OSCAR-P and aMLLibrary: Performance Profiling and Prediction of Computing Continua Applications

Enrico Galimberti
Politecnico di Milano university
Milan, Italy
Enrico.Galimberti@polimi.it

Bruno Guindani
Politecnico di Milano university
Milan, Italy
Bruno.Guindani@polimi.it

Federica Filippini
Politecnico di Milano university
Milan, Italy
Federica.Filippini@polimi.it

Hamta Sedghani
Politecnico di Milano university
Milan, Italy
Hamta.Sedghani@polimi.it

Danilo Ardagna
Politecnico di Milano university
Milan, Italy
Danilo.Ardagna@polimi.it

Sebastián Risco
Universitat Politècnica de València
Valencia, España
serisgal@i3m.upv.es

Germán Moltó
Universitat Politècnica de València
Valencia, España
Gmolto@dsic.upv.es

Miguel Caballer
Universitat Politècnica de València
Valencia, España
Micafer@i3m.upv.es

ABSTRACT

This paper proposes an auto-profiling tool for OSCAR, an open-source platform able to support serverless computing in cloud and edge environments. The tool, named OSCAR-P, is designed to automatically test a specified application workflow on different hardware and node combinations, obtaining relevant information on the execution time of the individual components. It then uses the collected data to build performance models using machine learning, making it possible to predict the performance of the application on unseen configurations. The preliminary evaluation of the performance models accuracy is promising, showing a mean absolute percentage error for extrapolation lower than 10%.

CCS CONCEPTS

• Software and its engineering → Software creation and management; **• Computing methodologies → Artificial intelligence**; **Machine learning**; **Modeling and simulation**.

KEYWORDS

Edge Computing, Performance Profiling, Machine Learning

ACM Reference Format:
Enrico Galimberti, Bruno Guindani, Federica Filippini, Hamta Sedghani, Danilo Ardagna, Sebastián Risco, Germán Moltó, and Miguel Caballer. 2023. OSCAR-P and aMLLibrary: Performance Profiling and Prediction of Computing Continua Applications. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584941

1 INTRODUCTION

Cloud computing is widely adopted today and has been used for years as the standard computing paradigm for industrial-level distributed systems [17]. This model evolved over time, and recently edge systems are becoming more and more popular. Their main advantage is given by the fact that the data produced by devices at the edge of the network can be processed locally, instead of being moved to a centralized data-center and returned to the user after the processing.

As a result, a novel computing paradigm is emerging, creating a computing continuum that seamlessly integrates edge devices with remote cloud servers by moving a part of the computation close to where data are produced [15].

Another novelty in cloud computing is represented by the introduction and quick rise in popularity [4] of the Functions as a Service (FaaS) model. It breaks down complex applications in workflows of small, usually short-lived components that run on reusable services consisting of stateless containers activated by suitable events (e.g., a file upload). This greatly benefits data processing systems, as the execution of resource-intensive applications is triggered on demand. Using containers instead of full virtual machines reduces both the development/deployment complexity and the resource usage. FaaS model in public clouds is characterized by per-millisecond costs bound to the actual resource usage, instead of charging a fixed rate at all times as in other cloud models [1].

Achieving low latency and high throughput is one of the main drivers behind edge computing. However, evaluating the performance of a complex application, whose components may be allocated at different levels of the computing continuum, is a problem without an easy solution. There is a need for automated tools that can help with the profiling of those components, measuring the execution times with a varying amount of available resources. Furthermore, predicting the performance of a given application at a target configuration is a key task to support the proper planning and runtime management of the available resources.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584941

This paper proposes an integrated framework for the execution, profiling, and performance models training for FaaS-based applications running in computing continua. This framework has two main components, each of which represents a novel contribution: *OSCAR-P* and *aMLLibrary*. Application execution is supported by the OSCAR[1] framework [13], a state-of-the-art runtime environment for computing continuum applications which aims at creating a highly-parallel, event-driven, pipelined file-processing serverless environment to execute general-purpose applications. Application profiling is based on OSCAR-P, an auto-profiling tool built around OSCAR, which automatically tests a specified application workflow on different hardware and node combinations, obtaining relevant information on the timing of the execution of the individual components. The profiling data are then used by aMLLibrary, a machine learning library we designed to easily build performance models. A performance model is built for every service/resource pair, and these single models are composed together to obtain a prediction on the runtime of the full workflow, making it possible to predict the performance (average execution time) of an application on unseen configurations. The preliminary results we achieved on a testing application are promising, since the performance models for estimating application response time introduce a mean absolute percentage error in extrapolation scenarios lower than 10%.

This paper is organized as follows. Section 2 presents relevant literature works. Section 3 provides a summary of the OSCAR framework and its architecture. Section 4 illustrates in details the OSCAR-P goals and its components, while Section 5 illustrates the capabilities of aMLLibrary and the performance analysis it can support. Section 6 focuses on the experimental scenarios considered to validate our tools. Conclusions are finally drawn in Section 7.

2 RELATED WORK

This section overviews some recent works in technological fields that are relevant to this paper, such as FaaS and edge computing enabling frameworks, benchmarking tools for FaaS and edge systems, and machine learning (ML)-based performance modelling.

Sewak et al. [16], as a primer on the topics, present FaaS and more generally serverless computing, explore its advantages and limitations, and compare the options available in public clouds with different providers. Recently, frameworks that aim to support edge computing have also been proposed. For example, ONEedge allows to build and manage a distributed edge-cloud using resources from public clouds and edge infrastructure providers. The literature presents frameworks that support edge computing following the FaaS paradigm. This is the case of *faasd*, which provides a lightweight alternative to OpenFaaS, a popular open-source FaaS framework, that runs on machines with very modest requirements, allowing the deployment of embedded applications in IoT and edge use-cases. As the popularity of FaaS and edge computing rose, several cloud providers started offering their own FaaS platforms, each with different underlying technologies. At the same time, many researches tried to address the challenges of benchmarking applications deployed partially on the edge and partially on the cloud and presented their own software solution [3, 11].

Many researchers focus on analyzing and predicting the performance of applications running on edge/cloud systems, and recently their majority use ML models. For example, the work by Disabato et al. [5] proposes linear regression models to predict the execution time of Convolutional Neural Networks (CNNs) on edge devices, given constraints on memory and processing load. Instead, authors of [12] employ several ML models alongside anomaly detection to properly configure a cloud-based Internet of Things (IoT) device manager while respecting Quality of Service (QoS) constraints. In [10], a number of ML techniques are used to perform execution times prediction of Spark Cloud jobs with different types of workloads. Finally, performance modelling through ML is also applied to FaaS platforms. SuanMing [8] is an integrated framework for learning regressors using different algorithms for microservice-based systems, with the goal of identifying potential sources of performance loss in complex applications. Different from the previous works, our performance profiling solution, OSCAR-P, is focused on benchmarking the OSCAR framework, which can be deployed on top of any commercial cloud, and hence has the important benefit of being cloud provider agnostic. Moreover, the aMLLibrary can provide the average execution time of an application workflow with acceptable precision even for unseen configurations and with a limited testing campaign.

3 OSCAR

OSCAR is an open-source framework that easily and efficiently supports event-driven, file-processing, serverless applications packaged as Docker images in the computing continuum [14]. These are executed in elastic Kubernetes clusters, dynamically provisioned and horizontally scaled across multiple Cloud back-ends. OSCAR is used as runtime framework in European research projects, to support inference of AI/ML models in AI-SPRINT and AI4EOSC and to provide event-driven data ingestion capabilities in InterTwin.

The framework architecture (see Figure 1) includes: i) MinIO, an S3-compatible high-performance multi-cloud object storage server, native to Kubernetes; ii) Knative, an open-source enterprise-level solution to build serverless and event-driven applications that supports synchronous invocations, iii) OSCAR Manager, the main service that manages the integration of the separate components.

The supported storage providers are: i) MinIO, deployed either internally or externally to the cluster; ii) Amazon S3, AWS's object storage service and industry leader in terms of scalability, data availability, security, and performance in the public Cloud; iii) Onedata, the global data access solution for science used by the EGI Federated Cloud.

Applications are created as services on the OSCAR cluster, by specifying: i) a Docker image to be used, ii) the input and output buckets of each service, and iii) by providing a shell script to be executed inside the container. The services can be created manually one by one, or all at once through a Function Definition Language (FDL) configuration file. Once the application is configured, the execution of a service is automatically triggered by uploading a file into its input bucket. The result is delivered into the output bucket; this may be the input of another service that, if so, is automatically triggered and either executed immediately, if there are enough available resources, or added to the job queue. Uploading multiple

[1] OSCAR - https://oscar.grycap.net

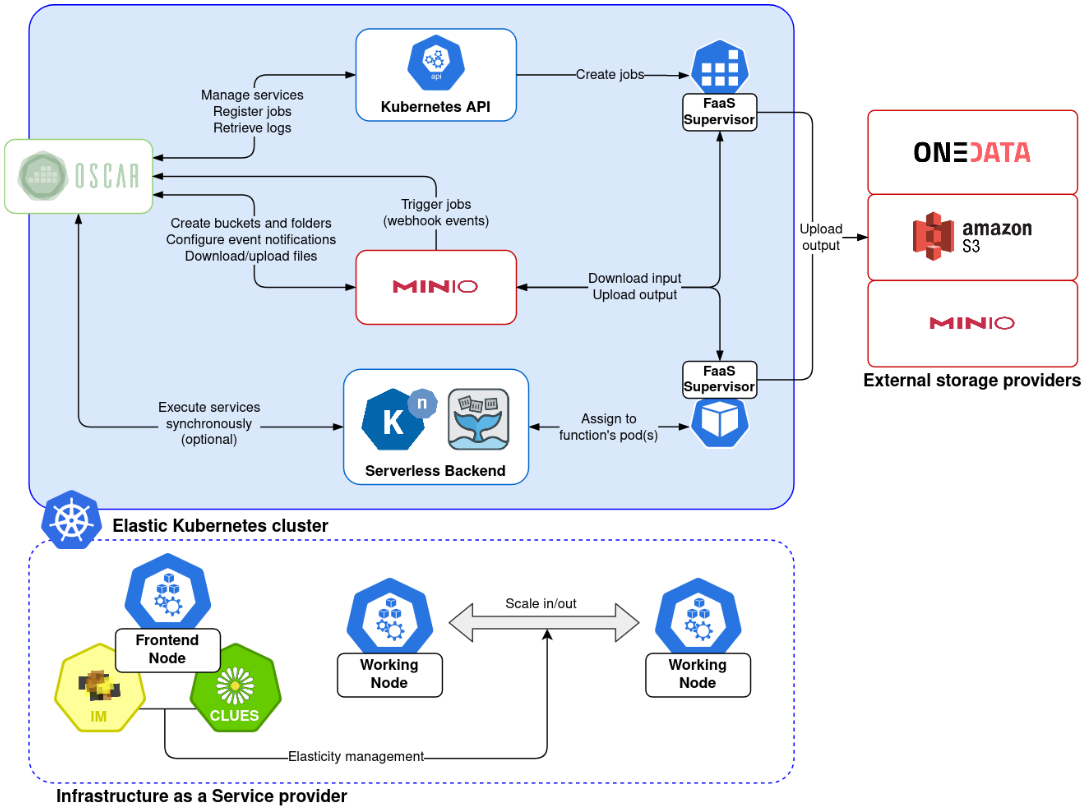


Figure 1: OSCAR architecture

files triggers the highest possible number of parallel invocations supported by the specific cluster; all the remaining function calls are scheduled to be executed as soon as the running services complete.

A component may have multiple versions, and it may be replaced by two or more equivalent components running sequentially. This is crucial for AI applications, since we can partition DNNs to run on (possibly) different devices, exploiting at best the computing continuum resources.

OSCAR can run on minified Kubernetes distributions (e.g., k3s) and on arm64 architectures (e.g., Raspberry Pi clusters).

4 OSCAR-P

This section presents OSCAR-P, the OSCAR profiler which provides the first novel contribution of this paper.

OSCAR-P is built around OSCAR and its components, and it acts as a director, configuring and coordinating the profiling activities and collecting the required data once the profiling is completed. The aim of OSCAR-P is to simplify and fully automate the testing of specific OSCAR application workflows on different hardware configurations, and collect data to train machine learning performance models.

Specifically, OSCAR-P receives as input:

(1) a description of resources that needs to be tested, with a detailed overview of their hardware and software architecture (the number of available nodes, the memory amount and number of cores of every node);
(2) the components, their Docker images, and their hardware requirements (the needed memory amount and number of cores of every single instance);
(3) a set of parameters specifying how to test the application (which input files to use, the number of batches and their sizes, the time interval between uploads and their distribution). Moreover, for every component, the parallelism levels to be tested (i.e., the maximum number of parallel instances that a component is allowed to have) is also specified;
(4) the machine learning models to consider in the performance models training and their hyperparameters.

The tests are performed by varying the used resources, either by changing the number of active nodes or by changing one resource with another.

Once all the required resources are in place and correctly configured, the testing campaign is controlled by a single YAML configuration file (see Figure 2) containing the list of services, the description of the clusters and their worker nodes and information on how the application needs to be tested; this YAML file is paired with the results of its associated run, to simplify debugging and allowing replicability. Every combination of hardware and nodes is tested more than once to cope with execution time measurement noise and, after testing the full workflow, the individual components are also tested on their own.

The profiling activities have to follow a precise sequence of steps (illustrated in Figure 2), each managed by a separate OSCAR-P sub-component.

1. Input files parser. Starting from the input files, OSCAR-P lists all the "testing units", i.e., all the valid component / resource tuples; if a component is partitioned (OSCAR supports also the execution of partitioned Deep Neural Networks), all the partitions are considered as part of the same "testing unit", thus ensuring that they are always tested together. As a simple example (see Figure 3), we can consider an application including a single component (*Component 1*), which can be split in two partitions (*Component 1.1* and *Component 1.2*) available for different architectures (ARM64 and AMD64, respectively). The available resources include a cluster of Raspberry Pi (ARM64) and a cluster of Virtual Machines (AMD64). In this scenario, the testing units would be: 1) Comp. 1 on the RasPi cluster; 2) Comp. 1 on the VM cluster; 3) Comp. 1.1 on RasPi and Comp. 1.2 on VM cluster.

OSCAR-P then creates a list of all the possible deployments, that is all the possible combinations of the testing units (see Figure 4). Finally, each deployment contains a list of "runs" to be tested: every run includes the same list of components, but each component "parallelism" (i.e., its maximum number of parallel instances) varies from one run to the other according to what has been specified. An example is shown for Deployment 3 in Figure 5.

2. Cluster configurator. Before testing a deployment, the involved clusters need to be setup and correctly configured for the first run. As an example, for configuring physical clusters, OSCAR-P connects via SSH to the front node of the Kubernetes (K8s for short) cluster, and then cordon or uncordon the worker nodes to reach the required number. Once the clusters are configured correctly, the profiling of a specific run can take place. In order to modify the configuration of the clusters to adapt it to the next runs, the cluster configurator can change the number of worker nodes.

3. Description generator. After configuring the clusters to suit the requirements of a particular run, OSCAR-P creates a descriptive YAML file detailing all the information needed to run the test in a single location. This file contains the list of all the services, reporting for each one their requirements in terms of memory and cores, their input and output buckets and the associated Docker image. It also contains a description of the clusters in use, their endpoints, credentials and configuration, as well as information on the input files to be used to start the run, their number and the timing of their uploads. This descriptive file is updated with all the subsequent runs of a deployment, and in the end it serves as a detailed summary of the whole testing campaign.

4. Run manager. The run description YAML file is parsed to extract all its relevant information before the run can start. OSCAR-P then generates an FDL file containing the information needed to build the new workflow, meaning the required services and buckets, and uploads it to OSCAR so that it can be applied. The run is started by moving the required files in the input bucket of the first service, which triggers its execution.

For the full application workflow tests, once the files are moved into the input bucket of the first component the workflow proceeds by itself, since every component output bucket is the input of another component (the input parser checks this assumption in the first steps), and OSCAR-P simply monitors the execution until its completion.

When testing single services instead, a component cannot write its output files in its assigned bucket or else it would trigger the execution of the next function. The solution we adopted is connecting the tested components to a temporary empty input bucket, and to a temporary output bucket which does not trigger the execution of other components. The contents of the "real" input bucket are then copied to the dummy input bucket, just like at the start of the run with the storage bucket, triggering its execution.

5. Log retriever. After finalising each run, OSCAR-P proceeds to collect and process the logs. The logs are retrieved both from OSCAR and kubectl, and together they provide information on when a job (that is a single component execution) was scheduled, when its pod was created, and when it was actually started and finished; all this information is useful for checking delays, waits and overheads. The relevant logs information, also across multiple runs are collected in a single CSV file which is used by the aMLLibrary to train a performance model for every service/resource pair.

5 AMLLIBRARY

aMLLibrary[2] is an open-source, high-level Python package that allows training of multiple performance models for the individual components and the full workflow, supporting feature selection and hyperparameters tuning. It is based on the scikit-learn toolkit, and uses supervised machine learning techniques to generate regression models which can be used to predict applications performance. Overall, aMLLibrary implements an autoML solution, i.e., it performs training of multiple regression models and automatically selects the most accurate one based on the validation metric chosen. The execution of the library is controlled by a simple configuration text file (or equivalently, a Python dictionary), where the user can specify the dataset to be used, the training settings, the regression models to be tested and their ranges of hyperparameters, and the validation method.

aMLLibrary has several useful perks for building performance models. Individual analyses can compare in a single run multiple alternative ML methods, and parallel processing of the training phase of the models is supported. The user can specify the number of parallel cores to be used, and the library automatically distributes the training experiments evenly among the parallel workers, even if the underlying scikit-learn model is limited to single-thread execution. Furthermore, the library implements a fault tolerance mechanism by saving incremental progress checkpoints. If the experimental campaign is interrupted, e.g., because of the failure of the server the library is running on, it can recover the previous results and resume from there.

The main strengths of aMLLibrary are its ease of use, customizability, and extensibility. A simple configuration text file is required to launch a full experimental campaign for all implemented models

[2] https://github.com/aMLLibrary/aMLLibrary

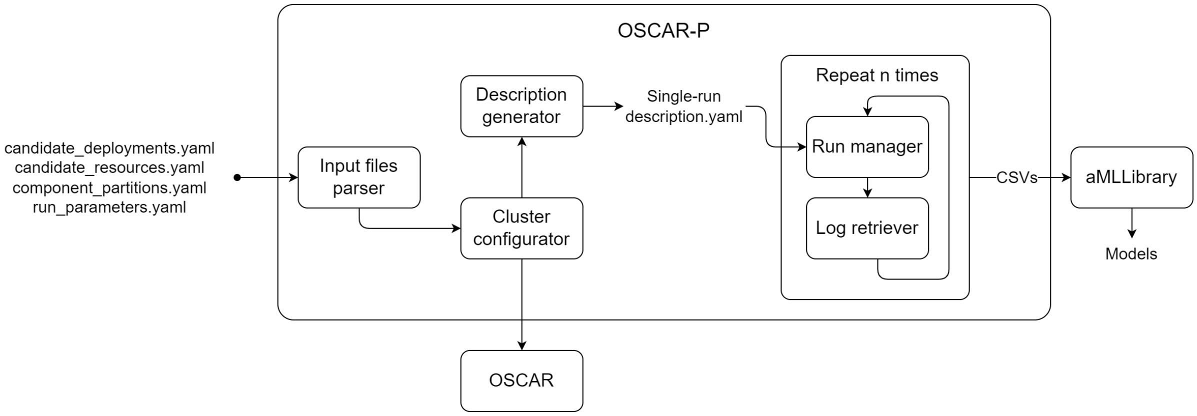


Figure 2: Profiling activity steps and OSCAR-P sub-components.

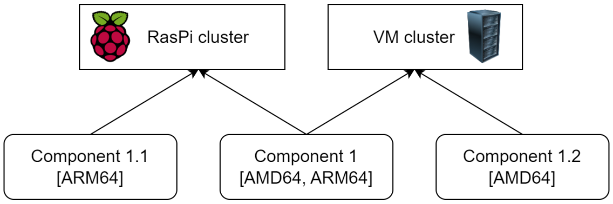
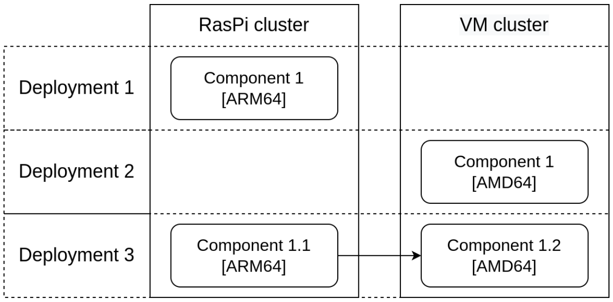


Figure 3: Simple application example

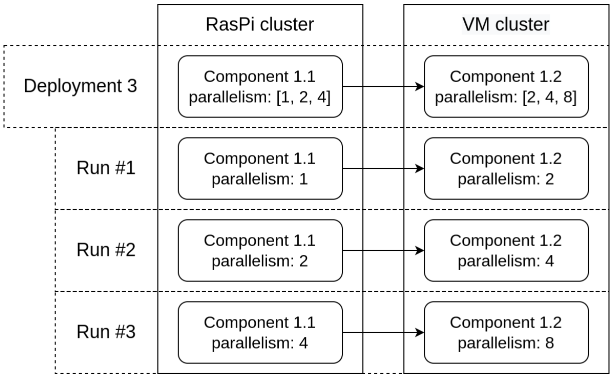


	RasPi cluster	VM cluster
Deployment 1	Component 1 [ARM64]	
Deployment 2		Component 1 [AMD64]
Deployment 3	Component 1.1 [ARM64]	Component 1.2 [AMD64]

Figure 4: Testing units example

without writing a single line of Python code. Default probability distributions for hyperparameters are provided, and they are general enough to allow the automatic tuning mechanism to find the appropriate parameter values without further input. At the same time, the user has full control over the experimental campaign thanks to the many options and flags available in the configuration file. Extensibility is another major advantage for advanced users who wish to implement new techniques. One can simply write a plugin or a model wrapper and add it to the library, building on the existing features already available.

The library currently supports Decision Tree (DT), Non-Negative Least Squares (NNLS), Random Forest (RF), Ridge Linear Regression, Stepwise (a linear regression model integrating the Draper-Smith feature selection technique [6]), Support Vector Regression (SVR), and XGBoost. Hyperparameter tuning of these models can be performed either via grid search, by specifying the lists of values to be



	RasPi cluster	VM cluster
Deployment 3	Component 1.1 parallelism: [1, 2, 4]	Component 1.2 parallelism: [2, 4, 8]
Run #1	Component 1.1 parallelism: 1	Component 1.2 parallelism: 2
Run #2	Component 1.1 parallelism: 2	Component 1.2 parallelism: 4
Run #3	Component 1.1 parallelism: 4	Component 1.2 parallelism: 8

Figure 5: Deployment example

tested, or automatically via Bayesian Optimization (BO), using the HyperOpt library[3], with which aMLLibrary is integrated.

aMLLibrary includes plugins for several data pre-processing techniques (e.g., data normalization and one-hot encoding for discrete features), as well as other convenient tools such as row selection and data validity checks. It also supports automatic feature engineering, computing logarithms, inverse values, and feature products/polynomial expansion up to a given degree. These tools can be useful to unearth potentially relevant information hidden in the input features, such as quadratic dependencies and interaction terms. Feature selection techniques are also supported, including forward Sequential Feature Selection (SFS) [7] and importance weight selection by using the XGBoost regression model. The user can choose among several validation methods to compute the Mean Absolute Percentage Error (MAPE) of a model, computed as $MAPE(y, \widehat{y}) = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{y_i - \widehat{y}_i}{y_i} \right|$, where y is the vector of true values and $\widehat{y}$ is the vector of predicted values by the ML model. Finally, the library has a prediction module that can be used to make interpolation and extrapolation with a trained regression model. A summary of the structure of aMLLibrary is presented in Figure 6.

[3] https://github.com/hyperopt/hyperopt

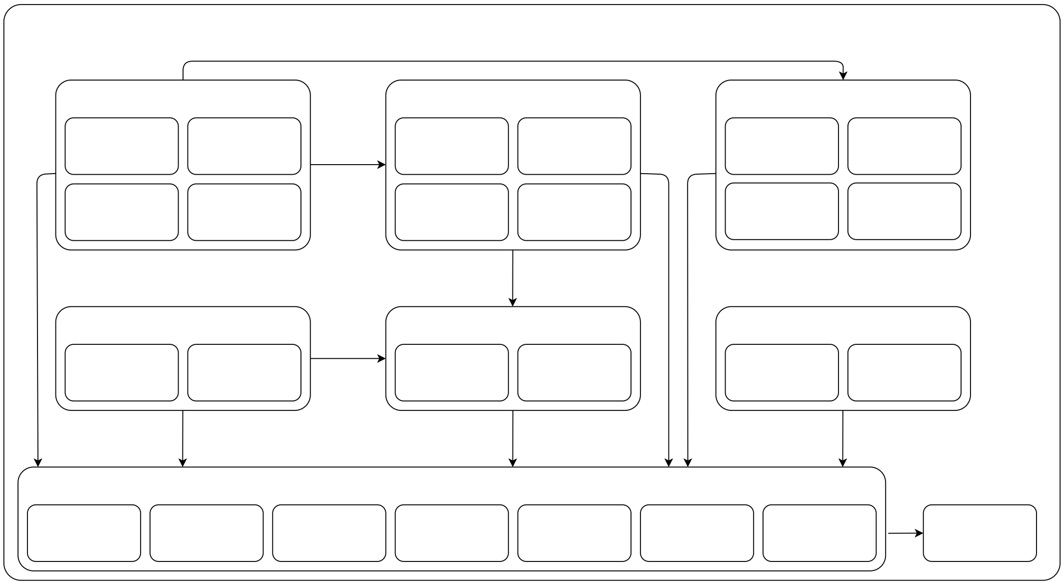


Figure 6: Block diagram of the aMLLibrary components

6 EXPERIMENTAL ANALYSIS

This section provides an overview of the preliminary experiments we performed to test and validate our framework.

Section 6.1 introduces the considered application and briefly overviews the tests that were performed. Section 6.2 describes the machine learning techniques that were exploited, and their hyperparameters. The results obtained in the testing campaigns are presented in Section 6.3.

6.1 Experimental setup

For our experiments we considered a mask-detection application initially proposed in [14], whose workflow includes two components, "blur-faces" and "mask-detector".

The "blur-faces" component receives a video as input, extracts a frame every 5 seconds using the *FFmpeg* tool, and then "anonymises" each frame by detecting the faces with a neural network and then modifying the corresponding pixels. The "mask-detector" component receives an image as input and uses a neural network to detect the faces appearing in it and decide whether or not they are wearing a mask, drawing a green or red box on them accordingly. The result confidence is also reported in the box border, while a summary of the mask count is written on top of the image.

YOLO (You Only Look Once) networks, powered by the TensorFlow object detection API and trained on the *WIDER FACE* dataset, are used for both tasks. As the name implies, these networks only examine the image once to detect multiple objects present in it, yielding significantly faster performance while maintaining high accuracy, which is crucial in a context where time is an important criterion.

The goal of this application is to monitor various areas of a city with cameras and determine which one has the highest percentage of people not wearing a mask. Ideally, the cameras are attached to an OSCAR cluster running at the edge on a cluster of Raspberry Pi. Videos are uploaded to the input bucket of the edge cluster, which performs the split and frames anonymization. The frames are then uploaded to the input buckets of the main OSCAR cluster, running in the cloud, where are processed by the mask-detector.

By processing the videos at the edge of the network, we ensure the privacy of the people appearing in them, as all the data preceding the anonymization never reach the cloud servers. At the same time, we improve performance by lowering the load on the cloud datacenter, and by eliminating the additional latency caused by the round-trip-time delays required to access it. In the experiments, we used a private cloud with 9 VMs (each with an Intel Xeon quad-core processor and 8GB of memory) to represent the cloud side, and a cluster of four Raspberry Pi 4 Model B (each with a quad-core processor and 4GB of memory) to represent the edge cluster.

The first component, "blur-faces", had an image for both x86 and ARM64 architectures and was tested and profiled on both clusters, while the second component, "mask-detector", was only profiled on the VM cluster.

6.2 ML models and hyperparameters settings

In the performance evaluation literature, the models accuracy is usually measured through the Mean Absolute Percentage Error (MAPE) (see also Section 5). An error lower than 30% is usually considered enough to support performance evaluation and capacity planning decisions [9].

The data collected during the tests were used to generate and evaluate different machine learning models (exploiting XGBoost, Ridge Regression, Decision Tree and Random Forest). The hyperparameters used for each model are reported in Table 1, and were tuned through the HyperOpt framework (see Section 5), setting to 10 the maximum number of evaluated hyperparameter sets.

Algorithm	Hyperparameter Name	Values
Ridge Regression	alpha	loguniform(0.01,1)
XGBoost	min_child_weight	1
	gamma	loguniform(0.1,10)
	n_estimators	1000
	learning_rate	loguniform(0.01,1)
	max_depth	100
Decision Tree	criterion	mse
	max_depth	3
	max_features	auto
	min_samples_split	loguniform(0.01,1)
	min_samples_leaf	loguniform(0.01,0.5)
Random Forest	n_estimators	5
	criterion	mse
	max_depth	quniform(3,6,1)
	max_features	auto
	min_samples_split	loguniform(0.1,1)
	min_samples_leaf	1

Table 1: Hyperparameters used for each ML algorithm

For every experiment, we trained models with all the four ML algorithms, with and without Sequential Feature Selection (SFS). The main feature is the *parallelism level*, which denotes the maximum number of available cores for all the services. Since all the components are single-core, this is directly translated to the maximum number of components that are running concurrently. For example, a parallelism level of four means that at any given time we will have, at most, four containers running in parallel; for the sake of simplicity, this feature is just reported as *cores*.

Aside from *cores*, we also used *1/cores* and *log(cores)* as features, as they are relevant for parallel systems [10]. We also exploited the polynomial expansion capability offered by the library, considering products up to the second degree.

The models were tested to assess their ability to perform interpolation (i.e., to predict values in areas of the features space that have been sufficiently observed during the training phase), and extrapolation (i.e., to predict values in regions of the parameters space not sufficiently explored) [2, 10]. Our goal when testing the models interpolation capabilities was to understand how dense our profiling campaign must be to achieve good results, and if we can obtain accurate models with smaller datasets. At the same time, extrapolation tests aim at understanding the behaviour of the models on unseen configurations, determining, e.g., if it is possible to predict the performance we will achieve on larger inputs, from experiments ran on a limited number of files.

The models were generated on a server with 2 Xeon E5-2620 v2 processors, six cores each, and 32 GB of RAM. More details on the training set and the prediction results are reported in the next section. The source code of aMLlibrary and OSCAR-P, the input files, the log data and the model results are available in Zenodo[4].

6.3 Testing scenario and results

In our tests, we considered different numbers of 10-seconds videos, running the application three times on VMs configured with 4 cores and 8 GB of memory. We tested both the full workflow and the single components one by one, exploiting the cores/node combinations reported in Table 2. Note that, when testing the full workflow, the parallelism level is comprehensive of both services, meaning that the sum of the number of cores of both services cannot exceed the parallelism level. As mentioned in Section 6.2, the services implement a sequential code, thus assigning more cores to a container does not yield any performance improvement. However, it regulates the number of containers that can be scheduled on one node at a given time. For instance, assigning 4 cores to a single container guarantees that only that container will run on a node with 4 cores; moreover, if only one node is active, this means that we will experience a sequential execution, with a parallelism level equal to 1.

Parallelism level	1	2	4	8	12	16	20	24	28	32
Cores reservation per container	3.9	1.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
Active nodes	1	1	1	2	3	4	5	6	7	8

Table 2: Cores-nodes combination used for the tests

Note that the number of reserved cores is not rounded to leave some room for the underlying OS. If we assigned, e.g., one core per container on a node, we would see at most three containers running concurrently. Indeed, the fourth one would consume all remaining resources leaving nothing for the OS, and therefore it will never be scheduled.

Finally, even though the cluster has 9 nodes, one of them is a master node where no jobs can be executed. Therefore, at most 8 active nodes are considered in the listed combinations.

We run experiments considering 5, 10, 15, and 20 videos. The data from the first 3 tests (5, 10, 15) were fed to aMLLibrary to train two performance models, one for every service, using Sequential Feature Selection (SFS) and the feature engineering options offered by the library. The MAPE obtained after training the models with and without SFS is reported in Table 3 (best results in bold).

		MAPE	
Algorithm	SFS	blur-faces	mask-detector
Ridge Regression	Yes	**9.6**	17.19
	No	43.01	73.63
Decision Tree	Yes	22.01	24.73
	No	27.8	34.04
XGBoost	Yes	17.6	**10.09**
	No	21.3	24.37
Random Forest	Yes	19.12	13.24
	No	60.54	32.53

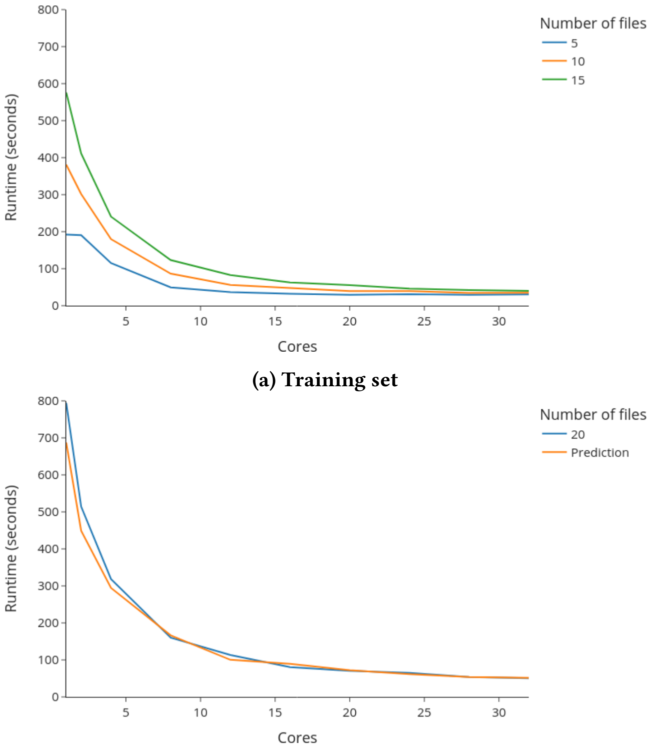
Table 3: MAPE [%] on extrapolation

We combined the two best models for the individual services and performed extrapolation on the data from the last test (20 videos). The original data and prediction results are shown in Figure 7. Note that the runtime measures the interval between the start of the first job and the completion of the last, including wait times and overhead. The MAPE of the combined models is 9.75%, proving the good extrapolation capabilities of the models, which allow us to make reliable predictions on larger and unseen inputs.

7 CONCLUSIONS AND FUTURE WORK

In this work, we developed an auto-profiling and performance prediction framework for OSCAR that can generate machine learning performance models from the collected data. The framework proved its efficiency, greatly reducing the time needed to set up OSCAR, collect the logs and process them manually. The results obtained from the experimental campaigns are promising, with the performance models having a MAPE around 10% on extrapolation.

[4] https://doi.org/10.5281/zenodo.7561987



(a) Training set

(b) Test set and prediction

Figure 7: Extrapolation results

Future works include expanding *OSCAR-P* capabilities to enable the profiling of heterogeneous clusters, and testing the whole framework on industrial case studies.

ACKNOWLEDGMENT

The European Commission has funded this work under the H2020 grant n. 101016577 AI-SPRINT: AI in Secure Privacy pReserving computINg conTinuum. OSCAR has also been developed under the project PDC2021-120844-I00, funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR, and under grant PID2020-113126RB-I00, funded by MCIN/AEI/10.13039/ 501100011033. Finally, aMLLibrary has been funded under the H2020 grant n. 956137 LIGATE: LIgand Generator and portable drug discovery platform AT Exascale.

REFERENCES

[1] Albuquerque Jr, L. F., Ferraz, F. S., Oliveira, R., and Galdino, S. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. In *ICSEA* (2017), pp. 206–212.

[2] Ataie, E., Evangelinou, A., Gianniti, E., and Ardagna, D. A hybrid machine learning approach for performance modeling of cloud-based big data applications. *The Computer Journal* (2021).

[3] Copik, M., Kwasniewski, G., et al. Sebs: A serverless benchmark suite for function-as-a-service computing. In *ICM* (2021), pp. 64–78.

[4] Das, A., Patterson, S., and Wittie, M. Edgebench: Benchmarking edge computing platforms. In *UCC* (2018), IEEE, pp. 175–180.

[5] Disabato, S., Roveri, M., and Alippi, C. Distributed deep convolutional neural networks for the internet-of-things. *IEEE Transactions on Computers 70*, 8 (2021), 1239–1252.

[6] Draper, N. R., and Smith, H. *Applied regression analysis*, vol. 326. John Wiley & Sons, 1998.

[7] Ferri, F. J., Pudil, P., Hatef, M., and Kittler, J. Comparative study of techniques for large-scale feature selection. In *Machine Intelligence and Pattern Recognition*, vol. 16. Elsevier, 1994, pp. 403–413.

[8] Grohmann, J., Straesser, M., Chalbani, A., et al. Suanming: Explainable prediction of performance degradations in microservice applications. In *ICPE* (2021), pp. 165–176.

[9] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik., K. C. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, 1984.

[10] Maros, A., Murai, F., et al. Machine learning for performance prediction of spark cloud applications. In *CLOUD* (2019), IEEE, pp. 99–106.

[11] McChesney, J., Wang, N., Tanwer, A., de Lara, E., and Varghese, B. Defog: fog computing benchmarks. In *ACM/IEEE SEC* (2019), pp. 47–58.

[12] Nawrocki, P., and Osypanka, P. Cloud resource demand prediction using machine learning in the context of qos parameters. *Journal of Grid Computing 19*, 2 (2021), 1–20.

[13] Perez, A., Risco, S., Naranjo, D. M., Caballer, M., and Molto, G. On-Premises Serverless Computing for Event-Driven Data Processing Applications. In *CLOUD* (jul 2019), IEEE, pp. 414–421.

[14] Risco, S., Moltó, G., Naranjo, D. M., and Blanquer, I. Serverless workflows for containerised applications in the cloud continuum. *Journal of Grid Computing 19*, 3 (2021), 1–18.

[15] Satyanarayanan, M. The emergence of edge computing. *Computer 50*, 1 (2017), 30–39.

[16] Sewak, M., and Singh, S. Winning in the era of serverless computing and function as a service. In *I2CT* (2018), IEEE, pp. 1–5.

[17] Vu, K., Hartley, K., and Kankanhalli, A. Predictors of cloud computing adoption: A cross-country study. *Telematics and Informatics 52* (sep 2020), 101–426.