OSCAR-P and aMLLibrary: Profiling and Predicting the Performance of FaaS-based Applications in Computing Continua

Roberto Sala, Bruno Guindani, Enrico Galimberti, Federica Filippini, Hamta Sedghani, Danilo Ardagna[a], Sebastián Risco, Germán Moltó, Miguel Caballer[b]

[a] *Politecnico di Milano, Italy*
Email: name.surname@polimi.it
[b] *Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022, Valencia, España*
Email: gmolto@dsic.upv.es,{serisgal,micafer}@i3m.upv.es

Abstract

This paper proposes an automated framework for efficient application profiling and training of Machine Learning (ML) performance models, composed of two parts: OSCAR-P and aMLLibrary. OSCAR-P is an auto-profiling tool designed to automatically test serverless application workflows running on multiple hardware and node combinations in cloud and edge environments. OSCAR-P obtains relevant profiling information on the execution time of the individual application components. These data are later used by aMLLibrary to train ML-based performance models. This makes it possible to predict the performance of applications on unseen configurations. We test our framework on clusters with different architectures (x86 and arm64) and workloads, considering multi-component use-case applications. This extensive experimental campaign proves the efficiency of OSCAR-P and aMLLibrary, significantly reducing the time needed for the application profiling, data collection, and data processing. The preliminary results obtained on the ML performance models accuracy show a Mean Absolute Percentage Error lower than 30% in all the considered scenarios.

Keywords: edge computing, computing continuum, performance profiling, machine learning

1. Introduction

Cloud computing is widely adopted today and has been used for years as the standard computing paradigm for enterprise-level distributed systems [1]. Despite its significant advantages in terms of costs and computational power accessibility, it is associated with significant challenges since data offloading leads to potential delays and increased expenses. Therefore, it falls short in meeting the demands of contemporary applications, often Artificial Intelligence (AI)-based and associated with strict or almost real-time processing constraints. Cloud users exhibit high sensitivity to delays and fluctuations, and they benefit from a newly-emerging paradigm called edge computing, striving to relocate applications closer to the point of data generation [2]. This approach offers several advantages: i) lower latency: by moving part of the computation where the data resides, we remove the round-trip-time delays needed to access the remote cloud data centers, resulting in faster response times and better performance; ii) reduced bandwidth usage: local processing at the edge removes the need to send vast amounts of data through the network, saving bandwidth and avoiding bottlenecks; iii) improved privacy: data processed on edge devices can be anonymized on the spot before communication, ensuring that potentially sensitive information is never shared with a central server; iv) better scalability: edge devices are usually cheap, and adding more to help with data processing is easy to implement and economical. However, edge resources cannot be seen as a possible replacement for cloud computing since they are usually characterized by lower computational capacity and consequently become a bottleneck in the computation. An integrated edge-cloud computing continuum enabling application components with different demands to be executed on the most appropriate resources is crucial to supporting complex application workflows effectively.

Together with the introduction of the computing continuum paradigm, another significant aspect characterizing the computational landscape in recent years is represented by the quick rise in popularity [3][4] of the Function as a Service (FaaS) model. It breaks down complex applications in workflows

of small, usually short-lived components, which run on reusable services consisting of stateless containers activated by suitable events (e.g., a file upload). Using containers instead of virtual machines (VMs) reduces both the development/deployment complexity and the resource usage. Containers can be dynamically created or destroyed in response to workload variations, and their stateless nature allows them to be reused to serve another event as soon as the previous computation completes. This increases the flexibility and makes the FaaS model suitable for scenarios characterized by light average workload interleaved with activity peaks. Finally, the FaaS model in public clouds is characterized by per-millisecond costs bound to the actual resource usage [5].

Despite the benefits associated with distributed computing and FaaS models, evaluating the performance of a complex application whose components may be allocated at different levels of the computing continuum poses significant challenges. Automated tools are needed to support the components profiling, measuring their execution times while exploiting variable resources and hardware configurations. Furthermore, accurately predicting the performance of a given application at a target configuration is the key to proper planning and runtime management of the available resources. As mentioned, edge devices with limited capacity often become the system bottleneck in case of workload spikes, and actions need to be taken to meet the performance objectives (e.g., limiting application execution time with a fixed threshold).

This paper proposes an integrated framework for the profiling and performance modeling of FaaS-based applications running in computing continua. In our setting, the application execution is supported by OSCAR [6], a state-of-the-art runtime environment that aims to create a highly parallel, event-driven, pipelined serverless environment to execute general-purpose data-processing computing applications. This includes the usage of AWS Lambda to execute event-driven Docker-based applications. On top of OSCAR, we developed *OSCAR-P* (OSCAR-Profiler), a novel auto-profiling tool that can automatically test application workflows on different hardware and node combinations, gathering relevant information on the execution time of the individual components.

Moreover, OSCAR-P leverages *aMLLibrary* [7], an open-source Machine Learning (ML) package we designed to automatically develop performance-predicting models for every service/resource pair, which can be combined to forecast the runtime of the entire workflow. Compared to tools such as Kubeflow Katib [8], aMLLibrary does not require extensive configuration and deployment of computational resources, is more straightforward and portable, and supports utilities specific to performance modeling such as feature augmentation and selection.

The usage of ML for performance prediction is motivated by the ever-increasing complexity of modern software. Often, the impact of input configurations and settings on software performance is not straightforward, preventing the use of analytical, white-box methods such as Petri nets [9] and queuing networks [10]. Even in the simplest scenarios where accurate modeling is possible, the hypothesis and assumptions underlying analytical formulations prevent them from covering all use cases. Therefore, an approach that does not require any knowledge of the internal details of the system, generally referred to as a black-box technique, is often preferred. In particular, ML is the prominent category of black-box approaches for performance analysis [11]. The models built by aMLLibrary make it possible to predict the performance (e.g., the average execution time) of an application on unseen configurations with high accuracy. This allows to limit the initial application profiling campaign since some configurations do not need to be tested directly. In general, this prediction capability enables efficient design-time decision making [12] and runtime resource management [13, 14, 15], which are essential tasks for numerous cloud systems.

This paper extends our initial works in [16] and [7]. Compared to these papers, we i) discuss our contributions with a much larger level of detail, ii) add support for synchronous calls, partitioned applications, and AWS Lambda functions into the framework, and iii) extend our experimental campaign with four new applications representing different real-life use cases.

This work is organized as follows. Section 2 presents relevant literature works. Section 3 provides a summary of the OSCAR framework and its architecture. Section 4 thoroughly illustrates the OSCAR-P goals and its architecture,

while Section 5 illustrates the capabilities of aMLLibrary and the performance analysis it supports. Section 6 describes the performance models used in our experiments. Section 7 focuses on the experimental scenarios to validate our tools. Conclusions and future works are discussed in Section 8.

2. Related work

This section overviews recent works in technological fields relevant to this paper, namely benchmarking tools for FaaS and edge systems (Section 2.1) and ML-based performance modeling (Section 2.2).

2.1. Benchmarking

As FaaS and edge computing rose to popularity, several cloud providers started offering their own FaaS platforms, each with different underlying technologies. At the same time, many researchers tried to address the challenges of benchmarking applications deployed partially on the edge and partially on the cloud. SeBS (Serverless Benchmark Suite) [17] is a comprehensive benchmarking tool that systematically supports various applications, cloud resources, and commercial providers. EdgeBench [3] focuses instead on two providers, AWS IoT Greengrass and Microsoft Azure IoT Edge, using different performance metrics, and also compares the edge frameworks performance to the respective cloud-only implementations. The framework in [18] allows the user to specify the latency and cost requirements of application services, and it determines whether it is better to deploy them on the cloud or the edge. It is meant for applications on small smart edge devices, like smart cameras. DeFog [19] presents a benchmarking tool that tests an application across a cloud-only, edge-only, and cloud-edge environment by comparing performance across different deployments. The tool collects relevant metrics to understand how the application services can be better distributed across the computing continuum. [20] presents Ilúvatar, a low-latency FaaS control plane. Its design principles significantly reduce overhead compared to state-of-the-art control planes. It introduces function-size-aware queueing policies to regulate worker load, improve latency, and balance utilization. Finally, [21] proposes a methodology for generating synthetic traces from

large-scale production workloads, focusing on Azure Functions across various scales and load factors. The authors introduce In-Vitro, which uses an iterative approach to sample functions, minimizing the Wasserstein distance between the full trace and sampled trace distributions of essential workload characteristics. Their experiments show that In-Vitro significantly enhances representativeness compared to random sampling methods. Finally, [22] introduces FaaSRail, a tool that reconciles open-source workloads with real-world FaaS traces in production environments. FaaSRail maps real traces to benchmarking suites, downscales invocations, emulates time variability, and replicates burstiness.

2.2. Performance models

Many researchers focus on analyzing and predicting the performance of applications running on edge systems and, recently, the majority use ML models instead of analytical models. For instance, [23] proposes some linear regression models to predict the execution time of Convolutional Neural Networks on edge devices, given constraints on memory and processing load. [15] focuses predominantly on mobile devices, specifically the cloud-only data-processing approach, and compares its efficiency to a partitioned approach where the computation is split between the cloud and mobile devices. In both [23] and [15], the ML performance models predict the application execution time by considering the number of floating point operations per second of the devices. Authors of [24] 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. Similarly, [25] applies popular ML techniques to a workload prediction analysis on HTTP servers, showing that these techniques all achieve good predicting capabilities.

Performance modeling through ML is also applied to FaaS platforms. Suan-Ming [26] is an integrated framework for learning performance degradation of microservice-based systems running in public and private clouds, using several ML algorithms. Authors of [27] propose the creation of a model to predict performance metrics by considering the application average response time for warm

and cold requests, the requests arrival rate, and the system expiration threshold. In [28], the same authors propose a steady-state analytical performance model for improving the QoS of FaaS and reducing operating costs, trading off cost and performance by making the platform workload-aware.

Unlike previous works, OSCAR-P is focused on benchmarking the OSCAR framework, which can be deployed on top of any commercial and on-premises Cloud Management Platform (e.g., OpenStack) and hence has the critical benefit of being cloud-provider agnostic. To the best of our knowledge, the literature proposes no other automated framework that integrates profiling with ML performance model building for the computing continuum.

3. The OSCAR Framework

OSCAR[1] is an open-source framework to quickly and efficiently support event-driven, data-processing, serverless applications packaged as Docker images along the computing continuum [29] (see Figure 1). They are executed in elastic Kubernetes (K8s) clusters that can be dynamically provisioned across multiple cloud back-ends thanks to the Infrastructure Manager[2] (IM), an open-source tool that automatically creates and manages virtual infrastructures. Horizontal elasticity is provided by CLUES[3], a management system that provisions and terminates the worker nodes to accommodate the cluster workload. While one could choose a different container orchestration tool like Docker Swarm[4], Kubernetes is preferred due to its greater scalability, richer ecosystem, and more robust features for complex deployments, making it a more versatile and powerful platform. OSCAR can also perform cloud bursting into SCAR [30], facilitating the execution of Docker-based applications in AWS Lambda.

The framework architecture is shown in Figure 2. The following components are deployed inside the K8s cluster to support the OSCAR platform: i) MinIO[5],

[1] https://oscar.grycap.net
[2] http://www.grycap.upv.es/im
[3] http://github.com/grycap/clues
[4] https://docs.docker.com/engine/swarm/
[5] http://minio.io

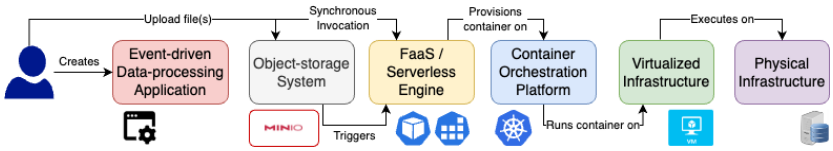


Figure 1: High-level architecture for event-driven container-based data-processing app.

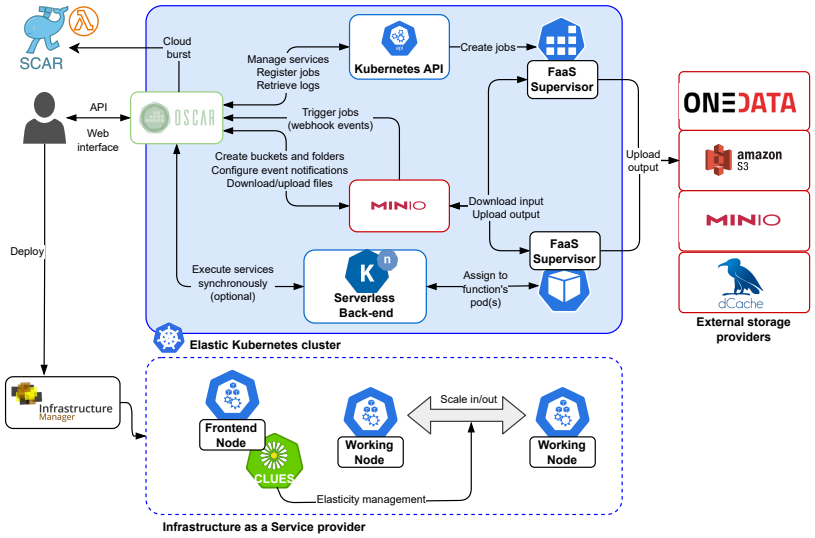


Figure 2: OSCAR architecture.

a high-performance multi-cloud object storage server ; ii) Knative[6], an open-source enterprise-level solution to build serverless and event-driven applications, used to support synchronous invocations, and 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 object storage service that provides 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; and iv) dCache, a system for storing and retrieving vast amounts of data, distributed among a large number of server nodes. The cluster can be managed through its REST API, either via web interface or command line.

[6] `https://knative.dev/docs/`

Applications are created as services on the OSCAR cluster by providing: i) a Docker image to be used, ii) the input and output buckets of each service, and iii) a shell script to be executed inside the container. OSCAR can create services by writing and uploading a Function Definition Language (FDL) configuration file on the cluster. Once 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, which may be the input of another service. If so, that service is automatically triggered and either executed immediately (if possible) or added to the job queue. Uploading multiple files triggers the highest possible number of parallel invocations supported by the specific cluster; all remaining function calls are scheduled for execution as soon as the running services are complete.

Serverless systems typically suffer from *cold start* (see analyses in Sections 7.3.2 and 7.6), where the first invocations incur increased latency due to the underlying capacity allocation. This is mitigated in OSCAR in two ways: pre-fetching the Docker images into the K8s cluster nodes to rapidly deploy the containers and maintaining a pool of user-defined active containers so that synchronous invocations via Knative can execute quickly. Synchronous calls are made exclusively on the first component, while subsequent components are triggered by the arrival of files in the corresponding input buckets.

When performing inference via a Deep Neural Network (DNN), a single component can be replaced by two or more equivalent components running sequentially. This is particularly relevant for AI applications since we can partition DNNs to run on different resources, exploiting more efficient alternative deployments according to the workload conditions [12, 13].

Despite our work being based on OSCAR, OSCAR-P and the prediction techniques we propose apply to other serverless frameworks too. Indeed, the K8s and Knative platforms leveraged by OSCAR are widely used for FaaS both on-premises and in the cloud (see, e.g., OpenFaaS and Fission). Thus, our approach can extend to any framework with compatible underlying technology.

4. OSCAR-P

This section describes OSCAR-P, the OSCAR Profiler that provides, alongside aMLLibrary, the novel contribution of this paper. OSCAR-P is built around OSCAR and its components (like K8s and MinIO) and acts as a director, configuring and coordinating the profiling activities and the data collection. The profiler aims to simplify and automate testing specific OSCAR application workflows on different hardware configurations, gathering data to train ML models through the aMLLibrary. These models perform predictions on the response time of OSCAR/SCAR workflow components.

OSCAR-P requires the following input information: i) the description of the physical or virtual resources to test, including access keys to AWS services and the private keys of physical clusters; ii) the description of the application components, e.g., their Docker images and hardware requirements; iii) the application parameters, including the input files, the distribution of data uploads (for asynchronous calls) and HTTP requests (for synchronous ones), and the parallelism levels; iv) the settings for ML models training in the aMLLibrary (see Section 5). All these inputs are encapsulated in YAML files, eliminating the necessity for user interaction with the code. A template of the individual configuration files is accessible on Zenodo [31].

In the following sections, we will refer to a specific cluster setting to run the components of an application as a *deployment*. OSCAR-P efficiently tests individual components after running the full workflow once, to reduce the time needed to test all possible deployments. Each OSCAR-P sub-component manages one step of the profiling activities, as illustrated in Figure 3.

4.1. Input files parser

Starting from the input files, OSCAR-P lists all the *testing units*, i.e., the valid component-resource assignments within a deployment. If a component is partitioned, all the partitions are considered part of the same testing unit.

As an example, illustrated in Figure 4, we consider an application including a single component (*C1*) that can be deployed on a cluster of Raspberry Pi

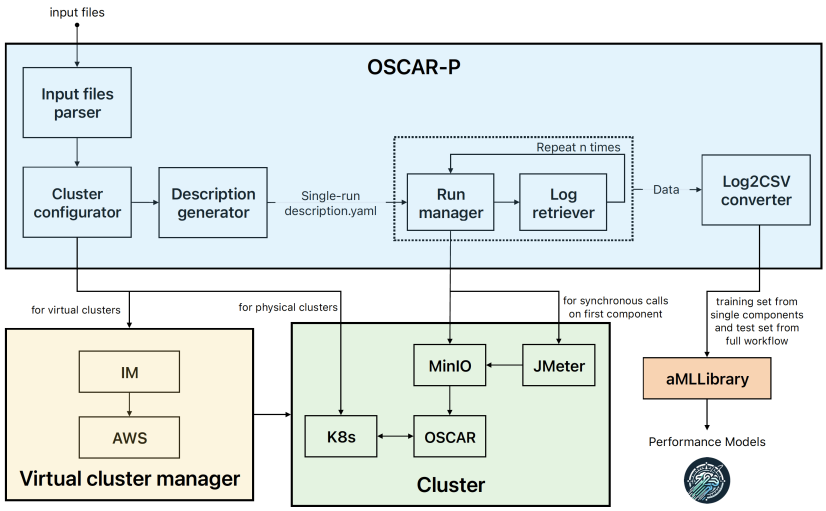


Figure 3: Profiling activity steps, OSCAR-P sub-components and interactions.

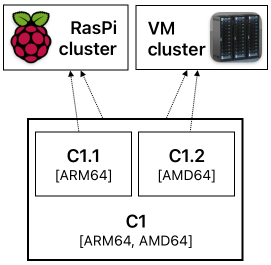


Figure 4: Example app.

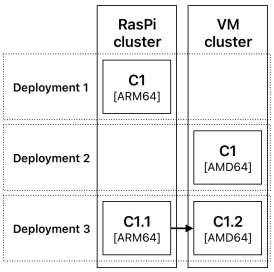


Figure 5: Testing units.

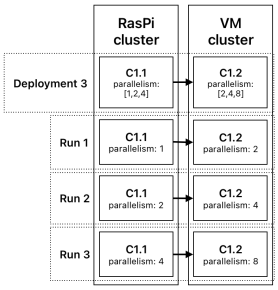


Figure 6: Deployment.

(arm64 architecture) or VMs (amd64 architecture). The component can also be partitioned in *C1.1* and *C1.2*, each available for a single architecture (arm64 and amd64, respectively), generating the following testing units: i) C1 on the Raspberry Pi cluster; ii) C1 on the VM cluster; iii) C1.1 on the Raspberry Pi cluster and C1.2 on the VM cluster. In this case, OSCAR-P creates a list of all the deployments to test, i.e., all the possible combinations of testing units of the different components (see Figure 5). Each deployment contains a set of configurations, i.e., the cluster settings, but with different *parallelism levels* for each component (i.e., the number of parallel instances executed). Figure 6 shows an example for a specific deployment.

4.2. Cluster configurator and description generation

Before testing a deployment, the clusters need to be set up and correctly configured. As shown in Figure 3, OSCAR-P generates the necessary files for IM and K8s to instantiate both physical and virtual nodes. During profiling, the cluster configuration undergoes periodic adjustments for subsequent runs, directly altering the number of worker nodes. When OSCAR-P completes its runs, the cluster configurator mandates IM to destroy the virtual clusters.

Once the cluster configuration for a specific test is completed, OSCAR-P generates a descriptive YAML file for that deployment detailing all information needed to run the test. This file contains the list of all the service requirements and a description of the clusters in use, and is updated with all the subsequent deployment runs, serving as a summary of the entire testing campaign.

4.3. Run manager

At this stage, the description YAML file (mentioned in Figure 3) is parsed to extract all the relevant information for the current run. The OSCAR cluster is cleaned by removing all the remnants of past executions (if any) to ensure they do not interfere with the current execution. Finally, OSCAR-P generates an FDL file with the information needed to build the new workflow (i.e., the required services and buckets) and applies it to the OSCAR cluster. Since OSCAR (and FaaS in general) is event-driven, moving the required files in the input bucket of the first service triggers its execution and starts the run. While for asynchronous calls the file upload is managed directly by the machine running OSCAR-P, for synchronous calls it is handled by JMeter[7], an open-source software for load testing. In particular, the JMeter client is automatically instantiated by IM before the run starts. Given the input files and the desired load, JMeter can upload files either at a constant rate or by sampling arrival times from an exponential distribution. Note that in the case of asynchronous calls, OSCAR-P input files are uploaded to a MinIO storage bucket and then

[7] `https://jmeter.apache.org`

moved to the input bucket. This method allows to measure the processing time of an external request independently of the network connecting the end-user to the OSCAR cluster, something especially useful for simulating peak loads with multiple file uploads. For the same reason, JMeter clusters are placed at the front-end component location for synchronous calls.

When testing the full application workflow, once the execution of the first component is triggered, the process proceeds automatically, as the output of each component serves as the input for the following one. On the other hand, single services are connected to temporary input/output buckets to control their execution, as depicted in Figure 7.

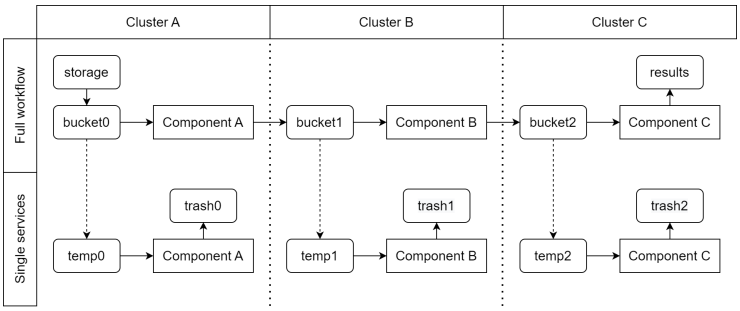


Figure 7: Full workflow and single component testing.

4.4. Log retriever

Upon completion of a run, OSCAR-P collects and processes the relevant logs from OSCAR, Kubernetes (K8s), and potentially JMeter. These logs provide detailed information, including the scheduling time of a job (i.e., a single component execution), the creation time of the corresponding *pod* (the deployable computing unit in K8s), as well as the application's execution start and end times. This data is crucial for monitoring delays, wait times, and overheads.

The collected data are organized in four time intervals, as reported in Figure 8. The Network Time Protocol guarantees synchronization among the cluster nodes. The initial *wait* interval starts when a file is uploaded in the input bucket. The next interval corresponds to the *pod creation* by K8s. *Overhead* starts when the pod is created and the underlying container is started, while

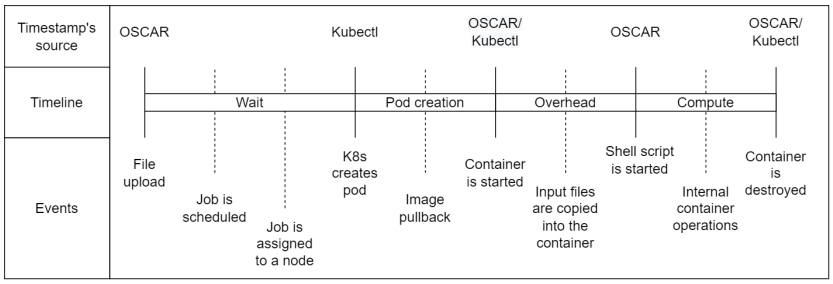


Figure 8: Timeline of a single job.

compute begins when the container log (obtained from OSCAR) reports that the application execution has already started. This last interval ends when the job terminates, and the corresponding container is destroyed.

The *Log2CSV-converter* OSCAR-P component merges the data collected for each tested configuration into a single CSV file, used by aMLLibrary to train a performance model for every service/resource pair.

5. aMLLibrary

aMLLibrary [7] is the last component in the OSCAR-P pipeline and builds the performance prediction models that are the final artifact produced by the framework. aMLLibrary is an open-source, high-level Python package that is based on the scikit-learn toolkit[8] and allows parallel training of multiple supervised ML models, supporting several pre-processing techniques, feature selection, and hyperparameter tuning. It receives the profiling data collected by OSCAR-P and builds regression models to predict the performance of both the application individual components and the full workflow.

Overall, aMLLibrary implements an autoML solution, i.e., it trains multiple regression models and automatically selects the most accurate one. By leveraging a highly accurate ML model, the performance of a specific application configuration can be estimated without direct observation, consequently reducing the size of the initial OSCAR-P profiling campaign. Furthermore, the generalization capabilities of ML models allow to estimate the performance of

[8] `https://scikit-learn.org`

a given application component for design-time decisions (e.g., how many nodes to allocate to each component of the OSCAR pipeline) [12] or runtime adaptive resource management (e.g., scaling nodes up and down according to runtime load variations) [13]. Also, when profiling some components of an application on a different hardware, we can can exploit the results from previous profiling campaigns for the unchanged components to reduce the overall profiling time.

As mentioned in Section 1, general black-box approaches that do not require any knowledge of the internal details of the system, such as ML methods, are becoming popular in studying software performance because of the limitations of analytical, white-box models that often rely on unrealistic assumptions. Moreover, it would be impossible to formulate one single analytical model to cover all possible target applications. Creating multiple individual, domain-specific models, each requiring thorough domain expertise, extensive effort, and significant profiling costs, is hardly affordable for large computing environments and applications. On the other hand, simulation-based approaches are characterized by excessive runtime evaluation costs, which makes it difficult to employ them for effective resource management decisions.

Within the OSCAR-P pipeline, the main strengths of the library are its ease of use, wide range of options, and robustness and efficiency of its training operations. We discuss the first aspect in Section 5.1 and the rest in Section 5.2.

5.1. Configuration files

aMLLibrary is controlled by a simple text configuration file, given as input to the OSCAR-P framework (see Appendix A.1 for a basic example of a configuration file). The file includes general settings for the campaign configuration, such as the ML models to build and the methods for hyperparameter selection and validation. It also lists the data pre-processing steps, if any. The user can specify these settings at the beginning of the OSCAR-P profiling campaign in a concise and declarative way without the need to write any Python code. An equivalent Python script would require hundreds of lines of code to achieve the same result (see the equivalent script to the configuration file in the Zenodo

repository [31]). The provided default settings for hyperparameter tuning are general enough to allow the tuning mechanism to find the appropriate parameter values, usually without requiring modifications by the user.

5.2. Features

We show the high-level architecture of aMLLibrary in Figure 9. The library

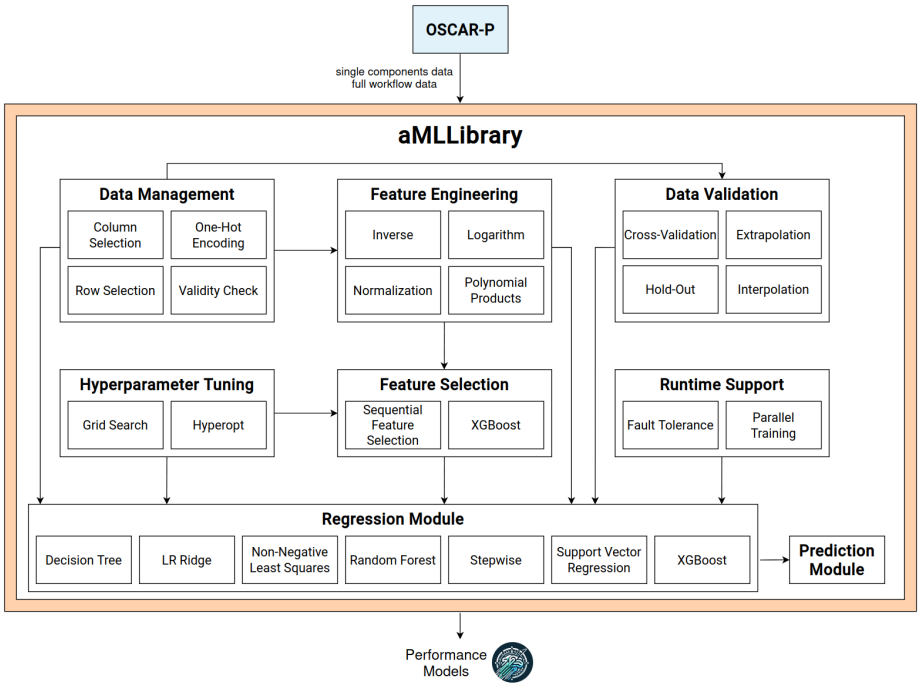


Figure 9: aMLLibrary components.

has several useful perks for building performance models. The parallel training of multiple models is supported: the user can specify the number of parallel cores to use, and the library automatically distributes the training experiments evenly among the parallel workers, even if the underlying scikit-learn models are limited to single-thread execution. Furthermore, the library implements a fault tolerance mechanism by saving incremental progress checkpoints.

Users have complete control over the experimental campaign thanks to the many configuration options and flags available. The library currently supports Decision Tree (DT), Non-Negative Least Squares (NNLS), Random Forest (RF),

Ridge Linear Regression, Stepwise (a linear regression model which integrates the Draper-Smith feature selection technique [32]), 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 tested, or automatically via Bayesian Optimization (BO). In the latter case, the integrated HyperOpt library[9] is used, and the user must provide prior probability distributions on the hyperparameters to optimize.

aMLLibrary includes plugins for several data pre-processing techniques, such as data normalization and one-hot encoding for discrete features, as well as other convenient tools, such as row selection and data validity checks. It supports automatic feature engineering, e.g., computation of logarithms, inverse values, and feature products/polynomial expansion up to a given degree. These tools can be helpful 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) [33] and XGBoost importance weight selection.

5.3. Validation methods

The user can choose among several validation methods to compute the Mean Absolute Percentage Error (MAPE) of a model, which is defined as $MAPE(y, \widehat{y}) = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \widehat{y_i}}{y_i}\right|$, where y and $\widehat{y}$ are the vectors of true and predicted values, respectively. In the performance evaluation literature [34], MAPEs lower than 30% for application execution times are usually considered enough to support runtime management decisions or capacity planning.

Besides classical validation methods such as train-test splitting and Cross-Validation (CV), aMLLibrary offers methods such as interpolation and extrapolation, often used in ICT settings to build custom test sets [35]. Here, interpolation means keeping some feature values within the feature space out of the training set and placing them in the test set to check the ability of the model

[9] `https://github.com/hyperopt/hyperopt`

to "fill in the blanks" of the feature space. The goal of such validation is to verify whether it is possible to reduce the dimension of the training set, therefore conducting a profiling campaign with a coarser granularity (e.g., varying a cluster size by 8 cores instead of 2 or 4 cores). Vice versa, extrapolation means keeping an entire area of the search space out of the training set to test the predicting capabilities of the model in an unexplored part of the feature space. After the validation phase, aMLLibrary chooses the best model according to its validation MAPE and saves it to a binary file so that it can be used for further inference. Finally, the library includes a prediction module that can be used to perform inference with an already-trained regression model.

6. Performance models

This section presents the performance models used in our experimental scenarios, discussed in Section 7. Our general objective is to train ML performance models for individual OSCAR workflow services and to predict the performance of the full workflow. Due to the substantial execution times and associated costs, we aim to avoid testing all possible combinations of a potentially complex application (see, e.g., "recipe-transcriber" in Section 7.5), where the number of experiments would scale exponentially with the number of components and resources to test. This prediction poses significant challenges, as the input rate of the subsequent component depends on the output rate of the previous component, further complicated by its configuration-specific nature. Additionally, we aim to generate ML models for individual components to facilitate reuse in case of setting changes, thereby avoiding the need to rerun the profiling campaign.

In tests with asynchronous calls, we seek to estimate the total processing time for a collection of input files, considering that these files are processed all at once. However, building a unified model encompassing all application components is not straightforward because of the *pipeline effect*. This effect arises during the execution of the full application, allowing subsequent components to start as soon as their predecessors have been completed. The performance models for single services do not capture this partial overlap (see Figure 10), as they assume

that a component only starts after all instances of the previous one have finished, resulting in an overly conservative estimate of execution time. To mitigate this issue, we adjust the predictions for each component by subtracting the average execution time for a single job from the previous component.

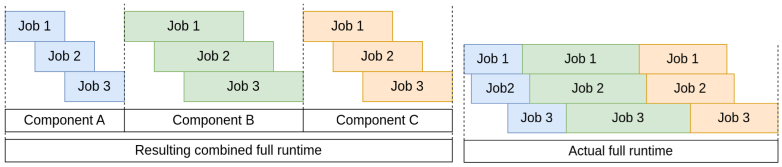


Figure 10: "Pipeline effect" illustration.

On the other hand, tests with synchronous calls aim to predict the execution time of individual inputs considering a continuous flow of requests with arrival rate λ. The models for each component need to account for not only the number of parallel pods, but also the arrival rate of requests from the previous component. In principle, the arrival rate of requests should match the value set by the user; however, this is not the case when one of the components saturates the available resources, which leads to a reduction in throughput compared to the initial rate. Consider for example the scenario depicted in Figure 11, where the application includes two components. If the first component saturates, the arrival rate of requests to the second component would be $\lambda_1 < \lambda$. Consequently, using the input arrival rate λ to predict the performance of the second component would lead to overly conservative predictions for the full workflow. To address this challenge, we generate models for all individual components to estimate their average execution times (T_1 and T_2 in the figure) and models to predict the throughput between one component and another (i.e., λ_1).

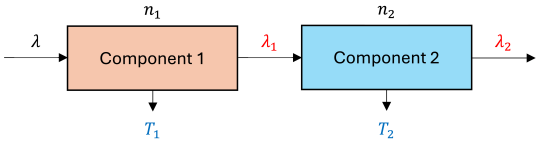


Figure 11: Overview of a test with synchronous calls on a two-component application, where the user-defined parameters for the specific run are (λ, n_1, n_2).

Note that this work focuses on the computational aspects of the cloud-edge continuum without addressing network communication delays. This is equivalent to the assumption that the network is not a performance bottleneck. Profiling and modeling its performance falls beyond the scope of our research.

7. Experimental analysis

This section overviews the experiments we perform with our framework, with the goal of demonstrating its usefulness in the collection of training data, post-hoc analysis of executions, and generation of highly accurate ML models. We consider four example applications (Section 7.1), testing them across different clusters and configurations. Section 7.2 overviews the ML techniques exploited for performance model generation and their hyperparameters. Sections 7.3 through 7.6 illustrate the profiling setup and results for the four applications. Finally, in Section 7.7, we discuss the cost and time of profiling applications, showing the advantages of using OSCAR-P and aMLLibrary. The source code, input files, log data, and experimental results are available on Zenodo [31].

7.1. Target applications

The four applications presented in the following subsections represent three examples of AI workflows characterized by a variable number of heterogeneous components, and one simple Fibonacci application. In particular, we consider object detection and video/audio transcription as representative AI tasks with different demands in terms of computational power. We focus on AI applications because of the blooming interest in this field, and especially in the deployment of these applications at the edge. Finally, the Fibonacci application differs from the previous cases in that the inputs are characterized by few bytes, allowing hundreds of requests per second. Note that while we conduct numerous test campaigns to validate our framework for the first application, we conduct fewer experiments for the larger ones due to time and budget constraints.

7.1.1. Mask detection

The first application we consider (see Figure 12) was initially proposed in [29] and consists of two components: 1) *blurry-faces*, which receives a video as input,

extracts a frame every 5 seconds via the *FFmpeg* tool, and anonymizes it by blurring the faces of any person appearing in it by performing face recognition; 2) *mask-detector*, which receives an image as input, detects the faces appearing in it, and decides whether or not they are wearing a mask.

The faces detection task is performed in both components via the YOLO (You Only Look Once) network, powered by the TensorFlow[10] object detection API and trained on the WIDER FACE dataset[11]. This type of network is designed to detect multiple objects while examining each image only once, resulting in significantly faster performance while maintaining high accuracy.

By deploying the *blurry-faces* component on the edge, we ensure the privacy of the people appearing in videos (because the server only sees anonymized frames) and reduce the latency associated with data transfer to the cloud.

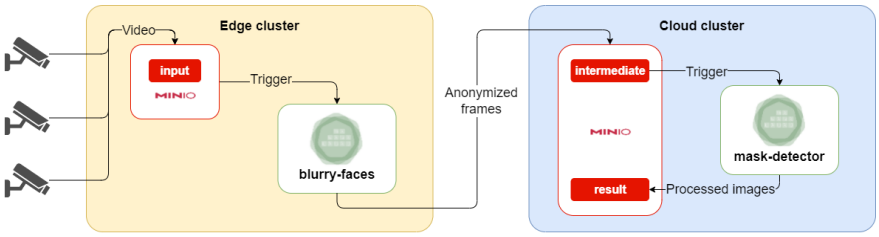


Figure 12: Mask detection workflow.

7.1.2. Video searcher

The second application (see Figure 13) creates the transcription of a video, searches for one or more user-specified words in it, and returns a clip of the original video for every match. It includes six components:

1. *ffmpeg_0* saves the audio track from the input video as a WAV file;
2. *Librosa* (based on the homonymous Python package[12]) analyses the audio track, looking for drops in the noise levels, which may indicate the end of a sentence. Each time the noise drops under a user-defined threshold, the component writes the corresponding timestamp in a text file;

[10] `https://www.tensorflow.org`
[11] `http://shuoyang1213.me/WIDERFACE`
[12] `https://librosa.org`

3. *ffmpeg_1* uses the timestamps produced by the previous component to cut the video into clips containing single sentences;
4. *ffmpeg_2* extracts the audio track and lowers its sample rate to 16 kHz;
5. *DeepSpeech* transcribes the audio track of the clip using an open-source recurrent neural network implemented by Mozilla [36];
6. *Grep* searches for the specified word(s) inside the transcription and copies the related video clip into the output bucket if it finds a match.

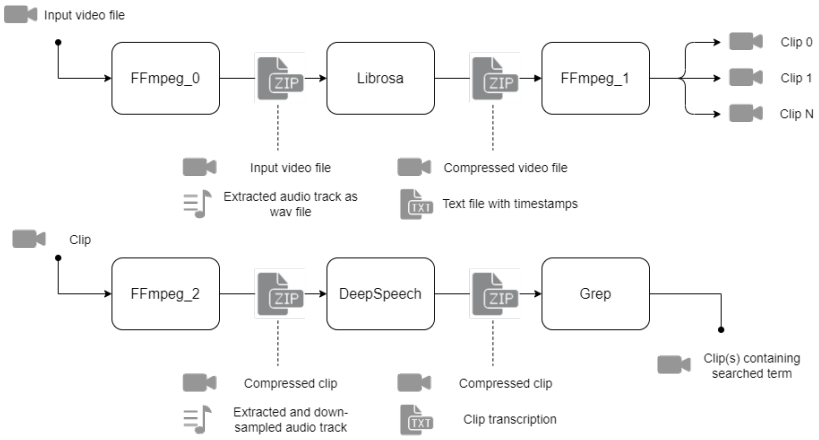


Figure 13: Video searcher workflow.

7.1.3. Recipe transcriber

The third application transcribes and recognizes the ingredients of a recipe, and it consists of seven components. Components 1-5 are the same as in the video searcher application. The last two are as follows:

6. *ffmpeg_3* extracts a frame every 5 seconds from the original video;
7. *object-detector* identifies and localizes objects in a box.

7.1.4. Fibonacci application

The last application (as in [37]) computes the N-th Fibonacci number where N is specified in the input file. In this case, the input file is only a few bytes, allowing exogenous loads up to hundreds of requests per second. The application

has a single component that performs an efficient, non-recursive computation of the requested Fibonacci number to avoid memory overheads.

7.2. ML models and hyperparameters settings

Once the profiling process is completed, aMLLibrary uses the collected data to build and compare the following ML models: XGBoost, Ridge Regression, Decision Tree, and Random Forest. The hyperparameters are obtained via BO tuning (see Section 5), fixing the maximum number of evaluated hyperparameter sets to 10 and considering the values or prior distributions listed in Table A.4 reported in Appendix A.2.

For each experiment, we train two instances of each regression model, one involving feature selection (see Section 5) and the other without it. Each row of the training dataset provided by OSCAR-P represents an individual application component execution, while columns (also referred to as "features") identify the corresponding configuration characteristics in terms of, e.g., parallelism level. To cope with the execution time variability in practical settings, OSCAR-P profiles each configuration multiple times (three in our case), resulting in as many dataset rows with potentially different performance measurements.

As mentioned, the primary factor influencing performance prediction is the *parallelism level*, which usually corresponds to the maximum number of available cores for all services (a relevant exception to this assumption will be discussed in Section 7.5). Since for all the applications and components we consider each pod requires exactly one core, the parallelism level directly translates to the maximum number of concurrently running jobs. For simplicity, we refer to this feature as *cores*. Additionally, in tests with synchronous calls, the *input workload/throughput* is also a pivotal feature for accurate performance prediction.

Aside from *cores*, we also use *1/cores* and *log(cores)* as features, as they are relevant quantities to predict the performance of parallel systems [35]. Moreover, we perform polynomial expansion of features up to the second degree. We also test the interpolation and extrapolation capabilities of the models (see Section 5). As mentioned, the goal of interpolation tests is to understand how dense

our profiling campaign must be to achieve good results and if we can still obtain accurate models with smaller datasets. At the same time, extrapolation tests aim to understand the behavior of the models on unseen, expensive-to-evaluate configurations. This means, e.g., analyzing whether it is possible to start from experiments run on a limited number of files and predict the performance for larger inputs. We perform interpolation tests in all cases and extrapolation tests when they are significant (as discussed in the following sections).

The aMLLibrary models are generated on a server with two Xeon E5-2620 v2 processors, with 6 cores each, and 32 GB RAM. More details on the composition of training sets in different tests are shown in the respective sections, along with the results tables. Note that, in the worst-case scenario, the library trains all ML learning models within approximately ten minutes.

7.3. Results with the Mask detection application

This section reports the profiling setup and the results obtained for *Mask detection*. We test the application both on physical nodes and cloud VMs, considering different workload injection models and evaluating the impact of neural network partitioning on the first component. In particular, Section 7.3.1 focuses on testing the performance of asynchronous calls, varying the components and resources configurations, the number of input videos, and the impact of different edge resources. Section 7.3.2 considers instead synchronous calls performed via JMeter, which allows to evaluate the prediction accuracy of our models when combining the impact of different resource configurations and throughputs.

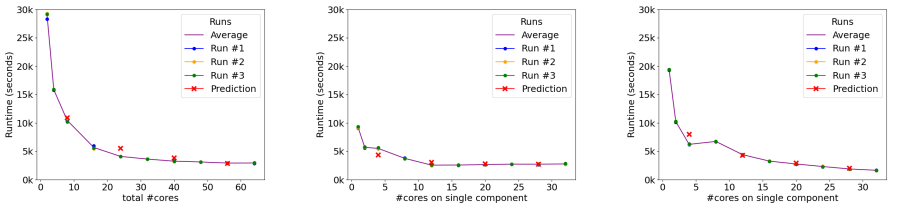
7.3.1. Tests with asynchronous calls

In this first scenario, we consider asynchronous calls to the *Mask detection* application, where the videos to process are uploaded to the input bucket of the first component in batches of variable size.

Scenario 1: impact of different parallelism levels. In the first scenario, we consider a single batch of ten videos of 15 minutes each. We analyze the impact of different configurations, running each component on up to 8 VMs with 4 cores and 8 GB of memory each, with a total of 32 cores, i.e., 32 parallel pods. A

runtime/core plot showcasing the impact of the number of cores on performance is reported in Figure 14. The runtime is the interval between the start of the first job and the completion of the last one, including wait times and overhead.

We use the collected data to generate prediction models for the runtime of the single components and of the entire application (see Figure 12). Moreover, we investigate whether combining the predictions obtained for *blurry-faces* and *mask-detector* (what we call the *combined model*) allows to predict the full workflow runtime with a reasonable level of accuracy. We then analyze the interpolation capabilities of our models, generating a test set with execution times collected on 4, 12, 20, and 28 cores while using the remaining data for training. Table 1 collects the MAPE on the test set for all trained regression models; the best models (corresponding to values in bold) are used to generate the predictions reported in Figure 14. The results show that our best model,



(a) Full workflow (b) *blurry-faces* component (c) *mask-detector* component

Figure 14: Scenario 1: interpolation results.

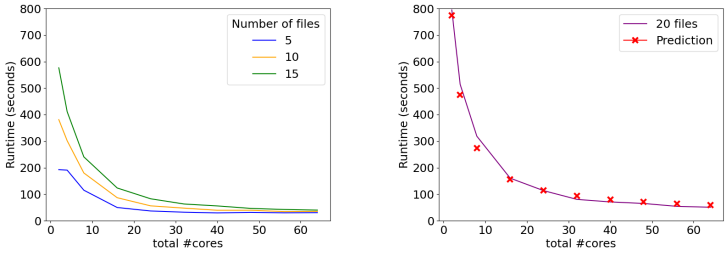
namely Ridge Regression without SFS, is able to predict the total runtime of the full workflow with an error of 14.73%.

		MAPE interpolation, *Scenario 1*			MAPE extrapolation, *Scenario 2*		
Algorithm	SFS	Full workflow	*blurry-faces*	*mask-detector*	Full workflow	*blurry-faces*	*mask-detector*
Ridge Regression	Yes	23.05	14.80	19.32	**9.82**	**9.60**	17.19
	No	**14.73**	11.48	**10.14**	70.60	43.01	73.63
Decision Tree	Yes	46.78	13.03	35.1	17.94	22.01	24.73
	No	30.28	9.29	38.12	23.40	27.8	34.04
XGBoost	Yes	46.99	13.12	39.48	17.23	17.6	**10.09**
	No	46.81	12.94	39.48	17.39	21.3	24.37
Random Forest	Yes	19.78	**3.34**	15.75	20.36	19.12	13.24
	No	28.41	16.15	27.45	89.19	60.54	32.53

Table 1: MAPE [%] on interpolation over *parallelism level* (*Scenario 1*) and on extrapolation over *batch size* (*Scenario 2*).

Scenario 2: impact of different batch sizes. We test the application on the same configurations considered in Scenario 1, evaluating the combined impact of var-

ious parallelism levels and input batch sizes on the components runtime. In particular, we consider 10-second-long videos uploaded in batches of 5, 10, 15, and 20. We aim to generate and test performance prediction models with good extrapolation capabilities, i.e., to check whether a model trained on data collected with batches of size 5, 10, and 15 can accurately predict the components runtime when 20 files are submitted. We report MAPEs for the trained models in Table 1. As in the previous scenario, we combine the two best models for the individual services to predict the whole application runtime, achieving a best MAPE of 9.82%. Figure 15 reports the execution times for the full application with 5, 10, and 15 files as input (Figure 15a), and the times for 20 files along with the corresponding extrapolation predictions (Figure 15b).



(a) Runtimes with 5, 10 and 15 files

(b) Extrapolation results

Figure 15: Scenario 2: extrapolation results for the full workflow runtime.

Scenario 3: evaluation on different edge resources. We conduct the third campaign on a cluster of 3 Raspberry Pi devices, processing batches of ten videos lasting 10 seconds each and exclusively testing the *blurry-faces* component.

Here, we aim to demonstrate the logging capability of OSCAR-P, as depicted in Figure 8. We assess the performance of the *blurry-faces* component in two different configurations: on the Raspberry Pi cluster alone, and with the addition of an Intel Movidius Neural Compute Stick 2 (NCS2)[13] USB accelerator. We conduct all tests with an 8-core configuration.

To precisely analyze how this device affects the component performance, we measure the duration of different computation phases: i) splitting the video into

[13] `https://intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html`

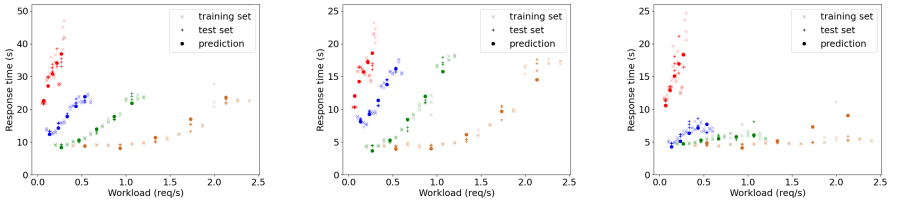
frames; ii) overhead for loading Python modules; iii) image reading from disk; iv) model loading, either in the device RAM or on the Intel NCS2; v) inference; vi) image blurring; vii) copying the results onto disk. It is worth noting that since the first task is executed on CPUs, we should not expect any impact on its performance. On the other hand, Figure A.17a in Appendix A.3 highlights that the model computation on the stick is over 7 times faster compared to the CPU, while the model loading is only one second slower.

Figure A.17b reports the *blurry-faces* component runtime with and without the NCS2 while varying the number of input files. The benefit of acceleration becomes more apparent as the number of files increases since they are processed sequentially. However, despite the significantly faster computation with the NCS2, the overall performance gains are mitigated by the model load phase which is slightly slower on the stick compared to the device RAM.

7.3.2. Tests with synchronous calls

In the fourth and fifth testing scenarios, we focus on synchronous calls to OSCAR services. We inject an input workload λ via JMeter, imposing a ramp-up of 10 seconds to avoid measuring runtimes during the warm-up period and testing different ranges depending on the number of cores. For example, for component configurations including 2 cores, we consider $\lambda \in [0.05, 0.3]$ requests per second (req/s). At 4 cores, since the system can sustain higher throughput values before reaching the saturation level, we test $\lambda \in [0.1, 0.6]$ req/s. Each workload value is kept constant for 10 minutes. Note that in each setup, half of the cores of the first component are warm to limit the impact of the cold start. The synchronous scenarios are more critical because, as the load increases, the individual configurations saturate, resulting in larger variability in execution time. This outcome is expected as it is consistent with results using other modeling techniques such as M/G/k queues [38]. Additionally, ML yields more effective results than the M/G/k approach as the latter requires expensive simulation time to estimate the components demand, whereas ML model predictions can be obtained in milliseconds.

Scenario 4: impact of the number of cores and throughput values. We deploy the *Mask detection* application on Amazon EC2 VMs, specifically using *m4.large* instances[14]. Profiling data are collected with the same number of cores (2, 4, 8, and 16) for both the *blurry-faces* and *mask-detector* components. Results for the ML performance models trained with this data are reported in Table 2. Figure 16 shows the interpolation predictions with the best models. In this test campaign, the test set comprises profiling data originating from specific workloads (which are five evenly-spaced values for each number of cores).



(a) *Full workflow* (b) *blurry-faces* component (c) *mask-detector* component

Figure 16: Scenario 4: Training set, test set, and predictions result for interpolation with 2 cores (red), 4 cores (blue), 8 cores (green), and 16 cores (brown) on each component.

The MAPE of the best combined model is 7.01%. Recall that in this case, we do not have to consider the pipeline effect, as synchronous calls represent a continuous flow of jobs, allowing our models to predict the average execution time of each individual job. For this reason, the execution time of the full workflow is the sum of the times of the individual components.

		MAPE interpolation, *Scenario 4*			MAPE interpolation, *Scenario 5*		
Algorithm	SFS	Full workflow	blur-faces	mask-detector	Full workflow	partition 1	partition 2
Ridge Regression	Yes	11.19	16.19	15.78	8.89	14.22	8.29
	No	**7.01**	8.57	13.05	6.67	10.43	**7.80**
Decision Tree	Yes	12.96	14.41	20.17	**5.97**	9.84	9.57
	No	17.15	17.99	19.41	11.21	9.94	21.34
XGBoost	Yes	28.60	44.55	**12.55**	9.57	12.02	12.66
	No	9.23	**6.09**	15.87	10.32	**9.64**	16.46
Random Forest	Yes	40.78	11.02	78.82	10.18	13.97	8.15
	No	31.39	14.70	60.96	7.90	12.64	7.95

Table 2: MAPE [%] on interpolation in *Scenario 4* and *Scenario 5*.

Scenario 5: impact of neural network partitioning. In the last scenario, we demonstrate the capability of OSCAR-P to manage individual application com-

[14] `https://aws.amazon.com/ec2/instance-types/`

ponents partitioned into multiple sub-components. Specifically, we focus on the *blurry-faces* component considering two partitions. The YOLOv4 network is partitioned at *layer 266*, corresponding to the layer that exports the smallest tensor. As in the previous scenario, the components are deployed on *m4.large* EC2 instances. Additionally, we explore the interpolation capability of ML models amidst variations in the number of cores and throughput. Figure A.18 shows the collected data and the corresponding predictions with the best models. Table 2 shows the MAPEs for the interpolation tests of all models, with a best value of 5.97%.

Note that the total execution time of the partitioned component in this particular setup exceeds that of the entire component mainly due to the transfer time between partitions. Despite this, DNN partitioning is still relevant in practice to optimize resource usage, particularly on constrained devices [39].

7.4. Results with Video searcher application

The video searcher application, which includes six compute-intensive components (see Section 7.1.2), is tested in the computing continuum. In particular, services are deployed on both local VM clusters, Raspberry Pi devices, and EC2 instances, as shown in Table A.5 in Appendix A.4. Notably, the full workflow is tested across a range of nodes for each component, from 1 to 8.

We perform asynchronous calls by uploading 1-minute-long videos in batches of 5, 10, and 15. The DeepSpeech component is also tested individually to assess whether increasing the number of cores from 1 to 4 improves its performance. We do observe an improvement in the test results, but it is not enough to justify running one component with 4 cores instead of 4 components with 1 core each. One reason is that only the actual computation benefits from the increased number of cores, while other operations, such as the container setup, do not. As in previous scenarios, we test both the full workflow and the single services.

First, we conduct an interpolation analysis including 16 and 24 cores in the test set. Subsequently, we use all the obtained values to generate the combined model to predict the runtime of the full workflow. We show results for the

best models obtained for data with 5, 10, and 15 input files in Figure A.19 in Appendix A.4: their MAPEs are 27.63%, 18.95%, and 7.88%, respectively.

Next, we perform extrapolation on the number of input files. In this case, the aMLLibrary training set is the collection of configurations with 5 or 10 files, while the test set includes the 15-video batch. We also apply the pipeline effect correction explained in Section 7.3.1. Figure A.20 in Appendix A.4 shows the training set, the test set, and the extrapolation prediction. Even though we are considering an application with many services distributed across three clusters and training our models with only two different input sizes, we obtain a MAPE of 22.33% for the full workflow. This outcome is especially remarkable given the constraints of our testing campaign.

7.5. Results with Recipe transcriber application

The recipe transcriber application is tested using both Amazon EC2 instances and AWS Lambda[15], which is one of the main platforms providing FaaS. This is the most complex scenario we consider, as the application consists of seven compute-intensive components (see Section 7.1.3). In the previous section, we analyze the first five components. Here, we focus on the final two components, deployed on AWS Lambda, and on estimating the execution time of the entire pipeline. We perform asynchronous calls using a batch of 100 videos, each with a duration of 10 seconds. In this scenario, we observe the ability of our performance models to predict the average execution time of the entire pipeline for each individual video input.

Note that for such a large batch of video inputs, it is crucial to consider that the arrival rate of each component depends on the output rate of previous components, which in turn hinges on exogenous load and component configurations.

Furthermore, unlike previous campaigns, we also assign different numbers of cores to each component. Table A.6 in Appendix A.5 displays all possible configurations of the individual components, listing the number of nodes, cores,

[15] `https://aws.amazon.com/lambda/?nc1=h_ls`

and the level of parallelism, i.e., the number of parallel instances per component.

Note that the EC2 instances are selected to ensure that two pods are instantiated on each node. Furthermore, for the *ffmpeg-2* component, the number of cores is larger than (specifically, a multiple of) the parallelism level due to the high computational load of the component. However, the parallelism level of Lambdas is undefined, since AWS Lambda guarantees automatic resource scalability and therefore jobs are handled with full parallelism by the platform with no limitations.

This variety of component configurations results in a total of $2 \times 3 \times 2 \times 4 \times 2 = 96$ possible configurations for the full workflow, a number that would increase exponentially if we tested more than two different configurations for all components. In general, testing all possible configurations would be unfeasible in terms of time and budget. Thus, we devise a strategy to limit the number of profiled setups to one third of the total by only considering the profiling configurations with the smallest and largest *parallelism levels* for each component. Therefore, the training set for our models consists of the $2^5 = 32$ combinations of configurations (see Table A.6 in Appendix A.5).

Moreover, for each component after the first one, we included in the training set only profiling data collected when considering the lowest parallelism level on all the previous components. This process guarantees an optimal balance between opposing scenarios: on one hand, we consider cases where we have performance dependency in prior components, and, on the other hand, cases in which files arrive in batches due to the large input parallelism. All the other combinations constitute the test set.

Since all intermediate configurations are excluded from the training set, linear regression is the most suitable model for estimating runtime. As we only use training data at the extremes of the intervals, models such as Random Forest and Decision Tree would not be able to discriminate between values within those intervals[16]. Thus, we train a Ridge Regression model for each component

[16]Random Forest and Decision Tree provide stepwise regression functions. Hence, interme-

without SFS, estimate their execution times on the test set, and evaluate the MAPE on the total time. The MAPE on the test set for the *ffmpeg-3* and *object detector* components are 5.01% and 2.48%, respectively, while the MAPE for the entire workflow is 26.78%. This result is noteworthy especially given the significant inherent variability present in the profiling data and the fact that our models do not consider the dependency of the component execution time on the parallelism of all preceding components.

7.6. Results with Fibonacci application

We test OSCAR-P on the Fibonacci application by relying on synchronous calls. In this scenario, we test an application with input files of few bytes to examine whether the trend in execution times differs from previous applications with larger inputs. Specifically, we deploy the unique component on a physical cluster consisting of 7 VMs, each provisioned with 4 GB of memory and 4 vCPUs. As outlined in Section 7.3.2, we test the application with varying numbers of cores and input workloads. In the 4-core setting, we fix the input workload at $\lambda \in [0.2, 0.6]$ req/s, with the upper bound representing the point at which we observe saturation. As the number of cores increases, we gradually raise this upper bound, up to 3.5 req/s in the 28-core case. We force this constant workload for 10 minutes via JMeter, with half of the cores being warm at the beginning of the test.

We report the results of the interpolation tests with ML models in Table 3. Also, Figure A.21 in Appendix A.6 displays the training and test sets along with the prediction obtained using the best model. These results indicate that the ML models achieve errors below 10%, showing the remarkable predicting capabilities of OSCAR-P even in the case of small inputs and large workloads.

7.7. Discussion

Using the OSCAR-P framework for application profiling brings notable efficiency in time and budget allocation for conducting performance profiling cam-

diate configurations would be approximated with the values of the minimum and maximum core configurations.

Algorithm	SFS	Full workflow
Ridge Regression	Yes	15.24
	No	13.15
Decision Tree	Yes	11.45
	No	11.87
XGBoost	Yes	10.98
	No	10.15
Random Forest	Yes	**9.64**
	No	9.91

Table 3: MAPE [%] on interpolation.

paigns. OSCAR-P automates and optimizes profiling, eliminating all periods of inactivity associated with manual profiling, during which machines remain idle and logs are downloaded manually. We provide an example of savings related to the ML performance models interpolation capabilities by considering the most complex application analyzed in this work, namely *Recipe transcriber* presented in Section 7.1.3. In that case, the collection of profiling data for all possible combinations of input configurations took about two weeks and had a total cost of about 200 USD. If we only consider executions associated with configurations in the training set of the ML models, the data collection time decreases slightly over five days. The performance of the remaining configurations (i.e., the ones in the test set) can be estimated with acceptable accuracy by the trained ML models, therefore in a real use case it would not be necessary to profile them directly, reducing the costs by about 55%. The reduction would have been even more significant if we had considered larger levels of parallelism for each component. Overall, we have shown that by profiling only 33% of all possible configurations, we can obtain a model with a prediction error of less than 30%.

8. Conclusions and future work

This work introduces OSCAR-P, an auto-profiling framework, and aMLLibrary, an autoML solution for performance prediction. These tools automate the testing of application workflows on various hardware configurations and generate ML-based performance models. OSCAR-P significantly reduces setup time and manual processing by automating the execution of profiling experiments and data collection. Additionally, aMLLibrary enables accurate performance model training without ML expertise. Experimental results show that our models

achieve a MAPE smaller than 30%, which makes them useful for design-time decisions and runtime resource management.

Future work will primarily address certain limitations of our study. First, we plan to test a broader range of hardware specifications for the same application. In particular, including the machine specifications (besides the number of cores) as input features would allow us to generalize our performance models across different setups. We aim to examine whether the performance of aMLLibrary is affected and if it can capture common trends across different kinds of machines.

Second, in this work, we attempted to minimize the network effect to focus on computational performance. However, real-world environments often face network-related issues such as congestions or jitters. A potential research direction would be to analyze how the network influences throughput. We expect that the network effect mainly translates into reduced input rates for the application components. Since our models for individual components also account for throughput, we will evaluate if they still remain effective in such conditions.

Finally, we intend to scale our tools to an industrial level. The most computationally intensive application we evaluated in this study, *Recipe Transcriber*, involved testing configurations using up to 82 total cores on AWS and Lambda functions. Our next step is to validate our models on applications including tens of components spanning across hundreds or thousands of nodes. We expect increased variance at this scale due to more severe resource contention and amplification of performance noise across component dependencies.

Acknowledgment

The European Commission funded this work under the H2020 project AI-SPRINT (grant n. 101016577). OSCAR was developed under project PDC2021-120844-I00 funded by MCIN/AEI/-10.13039/5011000-11033 and by the European Union NextGenerationEU/PRTR and under grant PID2020-113126RB-I00 funded by MCIN/AEI/10.13039/501100011033. aMLLibrary was developed under the H2020 HPC JU LIGATE (grant n. 956137).

References

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

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

[3] A. Das, S. Patterson, et al., Edgebench: Benchmarking edge computing platforms, in: UCC, IEEE, 2018, pp. 175–180.

[4] I. Baldini, P. Castro, et al., Serverless computing: Current trends and open problems, in: Research advances in cloud computing, Springer, 2017, pp. 1–20.

[5] L. F. Albuquerque Jr, F. S. Ferraz, et al., Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas, in: ICSEA, 2017, pp. 206–212.

[6] A. Perez, S. Risco, et al., On-Premises Serverless Computing for Event-Driven Data Processing Applications, in: CLOUD, IEEE, 2019, pp. 414–421.

[7] B. Guindani, M. Lattuada, et al., AMLLibrary: An AutoML Approach for Performance Prediction, in: 37th International Conference on Modelling and Simulation (ECMS), Vol. 37, ECMS, 2023, pp. 241–247.

[8] J. George, C. Gao, et al., A scalable and cloud-native hyperparameter tuning system, arXiv preprint arXiv:2006.02085 (2020).

[9] E. Barbierato, M. Gribaudo, et al., Modeling Apache Hive based applications in Big Data architectures, in: Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools, 2013, pp. 30–38.

[10] A. Gandini, M. Gribaudo, et al., Performance evaluation of NoSQL databases, in: Computer Performance Engineering: 11th European Workshop, EPEW 2014 Proceedings, Vol. 11, Springer, 2014, pp. 16–29.

[11] D. Didona, P. Romano, Using analytical models to bootstrap machine learning performance predictors, in: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2015, pp. 405–413.

[12] H. Sedghani, F. Filippini, et al., A random greedy based design time tool for AI applications component placement and resource selection in computing continua, in: Proceedings of EDGE 2021, IEEE, 2021, pp. 32–40.

[13] F. Filippini, H. Sedghani, et al., SPACE4AI-R: a Runtime Management Tool for AI Applications Component Placement and Resource Scaling in Computing Continua, in: IEEE/ACM UCC Proceedings, ACM, 2023, pp. 1–7.

[14] E. Li, L. Zeng, et al., Edge ai: On-demand accelerating deep neural network inference via edge computing, IEEE Transactions on Wireless Communications 19 (1) (2019) 447–457.

[15] Y. Kang, J. Hauswald, et al., Neurosurgeon: Collaborative intelligence between the cloud and mobile edge, ACM SIGARCH Computer Architecture News 45 (1) (2017) 615–629.

[16] E. Galimberti, B. Guindani, et al., OSCAR-P and aMLLibrary: Performance Profiling and Prediction of Computing Continua Applications, in: 1st Workshop on Artificial Intelligence for Performance Modeling, Prediction, and Control, 2023, pp. 139–146.

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

[18] A. Das, S. Imai, et al., Performance optimization for edge-cloud serverless platforms via dynamic task placement, in: CCGRID, IEEE, 2020, pp. 41–50.

[19] J. McChesney, N. Wang, et al., Defog: fog computing benchmarks, in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, 2019, pp. 47–58.

[20] A. Fuerst, A. Rehman, et al., Ilúvatar: A fast control plane for serverless computing, in: Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, 2023, pp. 267–280.

[21] D. Ustiugov, D. Park, et al., Enabling in-vitro serverless systems research, in: Proceedings of the 4th Workshop on Resource Disaggregation and Serverless, 2023, pp. 1–7.

[22] Faasrail: Employing real workloads to generate representative load for serverless research, in: Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, to appear.

[23] S. Disabato, M. Roveri, et al., Distributed deep convolutional neural networks for the internet-of-things, IEEE Transactions on Computers 70 (8) (2021) 1239–1252.

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

[25] D. F. Kirchoff, M. Xavier, et al., A preliminary study of machine learning workload prediction techniques for cloud applications, in: 2019 27th Euromicro international conference on parallel, Distributed and Network-Based Processing (PDP), IEEE, 2019, pp. 222–227.

[26] J. Grohmann, M. Straesser, et al., Suanming: Explainable prediction of performance degradations in microservice applications, in: ICPE, 2021, pp. 165–176.

[27] N. Mahmoudi, H. Khazaei, Temporal performance modelling of serverless computing platforms, in: WoSC, 2020, pp. 1–6.

[28] N. Mahmoudi, H. Khazaei, Performance modeling of serverless computing platforms, IEEE Transactions on Cloud Computing (2020).

[29] S. Risco, G. Moltó, et al., Serverless workflows for containerised applications in the cloud continuum, Journal of Grid Computing 19 (3) (2021) 1–18.

[30] A. Pérez, G. Moltó, et al., Serverless computing for container-based architectures, Future Generation Computer Systems 83 (2018) 50–59.

[31] R. Sala, E. Galimberti, OSCAR-P and aMLLib, experiments results and datasets (Apr. 2024). doi:https://doi.org/10.5281/zenodo.10927468.

[32] N. R. Draper, H. Smith, Applied regression analysis, Vol. 326, John Wiley & Sons, 1998.

[33] F. J. Ferri, P. Pudil, et al., Comparative study of techniques for large-scale feature selection, in: Machine Intelligence and Pattern Recognition, Vol. 16, Elsevier, 1994, pp. 403–413.

[34] E. D. Lazowska, J. Zahorjan, et al., Quantitative system performance: computer system analysis using queueing network models, Prentice-Hall, 1984.

[35] A. Maros, F. Murai, et al., Machine learning for performance prediction of spark cloud applications, in: CLOUD, IEEE, 2019, pp. 99–106.

[36] A. Hannun, C. Case, et al., Deep speech: Scaling up end-to-end speech recognition, arXiv preprint arXiv:1412.5567 (2014).

[37] G. R. Russo, V. Cardellini, et al., A framework for offloading and migration of serverless functions in the edge-cloud continuum, Pervasive and Mobile Computing (2024) 101915.

[38] D. G. Kendall, Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain, The Annals of Mathematical Statistics (1953) 338–354.

[39] U. Tadakamalla, D. A. Menasce, Autonomic resource management for fog computing, IEEE Transactions on Cloud Computing (2021).

Appendix A. Supplemental material

In this section, we provide some additional material that includes details about the configuration of aMLLibrary in Appendix A.1 and additional plots and analyses from our experimental campaign in Appendix A.3 through Appendix A.6.

Appendix A.1. aMLLibrary configuration file

We report the example configuration file for aMLLibrary mentioned in Section 5. Note that this file represents an equivalent Python script of hundreds of lines of code (see the equivalent script to the Appendix configuration file in the Zenodo repository [31]).

```
[General]
techniques = ['LRRidge', 'XGBoost',
              'DecisionTree']
hp_selection = KFold
validation = HoldOut
folds = 5
hold_out_ratio = 0.2
y = exec_time
hyperparameter_tuning = Hyperopt
hyperopt_max_evals = 10

[DataPreparation]
input_path = dataset.csv
normalization = True
product_max_degree = 2
inverse = [*]

[LRRidge]
alpha = ['loguniform(0.01,1)']

[XGBoost]
min_child_weight = [1]
gamma = ['loguniform(0.1,10)']
n_estimators = [500]
learning_rate = ['loguniform(0.01,1)']
max_depth = [10]
alpha = [0]
lambda = [1]

[DecisionTree]
criterion = ['squared_error']
max_depth = [3]
max_features = ['auto']
min_samples_split = ['loguniform(0.01,1)']
min_samples_leaf = ['loguniform(0.01,0.5)']
```

Appendix A.2. aMLLibrary hyperparameters

We list the values or prior distributions of ML models trained by aMLLibrary in our experimental campaigns in Table A.4. We recall that BO tuning samples a maximum of 10 parameter sets for each model.

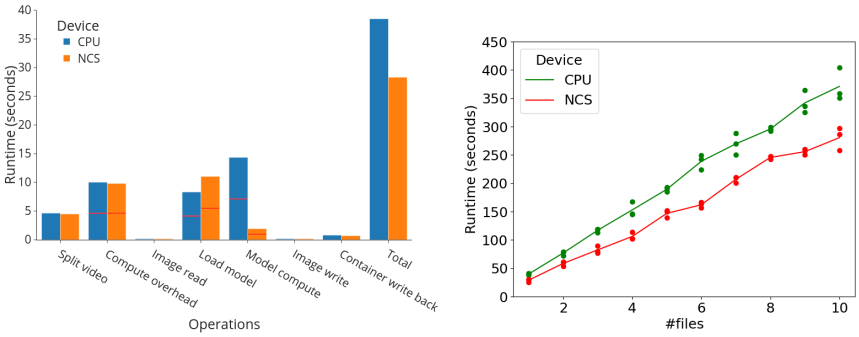
Appendix A.3. Mask detection experimental results

We depict the Figures related to the results with the Mask detection application, presented in Section 7.3.

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 A.4: Hyperparameters used for each ML algorithm.

Figure A.17 shows the results of executions with and without the NCS2 described in Section 7.3.1, *Scenario 3*. In particular, in Figure A.17a, some bars are split in half by red horizontal lines, representing each of the two produced frames for each input video.



(a) Runtimes of the computation phases

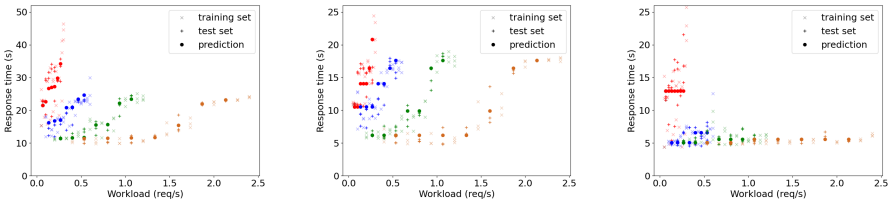
(b) Total runtime over number of input files

Figure A.17: Scenario 3: Comparison of the operations runtime with and without accelerator.

Figure A.18 shows the profiling data, divided into training and test sets, along with the predictions for the full workflow of the experimental campaign on the *blurry-faces* component partitioned as described in Section 7.3.2, Scenario 5.

Appendix A.4. Video searcher application experimental results

We show Figures and Tables related to the results with the Video searcher application, presented in Section 7.4.



(a) Full workflow (b) partition 1 (c) partition 2

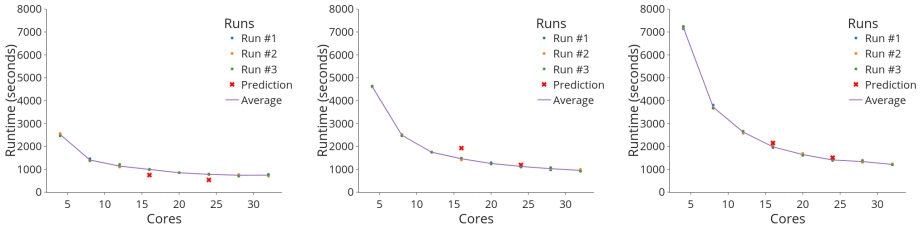
Figure A.18: Training set, test set, and predictions result for interpolation with 2 cores (red), 4 cores (blue), 8 cores (green) and 16 cores (brown) on each partition of the blur-faces component.

Table A.5 reports the application settings, including the type of computational resource, the number of nodes, and the corresponding number of cores for each component.

Service	Resource	#Nodes	#Cores
ffmpeg-0	local VMs	$1, 2, \ldots, 8$	$4, 8, \ldots, 32$
librosa	local VMs		
ffmpeg-1	Raspberry Pi		
ffmpeg-2	local VMs		
DeepSpeech	AWS t2.xlarge		
Grep	AWS t2.xlarge		

Table A.5: Setting of the simulations of video searcher application.

Figures A.19 and A.20 show the results of the interpolation and extrapolation tests on data collected with OSCAR-P.

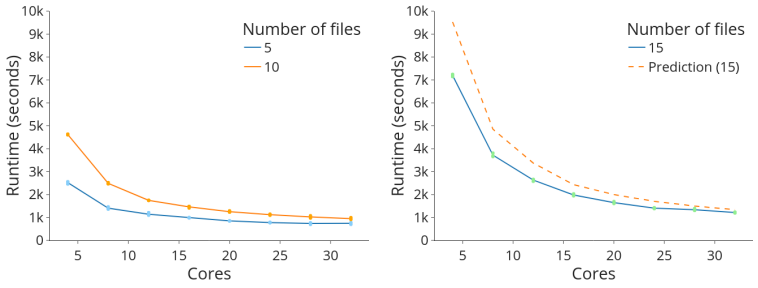


(a) 5 files (b) 10 files (c) 15 files

Figure A.19: Interpolation tests for the "Video searcher" application using combined model.

Appendix A.5. Recipe transcriber application experimental results

Table A.6 shows the settings for the experimental results with the Recipe transcriber application, described in Section 7.5, including the type of compu-



(a) Full workflow

(b) Test set and prediction on the combined model

Figure A.20: Extrapolation tests for "video-searcher" application.

tational resource, the number of nodes, and the corresponding number of cores for each component.

Component	AWS instance	Parallelism level	#Cores	#Nodes
ffmpeg-0	m4.large	**2**, **4**	2, 4	1, 2
librosa	m4.large	**2**, 4, **6**	2, 4, 6	1, 2, 3
ffmpeg-1	m4.large	**2**, **4**	2, 4	1, 2
ffmpeg-2	m4.4xlarge	**2**, 4, 6, **8**	16, 32, 48, 64	1, 2, 3, 4
DeepSpeech	m4.large	**2**, **4**	2, 4	1, 2
ffmpeg-3	Lambda	N/A	N/A	N/A
object detector	Lambda	N/A	N/A	N/A

Table A.6: Setting of recipe transcriber application. We highlight in bold the levels of parallelism used for the combinations of configurations in the training set.

Appendix A.6. Fibonacci application experimental results

Figure A.21 shows the profiling data, divided into training and test sets, along with the predictions for the Fibonacci application analysis, as described in Section 7.6.

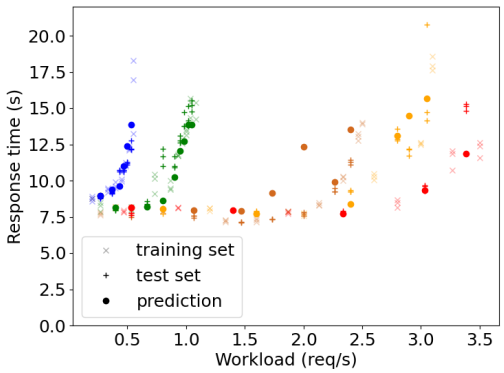


Figure A.21: Training set, test set, and predictions result for interpolation with 4 cores (blue), 8 cores (green), 16 cores (brown), 24 cores (orange) and 28 cores (red) on Fibonacci application.