas` parameter, a single command could run it in parallel on 64 `Learners`.

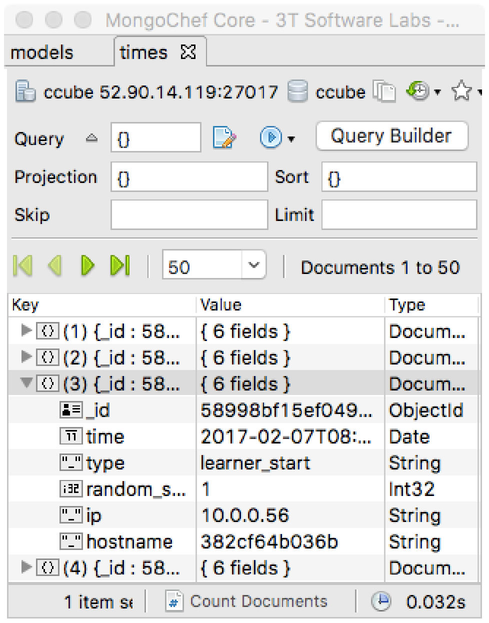
Figure 5 shows a screenshot of the RabbitMQ administration web page displaying the status of *cCube* messages at a point during the computation. During that moment, 14 `Learners` had completed their work and the other 50 `Learners` were still running. In the meantime, the `Fuser` was waiting for all the 64 `Learner` outputs.

Finally, Figure 6 shows the output from the `Fuser` we collected in the MongoDB database, storing both the models and the times.



Overview		Messages		
Name	Features	**Ready**	**Unacked**	**Total**
gpfunction@filter.outputs	D	0	0	0
gpfunction@filter.tasks	D	0	1	1
gpfunction@fuser.tasks	D	0	1	1
gpfunction@learner.outputs	D	14	0	14
gpfunction@learner.tasks	D	0	50	50

Figure 5: The message queues status during the *cCube* execution.



```
Query       {}
Projection  {}          Sort  {}
Skip                    Limit
50
```

Key	Value	Type
(1) {_id : 58...	{ 6 fields }	Docum...
(2) {_id : 58...	{ 6 fields }	Docum...
(3) {_id : 58...	{ 6 fields }	Docum...
_id	58998bf15ef049...	ObjectId
time	2017-02-07T08:...	Date
type	learner_start	String
random_s...	1	Int32
ip	10.0.0.56	String
hostname	382cf64b036b	String
(4) {_id : 58...	{ 6 fields }	Docum...

```
1 item se    Count Documents    0.032s
```

Figure 6: The output of the *cCube* execution on the MongoDB database.

5 CONCLUSIONS AND FUTURE WORK

We present *cCube*, an open source architecture for the comparison, competition and multi-party collaboration of EML algorithms and users. The goal of *cCube* is to provide an instrument for EML developers to run their algorithms on a large scale using parallel cloud machines without changing code. By using the "Bring Your Own Learner" model, we allow the collaboration of different EML algorithms to learn on the same dataset and combing the results into an ensemble after filtering and fusion.

A microservices architecture simplifies the development and the maintenance processes, and facilitates extension from the open source community. Moreover, we implemented the microservices in the form of software containers using Docker and demonstrated the execution of *cCube*. The main contributions are:

- let EML developers define the environment for the execution of their algorithms and easily inject and run *cCube*'s code;
- package the microservices in distributable images and make them available through a public repository;
- using the capability of Docker Swarm, we realized a flexible infrastructure, avoiding the "lock" in from specific cloud providers and possibly runnable on multi-cloud;
- allow users collaboration, in the form of crowdfunding.

Currently, our architecture allows to fuse models resulting from different learners only if the EML algorithms are available on the same container. For instance, the GP Function in Java can fuse with a another Genetic Programming learner written in Python. Therefore, the use of specific algorithms is possible by specifying it using a task parameter. It is in our future agenda to enhance the architecture to better support multiple EML algorithms running and their models fusion, adding a prior "executor" microservice whose purpose is to free the `Fuser` from the strict bond with the execution environments.

Also, future work will involve extending *cCube* further for comparison and competition, as well as testing its usability, testing the large scale capacity over more cloud providers and more nodes on big data challenges. In addition, deploy other Evolutionary Algorithm problems.

REFERENCES

[1] Ignacio Arnaldo, Kalyan Veeramachaneni, Andrew Song, and Una-May O'Reilly. 2015. Bring Your Own Learner: A Cloud-Based, Data-Parallel Commons for Machine Learning. *IEEE Computational Intelligence Magazine* 10, 1 (Feb. 2015), 20–32.

[2] P. Baldi, P. Sadowski, and D. Whiteson. 2014. Searching for Exotic Particles in High-Energy Physics with Deep Learning. *Nature Communications* 5 (Feb. 2014).

[3] Erhan Ekici. 2014. Microservices Architecture, Containers and Docker. (Dec. 2014). https://www.ibm.com/developerworks/community/blogs/1ba56fe3-efad-432f-a1ab-58ba3910b073/entry/microservices_architecture_containers_and_docker

[4] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. 2014. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems. In *IEEE International Conference on Cloud Computing (CLOUD)*. 887–894.

[5] Martin Fowler. 2006. Continuous Integration. (May 2006). https://www.martinfowler.com/articles/continuousIntegration.html

[6] Mario García-Valdez, Leonardo Trujillo, Juan Julián Merelo Guervós, Francisco Fernandez de Vega, and Gustavo Olague. 2015. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing* 13, 3 (Sept. 2015), 329–349.

[7] Guillaume Leclerc, Joshua E. Auerbach, Giovanni Iacca, and Dario Floreano. 2016. The Seamless Peer and Cloud Evolution Framework. In *Genetic and Evolutionary Computation Conference (GECCO)*. 821–828.

[8] James Lewis and Martin Fowler. 2014. Microservices, a Definition of This New Architectural Term. (March 2014). https://martinfowler.com/articles/microservices.html

[9] Juan Julián Merelo Guervós, Antonio Miguel Mora García, Carlos M. Fernandes, and Anna Isabel Esparcia-Alcázar. 2012. SofEA, a Pool-Based Framework for Evolutionary Algorithms Using CouchDB. In *Genetic and Evolutionary Computation Conference (GECCO)*. 109–116.

[10] K. Meri, M. G. Arenas, A. M. Mora, J. J. Merelo, P. A. Castillo, P. García-Sánchez, and J. L. J. Laredo. 2013. Cloud-Based Evolutionary Algorithms: An Algorithmic Study. *Natural Computing* 12, 2 (June 2013), 135–147.

[11] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. 2016. Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud. In *Genetic and Evolutionary Computation Conference (GECCO)*. 121–122.

[12] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[13] Jim Webber, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media.

[14] Yifeng Zhang and Siddhartha Bhattacharyya. 2004. Genetic Programming in Classifying Large-Scale Data: An Ensemble Method. *Information Sciences* 163, 1 (2004), 85–101.