Efficient Evolutionary Optimization using Predictive Auto-scaling in Containerized Environment

Milos Ivanovic[a,*], Visnja Simic[a]

[a] *University of Kragujevac, Faculty of Science, 12 Radoja Domanovica Street, 34000 Kragujevac, Serbia*

Abstract

Solving complex real-world optimization problems is a computationally demanding task. To solve it efficiently and effectively, one must possess expert knowledge in various fields (problem domain knowledge, optimization, parallel and distributed computing) and appropriate expensive software and hardware resources. In this regard, we present a cloud-native, container-based distributed optimization framework that enables efficient and cost-effective optimization over platforms such as Amazon ECS/EKS, Azure AKS, and on-premise Kubernetes. The solution consists of dozens of microservices scaled out using a specially developed PETAS Auto-scaler based on predictive analytics. Existing schedulers, whether Kubernetes or commercial, do not take into account the specifics of optimization based on evolutionary algorithms. Therefore, their performance is not optimal in terms of results' delivery time and cloud infrastructure costs. The proposed PETAS Auto-scaler elastically maintains an adequate number of worker pods following the exact pace dictated by the demands of the optimization process. We evaluate the proposed framework's performance using two real-world computationally demanding optimizations. The first use case belongs to the manufacturing domain and involves optimization of the transportation pallets for train parts. The second use case belongs to the field of automated machine learning and includes neural architecture search and hyperparameter optimization. The results indicate an IaaS cost savings of up to 49% can be achieved, with almost

*Corresponding address: University of Kragujevac, Faculty of Science, 12 Radoja Domanovica Street, 34000 Kragujevac, Serbia. Tel.: +381 34 336223; fax: +381 34 335040.

Email addresses: mivanovic@kg.ac.rs (Milos Ivanovic), visnja@kg.ac.rs (Visnja Simic)

unchanged result delivery time.

Keywords: Parallel metaheuristics based optimization framework, auto-scaling cloud resources, machine learning, resource usage prediction

1. Introduction

Complex real-world optimization problems can be successfully solved using evolutionary metaheuristics-based techniques such as a genetic algorithm (GA) [1]. However, the complexity of the problems often induces unreasonably long resolution times, especially when the computational cost of the fitness evaluation is extremely high.

Speeding up the execution of evolutionary algorithms (EAs) can be achieved by distributing population using master-slave (manager-worker), island, cellular, hierarchical, or pool model to parallelize an evolution task at the population, individual, or operation levels [2, 3]. Implementation of parallel metaheuristics has previously been mainly realized using HPC clusters, computing grids, GPUs, and volunteer peer-to-peer systems [4, 5, 6, 7, 8, 9]. Recent availability of Cloud on-demand massive computing resources offered new opportunities for the development of highly scalable and cost-effective optimization-as-a-service frameworks based on parallel evolutionary algorithms [10, 11]. On-demand access to cloud computing power makes it ideal for executing demanding optimizations without the need for ownership, operation and maintenance of computing infrastructure.

The ability to acquire and release resources on-demand to meet the end-users' needs is the most important property of Cloud Computing. However, deciding upon the appropriate amount of resources to meet the users' requirements is quite difficult, and as such needs to be automated in the form of an auto-scaling system. Many auto-scaling systems have been developed [12, 13] with different approaches to avoid both overprovisioning and underprovisioning of resources, and consequently, the cost increase or the violation of the Service Level Agreement (SLA). The main problem with automatic scaling is the timely response to sudden load changes. If not properly handled, a rapid increase in resource demands would lead to a short-term deterioration of performance, while a sudden decrease of demand would result in overprovisioning and in unnecessary expenses. The auto-scaling techniques handle resource demands variations using two main techniques: reactive and proactive [12]. Reactive techniques use scaling action as a reaction to a

change in the system, where the change itself is not foreseen. Proactive (predictive) techniques [14, 15] attempt to forecast upcoming changes in the system, estimate required provisioning, and perform necessary scaling actions before such changes occur.

To fully exploit the benefits of distributing EAs over Cloud resources and efficiently utilize the computing resources, researchers are turning to container-based technologies [16, 17, 18]. The containers are standalone self-contained units bundling the software and its dependencies together, replacing bulky virtual machine (VM) instances, and creating a lightweight environment for the applications [19]. The use of containers allows an application to be broken into smaller tasks, where each task is a microservice in a different container. Microservice-based applications lead to faster creation, operation, and removal of computing entities (containers) when compared to traditional VMs. The emergence of container technologies in the Cloud brought significant performance advantages compared to other virtualization technologies, with performance approaching bare-metal, but with much better flexibility. The density of computing entities can be significantly higher since operating system services are absent and more containers can be executed per machine. This is the main reason why the HPC community started to seriously consider Docker [20], Singularity [21], and Kubernetes-based [22] solutions. However, this imposes a challenge for auto-scaling, where more adaptable, precise, and capable algorithms are required to manage the elasticity of large-scale fleets of containers belonging to multiple applications with dynamic elasticity requirements. Keeping this in mind and considering the evolutionary-based optimization, a significantly different approach has to be taken regarding orchestration and scalability than in Simic et al. [11], or Dziurzanski et al. [17].

The existing auto-scalers, whether built-in Kubernetes or others such as Fission auto-scaler [23], and PASCAL auto-scaler [15] do not take into account the specific behavior of the evolutionary based optimization process, in particular the ones with the high computational cost of fitness evaluation. GA involves generational population improvement, where the entire population is evaluated in each generation, and the best individuals are selected to produce offspring. The need for a large number of distributed workers to perform evaluations is cyclic and follows the generational shift in the GA (Fig. 1).

This means that when faced with sudden resource demands at the beginning of a population evaluation, an auto-scaler must rapidly increase the

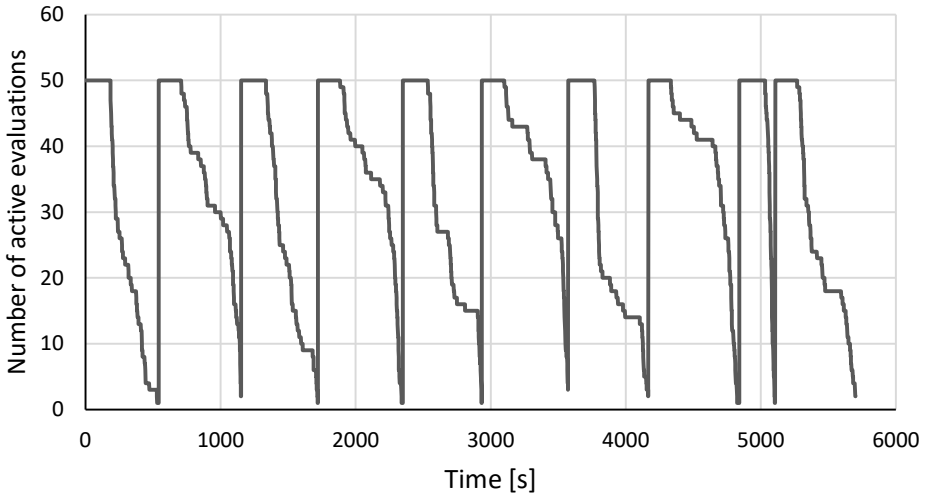


Fig. 1: The number of active evaluations of individuals over time

number of workers in the cloud to cope with additional load requirements and thus avoid under-provisioning. On the other hand, the duration of the fitness assessment in one population may vary from individual to individual. For example, real-world optimization problems often require a simulation model, such as a finite element model, to be executed as a black-box objective function evaluator. When a simulation has an adaptive time step, the evaluation of a certain number of individuals will take much longer than the evaluation of the rest of the individuals in the population. Therefore, most of the engaged workers would complete assigned evaluations and then idly wait for the remaining few to complete their work, leading to over-provisioning. To reduce unnecessary resource consumption and underlying infrastructure cost, idle workers should be stopped. Hence, the efficient auto-scaler would have to follow the GA's specific rhythm of demand for computing resources and avoid keeping idle workers, by elastically maintaining the adequate number of workers without affecting the quality of service.

With all that being said, the auto-scaler in compliance with the distributed evolutionary optimization in the container-based cloud environment should meet certain requirements. It must be able to establish a mapping between the characteristics of the individuals in each generation and the time needed for a generation's evaluation through some predictive model. This would allow auto-scaler to perform a scaling-out action at the most conve-

nient time to meet the incoming workload. On the other hand, there is a need to control the lifetime of each worker and adequately scale down to avoid unnecessary overprovisioning situations.

In this paper, we propose PETAS (Predicted Evaluation Time Auto-Scaling), a proactive, application-specific auto-scaler to provide efficient and cost-effective distributed evolutionary optimization in the containerized environment. PETAS operates by predicting patterns in the behavior of optimization processes to make the appropriate and timely scaling decisions and meet the above-mentioned requirements.

PETAS is developed to improve the already established distributed optimization framework WoBinGO [9, 11]. WoBinGO was developed for efficient solving of real-world optimization problems with high computational cost of the fitness evaluation. The framework is intended for parallel execution of GA to solve single-objective and multi-objective optimization problems. WoBinGO uses a manager-worker parallelization model and allows both: parallel evaluation of a population in GA and parallel execution of multiple instances of the parallel GA. It has been successfully employed for solving complex real-world optimization problems over HPC clusters, grids [9], and more recently, IaaS clouds [11]. The enhancements presented in this paper are intended to facilitate its use over the broadest spectrum of container-based cloud platforms such as Amazon ECS/EKS, Azure AKS, and on-premise Kubernetes. Additionally, the framework is modernized to a cloud-native microservice architecture. WoBinGO now consists of dozens of containerized microservices. Only three mandatory microservices are present throughout the whole optimization, while the rest are scaled by a specially developed application-level PETAS Auto-scaler, which uses machine learning techniques for workload pattern prediction.

Since our main concern is to enable end-users to efficiently, in terms of time and money, access solutions to their optimization problems, we evaluated our framework through solving two real-world optimization problems. The first is an intricate engineering optimization problem of transportation pallets design and production. The second benchmark problem concerns finding the optimal architecture of the artificial neural network to achieve the highest accuracy for a specific dataset. The experimental evaluation performed on these two problems demonstrates how PETAS proactive scaling, which uses a machine learning model to predict workload behavior, results in reducing operating costs without negatively impacting performance.

The rest of the paper is organized as follows: relevant related work is

reviewed in Section 2; the software framework's architecture is described in Section 3. The workflow of the optimization process using our solution is given in Section 4. Section 5 presents PETAS auto-scaling details. The experimental evaluation of the proposed software framework is described in Section 6 followed by the discussion of the obtained results in Section 7 and concluding remarks in the last section.

2. Related work

This section summarizes the related work in the area of parallel EAs distributed over cloud resources focusing on container-based approaches.

Distributing parallel EAs over cloud resources has been the focus of several research groups over the past decade. One of the first frameworks integrating parallel EAs with Cloud was proposed in [24] and it relied on [25] to distribute MOEAs on Enterprise Clouds. The other example of Cloud usage in evolutionary computation can be found in [26], which presents a distributed evolutionary computation system that relies on two different cloud storage services (Dropbox and SugarSync) for the file synchronization among islands, while each island is hosted by a different computer. Kurschl et al. [27] propose a reference architecture for cloud-based optimization services based on the Microsoft .NET framework and provide the prototype of a cloud-based optimization service (OaaS) intended only for the Azure cloud provider and using a Microsoft solution for automatic scaling of computational resources, which makes it highly platform dependent. Additionally, it requires users to establish a computing infrastructure on their own. In [28], a framework was introduced for distributing expensive fitness evaluations across an elastic, heterogeneous pool of computing nodes that include both personal computers of users/volunteers as well as a variable-sized pool of cloud computing nodes. Garcia-Valdez et al. [29] introduce the population storage model for the development of pool-based EAs that can be executed over cloud computing resources. The evolving population is stored in a centralized repository, while distributed clients asynchronously extract a subset of individuals and return a new subset of individuals after performing the search operators. The framework is configured to run on a cloud architecture using Heroku for the server and PiCloud for simulating workers. In [30], the authors present a proof of concept to map the island model of EA to the cloud-native format to be executed in a serverless manner, but with the fixed number of islands that could not be altered during the optimization

process. Lu et al. [31] propose to integrate the evolution operations of GA into Apache Hadoop or Spark to enhance the effectiveness and efficiency of GA parallelization. Authors devise the mechanism to fairly select the parents for crossover and mutation operations and avoid premature convergence in the parallel and distributed architecture. Their experiments have shown that the proposed architecture on Apache Hadoop as well as Spark performs more efficiently than the two reference architectures.

Numerous authors [32, 33, 34] have suggested using the MapReduce paradigm for parallelization of EAs over a cloud infrastructure, to relieve programmers of most of the distributed computation issues. An additional motivation for the use of this model stems from the fact that it is natively supported by several cloud infrastructures. All three different models of parallel GA are adapted to the MapReduce paradigm and issues and concerns about them are discussed.

As container-based virtualization became widely used in Cloud computing, many auto-scaling frameworks emerged to dynamically provision distributed services in responses to variations in the incoming load [15, 14]. PASCAL framework, for example is intended for a distributed stream processing (namely Apache Storm [35]) and as an auto-scalable distributed datastore (Apache Cassandra [36]). PETAS, in contrast, relies on a more general containerization frameworks such as Kubernetes and is specialized for evolutionary based optimization. While PASCAL carries out both scale-out and scale-in actions based on recent load, PETAS handles only scale-out action using predictive techniques. As opposed to PASCAL, PETAS scale-in policy is trivial, but most efficient for the case of evolutionary optimization - each pod goes down immediately when there is no work left.

Frameworks for distributed evolutionary algorithms are utilizing containers to gain more flexibility and efficiency. One of the first attempts in this area was carried out by Salza et al. [37]. In this paper, the authors propose the use of container technologies (Docker, CoreOS, and RabbitMQ), but they do not make an empirical assessment of the effectiveness of such systems. In [16], authors further propose a conceptual workflow describing the development, deployment, and execution of distributed GAs. Following that workflow, the master-slave parallel GA was implemented using containers to distribute fitness evaluation on a predefined number of slave nodes. In contrast, our solution elastically adapts the number of engaged nodes using a custom auto-scaler that interacts with the Kubernetes container orchestration system. The framework for master-slave parallelization of population-

based metaheuristics, presented in [38] uses containers to encapsulate each of the several microservices carrying out the tasks of the master and slaves. In contrast to our work, the number of engaged nodes is static and predetermined for the problem being solved. In [17] authors use Docker containers and Kubernetes to implement the Island Model of GA in the cloud, for solving process planning and scheduling problems in smart factories. Each container executes one or more islands, while the number of islands is determined using one of the proposed strategies. Each strategy dynamically adjusts the number of islands based on the quality and diversity of the currently obtained Pareto Front. The Fission auto-scaler steers the number of containers, which are executed in a serverless manner without requiring any VM to be instantiated in advance. The scalability of the proposed solution, unlike ours, is not aimed at saving time and money, but at improving the quality of obtained solutions by removing those islands which are not producing promising results and adding new ones. Valdez et al. [18] present reactive container-based application for the asynchronous execution of multi-population algorithms. Their solution allows the decoupling of the population and the population-based algorithm where different algorithms can be applied to each population. The model implemented by the framework is based on the asynchronous exchange of messages between stateless functions that react to a continuous stream of data; populations are the messages that flow in this stream. Though the suggested approach is very interesting from the research point of view, it can hardly be suitable for providing efficiency in terms of performance and monetary cost when it comes to real-world optimization problems with the high computational cost of fitness evaluation.

Although there is a significant number of software frameworks for distributed EAs that exploit the power of the cloud, not many of them use the containerized approach yet. Among those that do use containers, only [17] auto-scale them, but to obtain quality and diversity of Pareto Front. None of the papers we encountered so far reports elastic auto-scaling of containers to improve the cost-effectiveness of the optimization process.

3. Architecture

The adoption of Linux containers has recently led to the rise of microservice-based architectures in which monolithic applications are broken down into multiple services exposing interfaces. Containerization is a method for software packaging so that it can run in an isolated environment with its depen-

dencies. In contrast to Virtual Machines (VMs) which perform virtualization at a hardware level, containers do so at the operating system level. They run on a shared operating system but, being isolated in a self-contained environment prevents them from influencing the host machine or affecting processes running in other containers. Containers provide lightweight virtualization with fast and flexible deployment. Scheduling the execution of container-based workloads over the pool of shared computing resources is done by a container orchestration platform.

The architecture of the proposed system relies upon the usage of Docker [20], an open-source engine for containerization, and Kubernetes [22], container orchestration platform. Docker allows instantiating application containers, which are intended to contain all the components of an application. The engine creates containers out of images and provides an online registry to push/pull these images. An image is a lightweight, standalone, executable package of software that encompasses all necessary elements to run an application: code, runtime, system tools, system libraries, and settings. Images become containers when they run on Docker Engine.

Kubernetes is a framework designed to manage containerized workloads on clusters. A Kubernetes node may be a VM or a physical server. Each node contains services necessary to run pods. Pods are the smallest, most basic deployable objects in Kubernetes. A pod represents a single instance of a running process in a cluster and encapsulates one or more tightly coupled containers that are co-located and share the same set of resources. Multiple pods can be used to scale an application horizontally. However, as pods are not intended as durable entities, if a node fails, the pods will not survive. Kubernetes is designed to maintain the *desired state*, such as the desired number of pods.

The Kubernetes command-line tool, *kubectl* enables user to run commands against Kubernetes clusters: deploy applications, inspect and manage cluster resources, and analyze logs. Our custom PETAS Auto-scaler interacts with the Kubernetes API to launch pods. In our case, a pod corresponds to a single worker container that performs evaluation tasks. Fig. 2 shows the overall architecture of the proposed system comprising of four mandatory components which must be available through the entire optimization process: JARE, Binder manager, PETAS Auto-scaler, Binder worker gateway, and computationally demanding Binder worker microservice that is subject to scaling. The main functional modules of the system are:

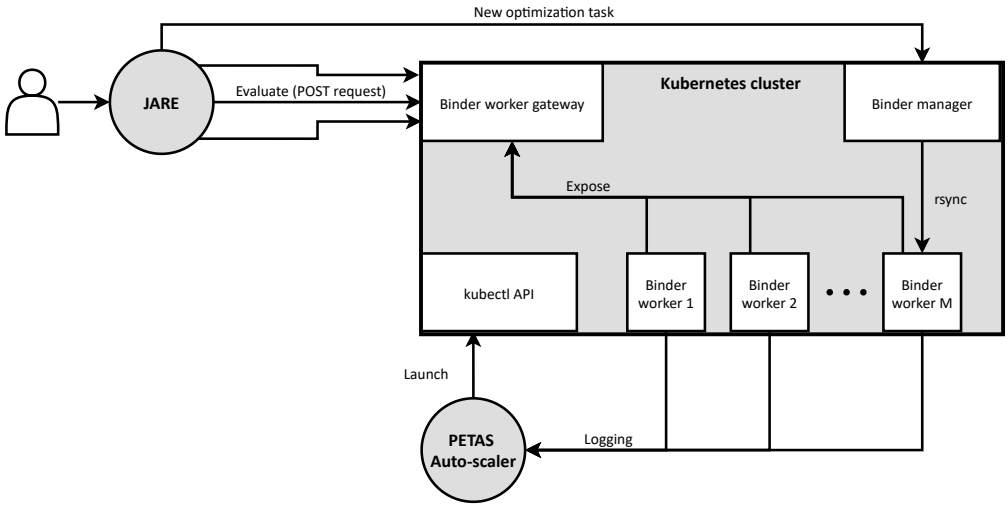


Fig. 2: Architecture of the proposed software framework

- **JARE** – the service that carries out the optimization process. It operates according to the manager-worker principle, i.e. when a generation is to be evaluated, JARE asynchronously sends POST requests to the gateway (one per individual) and waits for the results. When an evaluation of the whole generation is complete, the service applies the GA operators: selection, crossover, and mutation to obtain the next generation. The new generation is then being sent for the evaluation, and the whole process repeats until the GA stopping criterium is reached.

- **Binder manager** - accepts an evaluator from the JARE service and provides workers with a *rsync* [39] interface to download the evaluator. The evaluator contains everything needed to run a simulation (i.e. finite element), including executables, libraries, templates, and data files. It is often a few hundred megabytes in size.

- **Binder worker** – simple containerized microservice-based upon the Python-Flask framework [40]. The pod that is running a Binder worker exposes only a single API method that accepts an individual, runs the simulation internally, and returns results to the caller. It is easy to support any software requirement by adjusting the available Dockerfile of the Binder worker image. That way, the evaluator itself can be developed in any programming language and it may require any runtime.

The background process regularly checks for file compliance with the Binder manager using the *rsync* protocol. Binder worker may be in one of the following states:

  - *joining*: the underlying pod is launched and becoming ready to accept an evaluation task;
  - *ready* to accept an evaluation task;
  - *busy* performing a fitness evaluation;

  Binder workers operate in one of two modes: (1) constantly up and (2) automatic switch off. In the second mode, a potential infrastructure cost saving is enabled. The worker switches off if it has spent a given time T_{idle} in a *ready* state or has just completed an evaluation of an individual, and there are no more requests left in the queue. After the launch of the worker pods, it takes some time for them to become ready. This includes time to create containers from the Binder worker image, plus the time to synchronize files with the Binder manager. The overall duration depends on the size of the evaluator, the interconnection network throughput, and the total number of worker pods.

- **Binder worker gateway** - The unique endpoint that operates according to the reverse proxy principle. Nginx reverse proxy or Kubernetes service can be employed for this purpose. The round-robin and least-connections scheduling policies have been successfully applied. In this paper, we opted for the Kubernetes service and round-robin policy.

- **PETAS Auto-scaler** – The component that manages the number of Binder workers according to current and foreseen requirements. When a generation is to be completed, the number of available pods is usually low, due to adopted scale-in policy. Most of them have completed the assigned evaluations and turned off. For that reason, PETAS periodically checks if there is a need to launch a new portion of worker pods. Additionally, PETAS exposes an API method by which each Binder worker reports its current state. This way, PETAS is informed about the state of the distributed evaluation system at the time it decides to launch a new portion of worker pods.

The main contribution of this work lies in the way PETAS Auto-scaler determines the time point to launch a new portion of pods to make them available

at the moment the next generation emerges. Based on the predictive model trained by historical data, PETAS is capable of predicting the duration of an individual's evaluation and estimate the completion of the evaluation of the current generation. This is an advantage compared to a classic batch execution supported by Kubernetes[1] and other frameworks. If a classical batching system is employed, a new group of Binder worker pods has to be launched on each generation transition, and a certain amount of time is needed for them to become ready to carry out evaluations. With PETAS, on the contrary, pods are completely ready each time a new wave of evaluation requests emerges. While PETAS scale-in policy is identical to Kubernetes batching approach, PETAS scale-out policy is capable of providing clear performance benefit. Further details about the proposed auto-scaler are discussed in Section 5.

4. Optimization Execution Workflow

The first part of this section describes the system's internal workflow during the execution of the optimization task. The second part outlines the optimization workflow from the user's point of view.

1. JARE creates an optimization task by transferring the objective function evaluator and accompanying files to the Binder manager (Fig. 3). PETAS Auto-scaler, being informed that the optimization has just started, launches W_{max} Binder worker pods to listen on a unique Binder worker gateway.
2. Each Binder worker's background synchronization process notices a change within the Binder manager's *rsync* file system and starts synchronizing with it.
3. JARE begins the evolutionary process of GA. Each time a generation of N individuals has to be evaluated, the service sends N asynchronous requests to the Binder worker gateway.
4. The binder worker gateway further propagates the requests to worker pods which immediately start the evaluation of the individuals. When $W_{max} \geq N$, there is a big chance that all individuals are being evaluated simultaneously. Each worker reports to PETAS that the evaluation of an individual has begun.

[1] https://kubernetes.io/docs/concepts/workloads/controllers/job/

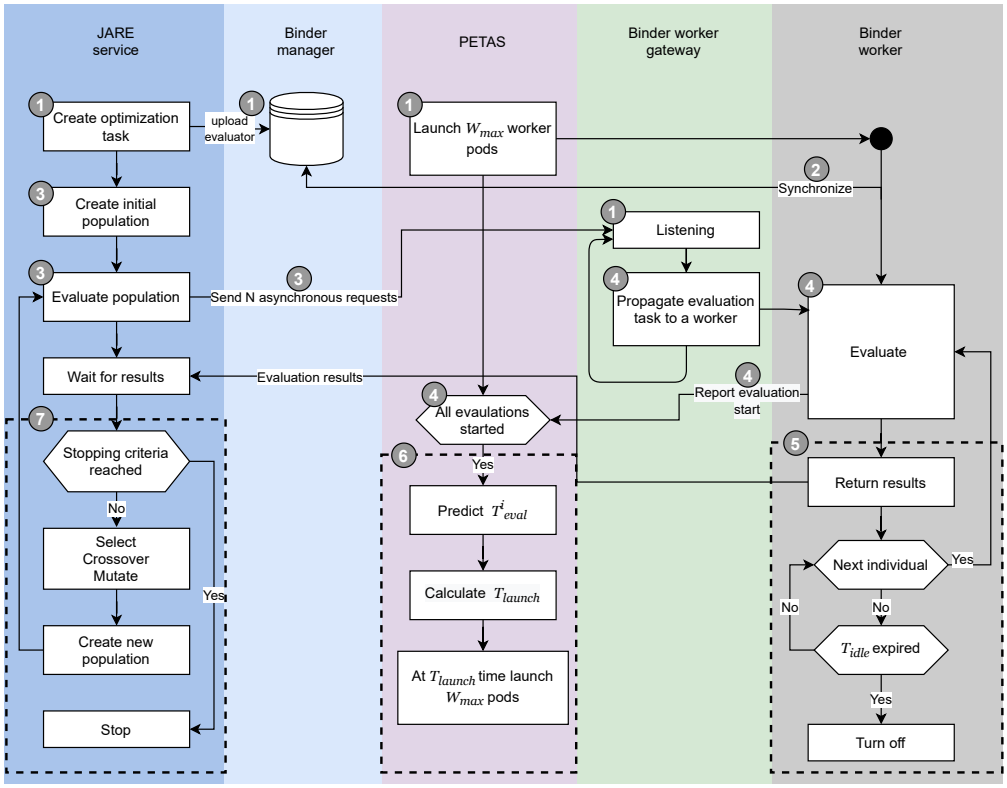


Fig. 3: The system's internal workflow during the execution of the optimization task

5. Upon completion of the evaluation, each worker returns the result and takes the next individual for evaluation, provided that there are more individuals in the queue. If there are not any, the worker pod automatically switches off. As the time passes, the number of workers decreases, which is a major contribution to infrastructure cost savings. A small number of workers does not even get a job before their idle time (T_{idle}) expires, and others complete all the evaluations in the current wave.
6. The moment PETAS is notified that an individual's evaluation has begun, due to a predictive algorithm described in Section 5.1, PETAS can predict when the individual's assessment will be completed, based on the values of the individual's decision variables. Upon receiving the notification that the last evaluation in the current generation has begun,

it is straightforward to predict the approximate time when the current generation's assessment will be completed and the next generation will begin. That way, PETAS "knows" when to launch the new portion of W_{max} pods to parallelize the evaluation of the next generation as much as possible.

7. JARE service continues with the rest of the evolutionary algorithm, and when the next generation is created, JARE sends N asynchronous requests to the Binder worker gateway. The whole process repeats as many times as necessary to meet the stopping criteria of GA (e.g. predefined number of generations, no improvements to the current best solution for a certain number of generations).

From the user's point of view, the system is a black box that allows her/him to perform optimization fully automatically, in the shortest possible time, and without any expertise regarding basic optimization methods or the underlying computing infrastructure. A user supplies the system with a definition of an optimization problem through the web portal. The definition includes a population size (N), the maximum number of generations in the GA, an objective function evaluator, as well as lower and upper bounds of the decision variables. The end-user does not, at any time, deal with the setting of the underlying cloud infrastructure. WoBinGO performs the optimization process as described in the first part of Section 4. An optimization can take several hours during which the user can monitor its progress. The obtained Pareto front is presented to the user through a web user interface and the user can further select one solution and analyze it.

5. PETAS Auto-scaler

PETAS Auto-scaler manages the number of Binder worker pods according to the current requirements. The optimization process initiates with W_{max} worker pods which are evaluating the individuals from the first generation. When there is no more work to do, those pods are switching off contributing to the IaaS cost savings. After an entire generation has been evaluated, there are almost no Binder workers left, and the number of pods running Binder workers can be increased through a scaling-out action. According to [14], the following four dimensions of a scaling action can be defined:

- *When* should scaling action be performed? The time can be determined using either reactive or predictive (proactive) techniques as explained

in the Introduction.

- *How* should the scaling be performed? Horizontal (scale-out/in) or vertical scaling (scale-up/down).
- *What* resource must be scaled to meet a given SLA?
- *How many* resources must be added or released to satisfy the SLA?

PETAS provides a proactive horizontal scale-out of CPU/memory resources whose number is determined by the W_{max}, a maximum number of worker pods. Our framework also includes scale-in action which is achieved by automatically shutting down idle pods after the predefined T_{idle} has expired or there are no evaluations left in the queue. Additionally, as pointed out in [14] a scaling action is defined by three points in time:

- The *demand point* (DP_{sa}) is the point at which a scaling action is required - the start of a next-generation assessment.
- The *triggering point* (TP_{sa}) is the point at which a scaling action is activated - when new W_{max} pods are launched.
- The *reconfiguration point* (RP_{sa}) is the point in time at which the scaling action has been completely terminated - all newly launched worker pods are ready to accept an evaluation task.

PETAS determines the TP_{sa} time when the launching of new W_{max} pods will occur. The choice of TP_{sa} must be made carefully: ideally, the scaling action should be completed at the same time a new generation's fitness evaluation begins. If the launch occurs too late, the scaling action will be completed after the beginning of a new generation's evaluation ($RP_{sa} > DP_{sa}$) and there will not be enough pods ($<< W_{max}$) to handle N individuals. On the other hand, if the pods are launched too early ($RP_{sa} << DP_{sa}$), the new pods may reach their T_{idle} and switch off, leaving an insufficient number of them to perform evaluations. The three crucial time points for scaling Binder workers that perform fitness evaluations are depicted in Fig. 4. The figure shows only a short period between the two generations. The blue line represents the number of required workers which almost drops to zero at the end of a generation, and then suddenly increases to the size of a generation, N. T_{safe} denotes the period between the moment when the new workers are

demanded and the moment when a scaling action covering that demand is concluded ($DP_{sa} - RP_{sa}$). It is the safe marginal time to ensure that pods will become fully ready to handle a new wave of evaluations. T_{safe} value does not affect the optimization process, but only the performance. It is the parameter that can be optimized according to the specific problem and specific underlying infrastructure.

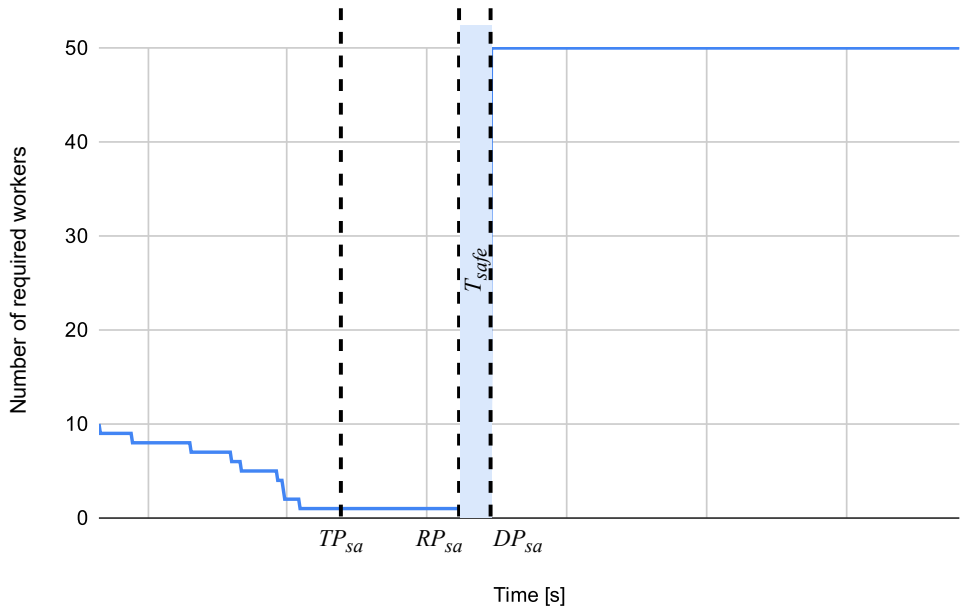


Fig. 4: The demand point (DP_{sa}) coincides with the beginning of a new generation. A scale-out action that is triggered in TP_{sa} ends in RP_{sa}. To avoid under-provisioning it is necessary to ensure T_{safe} marginal time between RP_{sa} and DP_{sa}.

PETAS operates in two consecutive phases: a *learning phase* and an *auto-scaling phase*. In the *learning phase*, PETAS utilizes ML to learn a prediction model of the optimization process behavior. If historical data for a specific optimization problem is unavailable, PETAS keeps W_{max} worker pods running in a static manner for a few generations, collecting the data. When sufficient quantity of historical data is available, PETAS builds an ML model of T_{eval}, which is then utilized at runtime, during the *auto-scaling phase*, to proactively scale the number of worker pods. The architecture of PETAS consists of two modules: (i) PETAS Evaluation Time Predictor, which is obtained through the learning phase, and (ii) PETAS Auto-scaler

for the second phase. Each of these modules operates as a black-box within the proposed framework, which allows seamless replacement of each of these modules.

5.1. PETAS Evaluation Time Predictor

This module is in charge of predicting the time that an individual's evaluation process will take (T_{eval}) depending on the values of its decision variables. PETAS Evaluation Time Predictor is a machine learning regression model for T_{eval} prediction. Whenever a new optimization problem is to be solved, a dedicated PETAS Evaluation Time Predictor is built through a learning phase. To generate the predictive model for a new optimization problem, the evaluation times of individuals and their decision variables' values have to be collected by executing GA for a few generations with a static number of workers (W_{max} pods running all the time). Collected records have the following form:

$$(x_1^i, x_2^i, ..., x_m^i, T_{eval}^i),$$

where T_{eval}^i is the evaluation time of individual i, $(i = 1, N)$ given its decision variables $x_1^i ... x_m^i$, and m denotes the number of decision variables. Using such a data set, the ML model is trained, validated, and tested. The resulting ML model is capable of predicting T_{eval} based on the values of an individual's decision variables. Once a model is chosen for a given optimization problem, it is used as such by the PETAS Auto-scaler for all subsequent executions of a genetic algorithm intended to solve that particular problem. Training of the predictor is explained in more detail in Section 6.1, where the specific implementations for the particular use cases are described.

5.2. PETAS Auto-scaler

PETAS Auto-scaler decides when to launch new W_{max} pods based on the information received from the PETAS Evaluation Time Predictor. As shown in Fig. 5, PETAS is notified whenever evaluation of an individual has begun by receiving the following: T_{start}^i - an evaluation beginning time for individual i, and its decision variables $x_1^i, ..., x_m^i$. For each individual i, PETAS Auto-scaler invokes the PETAS Evaluation Time Predictor and receives predicted value $\hat{T}_{eval}^i$ according to the values of decision variables $x_1^i ... x_m^i$. Upon the start of all evaluations, and after all $\hat{T}_{eval}^i$ predictions are obtained, PETAS Auto-scaler calculates the time to launch a new portion of W_{max} pods the following way:

$$TP_{sa} = \max_i \left[T^i_{start} + \hat{T}^i_{eval} \right] - T_{safe}, \tag{1}$$

where T_{safe} is a safe marginal time. Upon completion of the scaling action, a sufficient number of workers will be ready to receive the high workload generated by the beginning of a subsequent generation's evaluation process.

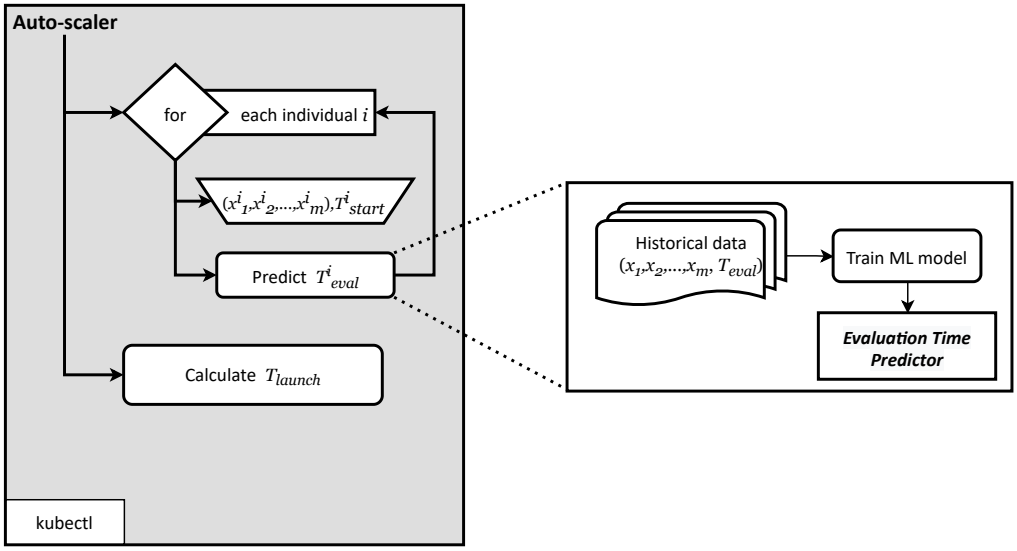


Fig. 5: PETAS simplified architecture

6. Experimental study

This section presents the evaluation of the proposed framework with the intelligent auto-scaling of containers. The first goal of the experimental study was to show that the pods' launching pace determined by the proposed PETAS auto-scaler is following the actual pace of change of generations. Secondly, we wanted to determine the performance of our solution in terms of the optimization time and cost. The proposed solution was tested using two real-world computationally demanding optimizations. The first use case belongs to the manufacturing domain and involves optimization of the transportation pallets for train parts. The second use case involves optimization as a part of the neural architecture search (NAS) and hyperparameter optimization of artificial neural networks.

Use case 1: Optimization of the transportation pallets

Transportation pallets are essential in various industries for the successful shipping of different products, including train parts. The pallet can be designed and its load capacity can be analyzed using commercially available software solutions, which already have modules for structural optimization. However, a pallet that is optimally designed and constructed regarding technical requirements and material consumption is not necessarily optimal from the point of view of the entire technological process, i.e. milling time, the number of welds, the costs associated with the welding process, the simplicity of the assembly process, the cost of transport to the end customer, etc. The optimization of the transportation pallets for train parts, not only from a structural point of view, but also taking into account the effectiveness of the production process in terms of all the above-mentioned parameters, is a complex multi-objective optimization problem. To solve this problem successfully and timely, one must possess expert knowledge in process modeling, optimization methods as well as distributed computing. Required human resources with an adequate level of expertise in any of these domains are scarcely available and expensive. Besides, the necessary hardware equipment and its constant maintenance are also expensive. In this regard, we evaluate the proposed cloud-based optimization service that allows manufacturers to optimize the design and production of pallets fully automatically in the shortest possible time without having to deal with purchasing, configuring, and maintaining specialized software and equipment. For solving the problem, we used the NSGA-II multi-objective genetic algorithm [41]. The objective functions aim to minimize pallet mass and its overall production cost. The core of the objective functions' evaluator was a finite element solver which checks the strength of any model constructed from the combination of 44 different decision variables. The following set of genetic algorithm parameters was adopted: population size of 50 individuals, a simulated binary crossover with a probability of 0.9 and a distribution index of 20, and polynomial mutation with a distribution index of 20 and probability of $1/l$, where l is the chromosome length. All results are taken from 10 independent runs. The average time needed to evaluate a single individual, T_{eval} was $127s$, with a standard deviation of $49s$, which means that the heterogeneity of the individuals was relatively high. This can be attributed to the fact that finite element analysis involves very diverse types of pallet constructions. Different types of constructions have a different total number of finite elements, and

consequently, the evaluation times vary significantly. The Pearson correlation between the decision variable *Type of construction* and the evaluation time of ≈ 0.8 promises a good base for modeling T_{eval}. The number of active evaluations throughout the optimization processes for this problem is shown in Fig. 1. Relatively high computational requirement and significant variance in evaluation time make this problem convenient for a proposed pod auto-scaling approach.

Use case 2: Optimization of the deep ANN architecture

The second use case belongs to the field of automated machine learning (AutoML) and involves joint neural architecture search and hyperparameter optimization. Automated discovery of artificial neural network models from the data is an optimization problem that can be solved by EAs. Evolutionary algorithms were successfully used for evolving the structure of feed-forward and recurrent neural networks [42, 43, 44]. Recently published papers evolve only neural architecture itself and use gradient-based methods for optimizing weights [45, 46, 47, 48, 49]. Despite their remarkable performance, architecture search algorithms and hyperparameter optimization are computationally highly demanding, requiring days of computing time, even when utilizing multiple GPUs. The experiments were conducted to evaluate whether the proposed container's elastic auto-scaling approach can enable the efficient solving of NAS and hyperparameter optimization problems.

To find optimal architecture and hyperparameters of deep feedforward and recurrent neural networks, specific evolutionary operators suitable for deep architectures must be used. The evolutionary algorithm evolves a population of models, i.e. deep neural networks (DNNs). Each individual in the population represents one ANN, such that genes encode the following: hidden layers count, number of neurons per layer, activation function, and training algorithm. In every evolution step, each network in a population is trained and its validation error is determined. The fitness is calculated such that individuals with lower validation error have higher fitness. Through a selection process, the best models (according to their fitnesses) are designated as parents to crossover and generate offspring. The selection is implemented using a standard tournament selection operator, while a one-point crossover operator performs the crossover of individuals. The mutation operator is applied randomly over a population to (1) change the number of neurons in a specific layer, (2) change an activation function of a randomly selected

layer, or (3) add a new or delete an existing layer. The new generations are created iteratively through selection, crossover, and mutation. The initial population consists mostly of individuals of modest complexity, and due to the DNN specific mutation, layers and neurons are added only if that will lead to an evident fitness gain. Therefore, the algorithm has an inherent preference towards simpler (and more efficient) architectures, which is a desirable feature in sense of production performance.

For this use case, we searched for an optimal deep neural network for the well-known benchmark dataset Boston housing [50]. The decision variables were the following: training algorithm, hidden layer count (1-10), neurons per layer (5-25), activation function, and dropout rate (0-1). The Boston housing dataset was divided into 70% training and 30% validation samples, and the number of epochs was fixed to 3000. The NAS and hyperparameters tuning was treated as a single-objective optimization problem to minimize the neural network's RMSE (root mean square error). The following set of GA parameters was adopted: population size of 50 individuals, a simulated binary crossover with a probability of 0.9 and a distribution index of 20, and polynomial mutation with a distribution index of 20 and probability of $1/l$, where l is the chromosome length.

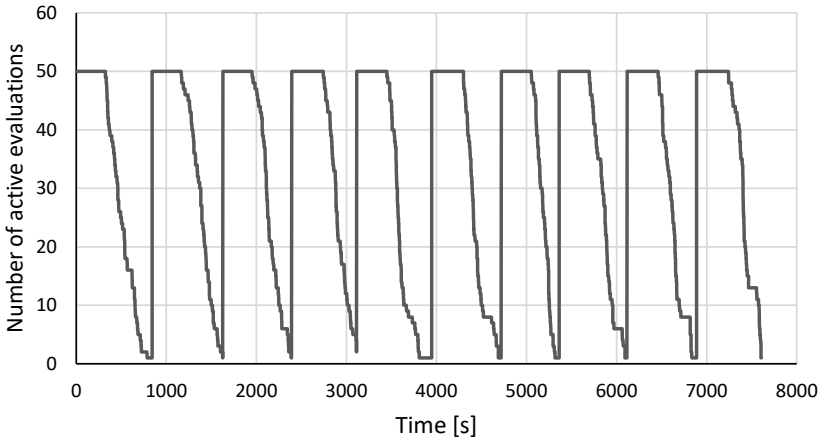


Fig. 6: Active evaluations for deep ANN architecture optimization problem

The average computational time to evaluate a single individual T_{eval} was $514.5s$, with a standard deviation of $111.7s$. The number of active evaluations throughout the optimization processes for this problem is shown in Fig. 6. Relatively high heterogeneity of computational time was expected, since the

training time of ANN largely depends on the number of layers, number of neurons per layer, etc. If we take a total number of neurons, it correlates with evaluation time by Pearson coefficient of 0.903, promising a good predictive potential. This optimization problem is, therefore, also suitable to be the benchmark for the proposed predictive auto-scaling approach.

To execute all experiments, we employed an on-premise Kubernetes 1.20 cluster consisting of 7 physical nodes. Each node is equipped with dual Intel Xeon E5-2683 v4 @ 2.1GHz CPU (32 physical cores), 128GB memory, and 10Gbps interconnection, totaling 224 cores and 896GB of RAM. The base OS platform is CentOS 7.7 x86_64.

6.1. Building PETAS Evaluation Time Predictors

In this section, we describe the implementation of the PETAS Evaluation Time Predictors for both use cases presented in the previous section. As previously discussed, PETAS Auto-scaler is in charge of launching worker pods timely to answer the requests for evaluation of a new generation. The decision is taken based on the prediction received from the PETAS Evaluation Time Predictor. This model is responsible for predicting an individual's evaluation time based on the corresponding decision variables' values. To generate the predictive model for each use case, the evaluation times of individuals and their decision variables' values have been collected by executing GA for a few generations with a static number of workers. This way, the two data sets, each containing 5000 examples, were obtained for the two use cases. As soon as PETAS Evaluation Time Predictor was trained employing obtained data sets, the PETAS Auto-scaler was capable of carrying out automatic scaling.

For both use cases, the datasets were used to generate various prediction models, after which the model with the lowest prediction error was chosen. We considered five well-established regressors: Polynomial regressor (POLY), Decision Tree regressor (DT), Random Forest regressor (RF), Support Vector regressor (SVR), and Multilayer perceptron (MLP). To evaluate the predictive capacity of these five models, k-fold cross-validation (k=10) [51] was performed. Fig. 7 and Table 1 show the results of 10-fold cross-validation comparing the predictive models for T_{eval} prediction. The presented results are the average scores obtained with the performance metrics for measuring the prediction error of the regression model (MAE) and the proportion of variance explained by the regression model (R^2):

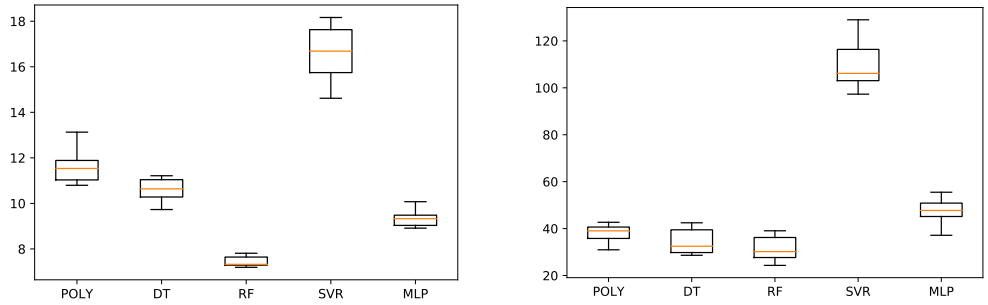
Mean Absolute Error (MAE): $MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$

R-squared (R^2) coefficient of determination: $R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$

where y_i denotes the actual output, $\hat{y}_i$ is the predicted value, $\overline{y}_i$ is the mean of actual outputs, and n is the size of the dataset.

Table 1: R^2 and MAE values obtained from the 10-fold cross-validation performed with models obtained using the following algorithms: Polynomial Regression (POLY), Decision Tree (DT), Random Forest (RF), Support Vector Regression (SVR), and Multilayer Perceptron (MLP).

	Use case 1		**Use case 2**	
ML algorithm	$MAE[s]$	R^2	$MAE[s]$	R^2
POLY	11.61	0.68	37.92	0.84
DT	10.59	0.83	34.47	0.85
RF	7.44	0.91	31.36	0.88
SVR	16.66	-0.03	109.58	-0.16
MLP	9.43	0.84	47.39	0.78



(a) (b)

Fig. 7: Spread of the mean absolute errors (in seconds) across each cross-validation fold for each of the five ML models for T_{eval} prediction: (a) Use case 1 - optimization of the transportation pallets; (b) Use case 2 - optimization of the deep ANN architecture. The box marks the 25th and 75th percentiles, with a line at the median. The whiskers extend from the box to show the range of the mean absolute errors.

From Table 1 and Fig. 7, it can be seen that for both use cases all the results go in favor of the Random Forest model. We further proceeded with hyperparameters' tuning of RF regressors trying to improve their prediction capacity. The adopted RF model for the optimization of the transportation palettes had the prediction error (MAE) value of $7.44s$ and its predictive capacity (R^2) was 0.91. In the case of the deep ANN architecture optimization, the adopted RF model had the prediction error (MAE) value of $31.36s$ and its predictive capacity (R^2) was 0.88. In both use cases, RF based PETAS

Evaluation Time Predictors were further utilized by PETAS Auto-scaler to determine the time to launch new pods based on the predicted evaluation times of the individuals that are currently being evaluated.

7. Results and discussion

In this section, we present and discuss the results of our experimental study. To show the efficiency of the proposed framework in terms of the achieved quality of the solution, the optimization results for both considered experiments are presented first. In the second part, we demonstrate that the proposed PETAS Auto-scaler leads to desirable provisioning of pods in real-world optimization tasks. The third part is focused on determining the performances of our solution in terms of the optimization time and cost.

7.1. Optimization results

The obtained optimization results for the considered manufacturing optimization problem are excellent. We considered two optimization scenarios, one regarding the optimization of production, and the other regarding the optimization of the overall solution. In the first scenario, we chose the solution that minimizes the welding costs while keeping the material costs at an acceptable level. In that case, welding costs were reduced by a remarkable 52%, while material costs were increased by 18%. In the second scenario, which includes all possible construction variants, the solution that showed improvement according to all criteria was chosen as the optimal solution. This solution provides a 12% reduction in pallet weight and reduces material costs by 14% and welding by an incredible 41%. It should be noted that this optimization with computationally hard, would take more than 20 hours if done sequentially, but with the proposed software framework it only takes an hour and a half, which means that the execution time is reduced by more than 90%.

The search for the artificial neural network that achieves the smallest validation error on the Boston housing benchmark resulted in the model with excellent performance without any human expert intervention. The architecture details, hyperparameters, and performance measures of the obtained model are given in Table 2.

7.2. Resource provisioning

The number of active pods throughout the optimization process is presented in Fig. 8. The results obtained throughout the palettes' optimization

Table 2: Architecture details, hyperparameters, and performance of the ANN model obtained using the proposed software framework

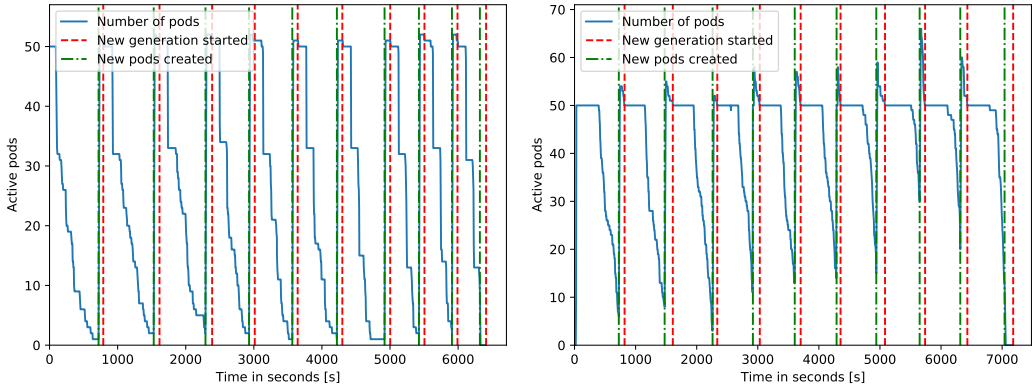
Architecture				**Hyperparameters**		**Model performance**	
Number of hidden layers	1	Activation function in hidden layer	ReLU	Training Algorithm	Nadam	RMSE [$1000]	2.78
Number of neurons per layer	8	Activation function in output layer	Sigmoid	Dropout rate	23%	R^2	0.92

are depicted in Fig. 8a, while the results obtained with the ANN architecture's optimization are shown in Fig. 8b. The number of generations for both experiments was set to 10 and the maximum idle time of a pod was set to $T_{idle} = 220s$. In both charts, the blue line represents the number of pods available to perform evaluations during the optimization process. The red vertical lines indicate the points when the evaluation of the new generation began. The green vertical lines indicate the moments when PETAS launched new pods. By comparing the positions of red and green lines for each generation on both charts, it is noticeable that the pod's launching occurs on time. Launching pods a little before the start of the new generation's evaluation (the green lines come before the red ones) gives pods sufficient time to become ready to accept their fitness evaluation tasks. As stated above, this intelligent scale-out feature represents a major advantage over classical Kubernetes batching approach, in which the whole optimization process has to wait for the pods to reach their readiness.

This proves that the proposed intelligent PETAS Auto-scaler is capable of determining the right moments to launch worker pods. If we compare use cases, it is noticeable that the intervals between pods' launching and a new generation commence (green and red vertical lines) are almost constant in use case 1, while variable for the use case 2. This is understandable, given the better performance of predictor for use case 1 in terms of prediction error.

7.3. Framework performance

Further, we considered the total optimization time (T_{opt}) and the cumulative uptime (T_{cum}). The total optimization time is the overall execution time of the optimization process. The cumulative uptime is the total time throughout which the cloud resources were used. It is directly and approximately linearly proportional to the monetary cost of an IaaS provider's service. The cumulative uptime, represented by the area under curve, can be calculated



(a) (b)

Fig. 8: The number of active pods throughout the optimization process: (a) Use case 1 - optimization of the transportation pallets; (b) Use case 2 - optimization of the deep ANN architecture

as follows:

$$T_{cum} = \int_0^{T_k} w(t)dt \tag{2}$$

where T_k is the time needed to execute k generations in GA, and $w(t)$ is the current number of active pods at time t. If we again observe Fig. 8, the area under the blue curve (T_{cum}) is relatively smaller for the first use case, expecting more cost-saving. This is due to a relatively higher variance of T_{eval} in the Palette optimization use case. Higher variance of T_{eval} gives more space PETAS scale-in policy to save pods' uptime.

Table 3: Comparison of PETAS, Batch, and Static solutions in terms of optimization time T_{opt} and cumulative pod uptime T_{cum}

	$T_{opt}[s]$			$T_{cum}[s]$		
Optimization problem	PETAS	BATCH	STATIC	PETAS	BATCH	STATIC
Palette optimization	6411	7167	6429	164621	250393	332852
ANN architecture opt.	7181	7830	6792	305091	318979	352160

To check the exact performance advantage of our approach, we compared PETAS with two generic solutions. The first was classic Kubernetes batch (**BATCH**), which launches W_{max} pods on each generation shift. One job corresponds to the evaluation of each individual in the generation. As stated above, it behaves identically as PETAS on scale-in, switching off pods that

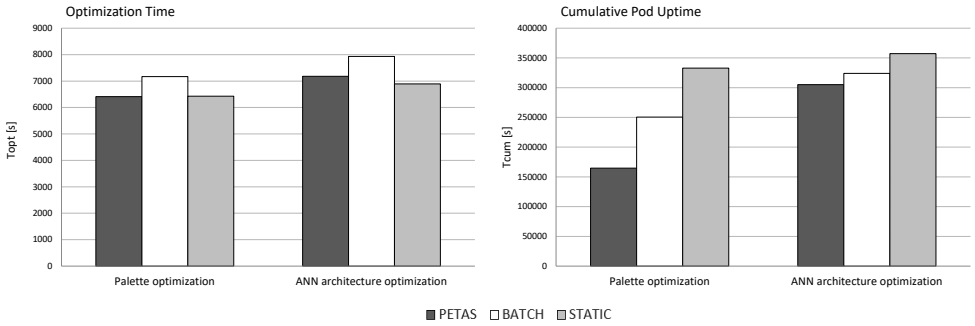


Fig. 9: Comparison of PETAS, BATCH, and STATIC approach in terms of *(a)* optimization time T_{opt} and *(b)* cumulative uptime T_{cum} for the two use cases

have completed their evaluations. However, on scale-out, significant time is required to set up all pods in the cluster. PETAS aims at avoiding that overhead. The second approach is **STATIC**, when scaling is off and we have W_{max} pods running and available all the time. STATIC is expected to have a very good T_{opt}, but since there is no scaling at all, a sub-optimal T_{cum}.

Table 3 and Fig. 9 reflect the main contribution of this work in terms of optimization time (left) and cumulative uptime (right) obtained from the two use cases. Considering T_{opt}, PETAS and STATIC deliver optimization result at almost the same time, while BATCH is lagging behind due to additional overhead of waiting pods to become ready on each GA generation shift. However, the major advantage of PETAS is obvious if we compare T_{cum} values in Fig. 9. In the case of the transportation palette optimization, the ratio between PETAS and STATIC suggests that there is 49% decrease of the cumulative uptime and 34% compared to BATCH. In the case of the ANN architecture optimization process, the savings are not that high (approx. 14%), but still significant, especially if optimization tasks are performed frequently and for a large number of generations.

Table 4: Mean absolute error and Maximum of absolute error of PETAS predictor for the two use cases

Optimization problem	$MAE[s]$	$\max(AE)[s]$
Palette optimization	5.5	18
ANN architecture optimization	22.5	43

The Table 4 gives the Mean (MAE) and Maximum of absolute error $max(AE)$ measured as the interval between the moment PETAS launched

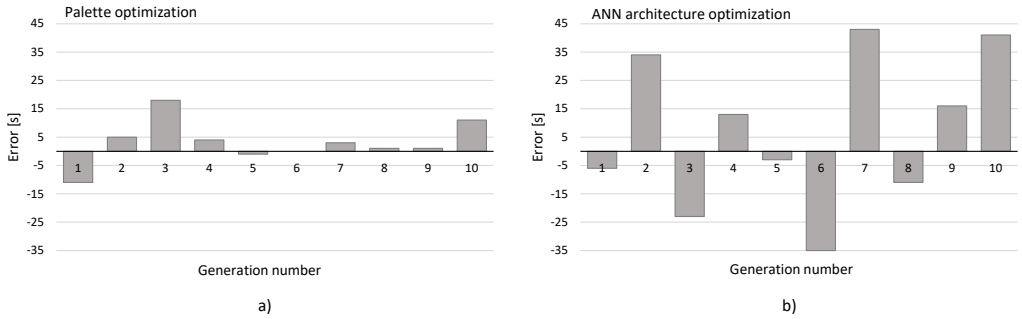


Fig. 10: The error in predicting the beginning of the evaluation of the new generation throughout 10 generations of GA: a) Use case 1 - optimization of the transportation pallets; b) Use case 2 - optimization of the deep ANN architecture

new pods, and the moment the evaluation of a new generation began. The diagram shown in Fig. 10 shows the prediction error throughout 10 generations. This error should ideally be positive and equal to the time needed for the pods to become ready to perform evaluations, or if we refer to equation (1), the error should be equal to T_{safe}. Therefore, knowing the values of MAE and $max(AE)$ allows the correct choice of the T_{safe} value, which defines how long before the predicted time to launch the pods. To ensure the proper operation, the value of T_{safe} has to keep the safe margin against $max(AE)$ to cover eventual prediction error. However, if T_{safe} value is too large and the T_{idle} value is too small, the pods will launch too early and switch off before the next generation begins, leaving the optimization process without enough workers. The values of MAE and $max(AE)$ express the level of certainty that sufficient number of pods will remain ready when the next generation starts. For both use cases, we chose $T_{safe} = 100s$ and $T_{idle} = 220s$, meaning that we covered the $100s$ window before predicted subsequent generation commencement and $120s$ after that point. Therefore, the optimization process stays on the safe side, since the pod window covers prediction errors of up to $100s$. In the case of the palette optimization (use case 1), even a tighter window would do the job, providing a bit more IaaS cost saving. However, it is up to the user to choose a safety factor according to the model accuracy, IaaS stability, and other factors.

8. Conclusion

In this paper, we have presented the software framework for solving optimization problems with a cloud-native, micro-serviced, and containerized architecture suitable for distributed, manager-worker execution of evolutionary metaheuristics. The main contribution of this paper is a novel application-level proactive auto-scaler of containerized evaluation workers, called PETAS. Aiming to improve the cost-effectiveness of the WoBinGO framework distributed over the cloud nodes, PETAS adapts to the specific requirements of evolutionary optimization. PETAS auto-scaler elastically maintains the adequate number of worker pods following the exact pace dictated by the demands of the optimization process. This is achieved by employing machine learning techniques to enable PETAS to predict patterns in the behavior of the optimization processes and, hence, make the appropriate and timely scaling decisions.

We have performed an empirical evaluation of the proposed software framework by utilizing it to solve two distinct real-world optimization problems. Through these computationally demanding optimization problems, it was possible to empirically demonstrate how PETAS, based on its predictive capacity, scales worker pods exactly in accordance with the pace of actual demands of the generational changes. For both optimization problems, the performance of our solution in terms of optimization time and infrastructure cost has been empirically determined. The results showed that due to predictive auto-scaling, the proposed solution offers significant benefits in cost savings, from a 14% to 49% decrease of the cumulative uptime. Thanks to PETAS and the adopted scale-out policy, these cost savings were achieved without compromising the optimization time, which is the most important quality for the end-user.

However, the system is still not sufficiently automated to reach a production readiness. We aim to extend this solution by integrating the PETAS learning phase into the software's framework workflow. According to the analysis presented in this research, this is fully achievable. For both real-world optimization problems, we showed that completely satisfactory PETAS models could be produced using a very simple ML technique since the data sets of historical runs are large and complete. Therefore, in the future, PETAS learning phase will be fully automated employing some of the popular AutoML tools. With the integration of the AutoML scheme into PETAS, together with the auto-tuning of a few remaining parameters such as T_{idle}

and T_{safe}, the proposed optimization system will reach the level of maturity and generality required from the production-ready PaaS.

Acknowledgements This work was supported by the Serbian Ministry of Education, Science and Technological Development (Agreement No. 451-03-9/2021-14/200122). The research was supported by the Scientific Fund of the Republic Serbia, PROMIS, 6062556, DyRes_System. Part of this research was supported by CloudiFacturing and DIGITbrain, European Innovation Projects, which receive funding from the European Union's Horizon2020 research and innovation program under grant agreements No 768892 and No 952071, respectively.

References

[1] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[2] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, J.-J. Li, Distributed evolutionary algorithms and their models: A survey of the state-of-the-art, Applied Soft Computing 34 (2015) 286–300.

[3] E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends, International Transactions in Operational Research 20 (1) (2013) 1–48.

[4] G. Luque, E. Alba, B. Dorronsoro, An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization, in: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, ACM, 2009, p. 1395–1402. doi:10.1145/1569901.1570088.

[5] N. Cole, T. Desell, D. L. González, F. F. de Vega, M. Magdon-Ismail, H. Newberg, B. Szymanski, C. Varela, Evolutionary algorithms on volunteer computing platforms: The milkyway@ home project, in: Parallel and distributed computational intelligence, Springer, 2010, pp. 63–90.

[6] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, B.-S. Lee, Efficient Hierarchical Parallel Genetic Algorithms using Grid computing, Future Generation Computer Systems 23 (4) (2007) 658 – 670.

[7] A. Munawar, M. Wahib, M. Munetomo, K. Akama, The design, usage, and performance of GridUFO: A Grid based Unified Framework for Optimization, Future Generation Computer Systems 26 (4) (2010) 633–644.

[8] T. Van Luong, N. Melab, E.-G. Talbi, Parallel hybrid evolutionary algorithms on GPU, in: IEEE Congress on Evolutionary Computation (CEC), Barcelone, Spain., 2010, pp. 1–8.

[9] M. Ivanovic, V. Simic, B. Stojanovic, A. Kaplarevic-Malisic, B. Marovic, Elastic grid resource provisioning with WoBinGO: A parallel framework for genetic algorithm based optimization, Future Generation Computer Systems 42 (2015) 44–54.

[10] S. Pimminger, S. Wagner, W. Kurschl, J. Heinzelreiter, Optimization as a Service: On the Use of Cloud Computing for Metaheuristic Optimization, in: International Conference on Computer Aided Systems Theory, Springer, 2013, pp. 348–355.

[11] V. Simic, B. Stojanovic, M. Ivanovic, Optimizing the performance of optimization in the cloud environment–An intelligent auto-scaling approach, Future Generation Computer Systems 101 (2019) 909–920.

[12] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, Journal of grid computing 12 (4) (2014) 559–592.

[13] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: state of the art and research challenges, IEEE Transactions on Services Computing 11 (2) (2017) 430–447.

[14] V. Rampérez, J. Soriano, D. Lizcano, J. A. Lara, FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services, Future Generation Computer Systems 118 (2021) 56–72.

[15] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, L. Querzoni, PASCAL: An architecture for proactive auto-scaling of distributed services, Future Generation Computer Systems 98 (2019) 342–361.

[16] P. Salza, F. Ferrucci, Speed up genetic algorithms in the cloud using software containers, Future generation computer systems 92 (2019) 276–289.

[17] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki, L. S. Indrusiak, Scalable distributed evolutionary algorithm orchestration using Docker containers, Journal of Computational Science 40 (2020) 101069.
URL https://doi.org/10.1016/j.jocs.2019.101069

[18] M. G. Valdez, J. J. M. Guervós, A container-based cloud-native architecture for the reproducible execution of multi-population optimization algorithms, Future Generation Computer Systems 116 (2021) 234–252.

[19] S. N. Srirama, M. Adhikari, S. Paul, Application deployment using containers with auto-scaling for microservices in cloud environment, Journal of Network and Computer Applications 160 (2020) 102629.
URL https://doi.org/10.1016/j.jnca.2020.102629

[20] Enterprise Application Container Platform, accessed: 2021-04-28.
URL https://www.docker.com/

[21] Singularity Documentation, accessed: 2021-04-28.
URL https://sylabs.io/docs/#singularity

[22] Kubernetes: Production-Grade Container Orchestration, accessed: 2021-04-28.
URL https://kubernetes.io/

[23] Fission Documentation, accessed: 2021-04-28.
URL https://docs.fission.io/docs/

[24] C. Vecchiola, M. Kirley, R. Buyya, Multi-Objective Problem Solving With Offspring on Enterprise Clouds, in: Proceedings of the 10th International Conference on HighPerformance Computing in Asia-Pacific Region (HPC Asia 2009), 2009, pp. 132—-139.

[25] C. Vecchiola, X. Chu, R. Buyya, Aneka: a software platform for .NET-based cloud computing, High Speed and Large Scale Scientific Computing 18 (2009) 267–295.

[26] K. Meri, M. G. Arenas, A. M. Mora, J. Merelo, P. A. Castillo, P. García-Sánchez, J. L. J. Laredo, Cloud-based evolutionary algorithms: An algorithmic study, Natural Computing 12 (2) (2013) 135–147.

[27] W. Kurschl, S. Pimminger, S. Wagner, J. Heinzelreiter, Concepts and requirements for a cloud-based optimization service, in: Computer Aided System Engineering (APCASE), 2014 Asia-Pacific Conference on, IEEE, 2014, pp. 9–18.

[28] G. Leclerc, J. E. Auerbach, G. Iacca, D. Floreano, The seamless peer and cloud evolution framework, in: Proceedings of the Genetic and Evolutionary Computation Conference 2016, ACM, 2016, pp. 821–828.

[29] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. F. De Vega, G. Olague, The evospace model for pool-based evolutionary algorithms, Journal of Grid Computing 13 (3) (2015) 329–349.

[30] J.-M. García-Valdez, J.-J. Merelo-Guervós, A modern, event-based architecture for distributed evolutionary algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO. ACM, New York, 2018, pp. 233–234.

[31] H.-C. Lu, F. Hwang, Y.-H. Huang, Parallel and distributed architecture of genetic algorithm on Apache Hadoop and Spark, Applied Soft Computing 95 (2020) 106497.
URL https://doi.org/10.1016/j.asoc.2020.106497

[32] P. Fazenda, J. McDermott, U.-M. O'Reilly, A library to run evolutionary algorithms in the cloud using MapReduce, in: European Conference on the Applications of Evolutionary Computation, Springer, 2012, pp. 416–425.

[33] P. Sachar, V. Khullar, Genetic Algorithm Using MapReduce - A Critical Review, i-manager's Journal on Cloud Computing 2 (4) (2016) 41–47.

[34] F. Ferrucci, P. Salza, F. Sarro, Using Hadoop MapReduce for parallel genetic algorithms: a comparison of the global, grid and island models, Evolutionary computation 26 (4) (2018) 535–567.

[35] A.S. Foundation, "Storm", accessed: 2021-12-06.
URL http://storm.apache.org

[36] A.S. Foundation, "Cassandra", accessed: 2021-12-06.
URL http://cassandra.apache.org/

[37] P. Salza, F. Ferrucci, F. Sarro, Develop, deploy and execute parallel genetic algorithms in the cloud, in: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, 2016, pp. 121–122.

[38] H. Khalloof, W. Jakob, J. Liu, E. Braun, S. Shahoud, C. Duepmeier, V. Hagenmeyer, A generic distributed microservices and container based framework for metaheuristic optimization, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2018, pp. 1363–1370.

[39] rsync - Linux man page, accessed: 2021-04-28.
URL https://linux.die.net/man/1/rsync

[40] Flask's documentation, accessed: 2021-04-28.
URL https://flask.palletsprojects.com/en/1.1.x/

[41] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE transactions on evolutionary computation 6 (2) (2002) 182–197.

[42] K. O. Stanley, D. B. D'Ambrosio, J. Gauci, A hypercube-based encoding for evolving large-scale neural networks, Artificial life 15 (2) (2009) 185–212.

[43] M. G. Epitropakis, V. P. Plagianakos, M. N. Vrahatis, Hardware-friendly higher-order neural network training using distributed evolutionary algorithms, Applied Soft Computing 10 (2) (2010) 398–408.

[44] R. Jozefowicz, W. Zaremba, I. Sutskever, An empirical exploration of recurrent network architectures, in: International conference on machine learning, PMLR, 2015, pp. 2342–2350.

[45] P. Vidnerova, R. Neruda, Evolving Keras Architectures for Sensor Data Analysis, in: 2017 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2017, pp. 109–112.

[46] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, K. Kavukcuoglu, Hierarchical representations for efficient architecture search, arXiv preprint arXiv:1711.00436 (2017).

[47] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, et al., Evolving deep neural networks, in: Artificial intelligence in the age of neural networks and brain computing, Elsevier, 2019, pp. 293–312.

[48] T. Elsken, J. H. Metzen, F. Hutter, Efficient multi-objective neural architecture search via lamarckian evolution, arXiv preprint arXiv:1804.09081 (2018).

[49] B. Stojanovic, M. Milivojevic, N. Milivojevic, D. Antonijevic, A self-tuning system for dam behavior modeling based on evolving artificial neural networks, Advances in Engineering Software 97 (2016) 85–95.

[50] D. Harrison Jr, D. L. Rubinfeld, Hedonic housing prices and the demand for clean air, Journal of environmental economics and management 5 (1) (1978) 81–102.

[51] T. Hastie, R. Tibshirani, J. Friedman, The elements of statistical learning: data mining, inference, and prediction, Springer Science & Business Media, 2009.