ISSN 2007-9737

A Container-Based Cloud Implementation of a Multi-Swarm PSO for Fuzzy Controller Optimization

Alejandra Mancilla[1], Mario García-Valdez[1,*], Oscar Castillo[1], Juan J. Merelo Guervós[2]

[1] Tecnológico Nacional de México, Tijuana, Mexico

[2] University of Granada, Department of Computer Engineering, Automatics and Robotics, Spain

{alejandra.mancilla,mario}@tectijuana.edu.mx, ocastillo@tectijuana.mx, jmerelo@ugr.es

Abstract. The adoption of cloud-native technologies for distributed computing presents a set of unique challenges. It is essential to carefully consider the technologies offered by the cloud provider to exploit the benefits of the architecture better while preserving the intended functionality. This paper introduces a container-based implementation of a cloud-native multi-swarm PSO on Amazon's Elastic Container Service. This paper is dedicated to a cost analysis, comparing multiple configurations of both local and cloud deployments. Furthermore, the paper proves that the proposed solution offers a robust alternative for both local and cloud environments. The results underscore the benefits of containerization for cloud-based bioinspired computing.

Keywords. Cloud Computing, multi-swarm optimization, fuzzy control.

1 Introduction

Cloud computing is becoming a standard way of running computer science experiments. This shift is primarily attributed to its cost-effective, pay-as-you-go model, eliminating the hefty upfront investment and ongoing management expenses associated with local computing infrastructures. Furthermore, cloud computing simplifies defining the infrastructure within the code itself, greatly enhancing the reproducibility of scientific computing results.

A basic example of the benefits of having easily reproducible code is the rise of web-based interactive computing platforms such as the Jupyter Notebook project [10] or Google's Colaboratory [3] platforms. When executing the interactive code, in these platforms, we can easily choose to scale the execution environment, using computing resources that greatly exceed what we have in our personal computers. This capability is a key advantage, enabling more complex and resource-intensive operations beyond our local hardware's limitations.

A cloud-native implementation of a computational experiment can also scale from a local development environment to a wide range of execution options via cloud computing services like Google Cloud Platform or Microsoft Azure. Cloud services evolved from primarily hosting monolithic applications on virtual machines to advanced distributed architectures adhering to the principles of microservices [28] and serverless computing [31].

Microservices architecture [14] emphasizes on decomposing applications into smaller, loosely coupled components, fostering flexibility and scalability. On the other hand, serverless computing further refines this paradigm by abstracting away infrastructure concerns, enabling developers to focus on coding while minimizing the management of computational resources. Combining these architectural patterns enhances the efficiency and reproducibility of computational experiments in cloud-native environments.

Cloud-native applications bring new methodologies and techniques [1] to the development of enterprise applications. To better exploit the capabilities of the cloud infrastructure, applications are implemented as reactive systems [2] that are generally more scalable, flexible, and fault-tolerant; these can be implemented as a loosely coupled collection of microservices. These application components are easily implemented and deployed to cloud environments, where processing nodes can be defined and deployed using scripts, and scalable message queue services are provided to send and receive events between them. Furthermore, best practices propose the use of continuous deployment procedures using software tools for automatic software provisioning, configuration management, and application deployment. Additionally, scalability in such systems is achieved through the automatic provisioning of additional computing nodes as demand increases or in response to node failures, ensuring consistent performance and reliability.

Microservices are typically executed within isolated runtime environments called containers [12]. These containers do more than provide a runtime environment; they encapsulate all the components required for the microservice to function autonomously. This includes software libraries, binaries, and configuration files. Essentially, containers package everything that a node needs to operate. The strength of a containerized application lies in its ease of operation on automation platforms. These platforms can take containerized microservice-based applications from a local computer to be deployed in a cloud or an ephemeral infrastructure [9, 11].

In our previous research [21], we highlighted that a reactive architecture is well-suited for implementing population-based metaheuristics if they require extensive computational resources. Building on this foundation, the current work details the deployment process of a reactive, container-based, multi-swarm PSO algorithm specifically designed for deployment in Amazon's AWS. This design decomposes the previous application into docker containers using an event-driven architecture.

The optimization problem tackled in this work is presented in [17] and consists of tuning the membership functions of a fuzzy controller. This optimization problem is computationally intensive, primarily because it requires evaluating the fitness of all potential solutions. This evaluation process involves running multiple simulations for each candidate solution, as explored in [19].

Despite the computational demands of this optimization problem, the independent nature of solution evaluation in evolutionary algorithms allows for the efficient parallelization of work. In the literature, we found only a few works [5, 23, 4] attempting to distribute fuzzy controller optimization; however, these works have not fully leveraged the advancements in cloud-native technologies. This gap presents an opportunity to explore how modern cloud-native solutions can enhance population-based optimization. This examination focuses on the intricacies of deploying such an algorithm within a cloud environment, leveraging the scalability and flexibility of AWS services.

The use of containerization is a key feature of this deployment because it offers the benefit of environment consistency and scalability. This approach aligns with the principles of reactive systems, ensuring that the deployed algorithm is efficient in resource usage and robust and adaptable to varying computational loads.

Furthermore, our application can be easily replicated and even run with other cloud providers. The code, including the container definitions, is available on GitHub (https://github.com/mariosky/fuzzy-control) with an open-source license. Our approach addresses a fuzzy control problem by applying a cloud-native distributed Particle Swarm Optimization (PSO) algorithm characterized by its minimal set of tunable parameters. We emphasize the significance of our design choices and elucidate the deployment process leveraging various cloud technologies provided by Amazon's AWS.

This paper is dedicated to a cost analysis, comparing multiple local and cloud deployments configurations. By scrutinizing the associated costs, we aim to provide insights into the economic considerations associated with implementing our proposed solution in different computing environments.

We structured the paper as follows: Section 2 explores contemporary research related to our work. In Section 3, we introduce the proposed approach, followed by a discussion on container-based cloud deployment in Section 4. We then outline the use case and experimental setup for evaluating deployment options, particularly focusing on time-to-solution performance, with an analysis of the results in Section 5. Finally, we provide conclusions and suggestions for future work in Section 6.

2 Related Work

The proliferation of container-based architectures [11] has given rise to a set of patterns that collectively define the landscape of cloud-native application development.

Integrating cloud-based evolutionary algorithms into cloud computing has been a gradual evolution within the field. Two notable examples of the integration of evolutionary algorithms with cloud computing include the Offspring framework developed by Vecchiola et al. [32] and the FlexGP system by Sherry et al. [27]. The Offspring framework implements a multi-objective Evolutionary Algorithm (EA) designed to operate on Aneka Enterprise Clouds. The system is built on top of a task model with a plugin architecture, enhancing flexibility and extensibility. On the other hand, the FlexGP system is a pioneering large-scale Genetic Programming (GP) system designed for cloud deployment. It adopts an Island model approach with a client-server architecture implemented on Amazon EC2. Furthermore, Valenzuela and García-Valdez [30] implemented a pool-based evolutionary algorithm using EC2 instances as workers, using a distributed pool for asynchronous collaboration between workers. Another feature of cloud computing is to provide infrastructure as a service (IaaS).

Several works starting to use these platforms, EvoSpace [8] used the PiCloud platform. PiCloud was a cloud for developers to run Python-based applications and tasks. It was designed to simplify the deployment and scaling of Python code in the cloud. Salza and Ferrucci took similar steps [25, 26] when they proposed an architecture to extend the reach of evolutionary algorithms into the cloud. This pioneering work laid the foundation for subsequent developments, as demonstrated in their subsequent paper [6].

The mentioned papers mark a transition toward incorporating cloud-native elements into the domain of evolutionary algorithms. Notably, they introduced aspects like the integration of messaging queues and adopting CoreOS as an operating system specifically tailored for efficient container utilization in evolutionary algorithms. However, it's worth noting that despite the infusion of cloud-native features, the management of containers in these instances still adheres to a more traditional approach from the perspective of distributed Evolutionary Computing (EC). Specifically, a master-worker architecture is employed, with communication facilitated through RabbitMQ, a messaging queue. In this setup, replicated workers are responsible for executing tasks in parallel, aligning with the distributed nature of EC methodologies.

While the papers introduce cloud-native elements, the underlying container management strategy retains a classical distributed EC framework, emphasizing the orchestration of tasks through a master node and distributed workers communicating via a messaging queue. This hybrid approach leverages both cloud-native technologies and established EC paradigms to enhance the scalability and efficiency of evolutionary algorithms in distributed environments. Dziurzanski et al. [7] implements an island model using a container-based architecture.

Another popular approach is offered by Pool-based systems [20, 22] in which a pool of candidate solutions is shared among all computing resources. In a pool-based system, worker nodes pull population samples from the pool and run several iterations of population-based algorithms instead of just evaluating candidate solutions. These systems behave more like serverless systems [14], which are closer to the nature of cloud-native systems.

Similar to the multi-population concepts in EC, other bioinspired algorithms employ the same ideas, currently, there are many proposals in the PSO research area exploiting Multi-Swarm configurations [13].

3 Cloud-based Deployment

Fig. 1 shows the container-based architecture proposed by García-Valdez and Merelo [29] the multi-swarm design. We adapted this initial design, adding components and strategies presented in previous papers. Adapting the initial parameters of the algorithm to use a heterogeneous strategy [16] together with an adaptation of these parameters considering the algorithm's diversity and the number of iterations. In these previous works, we implemented the event-driven design using containers running locally in a single workstation.

We first explain the overall design before explaining the cloud deployment in detail. Fig. 1 shows the system's main components as swim lanes. Following an event-driven design, we have two processing components (`combinator` and `worker`) that communicate via message queues. The data passed and interchanged between components are swarms (or populations) containing the current state of the proposed solutions (in this case, particles). Decoupling the population state from the search algorithm presents several advantages. We can scale the system by adding new populations and even have multiple algorithms processing the populations. The current implementation is considered a Multi-Swarm PSO because the PSO algorithm was used to process all the populations.

The `combinator` process is responsible for the first step, that is, creating a certain number of swarms, with each particle randomly positioned in the search space. Together with the collection of particles, the initial parameters of the PSO algorithm are also randomly generated within a range of values. This is called a heterogeneous strategy; this strategy gives acceptable results without the burden of searching the parameters experimentally [15]. As a second step (2), the data for each swarm is packed in a message in JSON format and pushed to the `input-queue`. The queue is constantly consumed by `Worker Containers`. We can have many workers; each worker is a daemon process that takes one message at a time from the `input-queue` (3), reads the PSO parameters contained in the message, and then runs the specified number of iterations on the received swarm. In the local PSO algorithm, there is not an initialization step. The algorithm takes the current state of the swarm and starts the iteration. After several iterations, the resulting state of the swarm is pushed to the `output-queue` (4). The `combinator` process pulls the resulting swarm messages and checks if the best solution has been found (5); if this is true, the algorithm ends. If the solution is not found, the best solutions are kept in a list structure storing only the k-best individuals. The `combinator` also has a buffer to store a certain number of swarms temporarily.

When the buffer reaches a certain number of swarms (in this case, two), the populations currently in the buffer are combined by swapping particles between the two in a process similar to a one-point crossover (6). This process is similar to the migration step of other Multi-swarm PSO algorithms. Before sending the modified swarms and depending on the current state of the algorithm, a fuzzy system adapts the *C1* and *C2* parameters of the swarms and pushes the newly generated swarms again to the `input-queue` starting a new cycle. The parameters *C1* and *C2* are acceleration coefficients that control the impact of personal best and global best values on the particle's movement.

We mentioned earlier that we have several options for deploying cloud-native applications to AWS; the most common approach is to use single or several virtual servers using EC2 instances. We used Amazon Elastic Container Service (ECS) in our current deployment. ECS is a fully managed container orchestration service, and it allows the user to run, stop, and manage Docker containers on a cluster, simplifying the process of deploying, managing, and scaling containerized applications. ECS uses *Task Definitions* as a blueprint for a set of containers that run together on the same host. It defines parameters such as which Docker images to use, the CPU and memory requirements, and networking information, among others. A task

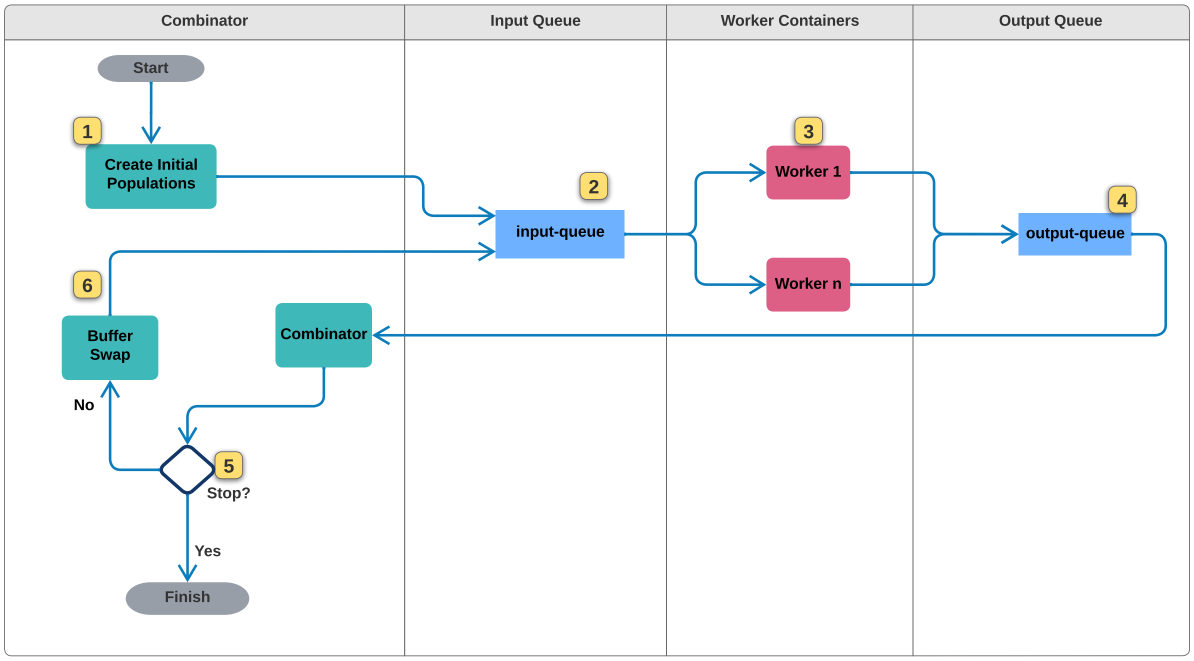

Fig. 1. Container-based architecture proposed by García-Valdez and Merelo [29] in which multiple swarms are created and added to an `input-queue`, multiple workers then evolve these swarms, and finally, they migrate in the `combinator` process. This cycle is repeated several times. In this work, we included dynamic adaptation of parameters, which are implemented inside the `combinator` module

definition is similar to a `docker-compose` file in the sense that it defines the interaction of several containers. To minimize the need to manage the underlying infrastructure, we use what is called `Fargate Tasks`, a serverless compute engine that eliminates the need for launching and managing EC2 instances ourselves.

We only need to specify the resource requirements needed for our task. Fig. 2, shows the main cloud computing technologies from Amazon's AWS used in the deployment of the multi-swarm algorithm. We had specified a Docker container definition for each component described previously using a `Docker` file. With this Docker definition, we could run the algorithm locally using a `docker-compose` script. As a first step for cloud deployment, we must create a repository for each container definition in the Amazon Elastic Container Registry (ECR) (1). ECR is a fully managed container registry service provided by AWS. This registry stores, manages, and deploys Docker container images. The advantage of having an image repository is that we can keep several versions of each image. As a second step (2), we defined a Fargare Task Definition for each component. We then created an ECS cluster is a logical grouping of container instances or tasks on which we can run our containerized applications. We now run the Task Definitions (3) and observe the execution logs in Amazon CloudWatch; this is a monitoring and observability service. These steps can be realized using a browser-based user interface or using a command-line interface.

Each of the services mentioned above has a cost that is charged monthly, the current prices are shown in Table 1.

First, we must consider the cost of storing the Docker container images in the ECR. We have

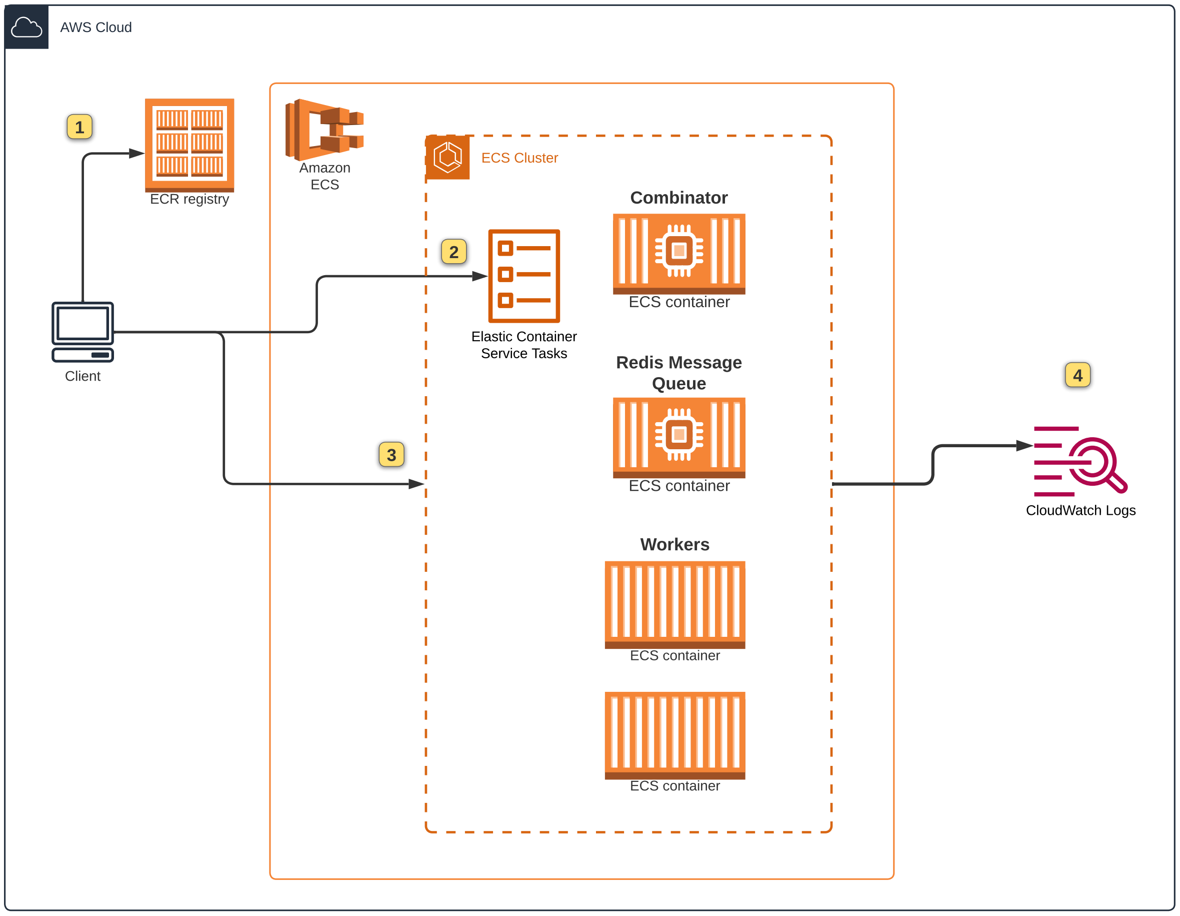

Fig. 2. Implementation using AWS cloud technologies

50GB a month for free, but it is important to remember that some container images can grow to several GB depending on the operating system, languages, libraries, and software included in the container. The data transfer cost for a private registry is negligible; there is no cost for uploading the images and no data transfer to the outside. When we specify the computing resources for running a specific Fargate Task, we select the number of vCPUs to assign, and depending on our selection, there is a minimum and maximum amount of memory we can assign to the task. At this time, when selecting one vCPU, the minimum memory is 2 GB, and the maximum is 8 GB, in 1 GB increments. We selected 3 GB memory for our containers. To run these experiments, we did not need additional ephemeral storage; we used the 20 GB included for free. Finally, we used the CloudWatch service extensively to keep track of the algorithm and summarize the results.

4 Use Case

To test the deployment, we compare the execution time of three configurations running on Amazon's Elastic Container Service against a local execution using a Mac Studio workstation. We describe next

Table 1. Cost of AWS Fargate Services, the Elastic Container Registry is needed to upload images for creating and running the containers; each container runs in a Task using one vCPU, 3 GB of RAM, and 20GB of ephemeral Storage. The results of the experiment are stored in Amazon Cloud Watch logs

Service	Price	Free tier
Elastic Container Registry		
Storage	$0.10 per GB	50 GB
Transfer IN	$0.00 per GB	
Transfer OUT		
AWS Fargate		
each vCPU per hour	$0.040480	
each GB per hour	$0.004445	
Ephemeral Storage GB per hour	$0.000111	20 GB
Amazon Cloud Watch		
Collect (Data Ingestion)		
Standard	$0.50 per GB	0 to 5 GB
Infrequent Access	$0.25 per GB	0 to 5 GB

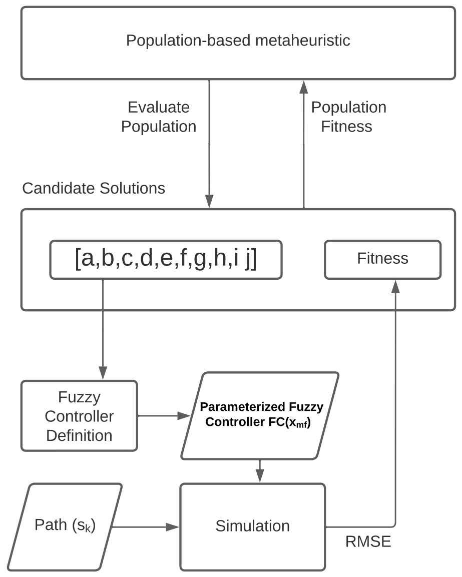

Fig. 3. Fuzzy Controller Problem [18]

the resource-intensive optimization problem used as proof-of-concept.

In this case, we want to optimize the membership functions for a fuzzy controller for a rear-wheel controller [24]. The fuzzy controller is presented in a previous paper [19]. This problem was selected because it consumes extensive computational resources to evaluate candidate solutions. Fig. 3 details the evaluation components for each candidate solution. Each solution consists of a list of floating-point parameters for establishing the components of the membership functions for a fuzzy controller definition. With this definition, we create a controller instance and simulate the control by following several paths with different degrees of difficulty. We then calculate the average of the RMSEs obtained for each path. This average is considered the fitness of the candidate solution. If a controller cannot follow the track to a certain degree, the simulation is terminated, and that controller is assigned the worst fitness value (RMSE=5000).

The current implementation uses the Python language, which means that even if there are multiple CPUs in the computer, the Python interpreter process uses only a single thread or CPU at a time. For this reason, we only run workers in Fargate instances using a single vCPU. Each vCPU is a hyperthread of an Intel Xeon CPU core. We validated this by running a few experiments using instances with 4vCPUs, but we did not find a substantial difference in the compute time, so the additional cost is not justified.

For the cloud configurations, we compare three configurations: 36, 40, and 90 workers. This means 36, 40, and 90 vCPUs. This number of processors is higher than what we normally find in a PC Workstation. We are comparing the results with a local implementation using a high-end workstation, in this case, a 2022 Mac Studio with an Apple M1 Ultra chip with 20 CPUs (16 performance and 4 efficiency). We use 16 workers in this configuration to fully exploit the maximum number of performance cores.

We usually set the number of swarms and the number of workers with a similar value; if we increase the number of swarms, many will remain in the `input-queue` waiting for a worker to be available. If we have more workers than swarms, the situation will be worse because we now have workers waiting for swarms to be available. To test how much the ratio of swarms/workers affects the time of execution, we test with these four configurations: the same number of swarms and workers with 36 each, 36 workers and 40 swarms, 40 workers and 36 swarms, and finally, 90 workers and 80 swarms.

The multi-swarm is composed of 36, 40, or 80 swarms with ten particles each. Although this is a small size for a traditional single swarm PSO, in the case of an MS-PSO, the total size of the swarm is the sum of all swarms. Each local PSO will run four iterations of the algorithm before returning the resulting swarm state to the `output-queue`. All swarms will complete ten cycles, meaning they will go through the combiner and back to the `input-queue` a total of ten times each.

The parameters of the multi-swarm version of the PSO algorithm are shown in Table 2. We use the number of function evaluations per second (EPS) to compare the time-to-solution performance. As we mentioned before, the fitness evaluation function is the most computationally demanding component of the PSO algorithm, and this is why the number of calls to this function is a commonly used metric to evaluate the amount of work needed to find a solution independently of the parameters or hardware used. In this case, we change the parameters of the MS-PSO algorithm depending on the number of swarms and workers available. These changes in the configuration yield different numbers of function evaluations for each configuration. To consider this, we compare the execution speed by EPS. Higher values are better.

5 Results

Table 3 shows the results for the proposed configurations, giving the average (n=15) RMSE obtained, the average time (in seconds), and the function evaluations per second of the experiments. We also included the cost per execution according to the number of workers (vCPUs) and memory multiplied by the time it takes to finish an average experiment. We discuss the results in two subsections. First, we focus on the number of function evaluations per second because this measure gives us the work each worker provides. In the second, we put our attention on the cost of each experiment configuration.

5.1 Number of Function Evaluations per Second

When changing the ratio between workers and swarms, we notice a significant increase in the #FE per second when the number of swarms equals the number of workers. These results can be explained by analyzing the timeline of three representative runs. The fourth configuration is not shown in the graph due to the complexity of the data handling, but it would be similar to Fig. 6. Figs. 4 to 6 show the timelines for the proposed configurations. The x-axis shows the time elapsed in seconds from the beginning of the experiment. Each row represents a particular worker, identified by a sequential number. Each box represents the time range in seconds, beginning when a swarm is pulled from the `input-queue` until pushed to the `output-queue`.

A sequential integer identifies each swarm. The elapsed time is not the same in all cases because the time to complete the path depends on the controller's performance. Moreover, when the error or distance to the path is large, the simulation is interrupted, thus taking less time. To identify each swarm, the color corresponds to the order in which each worker received it, and this is to avoid those cases where the same swarm is sequentially received by the same worker twice. In this case, it isn't easy to distinguish between the two boxes if the swarm has the same color. The vertical bold line indicates when the number of evaluations has been reached. At this point, no more swarms are pushed to the `input-queue`. Because workers take swarms asynchronously, the work already in the `input-queue` is still pulled by workers, and the work they are processing is also finished. This is why more work is processed beyond the bold line.

Fig. 4 shows the timeline for the experiment using 36 workers and 36 swarms. When we have this configuration, it is common for each swarm to be executed exclusively by the same worker. This is because the `input-queue` will usually be empty. Once a worker finishes processing a particular swarm, it will be available for the next swarm in the

Table 2. Experimental Setup, these values were obtained from initial experiments and these are the same parameters used in our previous work [18]. The number of Function Evaluations are different because some parameters are adjusted to the number of swarms or workers used

Algorithm	**Parameter**	**Range[min,max]**		
MS-PSO	Communication Topology	Fully connected		
	Speed.	Minimum = [-0.20, -0.30]		
		Maximum = [0.20, 0.30]		
	Coefficients C_1, C_2	[1.0, 2.0]		
	Local	Cloud		
Size	10	10	10	10
Swarms	16	36	40	80
Iterations	8	4	4	4
Cycles	10	10	10	10
Dimensions	15	15	15	15
# Function Evaluations	12800	14400	16000	32000

Table 3. Function Evaluations per second comparison between local and three configurations on AWS Elastic Container Service

	Local	AWS Elastic Container Service			
Configuration	Mac Studio	1	2	3	4
Workers	16	36	36	40	90
Swarms	10	36	40	36	80
#FE	12800	14400	16000	14400	32000
RMSE	0.00271	0.00328	0.00404	0.00303	0.00238
Time in seconds	754.9677	840.660	1059.593	936.064	1324.359
Evals. per second	16.954	17.129	15.100	15.383	24.163
Cost per run (USD)		0.4524	0.570	0.559	1.782

`input-queue`. The `combinator` returns the same swarm to the `initial-queue`, and the same worker is available to retake the same swarm.

Furthermore, workers do not have a noticeable waiting time when this condition happens. We can see that this happens in all the workers. It is essential to notice that *worker 21* takes a swarm, but this swarm is never pushed back into the `output-queue`. This could be because of an error in the swarm configuration or a bug in the code. For instance, we do not have a fault-tolerant solution to restart a worker if the work is not returned after a specific time. However, in any case, the nature of the event-driven solution still works well. The extra work needed to reach the number of evaluations required to complete the experiment is distributed among all the other workers. Fig. 5 shows when we have more swarms than workers; consequently, swarms are not always processed by the same worker as in the previous case.

We need to assess in a future work if this behavior benefits the search algorithm. The drawback of this case is that the work takes more time because we have fewer resources to do the computation. Finally, Fig. 6 shows the case where we have spare workers. The problem with this approach is that the workers must wait until a new population is available. For example, *worker 5* needs to wait until a swarm is available. In this case, population PSO-11 is received by *worker 5* after *worker 16* finished working on the swarm. Nevertheless, even if we have more computing

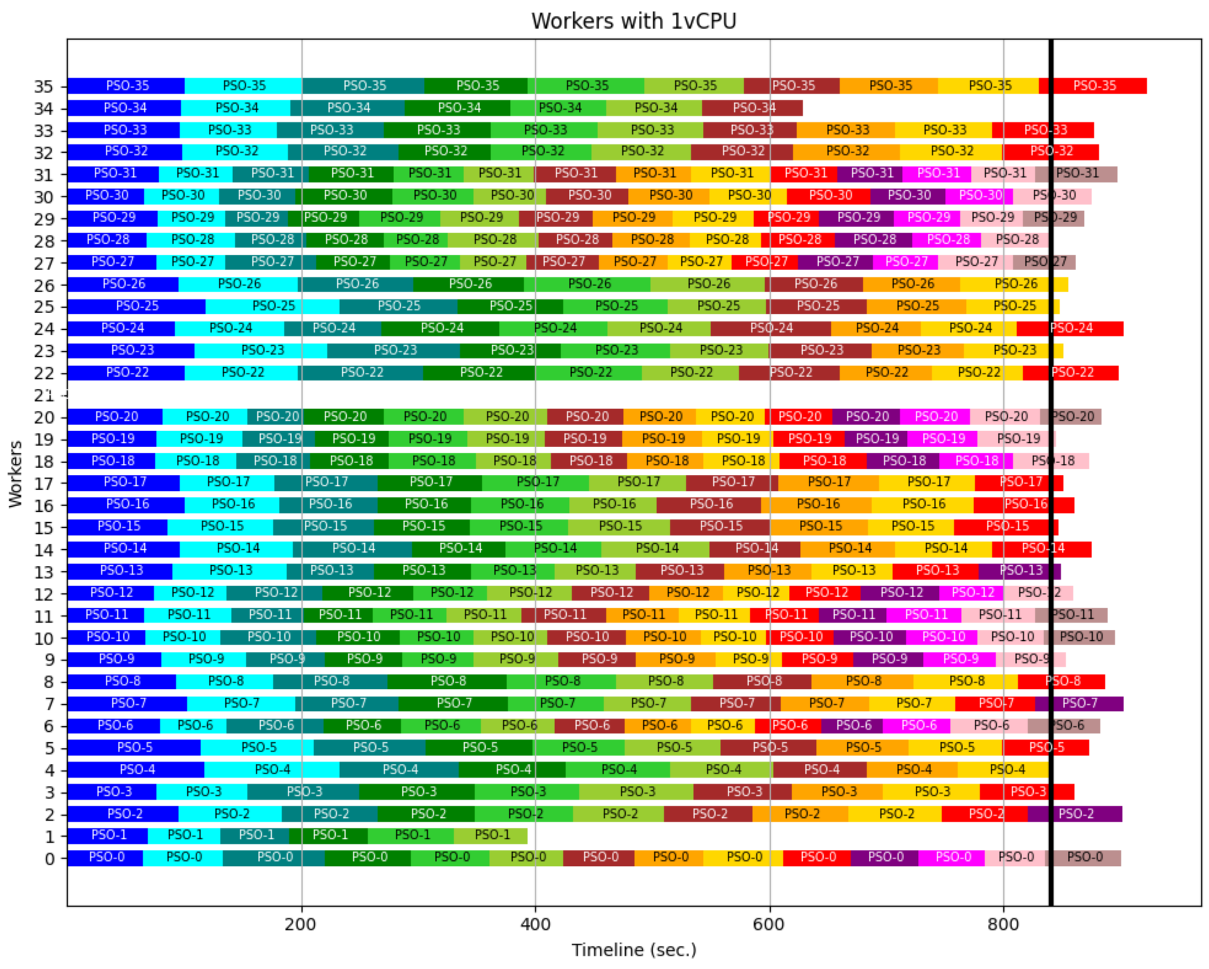

Fig. 4. Execution of 36 workers and 36 swarms, The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container

resources, because of the additional idle time of the workers, the overall time is not the best.

5.2 Cost per Run

When running our experiments in a cloud environment, we can use a certain amount of resources for free. Still, these are limited resources, so the cost of each run is an essential factor in deciding the configuration we use. The first configuration is less expensive and offers more evaluations per second, finishing an experiment in about fourteen minutes. On the other hand, the faster configuration gives us 57% more evaluations but costs almost three times more (293.90% increase).

When comparing against the local execution, results indicate that we needed about 2.25 times the number of workers to match the time-to-finish performance of a local execution. Several factors could have an impact on this difference. The communication time for a local container and the network is faster than in a cloud environment; the M1 chip with a fast memory bandwidth (800GB/s) is more rapid than a vCPU. Moreover, there could be differences in the virtualization environment

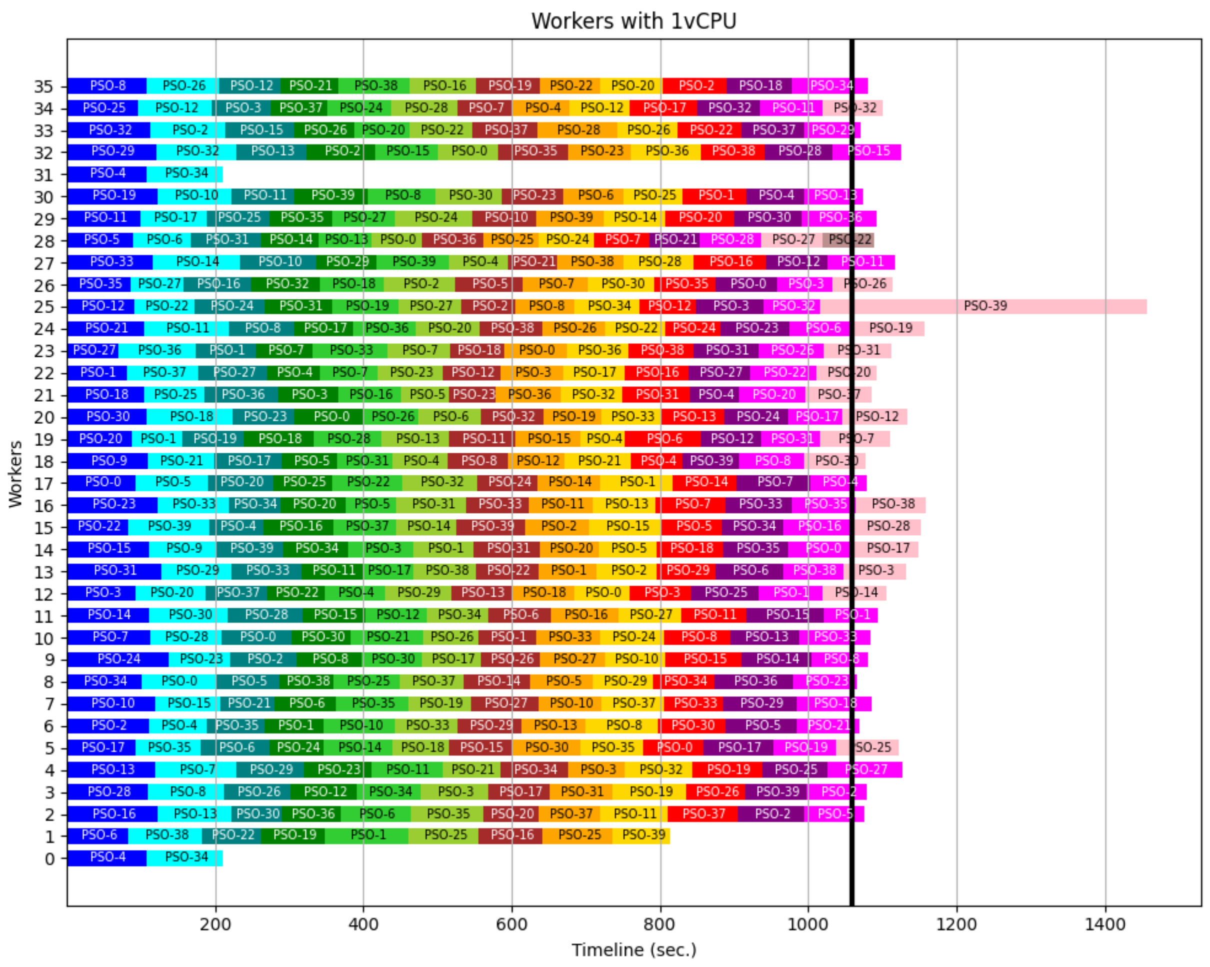

Fig. 5. Execution of 36 workers and 40 swarms. The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container

where the docker host runs; the Docker engine used in Mac Studio is optimized for the M1 chip. On the other hand, running these experiments in a cloud environment does not incur an initial investment on a workstation computer.

6 Conclusion and Future Work

This paper details the cloud-native design and deployment options for a multi-swarm PSO algorithm. The algorithm addresses the computationally demanding optimization problem of tuning the membership functions of a fuzzy controller applied to rear-wheel path tracking. We deployed the algorithm on Amazon's container platform and compared this solution against a local docker-based deployment. We compared the four configurations to highlight the differences between worker and swarm ratios.

We found that the most efficient configuration is to have the same number of workers, threads, and swarms.

We found also that for specific use cases, it is necessary to increase the number of workers when deploying the algorithm on a cloud provider, as we observed in the fourth configuration, where having

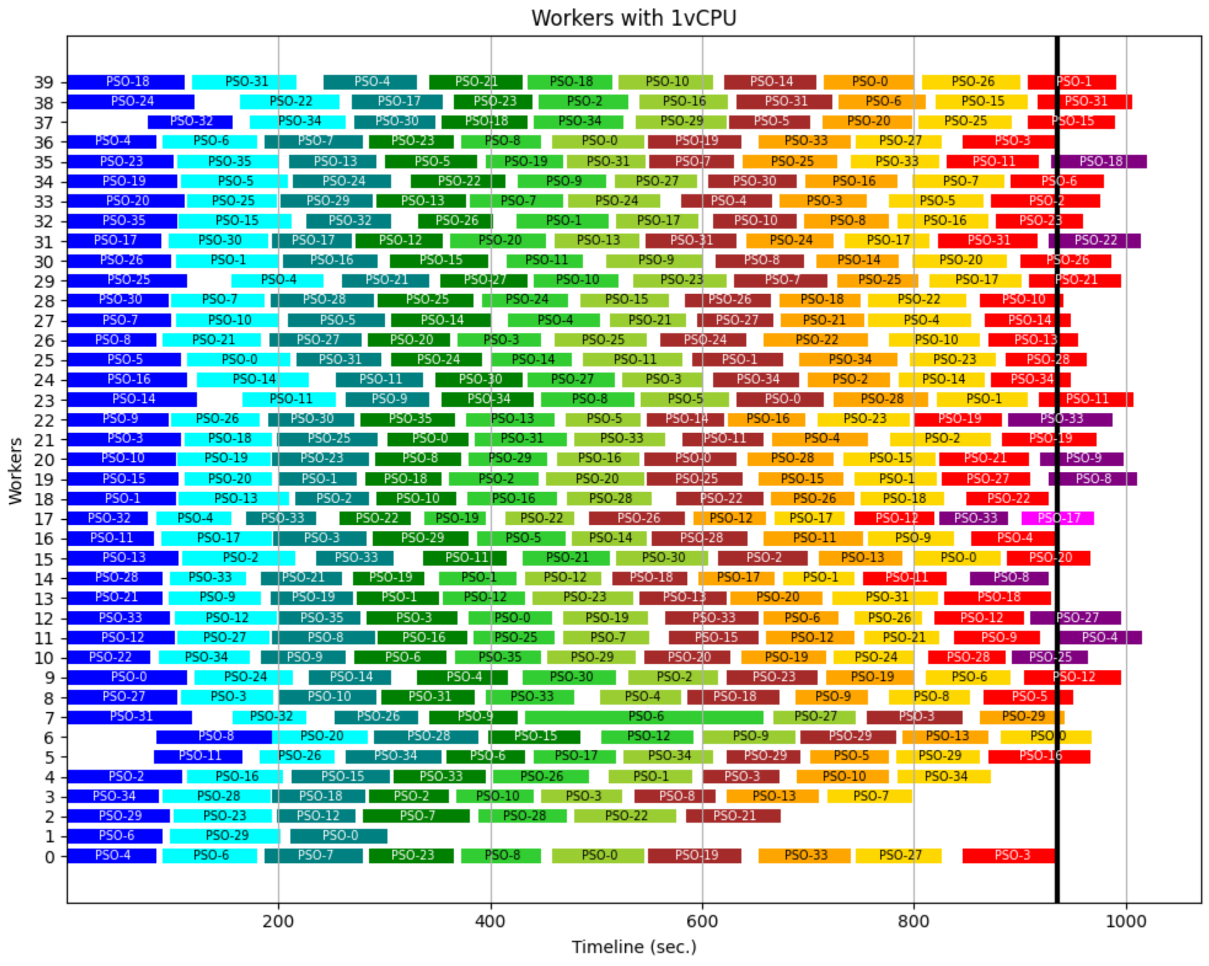

Fig. 6. Execution of 40 workers and 36 swarms.The timeline at the bottom marks the progression in seconds, indicating the duration of each PSO instance's activity inside a particular worker container

more workers than swarms resulted in a higher number of evaluations per second.

For future research, we aim to expand upon the design options on both the algorithmic and deployment sides. We can dynamically change the swarm size or the number of swarms (and workers) to ascertain potential performance gains.

When implementing these options, we can test the auto-scaling features of the cloud platform with the intent to optimize resource utilization without degrading the algorithm's exploratory capacities.

Acknowledgments

This work is funded by Project 21094.24-P of 2024 TecNM research grants.

References

1. **Baldini, I., Castro, P., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P. (2016).** Cloud-native, event-based programming for mobile applications. Proceedings - International Conference

on Mobile Software Engineering and Systems, MOBILESoft 2016, pp. 287–288. DOI: `10.1145/2897073.2897713`.

2. **Bonér, J., Farley, D., Kuhn, R., Thompson, M. (2014).** The reactive manifesto.

3. **Carneiro, T., Da Nóbrega, R. V. M., Nepomuceno, T., Bian, G.-B., De Albuquerque, V. H. C., Reboucas Filho, P. P. (2018).** Performance analysis of google colaboratory as a tool for accelerating deep learning applications. IEEE Access, Vol. 6, pp. 61677–61685.

4. **Ciurea, S. (2013).** Determining the parameters of a sugeno fuzzy controller using a parallel genetic algorithm. 2013 19th International Conference on Control Systems and Computer Science, IEEE, pp. 36–43.

5. **Cortes-Rios, J. C., Gómez-Ramírez, E., Ortiz-de-la Vega, H. A., Castillo, O., Melin, P. (2014).** Optimal design of interval type 2 fuzzy controllers based on a simple tuning algorithm. Applied Soft Computing, Vol. 23, pp. 270–285.

6. **De Lucia, A., Salza, P. (2017).** Parallel Genetic Algorithms in the Cloud. Ph.D. thesis, University of Salerno.

7. **Dziurzanski, P., Zhao, S., Przewozniczek, M., Komarnicki, M., Indrusiak, L. S. (2020).** Scalable distributed evolutionary algorithm orchestration using docker containers. Journal of Computational Science, pp. 101069.

8. **García-Valdez, M., Trujillo, L., Merelo, J.-J., De Vega, F. F., Olague, G. (2015).** The evospace model for pool-based evolutionary algorithms. Journal of Grid Computing, Vol. 13, No. 3, pp. 329–349.

9. **Gilbert, J. (2018).** Cloud Native Development Patterns and Best Practices: Practical architectural patterns for building modern, distributed cloud-native systems. Packt Publishing Ltd.

10. **Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B. E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J. B., Grout, J., Corlay, S., et al. (2016).** Jupyter notebooks-a publishing format for reproducible computational workflows.. Elpub, Vol. 2016, pp. 87–90.

11. **Kratzke, N., Quint, P.-C. (2017).** Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software, Vol. 126, pp. 1–16.

12. **Liu, G., Huang, B., Liang, Z., Qin, M., Zhou, H., Li, Z. (2020).** Microservices: architecture, container, and challenges. 2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C), IEEE, pp. 629–635.

13. **Ma, H., Shen, S., Yu, M., Yang, Z., Fei, M., Zhou, H. (2019).** Multi-population techniques in nature inspired optimization algorithms: a comprehensive survey. Swarm and evolutionary computation, Vol. 44, pp. 365–387.

14. **Malawski, M., Gajek, A., Zima, A., Balis, B., Figiela, K. (2020).** Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. Future Generation Computer Systems, Vol. 110, pp. 502 – 514. DOI: `https://doi.org/10.1016/j.future.2017.10.029`.

15. **Mancilla, A., Castillo, O., García-Valdez, M. (2023).** Optimization of fuzzy controllers using distributed bioinspired methods with random parameters. In Hybrid Intelligent Systems Based on Extensions of Fuzzy Logic, Neural Networks and Metaheuristics. Springer, pp. 189–197.

16. **Mancilla, A., Castillo, O., García-Valdez, M. (2024).** Fuzzy adaptation of parameters in a multi-swarm particle swarm optimization (pso) algorithm applied to the optimization of a fuzzy controller. In New Horizons for Fuzzy Logic, Neural Networks and Metaheuristics [in press]. Springer.

17. **Mancilla, A., Castillo, O., Valdez, M. G. (2021).** Optimization of fuzzy logic controllers with distributed bio-inspired algorithms.

Recent Advances of Hybrid Intelligent Systems Based on Soft Computing, pp. 1–11.

18. **Mancilla, A., Castillo, O., Valdez, M. G. (2022).** Evolutionary approach to the optimal design of fuzzy controllers for trajectory tracking. Intelligent and Fuzzy Techniques for Emerging Conditions and Digital Transformation, Springer International Publishing, Cham, pp. 461–468.

19. **Mancilla, A., García-Valdez, M., Castillo, O., Merelo-Guervós, J. J. (2022).** Optimal fuzzy controller design for autonomous robot path tracking using population-based metaheuristics. Symmetry, Vol. 14, No. 2, pp. 202.

20. **Merelo, J. J., Fernandes, C. M., Mora, A. M., Esparcia, A. I. (2012).** Sofea: a pool-based framework for evolutionary algorithms using couchdb. Proceedings of the 14th annual conference companion on Genetic and evolutionary computation, ACM, pp. 109–116.

21. **Merelo, J. J., Laredo, J. L. J., Castillo, P. A., García-Valdez, J.-M., Rojas-Galeano, S. (2019).** Scaling in concurrent evolutionary algorithms. Workshop on Engineering Applications, Springer, pp. 16–27.

22. **Merelo-Guervos, J. J., Mora, A., Cruz, J., Esparcia-Alcázar, A., Cotta, C. (2012).** Scaling in distributed evolutionary algorithms with persistent population. 2012 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. DOI: 10.1109/CEC.2012.6256622.

23. **Oh, S.-K., Jang, H.-J., Pedrycz, W. (2009).** The design of a fuzzy cascade controller for ball and beam system: A study in optimization with the use of parallel genetic algorithms. Engineering Applications of Artificial Intelligence, Vol. 22, No. 2, pp. 261–271.

24. **Paden, B., Čáp, M., Yong, S. Z., Yershov, D., Frazzoli, E. (2016).** A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Transactions on intelligent vehicles, Vol. 1, No. 1, pp. 33–55. Publisher: IEEE.

25. **Salza, P., Ferrucci, F. (2016).** An approach for parallel genetic algorithms in the cloud using software containers. arXiv preprint arXiv:1606.06961.

26. **Salza, P., Ferrucci, F., Sarro, F. (2016).** Develop, deploy and execute parallel genetic algorithms in the cloud. Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, pp. 121–122.

27. **Sherry, D., Veeramachaneni, K., McDermott, J., O'Reilly, U. M. (2012).** Flex-GP: Genetic programming on the cloud. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7248 LNCS, pp. 477–486. DOI: 10.1007/978-3-642-29178-4_48.

28. **Thönes, J. (2015).** Microservices. IEEE Software, Vol. 32, No. 1, pp. 116–116. DOI: 10.1109/MS.2015.11.

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

30. **Valenzuela, R. M., Valdez, M. G. (2015).** Implementing pool-based evolutionary algorithm in Amazon Cloud Computing Services. In Design of Intelligent Systems Based on Fuzzy Logic, Neural Networks and Nature-Inspired Optimization. Springer, pp. 347–355.

31. **Varghese, B., Buyya, R. (2018).** Next generation cloud computing: New trends and research directions. Future Generation Computer Systems, Vol. 79, pp. 849–861.

32. **Vecchiola, C., Kirley, M., Buyya, R. (2009).** Multi-objective problem solving with offspring on enterprise clouds. arXiv preprint arXiv:0903.1386.

Article received on 05/10/2024; accepted on 07/01/2025.
Corresponding author is Mario García-Valdez.