Dynamic Software Containers Workload Balancing via Many-Objective Search

Anwar Ghammam, Thiago Ferreira, Wajdi Aljedaani, Marouane Kessentini, and Ali Husain

Abstract—Software containers are becoming the new state of the art in the industry as they are extensively used to deploy systems. Indeed, the use of containers enables better modularity, reusability, and portability compared to other technologies. As the complexity of software systems is dramatically increasing, it is critical to enable optimal usage of the needed resources to execute them such as memory and CPU. Thus, different scheduling strategies are proposed to select the most suitable nodes to execute a set of containers. For instance, the default strategy in the Docker Swarm kit scheduling framework is based on an equal distribution of the containers between nodes independent of their sizes and consumed resources. However, balancing the containers' workload is a complex problem due to the conflicting objectives of minimizing the number of selected nodes, minimizing the number of containers per node, the number of changes compared to the original schedule, and the coupling between containers allocated to different nodes.
To deal with those conflicting scheduling objectives, we propose a scheduler based on a many-objective optimization approach for scheduling the execution of containers between multiple nodes. The proposed approach aims at finding the best allocation for containers in nodes that leads to efficient utilization of resources. To evaluate our approach, we compared the performance of multiple many and multi-objective techniques based on NSGA-II, NSGA-III, and IBEA algorithms using 48 Docker-related systems and the results show that NSGA-III outperforms the other algorithms in quality attributes as well as in CPU, Memory and Network usage.

Index Terms—Docker, container scheduling, many-objective optimization

1 Introduction

Software containers are becoming the new state of the art in the industry as they are extensively used to deploy systems [1]. Indeed, the use of containers enables better modularity, reusability, and portability compared to other traditional technologies [2]. For instance, the Docker container is considered one of the most important pillars for software deployment as it provides higher performance and flexibility in comparison with a traditional hypervisor-based virtualisation [1]. Thus, Docker and the software containerization technology are becoming the main parts of the cloud strategies of most industry organizations [3]. Despite the benefits and popularity of using Docker containers, there are several challenges associated with the optimal usage of the resources consumed by this technology as a large number of containers may need to be executed and orchestrated due to the high modularity of Docker architectures [4].

Due to the dramatic increase in the number and size of containers needed to build systems, there is a critical need for efficient mechanisms to schedule and orchestrate the execution of containers among many nodes in the cloud clusters. Thus, several container scheduling tools are proposed, such as Docker Swarm developed by Docker [5], Mesos by Apache [6], and Kubernetes by Google [7]. Generally speaking, despite their efficiency, these strategies are too simple to handle complex container execution. It is assumed that Docker Swarm has no prior knowledge regarding the workload or the container's resource requirements. The only available scheduling strategy is called Spread, which basically schedules a service task based on spreading the number of containers equally to all Docker hosts, and all the extra configuration has to be performed manually [8]. Thus, it is important to create more sophisticated, high-level, and adaptive allocation strategies in order to guarantee a balanced workload among devices, the service's performance requirements, efficient communications between containers, and more efficient utilization of resources in terms of CPU and memory.

To deal with these challenges, several scheduling techniques for containers were recently proposed [9], [10], [11], [12]. Most of them are based on the use of optimization techniques due to the complexity of the problem in terms of the number of scheduling alternatives. For instance, Kaewkasi et al. [9] adopted the Ant Colony Optimization (ACO) algorithm to implement a new container scheduler for SwarmKit which spread the containers over Docker hosts to balance the overall resource usages and therefore lead to increased performance of applications. The proposed approach continuously computes the available resources in every node every time it schedules a container. Guerrero et al. [10] proposed a genetic algorithm approach to implement a container allocation strategy and elasticity management by optimizing the elasticity of the currently deployed applications and maximizing the reliability of the micro-services by avoiding single points of failure. However, the proposed strategies are limited to only two objectives while many conflicting criteria should be taken into consideration within the container scheduling problem. Indeed, an efficient scheduling approach for containers may need to consider the

- *Anwar Ghammam, and Marouane Kessentini are with the Department of Computer Science and Engineering, Oakland University, Rochester, MI, USA.*
  E-mail: anwarghammam@oakland.edu, kessentini@oakland.edu
- *Thiago Ferreira is with the College of Innovation & Technology, University of Michigan-Flint, USA.*
  E-mail: thiagod@umich.edu
- *Wajdi Aljedaani is with the Department of Computer Science and Engineering, University of North Texas, USA.*
  E-mail: wajdi.aljedaani@unt.edu
- *Ali Husain with the Ford Motor Company, USA.*
  E-mail: ahusain4@ford.com

available resources in terms of memory and CPU but also reduce the changes in the current configuration (e.g. moving containers between nodes), the communications between containers located in different nodes (e.g. coupling) and balancing the software load between multiple nodes.

To address the above challenges, we propose a new many-objective optimization approach for the Docker containers scheduling problem. The goal is to find the best allocation of containers that can lead to a better workload balance and performance. The number of scheduling combinations to assign containers to nodes is high. Thus, the search space to explore is combinatorial which requires the use of an intelligent computational search technique. Furthermore, the different scheduling criteria are conflicting thus, we adopted a many-objective search algorithm, based on NSGA-III [13], to find a trade-off between four conflicting objectives. Our many-objective search-based software engineering approach aims at finding the rescheduling solution that: optimizes the structure of the cluster by optimizing some metrics such as the number of selected nodes in the cluster, the average of containers per node, optimizes the communications between containers by minimizing the coupling (dependent containers allocated to different nodes), and finally minimizing the number of required changes to move from the current scheduling to the new one (e.g., move container) to guarantee a fast allocation of containers to the cluster nodes.

To evaluate our approach, we compared the performance of multiple many and multi-objective techniques based on NSGA-II, NSGA-III [14], [15], and IBEA algorithms [16] using 48 Docker-related systems. The results show that NSGA-III can generate the best scheduling solutions considering the traditional quality indicators for computational searches, such as Hypervolume, IGD, and Contribution metrics, as well as other validation metrics such as CPU, memory, and network usage. We have also created an online appendix for a demo of our platform and related experiments material [17].

The primary contributions of this paper can be summarized as follows:

- We introduced a novel formulation of the containers scheduling problem as a many-objective problem that considers several conflicting objectives such as structural improvement, coupling, and the number of changes. The definition of the fitness functions was formulated based on the needs of our industry partner, the Ford Motor Company, to optimize specific objectives related to the usage of ECUs resources in the car.
- We compared three different many-objective optimization algorithms as it is the first formulation in the literature of container scheduling as a many-objective problem.
- We reported the results of an empirical study of our many-objective technique compared to the docker default approach. The obtained results provide evidence to support the claim that our proposal is, on average, more efficient than the existing techniques based on a benchmark of 48 open-source docker projects.

The remaining of this paper has been organized as follows: Section 2 reviews the Docker Container tool, as well as the three, used multi-objective optimization algorithms used in our approach. Section 3 then discusses the proposed approach, the population presentation and the objective functions. Section 4 is an empirical study to evaluate the feasibility of our approach by defining the research questions, quality indicators, used systems and algorithm configuration. Section 5 presents some related works. Finally, Section 6 concludes this paper.

2 Background

2.1 Docker and Container-based Projects

Docker [18], is one of the most popular container virtualization technologies [3], [19]. It packs the application's code and dependencies into a lightweight, standalone, and portable execution environment aiming to deploy containerized applications in a quick process.

Dockerfile is a document containing a sequence of instructions used for creating the computational environment, following the notion of Infrastructure-as-Code (IaC) [20] and it is used by Docker to build the container images.

```
1  FROM node:argon
2  # Create app directory
3  WORKDIR /usr/src/app
4  # Install app dependencies
5  COPY package*.json /usr/src/app/
6  RUN npm install
7  # Bundle app source
8  COPY . /usr/src/app
9  # Expose the app to the outside world
10 EXPOSE 8080
11 CMD [ "npm", "start" ]
```

Listing 1: Dockerfile example.

An illustrative example of a Dockerfile is shown in Listing 1. In this listing, the Dockerfile has seven instructions where the definition of each one is described in Table 1.

TABLE 1: Dockerfile setup instructions.

Instruction	Description
ENV	Setting the environment variables
ARG	Defining variables that can be set at build time
WORKDIR	Setting working directory for all subsequent instructions
COPY	Copying files from host to the Docker image
ADD	Similar to COPY instruction but supports two additional tricks. It supports the use of a URL instead of a local file and can recognize the archive format and extract it directly into the destination
LABEL	Key value pairs, indicating image metadata
RUN	Executing any command
EXPOSE	Informs Docker that the container is exposing a particular port
CMD	Setting a command and/or parameters, that executes when the container is starting and which can be overwritten at build time
ENTRYPOINT	Setting executable that will always run when the container is initiated and cannot be overwritten.

Docker-compose [21] provides a unified setup routine that deploys several containers using a YAML configuration file, as known as, `Docker-compose.yml` (or just Docker-compose). In the Docker paradigm, each container captures one particular component of the software (e.g., database). Thus, when creating a multi-component application using Docker, it is inevitable to combine multiple software components (containers) into a workflow. Then, Docker-compose can tackle this

problem by integrating containers and running them properly.

```
version: "3.7"
    services:
        server:
            build: .
            ports:
                - 8080:4040
            environment:
                - DB_ADDRESS=database-mongo
                - DB_PORT=27017
                - PORT=4040
            depends_on:
                - database
        database:
            image: mongo:latest
            volumes:
                - mydata:/data/db
    volumes:
        mydata:
```

Listing 2: Docker-compose example.

An example of a Docker-compose file is available in Listing 2. The example shows that the Docker-compose file is composed of two components/containers (SERVER and DATABASE). The SERVER component is represented by a local image (built from a `Dockerfile`, for instance, that one available in Listing 1), and the DATABASE component is created from the "mongo" image, hosted in DockerHub [22] (an online registry for Docker Images). Docker-compose file also provides a list of setup attributes which can be listed in Table 2.

TABLE 2: Docker-compose setup attributes.

Attribute	Description
BUILD	Setting path to the build context
IMAGE	Setting the image to start the container from
PORTS	Specify ports binding
ENVIRONMENT	Setting environment variables
DEPENDS_ON	Expressing dependency between services
VOLUMES	Setting volume bindings (host paths or named volumes)

Any typical Docker project includes the above files along with source code files written in typical programming languages, such as Java, to host the containers and enable their executions and synchronization with other features of the app that may not be containerized.

The way that tasks or containers are scheduled on a Swarm Mode cluster is governed by a scheduling strategy. Currently, Swarm Mode has a single scheduling strategy, called "Spread" [8]. The spread strategy attempts to schedule a service task based on an assessment of the resources available on cluster nodes. In its simplest form, this means that tasks are evenly spread across the nodes in a cluster. For example, if we create a service with three replicas, each replicated task will be scheduled on a different node.

2.2 Many-Objective Evolutionary Algorithm: NSGA-III

Non-dominated Sorting Genetic Algorithm III (NSGA-III) is a more recent optimization algorithm proposed by Deb et al. [14], [15], similar to NSGA-II, but with significant changes in its selection mechanism aiming to improve the results of many-objective problems. Unlike in NSGA-II, the diversity among population members in NSGA-III is aided by supplying a number of well-spread reference points.

NSGA-III demonstrates its efficacy in solving 2 to 15-objective optimization problems, and it is also extended easily to solve constrained optimization problems, and can be used with small population size (such as a population of size 100 for a 10-objective optimization problem). The algorithm is shown in Algorithm 1.

Algorithm 1: Generation t of NSGA-III. Adapted from [14], [15]

Input : H structured reference points Z_r or supplied aspiration points Z_a , parent population P_t
Output : P_{t+1}

1 $S_t = \emptyset$, i = 1;
2 Q_t = Recombination+Mutation(P_t);
3 $R_t = P_t \cup Q_t$;
4 (F_1, F_2, ...) Non-dominated-sort(R_t);
5 **repeat**
6 | $S_t = S_t \cup F_i$ and i = i + 1;
7 **until** $|S_t| \geq N$;
8 $F_l = F_i$(Last front to be included);
9 **if** $|S_t| = N$ **then**
10 | $P_{t+1} = S_t$, break;
11 **else**
12 | $P_{t+1} = \bigcup_{j=1}^{l-1} F_j$;
13 | Points to be chosen from F_l : K = N - $|P_{t+1}|$;
14 | Normalize objectives and create reference set Z_r : Normalize(f^n, S_t, Z_r, Z_s, Z_a);
15 | Associate each member s of S_t with a reference point: [$\Pi(s)$, d(s)] =Associate(S_t , Z_r) {Π(s): closest reference point, d: distance between s and $\Pi(s)$ };
16 | Compute niche count of reference point j $\in$ Z_r : φ_j=$\sum_{s \in S_t/Ft}$ ((Π(s) = j) ?1 : 0);
17 | Choose K members one at a time from F_l to construct P_{t+1} : Niching(K, φ_j,Π, d, Z_r , F_l, P_{t+1}) ;
18 **end**

First, same as NSGA-II, the parent population P_t is randomly initialized in the specified domain, then the binary tournament selection, crossover, and mutation operators are applied to create an offspring population Q_t (Line 1–2). Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected for the next generation.

Unlike in NSGA-II (which uses the crowding distance measure for selecting the best set of points from the last front that can be partially accepted), in NSGA-III the supplied reference points Z_r are used to select these remaining members. The chosen reference points can either be predefined in a structured manner or supplied preferentially by the user. To accomplish this, objective values and reference points are first normalized to have an identical range. Thereafter, the orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is calculated.

Next, the member is then associated with the reference point having the smallest orthogonal distance, and the niche counts φ for each reference point, defined as the number of members in S_t/F_l that is associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member in front last front F_l that is associated with the identified reference point is included in the final population. The niche count of the

identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

3 A Many-Objective Scheduling Approach for Software Containers

We describe, in this section, an overview of the proposed scheduling approach for assigning software containers to the nodes, then we explain the different adaptation steps of the computational algorithm to our problem, including the solution representation and the fitness functions.

3.1 Approach Overview

The main goal of the proposed approach is to schedule containers by considering four conflicting objectives to be optimized. Each solution generated by the evolutionary algorithm represents a possible container scheduling by assigning the containers into nodes. Figure 1 shows an overview of the proposed approach composed of three main components. The first component is a parser that automatically extracts from docker cli (e.g., command line) the initial swarm state in the cluster, including the number of nodes, the total number of containers, their images, and their distribution per node. The docker-compose file is also parsed to extract the dependencies between containers using the parser tool. The second component takes as inputs the different information collected by the parser, including the extracted dependency graph and the swarm state, to generate a new swarm state using a many-objective optimization algorithm to find a balance between the different objectives. The third component executes the best solution found by the multi/many-objective algorithm by updating the docker-compose file to specify a new placement for every container. Then, the docker-compose file is deployed again, and the Load Balancing module reallocates the containers as suggested.

In our approach, the user provides a docker-compose file as input, and then a Parser tool is used for generating a dependency graph $G = (V, E)$ where $V = \{v_1, v_2, v_3, \ldots, v_n\}$ means the set of containers or services and E is the set of calls or requests among them. The latter is written as a tuple $\{v_i, v_j\}$, where $v_i, v_j \in V$, and they are usually expressed in the docker-compose file as DEPENDS_ON or LINKS properties. Figure 2 shows an example of such conversion.

In this example, five services were converted to a graph G with five nodes and six edges. Aiming to ease the node's assignment, we assign a unique identifier (id) to every container/service and node to be used in the optimization process. Thus, let's consider 0, 1, 2, 3, and 4 as the id's for the following containers, respectively: CBEDB, CBEDBADMIN, CBEMQ, HAPROXY, CBEAPP. Then, the dependency graph generated for such example is $V = \{0, 1, 2, 3, 4\}$ and $E = \{\{1, 0\}, \{3, 1\}, \{3, 2\}, \{3, 4\}, \{4, 0\}, \{4, 2\}\}$.

The selected many-objective algorithm uses this dependency graph G and a Swarm State $P^{(t)}$ as input, where the latter means the current allocation of containers and nodes in Docker Swarm mode. Then, by taking into account the set of objectives to be optimized (details in Subsection 3.3), a new Swarm State $P^{(t+1)}$ is generated, and the docker-compose file is changed aiming to reflect the new scheduling (see Scheduler in Figure 1).

Docker Swarm service is based on a declarative model, which means that once the service runs, we are not allowed to move or replace containers when some node gets started. Thus, to bypass such limitations, we generate a new docker-compose file by changing the CONSTRAINTS property from the file. The Load Balancing module in our approach is responsible for monitoring the current state of Docker Swarm by considering several metrics such as CPU and Memory usage, network metrics, and so on. In our approach, we can define some thresholds for each of them, and once such thresholds are reached, the many-objective algorithm is automatically run to reschedule the containers. Finally, if the Swarm State is unavailable (for instance, when we are running the proposed approach for the first time), all containers are randomly assigned to nodes to compose this one.

In the following subsections, we describe the different adaptation steps of the multi/many-objective algorithms described in Section 2, including mainly the solution representation and the fitness functions.

3.2 Population Representation

An individual (or solution) in our population consists of an integer encoding, where each gene represents the node id, and the index is the container id. By using this approach, we have the advantage of the flexibility of having more than one container per node. Thus, let $S = \{s_1, s_2, s_3, \ldots, s_n\}$ be an individual with n containers. Figure 3 shows an example of an individual considering the containers available in Figure 2.

In this example, the chromosome representation uses five containers ($n = 5$) and a total of three nodes. The individual S represented in figure Figure 3 assigns containers 2 and 4 to node 1, container 3 to node 2, and containers 0 and 1 to node 3. Figure 4 shows a visual representation considering all containers and nodes (consider the dependency graph available in Figure 2).

3.3 Objective Functions

Even though different proposed scheduling techniques for containers [9], [10], [11], [12]. The proposed strategies are limited to at most two objectives, while many conflicting criteria should be taken into consideration within the container scheduling problem. We believe that our chosen objectives are extensively constructed based on preliminary research [13], [23] to obtain the optimal design that considers the most essential container attributes and scheduling limitations. We expect these functions to be valuable in future software container management efforts.

Consider $C = \{c_1, c_2, c_3, \ldots, c_n\}$ the set of all available containers, and $N = \{n_1, n_2, n_3, \ldots, n_m\}$ the set of all available nodes. The objective functions proposed in this work are described as follows:

*3.3.1 **First Objective**: Minimizing the number of selected nodes (Equation 1)*

The first objective corresponds to the number of selected nodes when rescheduling containers. Software containerization in many domains, such as smart automobiles [24], connected vehicles [25], [26], or different other domains [27], becomes critical, particularly in highly constrained environments For example, best practices [28] recommend that the load associated with one docker cluster node be lightweight to minimize congestion problems while running the applications, which explains the usage of the objective: Minimizing the number of

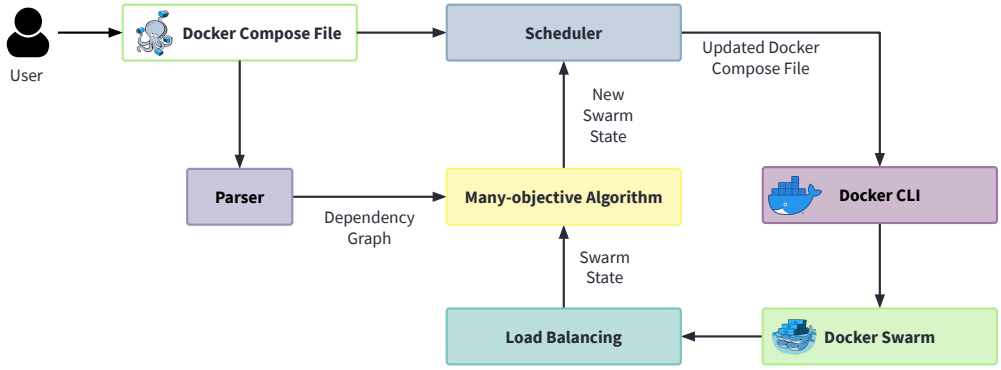


Fig. 1: Proposed Approach.

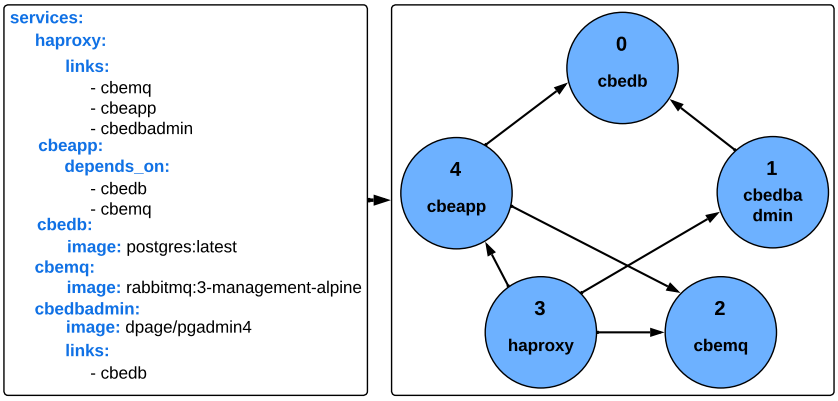


Fig. 2: Convert docker-compose file.

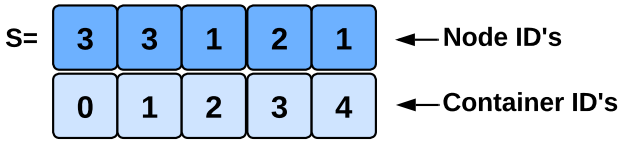


Fig. 3: Solution representation.

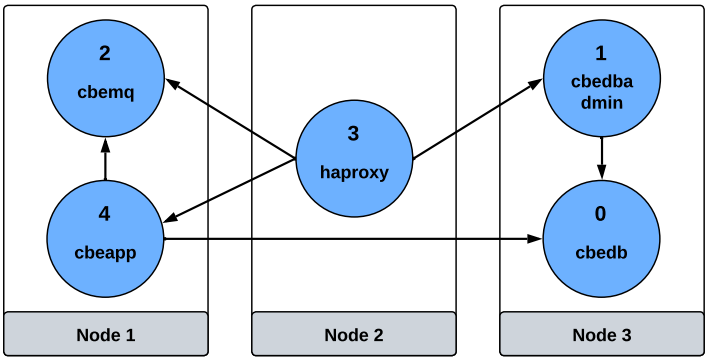


Fig. 4: Solution representation converted.

containers per node. This would avoid exceeding the resource consumption limits that might affect the behavior of the node that deploys the software.

This objective is expressed as the ratio of the number of selected nodes and the number of available ones. It is computed as follows:

$$NON(S) = \frac{|distinct(S)|}{|N|} \quad (1)$$

where $|N| > 0$ and $distinct(S)$ returns a distinct set of selected nodes from S. For example, if $S = \{2, 2, 2, 1, 2\}$, then $distinct(S) = \{1, 2\}$.

3.3.2 ***Second Objective:*** *Minimizing the average of containers per node (Equation 2)*

Minimizing the number of containers per node can avoid exceeding the resource consumption limits. However, minimizing the resource consumption of each node leads to activating a large number of nodes and spreading the load across them, on the other hand, would considerably increase the cluster's resource utilization, so adding considering minimizing the average number of containers per node in the cluster as a conflicting objective to the first is important in our context. For example, if we have 10 containers and 10 nodes, the worst solution would be to allocate every container to a single node; thus, the nodes will consume more resources. An optimal solution would be to choose a smaller number of nodes to distribute the containers while respecting the load between them and their resource limits. At the same time, we don't want to allocate all containers to one node so we can prevent violating the resource limits of the nodes and try to balance the node between the different nodes. Thus, minimizing the number of containers per node is a second objective to optimize despite conflicting with the first objective.

This objective is related to the average number of containers per node. The objective is calculated based on the normalized

standard deviation taking into account the number of containers for each node. The objective is defined as:

$$FRQ(S) = \sqrt{\frac{1}{|F|} * \sum_{f_i \in F} (f_i - \mu)^2} \tag{2}$$

where $F = \{f_1, f_2, f_3, \ldots, f_m\}$ is the number of containers for each node m divided by $|C|$ and μ is the mean of F. For example, if $S = \{2, 3, 2, 1, 2\}$, then $F = \{0.2, 0.6, 0.2\}$.

3.3.3 ***Third Objective**: Minimizing the nodes coupling (Equation 3)*

The third objective, "minimizing the coupling between containers allocated to different nodes", can be considered a security objective that helps save the data and the good performance of applications deployed in containers in case one of the nodes has been shut down for a software upgrade or operational failure: Containers sharing data or depending from each other are better to be running in the same physical ecus. (This reduces the risks of losing data or performance when for example, a container running in a different node and necessary for the work of another important container is shut down because of node failure), and also reduces the network transmission between the nodes. Although important, this objective conflicts with the objectives related to minimizing the number of containers per node.

This is expressed as a ratio between the number of inter-edges (calls or requests) in different nodes and the total number of edges E. The objective is defined as follows:

$$COP(S) = \frac{\sum_{\{a,b\} \in E} OP(S, a, b)}{|E|} \tag{3}$$

where $OP(S, a, b) = 1$ if $s_a \neq s_b$ for $s_a, s_b \in S$, otherwise, $OP(S, a, b) = 0$. If $|E| = 0$, then $COP(S) = 0$.

3.3.4 ***Fourth Objective:** Minimizing the number of changes (Equation 4)*

Finally, the fourth objective corresponds to the number of changes required to reschedule the containers. The fourth objective is considered a very important objective that previous works on container optimization did not consider. To ensure on-demand usage of the applications running in the containers, it is important to use a scheduler that is not only efficient but also fast to reallocate the containers to different nodes depending on the objectives without taking too much time that can affect the performance of the software, Thus a scheduler that does the minimal needed number of changes to balance the load between the nodes and respect the resources constraints of each of them is the one that we worked on in this project.

To this end, we compare the Swarm State $P^{(t)}$ and the solution S aiming to count the number of changes required to move a container to another node. The objective is defined as follows:

$$CHG(S) = \frac{hamming(S, P)}{|P|} \tag{4}$$

where $hamming(S, P)$ is the hamming distance between P and S and $|P| = |S|$. If $|P| = 0$, then $CHG(S) = 0$.

Therefore, the goal of our proposed approach is the following:

$$\underset{S}{\text{minimize}} \quad NON(S), FRQ(S), COP(S), CHG(S) \tag{5}$$

where all objective functions are normalized in the range [0, 1] where 0 is the best value and 1 the worst one.

To clarify how the objective functions are computed, consider $C = \{0, 1, 2, 3, 4\}$ as the set of available containers, $N = \{1, 2, 3, 4\}$ as the set of available nodes, $P^{(t)} = \{3, 3, 3, 1, 2\}$ as the current Swarm State and G as the dependency graph available in Figure 2.

Now, consider that a solution $S = \{3, 3, 1, 2, 1\}$ (the same available in Figure 3) was generated by an optimization algorithm to be evaluated, the objective functions are calculated as follows:

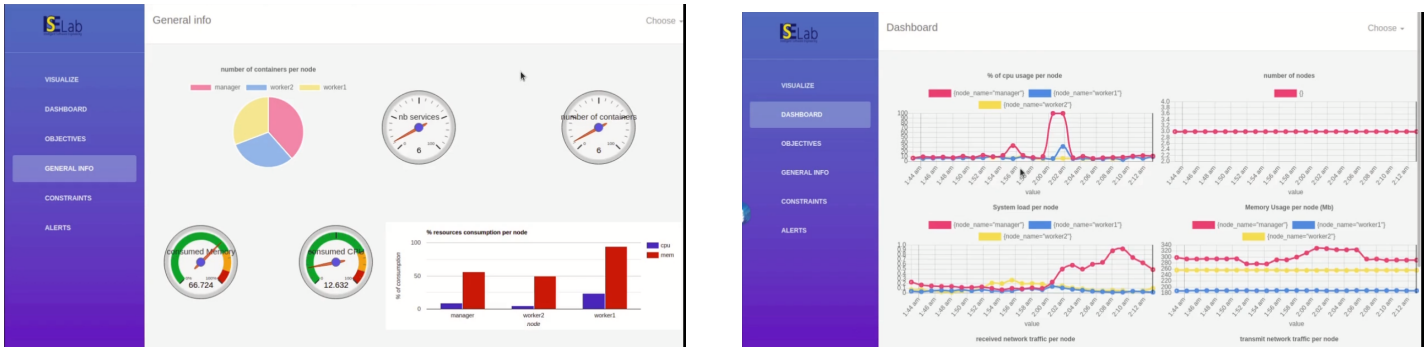
$$\begin{aligned} NON(S) &= \frac{3}{4} = 0.75 \\ FRQ(S) &= \sqrt{\frac{0.10}{3}} = 0.18 \\ COP(S) &= \frac{4}{6} = 0.66 \\ CHG(S) &= \frac{3}{5} = 0.60 \end{aligned} \tag{6}$$

Therefore, the objective values are S (0.75, 0.18, 0.66, 0.6).

3.4 Intelligent Software Containers Scheduler Framework

Based on the proposed many-objective formulation, We implemented a platform that helps the user to monitor the resource usage for every node in the cluster (CPU usage, memory usage, network I/O) and automatically rescheduled the containers using our many-objective approach. **??** shows a screenshot of our dashboard, which provides an overview of the current live status of the cluster, the number of activated nodes, the number of nodes, and the distribution per node. **??** shows real-time resource usage and the received and transmitted network per node. These metrics are extracted from the nodes using Node-exporter [29] and Cadvisor [30] to monitor the performance of each working node in the experiment, including CPU, memory usage, and network I/O. The tool also generates warnings when CPU and memory usage exceed 80%.

The candidate solutions are selected taking into account the preferences of the user interacting with the scheduler via the dashboard. Particularly, before running the scheduler, the user is asked to set the weight of each objective based on the user's preferences. These weights indicate the importance of the considered objective. For example, the user can specify that the container running the database is a high-priority container containing very sensitive data and needed for a good performance of the containers that depend on the database; other containers can have less priority. The user can manually choose to reschedule the containers if needed. Otherwise, the rescheduling is automatically based on the continuous monitoring of resource consumption metrics. Our tool periodically collects all the required data to calculate the objective functions defined by our approach and identifies the most suitable solution as detailed in Section 3. Once the rescheduling is performed, the user can view the differences in terms of resource usage and network transmission after applying our new approach. Furthermore, the user can select the option to return to the docker default scheduler if needed.



(a) The current state of the cluster.

(b) Real-time resource usage per node.

Fig. 5: Our dashboard.

4 Empirical Study

To evaluate our approach for software container scheduling using NSGA-III, we conducted a set of experiments based on 48 containers. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim of comparing our NSGA-III proposal with a variety of existing approaches. In this section, we first present our research questions and then describe and discuss the obtained results. Finally, we discuss the various threats to the validity of our experiments.

4.1 Research Questions

In our study, we assess the performance of our approach by finding out whether it could generate meaningful scheduling solutions for the software containers that improve the usage of resources. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. We define in the following the two main research questions that we are addressing:

RQ1. To what extent can the proposed NSGA-III approach provide efficient scheduling solutions based on different multi-objective (NSGA-II) and many-objective algorithms (IBEA)? This question aims to investigate the efficiency of our many-objective NSGA-III approach for container scheduling to find trade-offs between the different conflicting objectives compared to other multi/many-objective algorithms.

RQ2. To what extent can the proposed NSGA-III approach minimize the resources consumption in the cluster and balance the software workload (i.e., CPU and memory usage, the network I/O of each node) compared to the deterministic Docker Swarm's default scheduler? Since it is not sufficient to validate the outperformance of our approach compared to other search-based algorithms, this question evaluates the ability of our approach compared to the deterministic by default scheduler of the Docker Swarm in terms of the resources consumption (i.e., CPU and memory usage).

To answer **RQ1**, we considered the widely-used quality indicators in multi-objective optimization (described in Subsection 4.2) to evaluate the different search algorithms such as Hypervolume (HV), Inverted Generation Distance (IGD), Contributions (IC). These metrics validate the quality, spread, and diversity of the generated scheduling solutions on the Pareto front. Thus, we can determine which search algorithm performs better to find the best trade-offs between the conflicting scheduling objectives. Furthermore, we have also considered the execution time to compare the different algorithms since we are considering a large number of objectives. We did not compare our algorithm to random search as it is evident that the space to explore is too large, requiring an intelligent search. We have also did not compare with mono-objective search (aggregating all the objectives into one fitness function) as it is evident that the different objectives are conflicting: In a cluster where containers are connected to each other, minimizing the coupling between them will automatically increase the number of containers per node; and if we aim to decrease the number of containers per node we will automatically increase the number of selected nodes. Thus, we aimed in this research question to focus only on comparing our NSGA-III adaption and two other algorithms: IBEA and NSGA-II. We selected NSGA-II to evaluate its performance with a larger number of objectives than two, which may justify the need to use many-objective algorithms. We have also selected IBEA as it is known to be widely used in the current many-objective optimization literature after NSGA-III. We used the same adaptation for all three algorithms to enable a fair comparison.

We believe that the quality metrics results discussed in item **RQ1** would affect the results regarding resource consumption. The algorithm giving the best set of solutions will be able to give the best resource consumption compared to the other algorithms. Therefore, we only compared the default scheduler with the NSGA-III algorithm for resource consumption.

As a result, to answer **RQ2**, we compared the results from the default Docker Swarm's scheduling algorithm against the best search algorithm from item **RQ1**,

by considering ApacheBench [31] as a stress testing tool. Using ApacheBench, we set a total of 100000 requests that should be made when running each project and 100 requests concurrently (simultaneously) at a time, ensuring scalable testing settings. We also used Node-exporter [29] and Cadvisor [30] to monitor the status and performance of each working node in the experiment, including CPU and memory usage and network I/O. We used these evaluation metrics instead of the objective functions to avoid any bias when comparing the search-based and deterministic techniques. We have also created an online appendix for a demo of our platform, and related experiments material [17].

4.2 Quality Indicators

Aiming to compare the search-based algorithms, we considered the following sets of solutions [32]: i) PF_{approx}: set of non-dominated solutions obtained by one algorithm execution; ii) PF_{known}: set of non-dominated solutions of an algorithm obtained by the union of all the PF_{approx} from all the executions, removing the non-dominated and repeated solutions; and iii) PF_{true}: formed by all sets PF_{known} obtained from different algorithms by removing dominated solutions and repeated ones. The analysis was conducted by using the widely used quality indicators in the computational search field to evaluate both the quality and spread/diversity of the solutions:

- **Hypervolume** (HV) [33] measures the volume covered by members of a Pareto-front in objective space delimited by a reference point. An important feature of this metric is its ability to capture the diversity and convergence of solutions. A higher hypervolume value is desirable.
- **Inverted Generational Distance** (IGD) [34] is a convergence measure that corresponds to the average Euclidean distance between the approximate Pareto-front provided by an algorithm and the reference Pareto-front. Small values are desirable.
- **Contributions** (IC) [35] measures the proportion of solutions that lie on the reference front (RS) [36]. The higher this proportion, the better the quality of solutions.

4.3 Studied Docker Projects

In the experiments, we selected 48 Docker-based projects available on GitHub. Table 3 provides some descriptive statistics about all of them, such as the number of stars, contributors, services, and containers. We selected these projects for our validation because they range from medium to large-sized open-source projects, which were actively developed over the past 10 years, they are widely used. They are based on several programming languages. Regarding the number of containers, the figure shows that the smallest project has two containers (e.g., RAMMYGIT/MEWBASE), and the largest one has eleven containers to be scheduled (e.g., MARINANIEROD/-DOCKER_PRESTASHOP). Furthermore, the list of projects contains containers coded in several programming languages such as JavaScript, Python, Ruby, PHP, etc.

4.4 Parameter Settings

Parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each multi/many-objective algorithm and for each project, we perform a set of experiments using several population sizes [37], [38], [39]. Each algorithm is executed 30 times with each configuration, and then the comparison between the configurations is done based on IGD using the Wilcoxon test. In order to have significant results for each couple (algorithm, project), we use the trial and error method to obtain a good parameter configuration. Since we are comparing different search algorithms, we classify parameters into common parameters and specific parameters. If the results are similar for a given combination of parameters, the execution time was considered. As evolutionary operators, we adopted Integer SBX crossover, Integer Polynomial mutation, and binary tournament for selecting the individuals [40] because they have been designed to work with integer solutions. Therefore, the list of selected parameters used to answer the stated research questions is described in Table 4.

TABLE 3: Docker-based projects studied.

#	Name	Size (Kb)	Star	Contrib.	# of Serv.	# of Cont.
1	vegasbrianc/ prometheus	3133	2.9k	33	5	5
2	Zappelphilipp/docker-graylog-kibana-nginx-stack	23	2	1	5	5
3	joelanman/e-petitions	7.9k	0	38	7	7
4	Semprini/cbe-utilities	64	1	2	5	5
5	pjuu/pjuu	5.1k	55	7	5	5
6	blacktop/docker-bro	79.7k	124	3	3	3
7	miso-lims/miso-lims	129.3k	127	14	4	4
8	Screenly/screenly-ose	19.2k	1.1k	50	5	5
9	hamburml/docker-flow-letsencrypt	80	94	8	3	4
10	LaVestima/hanna-agency	5.2k	0	2	6	6
11	ucan-lab/docker-laravel	400	360	7	3	3
12	FedorSelitsky/ even-track	1.5k	5	3	5	5
13	MassDistributionMedia/ rc-ca-blinds	43.4k	2	113	3	3
14	stencila/hub	28.1k	23	13	11	11
15	dockerwest/compose-magento	77	2	3	6	6
16	dotnet-architecture/ eShopModernizing	133.6k	706	11	4	4
17	sameersbn/docker-gitlab	5.9k	6.6k	167	4	4
18	hcxp/hcxp	576	5	1	6	6
19	changami/froghouse-lightning-talk-docker	10k	0	1	3	3
20	Hygieia/Hygieia	75.6k	3.5k	140	4	4
21	bartTC/dpaste	1.9k	304	22	3	3
22	znly/docker-druid	13	25	2	7	7
23	benjefferies/gogo-garage-opener	16.1k	35	3	3	3
24	Merrick28/delain	48.5k	8	7	2	2
25	Djacket/djacket	352	60	1	3	3
26	p6spy/p6spy	59.6k	1.3k	27	5	5
27	memodir/cv	2.7k	0	2	3	3
28	marinanierod/ docker_prestashop	79	2	4	11	11
29	jfrog/artifactory-docker-examples	2.7k	291	30	2	2
30	zentby/dockerspace	22	0	2	3	3
31	zengoma/docker-magento2-apache-dev	25	1	1	2	2
32	envato/double_entry	722	267	19	3	3
33	camd67/moebot	1.2k	1	3	3	3
34	eclectic-boy/rhodonea_mapper	62	0	2	2	2
35	shin-/sakuya-blog	49	2	1	2	2
36	zebresel-com/ mongodm	73	179	3	2	2
37	josephspurrier/ gowebapi	648	55	1	2	2
38	RailsEventStore/ rails_event_store	8.3k	923	58	4	4
39	labpositiva/ pywork-place	5k	1	2	2	2
40	busino/ dj_farm_example	30	0	1	3	3
41	azerothcore/ azerothcore-wotlk	359.6k	1.2k	152	3	3
42	phillc/ rosterbater	503	3	1	4	4
43	vipulasa/laravel-meetup-v.2.0	201	1	1	2	2
44	tootsuite/ mastodon	115.2k	23.5k	416	4	4
45	rodrigogs/ github-metrics	295	6	2	2	2
46	netresearch/ timetracker	42.7k	8	5	3	3
47	chiefy/ terraform-linode-nextcloud	29	0	1	9	9
48	rammygit/mewbase	500	0	5	2	2

TABLE 4: Parameter Settings.

Parameter	NSGA-III	NSGA-II	IBEA
Population Size	200	200	100
Maximum Number of Generations	2500	2500	2500
Crossover Probability	0.95	0.9	0.9
Mutation Probability	0.05	0.01	0.01
Crossover Operator	Integer SBX		
Mutation Operator	Integer Polynomial		

As we have also measured the execution time, the algorithms were executed in a machine with an Intel(R), Core(TM) i7-5930K, CPU 3.50GHz with 40Gb RAM.

4.5 Statistical Tests

Since meta-heuristic algorithms are stochastic ones, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance. The obtained results are statistically analyzed using the Wilcoxon rank-sum test [41] with a 99% confidence level (α=1%). The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value less than or equal to α (<=0.01) means that we accept H1 and reject H0. However, a p-value that is strictly greater than α (>0.01) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing NSGA-II and IBEA search results with NSGA-III ones. This way, we determine whether the performance difference between NSGA-III and one of the other approaches is statistically significant or just a random result.

4.6 Results

4.6.1 Results for RQ1

Table 5 summarizes the results of mean values and standard deviations for HV, IGD, and IC indicators over 30 independent simulation run where the bold values represent the best ones. The results of Table 5 are based on the consideration of all 4 objectives for the evolutionary algorithms. The objectives values were normalized between 0 and 1 and set to be minimized; the order of the objectives is not important and has no impact on the results. The users can select the best solution based on their preferences (fitness function values) and programming behavior from the non-dominated (trade-off) set of solutions. All the results were statistically significant on the 30 independent simulations using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

When comparing NSGA-III against NSGA-II and IBEA using all three performance indicators, it is clear that NSGA-II has the weakest performance. On small-scale docker projects including up to 4 containers (e.g., docker-bro, miso-lims, docker-flow-letsencrypt, docker-laravel, rc-ca-blinds) all algorithms present similar results for IGD, HV, and IC. For example, for the rc-ca-blinds, both algorithms give the best results in terms of the three quality metrics (0 for the IGD, 0.065 for the HV, and 1 for the IC). On medium-scale docker projects with up to 7 containers (e.g., e-petitions, compose-magento, hcxp/hcxp), NSGA-III and IBEA present similar results, and both provide better results than NSGA-II. For hcxp/hcxp project, NSGA-III and IBEA output as results 0, 0.078, and 1 for the IGD, HV, and IC metrics compared to 0.056, 0.714, and 0.833 for NSGA-II respectively with the same metrics. Furthermore, the project compose-magento shows that both many objectives algorithms output 0 for the IGD, 0.078 for the HV, and 1 for the IC compared to 0.056, 0.071, and 0.833, respectively, when NSGA-II is used. For large-scale docker projects, NSGA-III is significantly better than NSGA-II and IBEA on most projects with a large number of containers (e.g., hanna-agency, Terraform-linode-nextcloud, docker-prestashop, and stencila/hub). Considering the example of the project "znly/docker-druid", NSGA-III outperformed both other algorithms by providing as results 0, 0.146, and 1 for IGD, HV, and IC compared to 0.004, 0.144, 0.978, and 0.103, 0.135 AND 0.806 respectively for both IBEA, NSGAII. This outcome is consistent with existing studies in other domains where NSGA-II is not able to handle more than 2-3 objectives.

For most of the test results, IBEA evaluation was consistent with the NSGAII algorithm and presented similar results, whereas, for the other docker projects, NSGA-III outperformed both algorithms. This could be explained by the interaction between (1) Pareto dominance-based selection and (2) reference point-based selection, which is the distinguishing feature of NSGA-III compared to other existing many-objective algorithms. For a better comparison between NSGA-III, and IBEA, since they showed similar results in different projects, we studied the execution time of all many/multi-objective algorithms used in our experiments. The execution time is critical when using evolutionary algorithms. This metric is important to compare the algorithms regarding the spread of identifying scheduling solutions. It is important to give not only efficient scheduling of the containers but also a fast and smooth reallocation required for normal behavior of the applications deployed in the containers when rescheduling.

Figure 6 shows the average running times of the different algorithms, over 30 runs, on the different projects used in our experiments. It is clear from Figure 6 that NSGA-III is the fastest, on average, compared to NSGAII and IBEA.

For hcxp/hcxp project, NSGA-III output ran in 122 seconds compared to 145 seconds for 145 and 133 respectively, for IBEA and NSGA-II. Furthermore, the project compose-magento shows the out-performance of NSGA-II with 107 seconds compared to 145 seconds for 152 and 143, respectively, for IBEA and NSGA-II. Also, NSGA-III is significantly better than NSGA-II and IBEA on most projects with a large number of containers (e.g., hanna-agency, Terraform-linode-nextcloud, docker-prestashop, and stencila/hub). Considering the example of the project "znly/docker-druid", NSGA-III outperformed both other algorithms by providing results 92 seconds compared to 126 and 132 seconds, respectively, for both IBEA and NSGAII.

This observation could be explained by the computational effort required to compute each solution's contribution (IGD) when using IBEA. Furthermore, NSGA-II may take longer time to find relevant solutions than many-objective algorithms due

TABLE 5: HV, IGD, and IC mean values with NSGA-III, NSGA-II, and IBEA. The results were statistically significant on 30 independent simulation runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%)

Docker Project	NSGAIII			IBEA			NSGAII		
	IGD	HV	IC	IGD	HV	IC	IGD	HV	IC
terraform-linode-nextcloud	**0.028 ± 0.013**	0.123 ± 0.013	**0.942 ± 0.013**	0.051 ± 0.010	**0.124 ± 0.010**	0.858 ± 0.010	0.089 ± 0.013	0.123 ± 0.013	0.725 ± 0.013
docker-grocery	**0.007 ± 0.000**	**0.150 ± 0.000**	**0.987 ± 0.000**	0.013 ± 0.000	**0.150 ± 0.000**	0.980 ± 0.000	0.141 ± 0.000	0.150 ± 0.000	0.800 ± 0.000
cbe-utilities	**0.000 ± 0.000**	**0.063 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.063 ± 0.000**	**1.000 ± 0.000**	0.118 ± 0.000	0.031 ± 0.000	0.500 ± 0.000
docker-bro	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
miso-lims	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.125 ± 0.000**	**1.000 ± 0.000**
screenly-ose	**0.000 ± 0.000**	**0.111 ± 0.000**	**1.000 ± 0.000**	0.008 ± 0.000	**0.111 ± 0.000**	0.987 ± 0.000	0.128 ± 0.000	0.111 ± 0.000	0.793 ± 0.000
pju	**0.000 ± 0.000**	**0.089 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.089 ± 0.000**	**1.000 ± 0.000**	0.098 ± 0.000	0.044 ± 0.000	0.625 ± 0.000
docker-flow-letsencrypt	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
hanna-agency	**0.000 ± 0.000**	**0.150 ± 0.000**	**0.987 ± 0.000**	0.013 ± 0.000	0.150 ± 0.000	0.980 ± 0.000	0.141 ± 0.000	0.150 ± 0.000	0.800 ± 0.000
stencila/hub	**0.023 ± 0.001**	**0.144 ± 0.001**	**0.644 ± 0.001**	**0.021 ± 0.007**	0.138 ± 0.007	0.612 ± 0.007	0.054 ± 0.020	0.108 ± 0.020	0.355 ± 0.020
eventrack	**0.000 ± 0.000**	**0.074 ± 0.000**	**1.000 ± 0.000**	0.002 ± 0.000	**0.074 ± 0.000**	0.993 ± 0.000	0.176 ± 0.000	**0.074 ± 0.000**	0.600 ± 0.000
docker-laravel	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
rc-ca-blinds	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
compose-magento	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
eShopModernizing	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
docker-gitlab	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
hcxp/hcxp	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
froghouse-lightning-talk	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
Hygieia/Hygieia	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
bartTC/dpaste	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
znly/docker-druid	**0.000 ± 0.000**	**0.146 ± 0.000**	**1.000 ± 0.000**	0.004 ± 0.004	0.144 ± 0.004	0.978 ± 0.004	0.103 ± 0.024	0.135 ± 0.024	0.806 ± 0.024
gogo-garage-opener	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
delain	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
Djacket/djacket	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
p6spy/p6spy	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
memodir/cv	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
docker_prestashop	**0.091 ± 0.000**	**0.210 ± 0.000**	**0.696 ± 0.000**	**0.084 ± 0.000**	**0.210 ± 0.000**	0.654 ± 0.000	0.108 ± 0.000	0.210 ± 0.000	0.583 ± 0.000
artifactory-docker	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
magento2-apache-dev	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
double_entry	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
camd67/moebot	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
rhodonea_mapper	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.005 ± 0.000	0.750 ± 0.000
dockerspace	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
sakuya-blog	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
mongodm	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
gowebapi	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
rails_event_store	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
pyworkplace	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
azerothcore-wotlk	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**
dj_farm_example	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
rosterbater	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
laravel-meetup-v.2.0	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
mastodon	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
timetracker	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.083 ± 0.000**	**1.000 ± 0.000**	0.228 ± 0.000	0.000 ± 0.000	0.750 ± 0.000
graylog-kibana	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.100 ± 0.000	**0.078 ± 0.000**	0.714 ± 0.000
e-petitions	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.095 ± 0.000	**0.078 ± 0.000**	0.714 ± 0.000
vegasbrianc/prometheus	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.078 ± 0.000**	**1.000 ± 0.000**	0.056 ± 0.000	0.714 ± 0.000	0.833 ± 0.000
github-metrics	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**	**0.000 ± 0.000**	**0.065 ± 0.000**	**1.000 ± 0.000**

to the limited spread of the solutions in the Pareto front when using more than 3 objectives. We note that the experiments were conducted on a single machine (i7 – 2.70 GHz, 8.0 GB – DDR3, SSD - 520MB/s); thus, the different algorithms will run faster on better hardware configurations.

Key findings: NSGA-III outperforms the different search algorithms based on NSGA-II and IBEA regarding the quality and spread of identified scheduling solutions in the Pareto front.

4.6.2 Results for RQ2

In this research question, we compare the NSGA-III results against the Docker Swarm's default scheduler on 30 independent runs.

We used the first three objectives NON, FRQ, and COP. CHG is not considered in this experiment because CHG corresponds to the number of changes required to reschedule the containers from a swarm state P generated by the default scheduler to a new swarm state generated by our tool. We believe that it is important to consider only the required changes when moving containers between nodes. Thus, we find the best and fastest solution that reallocates the containers for better resource usage while keeping the normal behavior of the applications deployed in the containers. However, we cannot compare the docker default scheduler with our new scheduler using this metric since we are calculating these changes when moving from the default state presented by the docker.

The default Docker Swarm's scheduler is based on a deterministic adhoc approach based on filters and strategies. Filters are used to narrow the domain of nodes for scheduling by taking the node and container properties as inputs, among other parameters. Strategies are used to decide on which node the next container runs using three alternatives: random, spread, and binpack.

As described in Table 6, our proposed approach provides significant improvements in terms of the number of selected nodes (NON), the average number of containers per node (FRQ), and the node's coupling values (COP) compared to Docker Swarm's default scheduler. This is an interesting result confirming that NSGA-III can find very good compromises between the different conflicting objectives and outperform

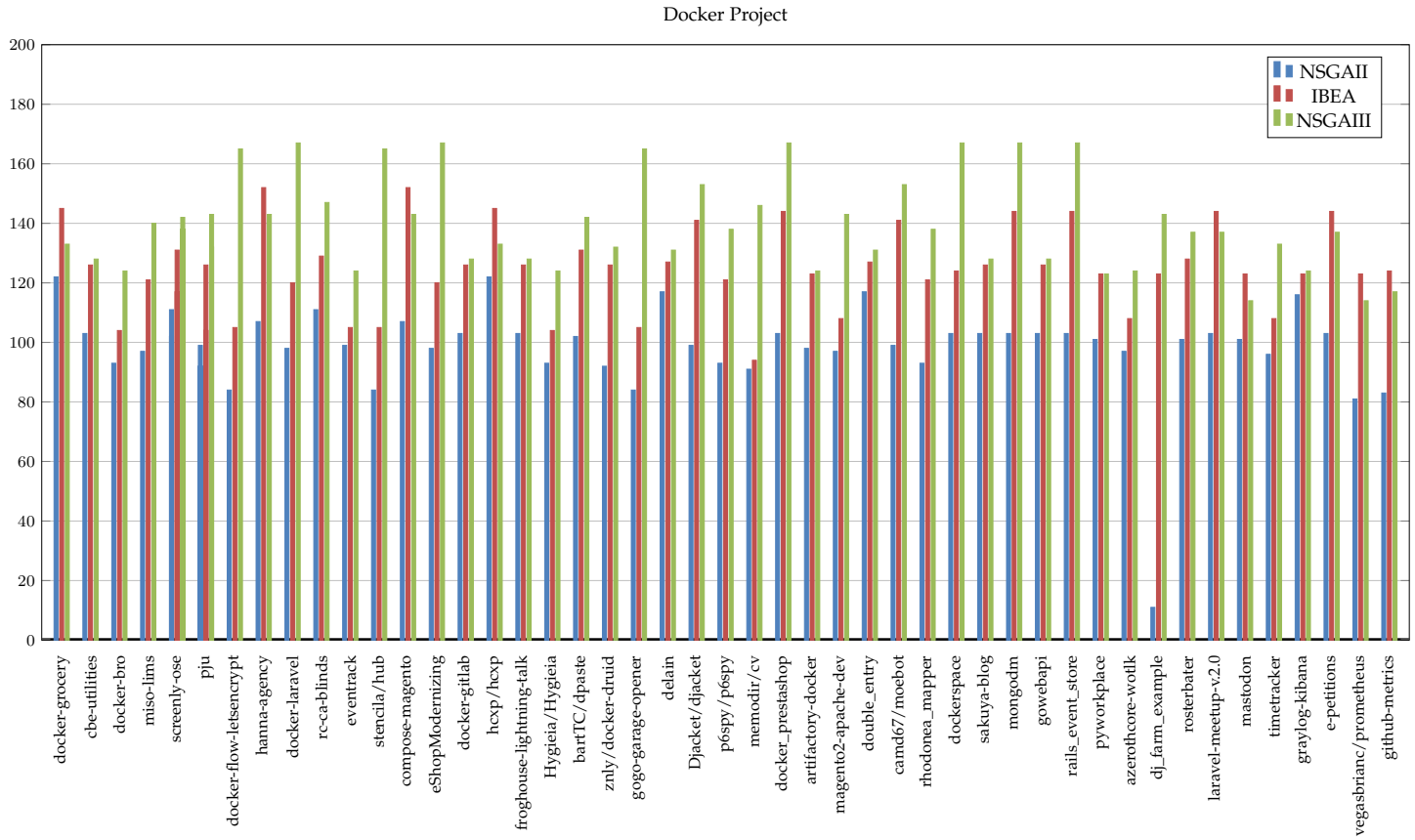


Fig. 6: Average Computational time values on 30 independent runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level ($\alpha < 1\%$).

those produced by the Docker default scheduler. In some cases, applying NSGA-III solutions give slightly higher values for the average number of containers per node (e.g., docker-grocer, cbe-utilities, screenly-ose, pju and examples shown in Table 6). This can be explained by the fact that decreasing the number of depending containers allocated on different nodes and decreasing the number of selected nodes will automatically increase the number of containers per node. Overall, the NSGA-III algorithm was able to find a good trade-off between all four objectives since most of them were significantly decreased comparing the initial state of the cluster before rescheduling.

Since comparing the performance of NSGA-III with the default scheduler using similar evaluation metrics to the fitness functions is not sufficient, we considered three evaluation metrics in terms of CPU usage, memory usage, and network transmission. For this purpose, we selected ten projects with different sizes and numbers of containers in a cluster composed of 3 nodes. The results described in Table 7 provide for every project the % of CPU usage, memory usage (average of the resources consumption for the three nodes), and the network usage (received and transmitted values in bytes per second) in the cluster using NSGA-III versus the default Docker swarm scheduler. Table 8 describes also in more details the % of resource consumption per node.

We found interesting results as described in Table 7, including a decrease in all resource usage for almost all projects using our approach compared to the Docker Swarm scheduler. Considering the example of project "cbe-utilities", the percentage of CPU usage for the cluster decreased from 16.89% to 13.90%, and the decrease in the memory usage exceeded 6% (from 40.3% to 33.90%) after applying our new many-objective approach. Further details in Table 8 about the results per node are described. We notice that the CPU usage for both node 2 and node 3 decreased from 14.860% and 15.130% to 11.5% and 5.650%, respectively, with a little increase within node 1 which can be explained by an increase in the number of containers in node1. Regarding memory usage, we notice a decrease for node 3 to half (from 40.5% to 20.210%) while keeping approximately the same memory usage for both other nodes. Also, in other projects such as docker-flow-letsencrypt, e-petitions, and docker-laravel, we notice a decrease of approximately 3% in the CPU usage for the hole cluster when using our approach and a memory decrease that exceeds 2%, 2%, and 7% respectively.

The results for each node for these projects detailed in Table 8 are promising. Continuing with the example docker-flow-letsencrypt, the CPU usage for both node1 and node2 was decreased from 15.620% and 19.520% to 13.140% and 10,260%, respectively, and the memory usage for node 1 was decreased from 55.23%51.850 to 55.230%. Also, for e-petition, we observe a decrease that exceeded 9% for node 2 in terms of CPU (from 11.930% to 2.7% and from 4.6 to 2.070% for node 3) and approximately the same average of memory consumption for both nodes. The balance in our approach between decreasing the number of nodes and the average of containers per node succeeded in distributing equally the containers achieving a balanced load between nodes that leads to reduced resource usage and more efficient utilization of these resources in the whole cluster. For a few projects (e.g., graylog-kibana, miso-

TABLE 6: Number of selected nodes (NON), Average number of containers per nodes (FRQ), number of coupling (COP) mean values of NSGA-III over 30 independent simulation runs. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α < 1%).

Docker Projects	Our Approach			Default Scheduler		
	NON(S)	FRQ(S)	COP(S)	NON(S)	FRQ(S)	COP(S)
terraform-linode-nextcloud	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
docker-grocery	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
cbe-utilities	0.66666	0.09999	0.33333	1.00000	0.09428	0.83333
docker-bro	0.66666	0.09999	0.80000	1.00000	0.09428	1.00000
miso-lims	0.33333	0.00000	0.00000	1.00000	0.00000	1.00000
screenly-ose	0.66666	0.00000	0.33333	1.00000	0.11785	0.66666
pju	1.00000	0.13608	0.71428	1.00000	0.00000	0.85714
docker-flow-letsencrypt	1.00000	0.09428	0.60000	1.00000	0.09428	1.00000
hanna-agency	0.66666	0.16666	0.00000	1.00000	0.00000	0.00000
docker-laravel	1.00000	0.06734	0.25000	1.00000	0.06734	0.50000
rc-ca-blinds	0.66666	0.16666	0.00000	1.00000	0.00000	0.00000
eventrack	0.33333	0.00000	0.00000	1.00000	0.00000	1.00000
stencila/hub	0.66666	0.09999	0.40000	1.00000	0.09428	0.60000
compose-magento	1.00000	0.04285	0.46153	1.00000	0.08570	0.53846
eShopModernizing	0.66666	0.07142	0.00000	1.00000	0.06734	0.00000
docker-gitlab	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
hcxp/hcxp	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
froghouse-lightning-talk	1.00000	0.18856	0.00000	1.00000	0.09428	0.50000
Hygieia/Hygieia	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
bartTC/dpaste	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
znly/docker-druid	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
gogo-garage-opener	0.66666	0.00000	0.00000	1.00000	0.00000	0.00000
delain	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
Djacket/djacket	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
p6spy/p6spy	0.33333	0.00000	0.00000	1.00000	0.11785	0.00000
memodir/cv	1.00000	0.09428	0.00000	1.00000	0.09428	0.00000
docker_prestashop	1.00000	0.11785	0.00000	1.00000	0.11785	0.00000
artifactory-docker	1.00000	0.08570	0.10000	1.00000	0.04285	0.30000
magento2-apache-dev	0.33333	0.00000	0.00000	1.00000	0.11785	1.00000
double_entry	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
camd67/moebot	0.66666	0.00000	0.00000	1.00000	0.11785	0.50000
rhodonea_mapper	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
dockerspace	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
sakuya-blog	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
mongodm	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
gowebapi	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
rails_event_store	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
pyworkplace	0.33333	0.00000	0.00000	0.66666	0.00000	0.00000
azerothcore-wotlk	0.66666	0.00000	0.66666	1.00000	0.11785	0.66666
dj_farm_example	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
rosterbater	0.66666	0.04545	0.00000	1.00000	0.04285	0.00000
laravel-meetup-v.2.0	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
mastodon	0.66666	0.00000	0.00000	1.00000	0.11785	0.00000
timetracker	1.00000	0.11785	0.00000	1.00000	0.11785	1.00000
graylog-kibana	0.33333	0.00000	0.00000	1.00000	0.09428	1.00000
e-petitions	0.33333	0.00000	0.00000	1.00000	0.09428	1.00000
vegasbrianc/prometheus	0.66666	0.25000	0.00000	1.00000	0.11785	0.50000
github-metrics	1.00000	0.00000	0.33333	1.00000	0.00000	0.50000

TABLE 7: Average of CPU, Memory, and Network usages comparing our approach and Spread in the cluster The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

Docker Project	CPU Usage		Memory Usage		Network Usage (bytes/s)			
					Received		Transmitted	
	Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread
graylog-kibana	**10.15%**	10.60%	**43.36%**	50.09%	**1164.92**	1603.96	**1161.17**	1652.10
e-petitions	**6.88%**	9.22%	**47.63%**	48.35%	**846.70**	1294.86	**1160.09**	1340.18
cbe-utilities	**13.90%**	16.89%	**33.90%**	40.30%	**1179.04**	1539.35	**1117.37**	1661.11
docker-bro	**14.97%**	17.65%	**44.31%**	46.63%	**1172.26**	1610.29	**1162.84**	1415.64
miso-lims	**2.34%**	2.38%	**15.36%**	20.60%	**521.93**	940.86	**102.54**	340.15
screenly-ose	**6.77%**	7.68%	**29.39%**	29.79%	**1099.01**	1505.93	**1334.49**	1429.68
pju	**20.18%**	20.78%	**36.84%**	42.53%	**719.01**	1459.63	**895.31**	1424.31
docker-flow-letsencrypt	**14.97%**	17.65%	**44.31%**	46.63%	**664.90**	1006.96	**720.01**	868.74
hanna-agency	**2.31%**	2.38%	**15.36%**	20.69%	**1319.59**	2158.03	**1386.06**	1975.92
docker-laravel	**20.60%**	23.65%	**45.63%**	52.50%	**664.90**	1006.96	**720.01**	868.74

lims, hanna-agency), the results show unremarkable differences between the two approaches in terms of CPU. These results can be explained by the low number of containers used in such projects. Thus, we obtained a very similar allocation of nodes for both approaches that do not affect resource usage by keeping almost the same number of containers per node.

We have noticed a significant decrease in network transmission for all projects using our approach compared to the Docker swarm scheduler. For instance, the project "cbe-utilities" has an average of received and transmitted network in the cluster that decreased from 1539.350 and 1661.106 bytes per second while the docker strategy provided values of 1179.043 and 1117.367 (bytes per second). Table 8 summarizes the results for each node separately. For the project graylog-kibana, for example, the received network was decreased from 2632.610, 1228.150, and 960.120 for node 1, node 2, and node 3 using the default scheduler to 1537.520, 997.570, 959.680 respectively using our approach. The Transmitted Network was decreased

TABLE 8: CPU, Memory, and Network usages comparing our approach and Spread for every node. The results were statistically significant on 30 independent runs using the Wilcoxon rank sum test with a 99% confidence level (α<1%).

Docker Project	Nodes	CPU Usage		Memory Usage		Network Usage (bytes/s)			
						Received		Transmitted	
		Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread	Our Appr.	Spread
graylog-kibana	1	19.700%	15.140%	53.390%	51.090%	1537.520	2632.610	3366.580	4830.660
	2	5.700%	11.930%	40.850%	40.500%	997.570	1228.150	59.120	60.610
	3	5.070%	4.600%	35.830%	40.500%	959.680	960.120	57.820	65.050
e-petitions	1	15.890%	11.140%	62.090%	62.390%	1542.520	1752.320	3366.580	3892.660
	2	2.700%	11.930%	40.500%	41.850%	997.570	1260.150	59.850	62.420
	3	2.070%	4.600%	40.300%	40.830%	658.680	872.120	53.840	65.444
cbe-utilities	1	24.560%	20.660%	41.560%	40.470%	1609.970	1832.780	3233.000	4830.660
	2	11.500%	14.860%	40.120%	40.500%	966.870	1825.150	58.660	91.610
	3	5.650%	15.130%	20.210%	40.621%	960.290	960.120	60.442	61.050
docker-bro	1	21.520%	17.820%	51.850%	55.230%	1456.520	2242.61	3366.580	4120.66
	2	13.140%	15.620%	40.500%	41.850%	1100.570	1628.15	59.120	63.610
	3	10.260%	19.520%	40.600%	42.830%	959.680	960.12	62.820	62.650
miso-lims	1	5.454%	2.200%	25.090%	21.390%	809.720	1100.320	210.970	892.660
	2	0.256%	2.500%	11.500%	19.850%	97.410	960.150	35.450	62.780
	3	1.233%	2.450%	9.500%	20.830%	658.68	612.250	62.200	65.000
screenly-ose	1	11.500%	11.620%	38.090%	38.500%	1409.970	1632.520	3830.660	4125.650
	2	10.700%	5.820%	37.000%	36.850%	1266.780	1925.150	99.800	75.125
	3	1.120%	5.600%	5.300%	12.830%	620.290	960.120	73.000	88.254
pju	1	24.820%	21.337%	41.120%	40.470%	1229.520	1856.610	2563.000	4120.660
	2	20.320%	21.023%	25.200%	42.500%	1100.570	1562.150	60.120	89.610
	3	17.200%	20.130%	44.200%	44.620%	450.680	960.120	62.820	62.650
docker-flow-letsencrypt	1	21.520%	17.820%	51.850%	55.230%	537.320	1132.610	2100.100	2530.560
	2	13.140%	15.620%	40.500%	41.850%	937.700	1228.150	32.120	40.610
	3	10.260%	19.520%	40.600%	42.830%	459.680	660.120	27.820	35.050
hanna-agency	1	5.454%	2.200%	25.090%	21.390%	2001.520	3500.810	3366.580	4730.660
	2	0.256%	2.500%	11.500%	19.850%	997.570	1428.150	478.120	650.610
	3	1.233%	2.450%	9.500%	20.830%	959.680	1545.120	313.487	546.500
docker-laravel	1	26.560%	25.230%	66.360%	62.500%	537.320	1132.610	2100.100	2530.560
	2	12.450%	24.180%	25.250%	47.000%	937.700	1228.150	32.120	40.610
	3	22.780%	21.550%	45.300%	48.000%	459.680	660.120	27.820	35.050

from 4830.660, 60.610, and 65.050 to 4830.660, 60.610, and 65.050 for each node, respectively (node 1, node 2, and node 3). The decrease in the network I/O values is explained by the fact that we are decreasing the container coupling in different nodes for better communication between nodes.

Key findings: Our new many-objective approach outperforms the docker default scheduler regarding resource usage: the CPU, memory, and network transmission in the docker cluster.

4.7 Threats to Validity

Conclusion Validity. Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Wilcoxon rank sum test with a 99% confidence level (α <1%). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the search-based software engineering community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy [42] for our approach so that parameters are updated during the execution in order to provide the best possible performance.

Construct Validity. Construct validity is concerned with the relationship between theory and what is observed. To evaluate the results of our approach, we selected solutions at the knee point when we compared our approach with fully-automated scheduling approaches, but the users may select a different scheduling solution based on their preferences to give different weights to the objectives when selecting the best solution. To mitigate this threat, we have to use the quality indicators of the Pareto fronts when comparing the different search algorithms and also the average values of the resource usage metrics.

External Validity. We selected such Docker projects because they are developed considering several programming languages and have been developed in the past 10 years. However, we cannot state that this is enough to generalize the results since the site may not reflect real-world projects. To minimize such threats, we tried to evaluate docker projects from different domains and sizes. The use of larger docker projects should be evaluated in a future experiment. Another threat is related to the generalizability of our findings.

Another threat is that we did not include existing approaches, including meta-heuristic algorithms and deep learning models, in the validation because they use assumptions different from ours. For instance, deep learning models require a very large dataset that is not available in practice. The other search algorithms use fewer objectives. Thus, they were considered as part of the multiple formulations that we proposed in our benchmark. We highlighted the lack of existing tools for container scheduling beyond Kubernetes and a few other commercial tools. None of the existing approaches provided their tools for the community, and they are very hard to replicate.

5 Related Work

We summarize, in this section, the most relevant studies to our approach, including two main categories of studies related to

1) software workload balancing and 2) search-based software engineering.

5.1 Software Workload Balancing

In this section, we focus on existing container scheduling strategies In the last few years, Docker containerization has gained widespread popularity due to its remarkable features, such as portability, high performance, agility, modularity, and scalability, which pave the way for Docker to step further to more practical usage in the industry. Despite its quick growth, Docker container scheduling is still a challenging problem, especially in optimizing the usage of available resources.

There are some container scheduling tools such as Docker Swarm developed by Docker [5], Mesos by Apache [6], and Kubernetes by Google [7]. Generally speaking, despite their efficiency, these strategies are not adequate to handle complex application scenarios to enable adaptive scheduling strategies.

It is assumed that Docker Swarm has no prior knowledge regarding the workload or the container's resource requirements. The only available scheduling strategy is Spread, which basically schedules a service task based on spreading the number of containers equally to all Docker hosts. All the extra configuration has to be performed manually [8]. Thus, it is crucial to create more sophisticated, high-level, and adaptive allocation strategies to guarantee a balanced workload among devices, the service's performance requirements, efficient communications between containers, and more efficient utilization of resources in terms of CPU and memory.

In their paper [43], Feifei Chen et al. propose a container Scheduling Method in Edge Computing. Based on the Min-Min algorithm, this approach aims to place a container on the physical machine with the most minor increase in energy consumption by the Min-Min to reduce the energy consumption of the cluster. It is an important approach but that targets only the power consumption in edge devices. Our work went on a similar path, but our target is to reduce the different resource consumption of the cluster, including CPU, memory, and network transmission, so we handle more complex objectives simultaneously.

Sureshkumar et al. [23] introduced a new scheduling algorithm based on load balancing. Their approach dynamically controls the load of each container within a certain threshold in the cluster by keeping it not too high or too low. If the container's load is too high, another container is created to balance that load, and if it is too low, the container will be closed to save energy. The approach controls the energy consumption of the entire cluster and the load balancing of the containers but does not consider the problem of allocating a container after a new node is added to the cluster. It will lead to the selection of an inappropriate node.

Our work not only considers conflicting objectives but also finds the best allocation of the containers depending on the cluster properties in a dynamic way. It considers the addition/shutdown of nodes and the regular change in the containers' resource consumption and properties. It dynamically finds the best allocation of the containers depending on the updated cluster properties.

Zhang et al. [12] proposed a solution to the scheduling problem based on a linear programming model. They designed a container task scheduling algorithm that aims to reduce the consumption of network transmission between server-side container and client, network consumption of pulling the required images from the remote repository, and energy consumption of the node itself. However, their work only simulates experiments using MATLAB without validating the container scheduling process in a not simulated cluster environment. In our work, we evaluated the approach using real-life docker containers and a cluster environment for realistic validation.

Kaewkasi et al. [9] focused on applying meta-heuristic algorithms. They adopted the Ant Colony Optimization algorithm (ACO) to implement a new container scheduler for SwarmKit, the purpose of which is to balance the use of resources so that applications in the container cluster will have better performance. A more recent work introduced by Li et al. [11] is also based on meta-heuristic algorithms. This paper proposes a Particle Swarm Optimization-based container scheduling (PSO) algorithm of the Docker platform to make the best use of each node's resources, avoid the problem of insufficient resource utilization and ensure a balanced load in the scheduling algorithm of the nodes cluster compared to the default Docker Swarm scheduler.

Liu et al. [44] proposed a new container scheduling algorithm based on multi-objective optimization, namely Multiopt. This approach aims to optimize the performance of docker containers using five key factors: the resource usage of every cluster node (CPU, Memory), the clustering of containers, the association between nodes and containers, and the time consumption transmitting images on the network.

Guerrero et al. proposed a genetic algorithm approach to implementing a container allocation strategy and elasticity management by optimizing the elasticity of the currently deployed applications and maximizing the reliability of the micro-services by avoiding single points of failure. However, the proposed strategy is limited to only two objectives, while many conflicting criteria should be considered within the container's scheduling problem. Indeed, our approach considers the available resources in terms of memory and CPU to balance the software load between multiple nodes but also aims to reduce the changes in the current configuration (e.g., moving containers between nodes), the communications between containers located in different nodes (e.g., coupling) for network and security purpose.

5.2 Search-Based Software Engineering

Search-based software engineering (SBSE) is a growing field about the design and application of computational search algorithms to address software engineering problems [45]. A comprehensive survey about existing studies can be found in [46]. Existing studies cover almost the whole software life cycle, including requirements engineering [47], software design [48], web application testing [49], software refactoring [50], etc. As discussed in the previous section, few studies addressed the problem of software container scheduling using SBSE techniques. Indeed, none of the existing studies formulated container scheduling as a many-objective problem considering different conflicting criteria. In the following, we will summarize some of the existing studies on the design and application of many-objective techniques in software engineering.

Different many-objective techniques are proposed in the literature. The first category is about objective reduction approaches. These techniques mainly look for the minimal subset of conflicting objectives. The objective reduction approach initially attempts to examine the degree of conflict among

objectives to eliminate objectives that do not construct the Pareto- front [51]. Regardless of the number of objectives, finding objective reduction opportunities in a problem has a favorable impact on search efficiency, computational cost, and decision-making. Although this technique has solved benchmark problems involving up to 20 objectives, its applicability in real-world settings is not straightforward, and it remains to be investigated since most objectives are usually in conflict with each other [52].

With increasing objectives, the Pareto optimal approximation involves investigating many Pareto-equivalent solutions. Consequently, the numerous variety of solutions makes the choice of the preferred alternative very hard for the human decision-maker (DM). More practically, DMs are not usually interested in the whole Pareto front rather than a portion of it that best fits their preferences, called the Region of Interest (ROI). The main idea is to incorporate the DM's preferences in the search space to distinguish between Pareto equivalent solutions that can evolve towards the ROI on problems involving more than 3 objectives [53]. Preference-based MOEAs have given many interesting results when addressing concrete problems in several engineering fields, including software design, by incorporating designer preferences [54].

The new preference ordering relations is an alternative approach that takes into account additional information, such as the rank of the particular solution regarding the different objectives and the related population [55] in order to overcome the inability to differentiate between solutions with the increased of the number of objectives; however, these methods do not necessarily agree with to the DMs preferences. Another category of work is the decomposition technique that decomposes the problem into several sub-problems that can be solved simultaneously by using evolutionary algorithms' parallel search capability such as MOEA/D [56].

The closest application of many-objective techniques to software engineering is a study related to software modularization [13]. In that work, the authors proposed to use coupling, cohesion, history of changes, and other structural metrics as fitness functions to guide the search toward finding relevant code restructuring actions. Although applying many-objective techniques in software engineering is not new, our study is the first to formulate the container scheduling problem as a many-objective one.

6 Conclusion

In this paper, we proposed a new dynamic workload balancing for containers that target more complex objectives than the typical default scheduler of existing Docker technologies. Our new approach aimed to achieve a balanced workload between the clusterś nodes and more efficient utilization of resources. Therefore, we considered in our approach to minimize the number of selected nodes to reduce resources consumption, minimize the average of containers per node for a balanced workload between them, taking into consideration the dependencies between containers and try to reduce the coupling (dependent containers allocated to different nodes) to minimize the network transmission and finally minimize the number of changes between the current scheduling and our approach (e.g., move container). The experiments performed on 48 docker projects provides strong evidence that our approach can significantly reduce resource consumption (CPU, memory, network I/O) compared with the default scheduler and other existing techniques.

As part of our future work, we plan to generalize our results with a more significant number of nodes and containers. We are also planning to consider more complex constraints in the scheduling based on real-world applications such as connected vehicles.

Finally, we plan to extend our study and investigate the efficiency of the proposed approach against other non-optimization-based algorithms, such as deep learning.

References

[1] G. Bhatia, A. Choudhary, and V. Gupta, "The road to docker: A survey." *International Journal of Advanced Research in Computer Science*, vol. 8, no. 8, 2017.

[2] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[3] Portworx, "2017 annual container adoption survey: Huge growth in containers," 2020, https://portworx.com/2017-container-adoption-survey/.

[4] R. Senington, B. Pataki, and X. V. Wang, "Using docker for factory system software management: Experience report," *procedia CIRp*, vol. 72, pp. 659–664, 2018.

[5] "what is docker swarm ?" 2021. [Online]. Available: https://docs.docker.com/engine/swarm/

[6] "what is mesos by apache?" 2021. [Online]. Available: http://mesos.apache.org/

[7] "what is kubernetes ?" 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[8] "Docker scheduling strategy," https://semaphoreci.com/community/tutorials/scheduling-services-on-a-docker-swarm-mode-cluster, accessed on 26/04/2021.

[9] K. Kaewkasi, C.and Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," *9th International Conference on Knowledge and Smart Technology*, no. 7886112, pp. 254–259, 2017.

[10] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, pp. 113–135, 2017.

[11] L. Li, J. Chen, and W. Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform," in *Proceedings of the 4th International Conference on Communication and Information Processing*, 2018, pp. 12–17.

[12] D. Zhang, B.-H. Yan, Z. Feng, C. Zhang, and Y.-X. Wang, "Container oriented job scheduling using linear programming model," in *Proceedings of the 3rd International Conference on Information Management (ICIM '17)*. Chengdu, China: IEEE, 2017, pp. 174–180.

[13] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–45, 2015.

[14] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, Aug 2014.

[15] H. Jain and K. Deb, "An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 602–622, Aug 2014.

[16] D. H. Phan and J. Suzuki, "R2-ibea: R2 indicator based evolutionary algorithm for multiobjective optimization," in *2013 IEEE Congress on Evolutionary Computation*. IEEE, 2013, pp. 1836–1845.

[17] Anwar Ghammam, Thiago Ferreira, Wajdi Aljedaani, Marouane Kessentini, and Ali Husain, 2023, https://anwarghammam.github.io/ContainersSchedulingWebSite/.

[18] Docker, 2020, https://docker.com.

[19] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, 2017, pp. 323–333.

[20] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code - an empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.
[21] Docker Compose, 2020, https://github.com/docker/compose.
[22] DockerHub, 2020, https://hub.docker.com.
[23] M. Sureshkumar and P. Rajesh, "Optimizing the docker container usage based on load scheduling," *2nd International Conference on Computing and Communications Technologies, ICCCT 2017; Chennai,*, no. 7972269, pp. 165–168, 2017.
[24] S. Meisenbacher, K. Schwenk, J. Galenzowski, S. Waczowicz, R. Mikut, and V. Hagenmeyer, "A lightweight user interface for smart charging of electric vehicles: A real-world application," in *2021 9th International Conference on Smart Grid and Clean Energy Technologies (ICSGCE)*, 2021, pp. 57–61.
[25] R. Morabito, R. Petrolo, V. Loscrì, N. Mitton, G. Ruggeri, and A. Molinaro, "Lightweight virtualization as enabling technology for future smart cars," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 1238–1245.
[26] S. Dabbene, C. Lehmann, C. Campolo, A. Molinaro, and F. H. P. Fitzek, "A mec-assisted vehicle platooning control through docker containers," in *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*, 2020, pp. 1–6.
[27] T. Mikkonen, C. Pautasso, K. Systä, and A. Taivalsaari, "Cargo-cult containerization: A critical view of containers in modern software development," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 93–98.
[28] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *2017 9th international conference on knowledge and smart technology (KST)*. IEEE, 2017, pp. 254–259.
[29] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.
[30] cAdvisor, 2020, https://github.com/google/cadvisor.
[31] ApacheBench, 2020, https://httpd.apache.org/docs/2.4/programs/ab.html.
[32] E. Zitzler, M. Laumanns, and S. Bleuler, "A tutorial on evolutionary multiobjective optimization," *Metaheuristics for multiobjective optimisation*, pp. 3–37, 2004.
[33] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
[34] D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," Citeseer, Tech. Rep., 1998.
[35] F. Ferrucci, M. Harman, J. Ren, and F. Sarro, "Not going to take this anymore: multi-objective overtime planning for software engineering projects," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 462–471.
[36] H. Meunier, E.-G. Talbi, and P. Reininger, "A multiobjective genetic algorithm for radio network optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 1. IEEE, 2000, pp. 317–324.
[37] R. A. Matnei Filho and S. R. Vergilio, "A mutation and multi-objective test data generation approach for feature testing of software product lines," in *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES'15)*. Belo Horizonte, Brazil: IEEE Computer Society, Sep. 2015, pp. 21–30.
[38] A. Strickler, J. A. Prado Lima, S. R. Vergilio, and A. Pozo, "Deriving products for variability test of feature models with a hyper-heuristic approach," *Applied Soft Computing*, vol. 49, pp. 1232–1242, Dec. 2016.
[39] T. N. Ferreira, J. N. Kuk, A. Pozo, and S. R. Vergilio, "Product selection based on upper confidence bound MOEA/D-DRA for testing software product lines," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '16)*. Vancouver, Canada: IEEE, Jul. 2016, pp. 4135–4142.
[40] K. Deb, "Multi-objective optimisation using evolutionary algorithms: an introduction," in *Multi-objective evolutionary optimisation for product design and manufacturing*. Springer, 2011, pp. 3–34.
[41] B. Rosner, R. J. Glynn, and M.-L. Ting Lee, "Incorporation of clustering effects for the wilcoxon rank sum test: a large-sample approach," *Biometrics*, vol. 59, no. 4, pp. 1089–1098, 2003.
[42] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.
[43] F. Chen, X. Zhou, and C. Shi, "The container scheduling method based on the min-min in edge computing," *ACM International Conference Proceeding Series*, pp. 83—-90, 2019.
[44] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang, "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.
[45] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
[46] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.
[47] Y. Zhang, A. Finkelstein, and M. Harman, "Search based requirements optimisation: Existing work and challenges," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2008, pp. 88–94.
[48] M. Harman, P. McMinn, J. T. De Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical software engineering and verification*. Springer, 2010, pp. 1–59.
[49] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 3–12.
[50] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, pp. 1–53, 2016.
[51] H. Wang and X. Yao, "Objective reduction based on nonlinear correlation information entropy," *Soft Computing*, vol. 20, no. 6, pp. 2393–2407, 2016.
[52] D. K. Saxena, J. A. Duro, A. Tiwari, K. Deb, and Q. Zhang, "Objective reduction in many-objective optimization: Linear and nonlinear algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 77–99, 2012.
[53] J. Horn, "F1. 9 multicriterion decision making," *Handbook of evolutionary computation*, vol. 97, no. 1, 1997.
[54] M. Li, S. Yang, and X. Liu, "Diversity comparison of pareto front approximations in many-objective optimization," *IEEE Transactions on Cybernetics*, vol. 44, no. 12, pp. 2568–2584, 2014.
[55] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: A tutorial," *Reliability engineering & system safety*, vol. 91, no. 9, pp. 992–1007, 2006.
[56] D. Jia and J. Vagners, "Parallel evolutionary algorithms for uav path planning," in *AIAA 1st intelligent systems technical conference*, 2004, p. 6230.

Anwar Ghammam is currently a PhD student in the intelligent Software Engineering group at the University of Michigan. Her PhD project is concerned with the application of intelligent search and machine learning in different areas such as Software Containers, web services, refactoring, and mobile app reviews. Her current research interests are AI, web services, refactoring, data analytics, and software quality.

Thiago Ferreira is an Assistant Professor in the College of Innovation & Technology (CIT) at the University of Michigan-Flint. He received his Ph.D. Degree in Computer Science from the Federal University of Parana in 2019. His research interests focus on the use of User Preferences, Optimization Algorithms, and Artificial Intelligence techniques to address several Software Engineering problems, such as Software Requirements, Software Testing, and Software Refactoring.

Wajdi Aljedaani received a bachelor's degree in Software Engineering from the Athlone institute of technology, Ireland, in 2014 and received his master's degree in Software Engineering from Rochester Institute of Technology, New York, in 2016. He received his Ph.D. in computer science and engineering at the University of North Texas. He worked as a teaching Fellow for two years (2021-2022) at the University of North Texas then he joined the ISE Lab at Oakland University as a post-doctoral researcher. His research interests are software engineering, mining software repository, accessibility, machine learning, and NLP.

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, four best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence at the University of Michigan.

Ali Husain is a senior scientist Leader in the Research department at Ford with a passion for Artificial Intelligence and Machine Learning in Vehicles. As for his professional experience, he has more than 13 years of Embedded Software experience in the Automotive Industry. His experience spans across Advanced Driver Assistance Systems, AI, Machine Learning, Software Virtualization, Safety-Critical Software Development, and Software Architecture.