IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 April 2024, accepted 20 May 2024, date of publication 23 May 2024, date of current version 4 June 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3404643*

RESEARCH ARTICLE

# Next-Gen Cloud Efficiency: Fault-Tolerant Task Scheduling With Neighboring Reservations for Improved Resource Utilization

**SHEIKH UMAR MUSHTAQ** [1], **SOPHIYA SHEIKH** [1], **AND SHEIKH MOHAMMAD IDREES** [2]
[1]School of Computer Application, Lovely Professional University, Phagwara, Punjab 144411, India
[2]Department of Computer Science (IDI), Norwegian University of Science and Technology, 2815 Gjøvik, Norway

Corresponding author: Sheikh Mohammad Idrees (sheikh.m.idrees@ntnu.no)

This work was supported by the Ministry of Minority Affairs, India, through University Grants Commission (UGC) [Maulana Azad National Fellowship (MANF), Junior Research Fellowship (JRF)] Fellowship under Grant 210510690926 and Grant 202122-PB04002658. The open access publication costs are covered by Norwegian University of Science and Technology.

**ABSTRACT** One of the main goals in any computational system like the cloud is to effectively allocate resources proficiently for task scheduling. However, the dynamic characteristics of the cloud make it more prone to faults and failures. The flexible and responsive changes are made to redistribute virtual machines (VMs) to address these faults and failures for maintaining continuous services. However, it may inadvertently lead to uneven load distribution. Therefore, thorough attention is required to ensure carefully monitored load equilibrium following fault tolerance. Addressing all these issues simultaneously with optimized Quality of Service (QoS) parameters is a good need of time. In this paper, a novel hybrid model: the Hybrid Fault-tolerant Scheduling and Load balancing Model (HFSLM) has been proposed to optimize the makespan of the dynamically arriving tasks and efficiently utilize the available VMs. Moreover, the model also provides solutions for several crucial concerns in cloud systems including VM failure, and VM/task heterogeneity. In the consequence of a VM failure, the approach offers a Neighbouring VM as a substitute for the corresponding task to complete its execution. Furthermore, the model is escorted by a load-balancing algorithm to maintain the equilibrium of load distribution after fault handling for further optimization of the considered QoS parameters. HFSLM is evaluated by comparing it with FTHRM, MAX-MIN, MIN-MIN, and OLB on a small task scale over diverse task and machine heterogeneities and with ELISA and MELISA on an extremely large task scale. The evaluation results show that the recommended HFSLM tops the compared approaches in all the considered cases and heterogeneities.

**INDEX TERMS** Cloud computing, task allocation, fault tolerance, resource reservation, load balancing.

## I. INTRODUCTION

The field of distributed computing is rapidly advancing technology which also includes cloud computing. Applications for cloud computing include data storage, data analytics as well as IoT applications [1]. The technology of cloud computing has altered the past ways of distributing services by companies. It offers a variety of online services to registered consumers so they may avoid investing in computing equipment. Users are only required to utilize the internet to communicate their demands to the service provider for each sort of service rather than install or maintain their own systems infrastructure. In response to the user's requests, the service provider is in charge of administrating the resources/VMs needed to satisfy customer requests.

Besides, scheduling algorithms are applied by service providers to effectively schedule incoming requests/tasks and manage their computational resources. The more efficient these scheduling algorithms/strategies are, the more efficiently the resources will be allocated by the service providers. The efficient planning of scheduling ensures the service providers optimize their revenue by making the most available resources. However, real-world cloud computing resource performance is hindered by inappropriate

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

scheduling and resource allocation issues. Making scheduling more efficient and properly utilizing cloud resources is the key aim of the research in cloud computing. A scheduler should arrange tasks in a way so that task execution time will be kept to a minimum. This resource management issue is a significant factor for any distributed system [2].

Additionally, due to the geographical dispersion of the VMs in the cloud, they may enter and exit the system due to a variety of potential causes at any moment while task executions are being carried out. Moreover, any of the VMs can fail at any point in time which may result in the interruption of the corresponding processing of the task [3]. Failure is the system's condition when it is unable to achieve the desired result within the stipulated time. To handle this issue, it is necessary to assign some alternative VM to the affected task if the VM failure happens. This makes the system fault-tolerant and adaptable to faults [4].

Furthermore, fault-tolerant scheduling may lead to inappropriate load distribution because to handle the faults we need to go for the redistribution of VMs. This difficulty can be solved by efficiently balancing the task distribution among the processors. An efficient load-balancing strategy will make it possible to distribute the incoming load over all VMs uniformly thereby giving an increasing edge to the average resource utilization. Since the task scheduler in a dynamic environment has no precise knowledge of the sizes or different arrival times of the task. Therefore, the enhancement of the cloud environment by implementing efficient scheduling and load balancing is important to address. In a nutshell, a model must focus on all discussed issues and accomplish the performance objectives.

To address these issues, HFSLM is introduced as a hybrid model that employs task and VM sorting for task scheduling, a Neighbouring reservation approach for fault tolerance, and a novel task reallocation strategy for load balancing. The proposed hybrid model focuses on three issues i.e., efficient scheduling, fault tolerance, and load balancing. The model initially schedules the arriving tasks and maps them to the most suitable VM thereby focusing on the optimized makespan and efficient utilization of VMs. Moreover, the proposed model adapts the system to respond to the faults by using the neighboring-based advance reservation technique. The neighboring-based advance reservation technique is the technique where the reservation slot is estimated in advance and the neighboring VM is reserved as an alternative VM for the affected task to reserve the processing of the task till completion. In this case, the neighboring VM with the least history of the load (Ready Time) is preferred to be selected as an alternative VM. Furthermore, the model also escorts with a load-balancing strategy to make further optimizations in various QoS parameters. The proposed model was evaluated for parameters like makespan and average resource utilization by comparing it with FTHRM, MAX-MIN, MIN-MIN, and OLB on a low task scale (less than 1000 tasks). The evaluation has been done by adjusting the number of tasks and VMs, and size of tasks, and the capacity of VMs in four different heterogeneities given by Braun [5]i.e., low task-low machine heterogeneity, low task-high machine heterogeneity, high task-low machine heterogeneity, and high task-high machine heterogeneity. Besides, the proposed HFSLM was compared with ELISA and MELISA on very high task scales (greater than 10,000 tasks, the high scale is categorised because the \) and was evaluated using an average makespan, and the resource utilization was taken into consideration for minimum, average, and maximum cases. Its effectiveness is assessed across both small and large task scales through comparisons with established models like FTHRM, MAX-MIN, MIN-MIN, OLB, ELISA, and MELISA. The findings highlight the efficiency of HFSLM, as outlined in a dedicated section.

The following is a summary of the major contributions.

- *First, we set up a task scheduling framework for a cloud environment. This framework includes components such as the Application layer, Middleware, and Host/VM layer.*
- *An efficient allocation strategy has been carried out by sorting the positions of the arriving tasks and available VMs.*
- *The scheme for dynamic task allocation has been presented in the model while concurrently adding and deleting the restored and faulty VM respectively.*
- *Afterward, we utilized a novel Neighbouring-based reservation technique, which involves the reservation of Neighbouring VMs to be used in case of faults.*
- *To sustain optimizations following fault management, we combine the suggested fault-tolerant scheduling with a proficient load-balancing approach which is lacking in most scheduling algorithms.*
- *Subsequently, we explore a motivating example that mathematically demonstrates the efficacy of the proposed hybrid model in a practical working environment. Simultaneously, we frame the fault-tolerant and load-balancing issue as an optimization problem.*
- *Moreover, we evaluated the performance of the HFSLM on small and large task scales to assess its effectiveness.*
- *Throughout the evaluations, we compared the model with similar existing models across four different tasks and VM heterogeneities given in [5] i.e., HH, HL, LH, LL task, and machine heterogeneities respectively.*

The remainder of the paper is structured as follows: Section II explains the relevant prior literature in four different categories Resource allocation approaches, Fault tolerance approaches, Load balancing approaches, and Hybrid approaches with the motivation of the work. Section III discusses the intended work as the proposed model having subsections as System model, Problem formulation, and Computational Complexity of HFSLM. The illustrative example has been presented in Section V with in-depth mathematical calculations and pictorial flow. Section VI of the paper explains the model's findings and discussion in two task scales and four task and machine heterogeneities, and section VII brings everything to a close conclusion.

## II. RELATED WORK

Efficient resource allocation along with fault tolerance and load balancing is one among several major objectives in any distributed/computational environment to ensure effective resource usage of the system. The level of resource usage brought on by fault tolerance depends on the allocation method being used. By utilizing computing resources efficiently, proficient results can be obtained with an effective strategy [6]. Additionally, the current fault tolerance strategies typically compromise resource availability and system efficiency in the event of unforeseen breakdowns. The system's effectiveness is also influenced by average resource utilization and throughput. However, in any dynamic environment, the resource behavior renders the system unstable and fault-prone, which might result in the suspension of application processing and an imbalanced load. Besides, fault tolerance inclines to deliver the expected services at the right time even when systemic problems or errors are revealed. To achieve this, steps are made to identify and correct errors and reduce the hazards in the system so that the execution of the tasks continues till completion [7]. Numerous researchers have put forward solutions to these issues. However, there is still room for developing the new and most efficient techniques for better utilization of the available resources after fault tolerance. A review of the categorical literature has been done about four main categories. i.e., resource allocation, fault tolerance, load balancing, and hybrid models. Moreover, several recent task scheduling and resource allocation techniques, different fault tolerance, and load-balancing approaches are given in this portion of the paper.

### A. RESOURCE ALLOCATION APPROACHES

A multi-object strategy using the enhanced differential evolution algorithm has been developed and proposed in [8]. This technology offers a time and cost model for cloud computing. However, this model lacks dynamism i.e., variations in tasks. A vacation queuing model for scheduling was proposed in [9]but does not demonstrate the efficient use of resources. The real-time tasks are scheduled using the newly developed rolling-horizon scheduling paradigm [10]. By allocating resources, the authors have focused on showing how job scheduling and energy saving are related. This model again lacks emphasis on resource utilization. Authors in [11] have utilized the FCFS technique to schedule the tasks. However, the proposed method needs to consider the elimination of the jobs. To perform task scheduling, authors in [12] suggest a heuristic method that combines the MAHP (Modified Analytic Hierarchy Process, BAR +bandwidth aware divisible scheduling (BATS) optimization, longest expected processing time preemption (LEPT), and divide-and-conquer methods. This method allocates resources by using combined approaches of BAR + BATS methods. Q-learning and HEFT (Heterogeneous Earliest Finish Time method have been combined in [13] to create a novel job scheduling system namely QL-HEFT to shorten makespan. However, this method has not considered resource utilization. Each task is sent to the system with the lowest predicted finishing point time for that task using the MCT (Minimum Completion Time) algorithm [14] in an arbitrary sequence. Later, the MIN-MIN allocation algorithm was proposed in [5] which initially takes all the unmapped tasks as a separate set and estimates the group of minimum completion times for each unexecuted task. The task with an overall minimum completion time is then chosen and allotted to the subsequent machine. The authors also highlighted the MAX-MIN algorithm which is the same as MIN-MIN. However, to allocate the work to the relevant VM, MAX-MIN chooses the task having an overall maximum end time from the set of minimum completion times. Besides, in [15], a three-step method was proposed to minimize makespan where the precedence graph is constructed in the first step, which is then transformed into a two-machine Johnson Sequencing Problem in the second step, and in the third step, the best sequence has been determined by utilizing the Dynamic Heuristic Johnson Sequencing method. In [16], cloud computing platforms like AWS(Amazon Web Service) have been utilized to present PMW(Periodic Min-Max Algorithm) which schedules multi-robot fast computation tasks. Recently in [17], PBFS (Priority Based Fair Scheduling) was proposed which prioritizes the definite order based on EG-SJF (Earliest Gap Shortest Job First) for filling schedule gaps. Moreover, the new backfilling strategy termed SG-PBFS (Shortest Gap-Priority-Based Fair Scheduling) was developed recently in [18], for attempting to direct the cloud gaps and scheduling tasks.

### B. FAULT TOLERANCE APPROACHES

The fault-proof strategies found in the literature have been categorized into three broad categories. i.e., Reactive, Proactive, and Resilient approaches. In reactive approaches, the faults are handled once they occur. The Proactive approaches offer the process's pre-planned alternate routes for handling errors. A proactive technique involves the system being continuously monitored to assess flaws and fix them before they manifest. The Resilient approaches differ from proactive approaches in a way that they use intelligent learning to avoid faults.

In [19], the task resubmission method was used to handle the faults, and the unsuccessful task was continuously assigned and re-assigned to the same or different resource. However, this assigning and re-assigning of the unsuccessful task causes resource loss. Software rejuvenation technique is used in [20] to handle faults, the system is rebooted regularly and each time starts in a fresh state. Resilient approaches have been used for handling faults. This approach involves regularly checking the system for errors. As a result, it qualifies as an adaptive fault tolerance strategy [21].

A kind of resilient approach was used in [22]. This technique was recently employed in a cloud environment. In this technique, the assertions based on the system's response are used to manage failures. However, The Checkpointed League Championship Algorithm (CPLCA) scheduling technique was suggested in [23]as a fault-tolerance conscious task scheduling method. This approach offers task migration and fault tolerance by employing the checkpointing strategy. Besides, CPLCA lacks any load-balancing strategy. Additionally, authors of [24]addressed the issue of job scheduling in heterogeneous systems. It was suggested to use the heuristic algorithm DBSA to solve the NP-hard issue of the scheduling problem. Later, in [25], the Dynamic Clustering League Championship algorithm (DCLCA) for fault-tolerant-based scheduling was suggested to minimize the early failure of the jobs. Advance reservation technique was used in [26] for handling faults. This reserves the VM for the task for a specified amount of slots until the task is completed.

## C. LOAD BALANCING APPROACHES

The difficult research problems include how to distribute the workload across the VMs and how to shorten and maximize the makespan and resource utilization respectively. To overcome the aforementioned difficulty, several methods have been developed. In this section, we'll discuss a few recent studies on load balancing and scheduling problems in a cloud context. Infrastructure as a service (IaaS) cloud load balancing method namely HBLBA (Heuristic Based Load Balancing Algorithm) has been proposed in [27]. To select appropriate VMs for allocation and to get the most out of the available computing resources, an effective method was designed for organizing the servers depending on the quantity and size of tasks. Based on clustering and the Bayes theorem, the novel heuristic-based load-balancing method was devised in [28]. These two methods obtained the optimum clusters of hosts. However, is not focusing much on utilization. In a public cloud setting, [29] offered a load-balancing model based on cloud partitioning and game theory. The authors in [30] and [31] modified the HEFT algorithm by including load balancing in HEFT and proposed E-HEFT and LB-HEFT (Load Balancing-Heterogeneous Earliest Finish Time) respectively. The PTAL technique was put in [32] for efficient resource reallocation and load distribution to achieve better QoS results.Further in [33] the author proposed WAMLB (Weighted Active Monitoring Load-Balancing in Cloud Computing) for effective utilization of resources. However, both PTAL and WAMLB lack the dynamism of the system concerning both tasks and VMs. Furthermore, it was noted that there is a good need to enhance the load balancing model in a dynamic environment for optimal makespan and efficient utilization of resources. The whole expected completion time while executing a task is considered and OLB was proposed in [34]. The benefit of OLB is to maintain optimal load balancing for the system and to keep all hosts as active as possible, which improves efficiency. OLB suffers from a low makespan when numerous objectives are considered at once, making it unsuitable for cloud environments. ELISA [35] was modified and MELISA [36] was proposed. The literature shows that these models are most suited to large-scale systems. Both ELISA and MELISA were considering load balancing by taking resource heterogeneity and migration cost into account. Recently, LDRA (Load Distribution Based Resource Allocation) was established [37] to earn boosted resource utilization and minimum likely execution time for distributed task.

## D. HYBRID MODELS

In [38] a decision model for central load balancing was developed that may be used in cloud systems; this approach automates scheduling and lessens the need for human administrators. This model, however, falls short in identifying and handling the faults in the system. Additionally, in the event of unforeseen failures, the present fault tolerance strategies typically compromise the reliability and efficiency of the resource management in the system. However, the cloud environment is unstable and fault-prone due to the dynamic nature of its VMs, which might result in the ceasing of the task execution and imbalanced load. Additionally, even when system problems or errors are notified, making the system fault-tolerant tends to deliver the anticipated services at the right time to prevent any task from premature termination [39]. Scheduling, load balancing algorithms, and fault tolerance strategies have all been mentioned in a survey that has been put together in [40]. A similar survey has been obtained from another study, in which several fault tolerance approaches are assessed using a variety of QoS metrics, including waiting time, throughput, turnaround time, and network latency. Additionally, it demonstrates that the aforementioned techniques can help to raise the optimization of the computing infrastructure [41]. The authors of this study are motivated to perform more research on fault-tolerant task scheduling and load balancing because of this state-of-the-art. In [42], the model namely, FTHRM was proposed where a reservation was used for fault tolerance. However, the model has charted the concept of advance reservation but not implemented the advance reservation. The Proactive and Reactive Fault Tolerance Framework (PFTF) in combination with ECB (Elastic Cloud Balancer) was presented in [43], and they prevent the cloud scenario in which some nodes are idle or lightly loaded while others are significantly loaded (overloaded). Proactive Load Balancing Fault Tolerance (PLBFT) was proposed in [44] where the faults were handled proactively and simultaneously the load was balanced. For further enhancements of the availability, A blend of load balancing and fault-tolerant approaches has been suggested in [45]. A resilient approach-based strategy, called FTLB (Fault Tolerant Load Balancing), was introduced in [46], and it incorporates fault tolerance into load balancing such that the load balancing algorithm can manage failures as well.

• *Motivation*

The consideration of the dynamic character of the cloud motivated us to propose the hybrid scheduling model integrated with fault tolerance, and load balancing for cloud setup. Although there are many scheduling algorithms available in the literature, the researchers are highly attracted and conservative towards developing various scheduling, fault tolerance, and load balancing algorithms. However, it is observed that there are very few dynamic scheduling algorithms that integrate both fault tolerance and load balancing models to optimize the QoS parameters. The integration of load balancing models with fault tolerance is a peak demand of time. Because the fault tolerance mechanisms may often reorder the prior scheduling VM assignment to fit and strong VMs in the occasion of a failure or fault, leading to uneven VM reassignment. This uneven VM reassignment becomes the cause of QoS degradation even if the prior Scheduling algorithm is highly optimized. Besides, there are various demanding reasons why the integration of load balancing is important for optimal overall system performance. Some of the mounted demands are listed below:

- *Fault tolerance often necessitates redundant resources to grip failover circumstances, which can lead to resource overprovisioning and over-cost. Load balancing can be helpful in such cases as the integration of load balancing can dynamically adjust the load over VMs thereby dropping the requirement of additional capacities.*
- *Implementing fault tolerance can introduce the overheads associated with it. However, using load balancing with fault tolerance can reduce operational burdens and other complexities.*
- *Various bottlenecks and other congestion can be created on healthy VMs in fault-tolerant systems. This can impact overall system performance. This can be eased by intelligently distributing load flow across VM post to fault tolerance.*
- *Similarly, other factors should be considered in fault-tolerant systems such as augmented latency, partial scalability, suboptimal resource utilization, etc.*

Fault tolerance integrated with load balancing can help organizations overcome these limitations and create stronger, more effectual, and mountable distributed systems that can adapt to the altered loads because of fault tolerance. Therefore, we converge towards developing the dynamic scheduling model in this paper which not only handles faults but also handles the uneven VM reassignments by integrating effective load balancing constraints post to fault tolerance.

The scheduling in the proposed model has been done by initially rearranging both arriving tasks and available VMs. The newly incoming tasks and freshly installed or deleted VMs are also taken into consideration while performing the recommended scheduling. This consideration makes it the most suitable scheduling for fully dynamic computing infrastructures. Additionally, the scheduler offers efficient allocation concerning the user's needs at selected timeslots by using a reservation. Reservation is the technique where the VMs are reserved for the task till it completes its execution thereby resulting in the assurance of task completion. However, if the VMs are not reserved, they might fail permanently or stop working at any time which may result in the termination or interruption of the corresponding task. Therefore, the model delivers the system the fault tolerance that it needs to manage runtime system errors after conducting effective scheduling. Apart from fault tolerance, the model reallocates the load to reduce the imbalance caused by fault tolerance. The evaluations are conducted by assessing the proposed model with existing similar models such as MAX-MIN, MIN-MIN, OLB, FTHRM, ELISA, AND MELISA. The MAX-MIN algorithm was found optimal for resource allocation. The makespan and utilization obtained by MAX-MIN on low task heterogeneity were also found efficient. However, for high task heterogeneity, MAX-MIN was not found significant. Furthermore, the QoS parameters obtained in MIN-MIN were not optimized in varying task and machine heterogeneities. Apart from this, these are allocation algorithms and do not support any fault handling or load-balancing procedure. The most recent FTHRM model for fault tolerance was using advance reservation. However, this model did focus on uniform load distribution. Moreover, this has not migrated the tasks from the faulty VM to the reserved VM. For large sizes, ELISA and MELISA were shown to be the best tests; however, for small scales, these models were insignificant. Additionally, these models are load-balancing models and do not support any fault tolerance mechanism. After analyzing the related literature, it was determined that the models implemented so far, particularly the hybrid models, needed to be improved for better QoS parameters. This is where we were motivated to propose a neighboring-based reservation technique for fault tolerance for the real-time cloud. In this paper, we suggested the HFSLM model with effective fault tolerance and load-balancing strategies for better outcomes. Table 1 shows a comparative analysis of all the considered issues and parameters between the proposed HFSLM and related models.

## III. THE PROPOSED MODEL

The proposed model is discussed in four subsections. Initially, the System Model explains the System architecture of the proposed model. The Problem Formulation provides the mathematical explanation for the proposed HFSLM. The Proposed Algorithm and Pseudocode present the HFSLM in a semi-formal form as an algorithm and pseudocode. Later, in the Motivational Illustrative Example, the proposed model's operation is demonstrated as an example. Furthermore, the following list contains the notions that were utilized in the illustration and demonstration.

**TABLE 1.** **Comparative analysis of existing models and the proposed model.**

| Technique/Parameters | FTHRM | MAX-MIN | MIN-MIN | OLB | ELISA | MELISA | Proposed HFSLM |
|---|---|---|---|---|---|---|---|
| Task scheduling | √ | √ | √ | √ | √ | √ | √ |
| Fault tolerance | √ | × | × | × | × | × | √ |
| Load balancing | × | × | × | × | √ | √ | √ |
| Task/VM Heterogeinity | × | × | × | × | × | × | √ |
| Makespan | √ | √ | √ | × | √ | √ | √ |
| Resource Utilization | √ | √ | √ | √ | √ | √ | √ |
| Dynamic scheme for inserting and deleting task/VM | × | × | × | × | × | × | √ |

*Notations Used:*

| | |
|---|---|
| $T$= set of tasks | $AFT_{ij}$ = Actual Finish Time of $t_i$ on $v_j$ |
| $n$ = Number of tasks | $t_p(t_i, v_j)$ = Total Processing Time of $t_i$ on $v_j$ |
| $V$= set of VMs | $o(VMf)$ = Order of $Vf$ |
| $m$= Number of VMs | $p$ = Number of Failed Resources in $VMf$ |
| *t_id* = task id | $Vf$: $f$th Failed VM in $VMf$ |
| *t_size* = task size | $Tf$: Set of Failed Tasks |
| *V_id* = Virtual Machine id | $o(Tf)$: Order of $Tf$ |
| $VM_f$ = Set of Failed Virtual Machines | $q$: Number of Failed Tasks in $Tf$ |
| $S$ = Speed of VM | $TAT$ = Turnaround Time |
| *MIPS* = Million Instruction Per Second | $U$ = Tasks executing on minimum underloaded VM |
| $M$= mapping | $UT$ = Average Utilization |
| $AR$ = Advance Reservation | $O$ = Tasks executing on maximum overloaded VM |
| $RT_j$ = Ready time of VM | $U$ = Tasks executing on minimum underloaded VM |
| $TET_j$ = Total Execution Time of VM | € = Average execution time of tasks allocated over the maximum overloaded VM |
| $EST_{ij}$= Early Start Time of $t_i$ on $v_j$ | |

## A. THE SYSTEM MODEL

This paper considers a heterogeneous system concerning both tasks and VMs. The group of VMs has varying processing speeds and so do the sizes of the incoming tasks. The system architecture of the proposed HFSLM is shown in Figure 1.

There are three basic levels in the HFSLM system architecture, i.e., the Application Layer, Middleware, and Host/VM Layer. The application layer receives the user's tasks (incoming), and the Task Sorter sorts them in the ascending order of their size as they arrive. On the other hand, the VMs that are accessible are in the Host/VM layer, the VM sorter sorts the VMs in the ascending order of their speed. The middleware handles the primary allocation and fault tolerance. The middleware is made up of two primary parts: the VM allocator, which creates the schedule for receiving task information, and the Failure Handler, which functions when any VM has a fault. Both parts effectively work together to schedule incoming tasks and reserve VMs. The different components in the VM allocator work in coordination and oversee the incoming task information for selecting the appropriate VM for accomplishing tasks. The task of identifying every accessible VM in the VM layer falls within the purview of the VM Discovery component. Once the available VMs are discovered, the suitable VM for the task is selected by the VM selector. After identifying the most appropriate VM for the task, the VM Producer allocates the specified VM to the task. Further, the VM Allocator communicates the schedule generated by it to the AR Module and the Load Balancer. In response, the AR Module activates its components and generates the reservation in case of faults and breakdowns. The Time Manager component of the AR Module forecasts the AR Slot for the affected task and reserves the suitable VM for the computed AR slot in advance. After calculating AR Slots, the System Matching verifies if the task and VM are a good fit for generating reservations, and the Reservation Producer commits the produced reservation for the estimated AR Slot in the event of a fault. Additionally, the load balancer analyzes the generated schedule and plays the key role in uniformly distributing the load among VMs by identifying the maximum overloaded and minimum underloaded VM and reallocating the tasks between them.

## B. PROBLEM FORMULATION

Initially, the set of incoming tasks represented by $T=\{t_1,t_2,\ldots t_n\}$ and the set of Virtual Machines signified by $V=\{v_1, v_2,\ldots v_m\}$ has been taken over the proposed HFSLM. Every task ($t_i$) is executed on the allocated *VM* till the execution of the task is completed. The task is pre-empted, in case the assigned VM fails or becomes unavailable at any point in time.

The execution of an affected task will start from the beginning on an alternative VM assigned to it. Further, each task has its parameters like *t_id* and *t_size*. However, each VM has its parameters like *V_id,* and S. Apart from this, a few characteristics considered for VMs are:

- *The model considers "m" VMs for the mapping of "n" tasks.*
- *S of VM is taken in MIPS (Million Instructions Per Second)*

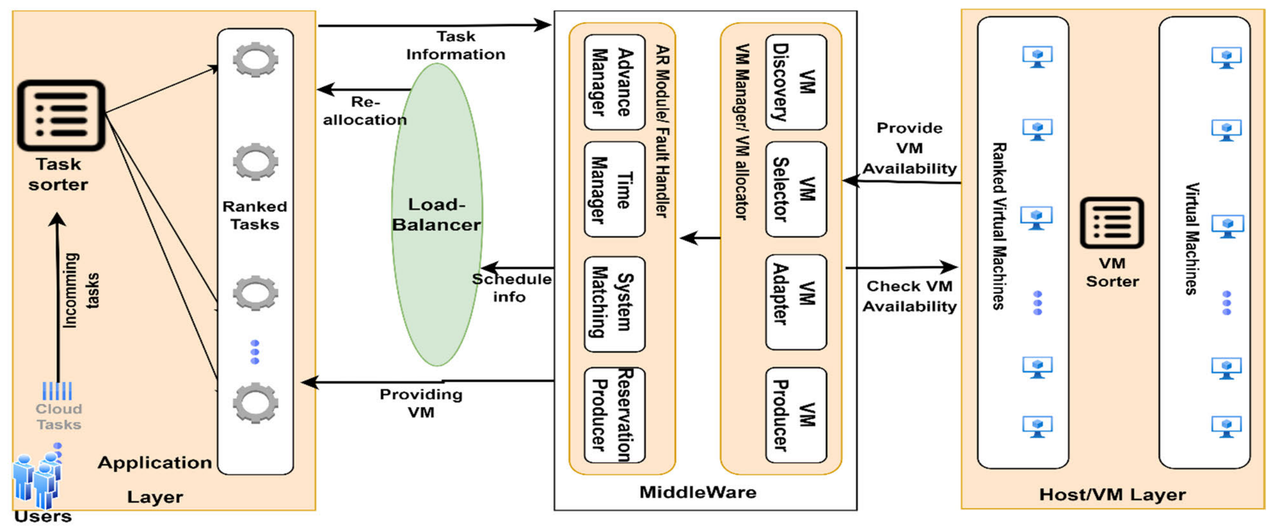

**FIGURE 1.** **The proposed system architecture.**

- *The available VMs do not apply to other applications.*
- *Each VM has its $RT_j$ associated with it. $RT_j$ is the time experienced to execute the load history on the VM.*

The problem modeled here is to generate a fault-tolerant allocation schedule in a dynamic environment like the cloud in a way that will optimize makespan and increase the average VM utilization. Mathematically, the problem can be viewed as an effective mapping ($M$) (eq 1) between two sets i.e., set $T$ and set $V$, which will optimize the given parameters.

$$M \ : T \rightarrow V \tag{1}$$

The mapping between tasks and VMs graphically can also be treated as a bipartite graph as shown in Figure 2.

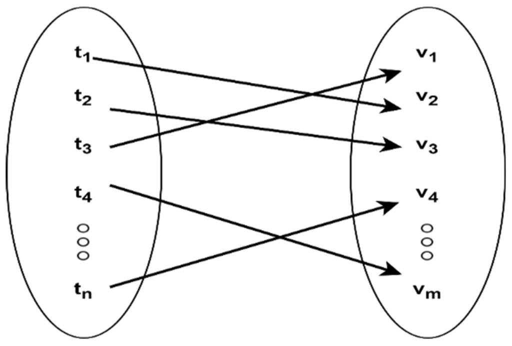

**FIGURE 2.** **Mapping between tasks and VMs.**

***Pseudocode of the Proposed Work:***

```
1. Sort tasks based on Task_size //task sorting
2. Sort VMs based on Speed (S).//VM sorting
3. Allocate the sorted tasks to the sorted VMs. //task allocation
4. Allocate the dynamically arriving tasks. //dynamic allocation
5. Calculate the AR slot for advance reservation.
6. Perform Fault tolerance by using the neighbouring-based reservation algorithm.
7. Perform load balancing or task reallocation. // Load Balancing Algorithm
8. Calculate the considered parameters and perform the comparative analysis.
```

### 1) TASK MAPPING MODEL

The main difficulties in mapping between T and V are the dynamism of the system and the limited number of available VMs. Achieving fault tolerance in such a dynamic system is a challenging task. This section explains the detailed methodology to deal with the modeled problem. Initially, in the first algorithm, the tasks are allocated to the available VMs. The allocation process first sorts the incoming tasks and VMs based on *task_size* and VM speed respectively. Thereafter maps the sorted task set to the sorted VM set. After sorting incoming tasks and VMs, the VM is assigned to the tasks in the order until the ready time of any one of the available VMs is zero. Once the ready time of all VMs becomes greater than zero that means currently all the VMs have some load history. After this point, the allocation of further arriving tasks will be done to the VM having the least ready time. Doing this will again minimize its response time. The proposed allocation strategy handles the dynamically arriving tasks by employing a Neighbouring insertion policy. The newly arriving task will be inserted based on the arriving task size. i.e., the immediate greater and immediate lesser task (neighboring tasks) than the arriving task is identified, and the newly arriving task is allocated to that task's VM which has less ready time as shown in Figure 3. This insertion policy of tasks will again play a critical role in allocating the most suitable VM for the dynamically arriving task. Similarly, the newly added VMs are inserted in their correct position by employing the same insertion policy as shown in Figure 4. This insertion policy of incoming tasks and VMs in their respective positions allows the system to handle tasks and VMs runtime.

Because of the sorting and neighbouring approach, every task will be allocated to the most suitable VM. The allocation of VMs to tasks is done in this order until the ready time of any one of the available VMs is zero. Once the ready time of all VMs becomes greater than zero that means currently all the VMs have some load history. After this point, the allocation of further arriving tasks will be done to the VM having the least ready time. Doing this will again minimize its response time.

**Algorithm 1** Task Sorting

```
def task_sort(incoming_tasks) :
        // incoming_tasks is a list of tuples (task_id, task_size)
        incoming_tasks.sort(key=lambda x: x[1], reverse=True) // Sort based
on task_size in descending order
        return incoming_tasks
    VM Sorting:
    def vm_sort(available_vms):
        // available_vms is a list of tuples (VM_id, VM_Speed)
        available_vms.sort(key=lambda x: x[1], reverse=True) # Sort based
on Speed in descending order
        return available_vms
    Task Mapping:
    def task_mapping(incoming_tasks, available_vms):
        mapped_tasks = [] // List to store allocated tasks
        for task in incoming_tasks:
            if task_status[task] == 0:
                for vm in available_vms):
                    if ready_time[vm] == 0:
                        map_task_to_vm(task, vm) # Map task to VM
                        mapped_tasks.append(task)
                        break // Move to the next task
    # Map remaining tasks to VMs with the least ready time
        for task in incoming_tasks:
                if the task is not in mapped_tasks:
                min_ready_time = min(sorted_vms,
 key=lambda vm: ready_time[vm])
                map_task_to_vm(task, min_ready_time)
    # Map task to VM with the least ready time
                mapped_tasks.append(task)
    # Update task status to 1 (mapped)
        for the task in mapped_tasks:
                task_status[task] = 1
    Dynamically arriving Task Mapping:
    while (there are upcoming tasks):
        upcoming_task = get_next_upcoming_task()
    greater_task=find_right_neighbor(upcoming_task)
    lesser_task = find left neighbor(upcoming_task)
    greater_vm = find_vm(greater_task)
    lesser_vm = find_vm(lesser_task)
        if (greater_vm.ready_time< lesser_vm.ready_time):
            map_task_to_vm(upcoming_task, greater_vm)
        else:
            map_task_to_vm(upcoming_task, lesser_vm)
```

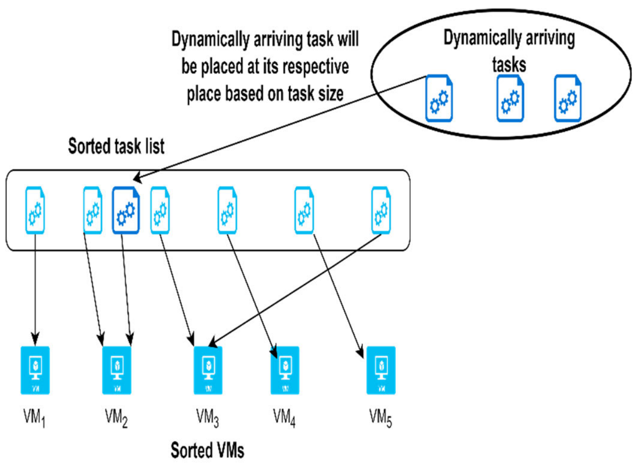
Dynamically arriving task will be placed at its respective place based on task size
Dynamically arriving tasks
Sorted task list
VM1
VM2
VM3
VM4
VM5
Sorted VMs

**FIGURE 3.** Allocation of dynamically arriving task.

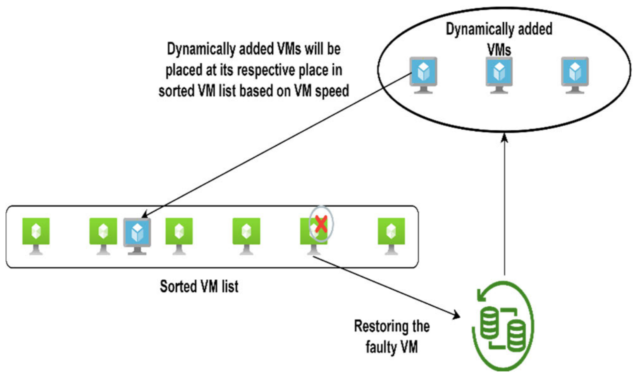
Dynamically added VMs will be placed at its respective place in sorted VM list based on VM speed
Dynamically added VMs
Sorted VM list
Restoring the faulty VM

**FIGURE 4.** Adding and deleting VMs from the system dynamically.

## 2) NEIGHBOURING-BASED RESERVATION FOR FAULT TOLERANCE

After VM allocation, a fault handling algorithm that enables the proposed work to win fault tolerance if any VM fails or leaves the system is proposed. This fault handling algorithm has been developed by employing the technique of advance reservation of neighboring VMs. The advance reservation is the technique where the AR time slot is computed or estimated and the VM is reserved for that predicted time slot to guarantee the task execution till completion. In the beginning, the TET$_j$ for all VM$_s$ is taken as zero. It means that currently, the particular VM has executed no task. Afterward, TET$_j$ is updated after the finishing of each task on the VM. Moreover, every VM has some load history which is termed as the ready time of the VM. Initially,RT$_j$ is taken as TET$_j$ as shown in eq. (2)

$$TET_j = RT_j \tag{2}$$

After the mapping of $t_i$ and $v_j$ as per the allocation algorithm explained above every ti will start its execution on some $\mathrm{v_j}$. This starting time of the execution of $t_\mathrm{i}$ on $v_\mathrm{j}$ is termed as $EST_{ij}$

and is calculated as in eq. (3)

$$EST_{ij} = TET_j \tag{3}$$

After the execution of $t_i$ on $v_j$ is over, $AFT_{ij}$ is determined by adding the total processing time of $t_i$ on $v_j$ $\{t_p(t_i, v_j)\}$ to the $EST_{ij}$ as shown in eq. (4)

$$AFT_{ij} = EST_{ij} + t_p(t_i, v_j) \tag{4}$$

where $t_p(t_i, v_j)$ is the time taken to process $t_i$ by $v_j$ and is calculated as in eq (5)

$$t_p(t_i, v_j) = \frac{t_{\text{size}}}{s} \tag{5}$$

Furthermore, $TET_j$ is updated after every execution of $t_i$ and will be equal to $AFT_{ij}$ as shown in eq. (6)

$$TET_j = AFT_{ij} \tag{6}$$

However, for calculating AR slots, the proposed algorithm takes Early Start Time and Actual Finish Time as input parameters and estimates the AR slot as the difference between $EST_{ij}$ and $AFT_{ij}$.

If the advance reservation strategy is not employed, the task $(t_u)$ may fail to execute on $VM_f$ i.e., the failed VM, or if the VM leaves the system for a certain time will result in the suffering of the corresponding task. To handle this situation, p failed VMs are defined as:

$$VM_f = \{V_f : V_f \in V \ \& \ o(VM_f) = p\}$$

and q corresponding affected tasks are defined which were executing on these failed VMs. The set of failed tasks is defined as:

$$T_f = \{t_f : t_f \in T \ \& \ o(T_f) = q \ \& \ q <= n\}$$

Now, these failed tasks need to be reallocated to some other suitable healthy VMs so that $T_f$ will execute without any interruption. On reallocating, all the failed tasks $T_f$ are migrated from $V_f$ to $VM_j$ such that:

$$VM_j \in V \ \& \ VM_j \notin V_f$$

The model reserves the neighboring VM of the corresponding failed task as an alternative VM. Later, the $TET_j$ is again updated as shown in eq. (7):

$$TET_j = TET_j + t_p(t_f VM_j) \text{ where } VM_j \bar{E} V \&\& VM_j \notin V_f \tag{7}$$

Makespan is taken as the highest or maximum among all $TET_j$ and can be expressed as eq. (8).

$$Makespan = max(TET_j), \ \forall Vj \tag{8}$$

Finally, the Average VM utilization of the system is defined as in eq. (9).

$$UT = \frac{\sum_1^k (TET - tp\,(tf \in Vf\ ,\ Vj \in Vf\ ))}{k * \text{Makespan}} \ \forall Vj \tag{9}$$

**Algorithm 2** Neighbouring-Based Reservation Algorithm for Fault Tolerance

```
Initialize_ARM (incoming_tasks, Mapped VMs, task_size, VM_Speed, $EST_{ij}$, $AFT_{ij}$, $AR_{ij}$, Status)
Compute_$AR_{ij}$()
fail_tests ={ $t_f$| $t_f \in$ T, o($t_f$) = q and q $\le$ n}
for each $t_f$ in fail_tests:
        while Status($t_f$) = 1:
                if ($t_{i-1}$, $t_{i+1}$ $\in$T && RT($t_{i+1}$.VM) < RT($t_{i-1}$.VM)):
                    Select $t_{i+1}$.VM (right neighbor) as alternative VM for $t_f$ for $AR_{ij}$
                    // Reserve the time slot for the selected task
                else if ($t_{i+1}$ $\in$T && ti$-1 \notin$T):
                    Select $t_{i+1}$.VM as alternative VM of $t_f$ for $AR_{ij}$
                    // Reserve the time slot for the selected task
                else:
                    Select $t_{i-1}$.VM as alternative VM of $t_f$ for $AR_{ij}$
                    // Reserve the time slot for the selected task
                Update_ARM (incoming_tasks, Mapped VMs, task_size, VM_Speed, $EST_{ij}$, $AFT_{ij}$, $AR_{ij}$, Status(tf) = 1)
                    // Status(tf) = 1 implies the AR slot is reserved for $t_f$
```

## IV. LOAD BALANCING MODEL

Apart from all this, a load-balancing algorithm is also proposed which escorts the whole system for uniform load distribution that might be disturbed after fault-handling throughout the system and further improves the makespan and utilization. The under and overloaded VMs are identified by the Load balancing algorithm and the load is shifted from the overloaded VM to the underloaded VM for uniform distribution of load among VMs. The VMs having the highest and lowest makespan are taken as maximum overloaded and underloaded VMs respectively. Then (€ ), the average execution time of tasks assigned over the maximum overloaded VM is calculated as in eq (10). The tasks with execution time less than the €are taken as separate sets ($\Sigma$). i.e., $\Sigma = \{t_i | E(t_i, v_j) <$ €

$$€ = \sum_{i=1}^{o} \frac{tp(ti,\ vj)\ \&\&\ ti\ \bar{e}\ O}{|O|} \tag{10}$$

Finally, the load is shifted from an overloaded VM to the underloaded VM as described in the load balancing algorithm.

Figure 5 depicts the flowchart of the proposed work.

### A. THE PROPOSED HFSLM

The allocation in HFSLM is done in three phases: In the allocation phase, we perform Task Sorting, VM Sorting, and Task Allocation. However, for dynamically arriving tasks, HFSLM provides a distinct algorithm for the allocation. In the second phase: fault tolerance is achieved by proposing an innovative fault-tolerant algorithm namely Neighbouring-based Reservation Algorithm for Fault Tolerance. Following fault tolerance, the model addresses the evenly distributed load among VMs by recommending a load-balancing technique.

**HFSLM(O, €)**

```
Call Algorithm 1;
Call Algorithm 2;
Σ = {ti | E(ti, vj) < €}
Sort set Σ in descending order of execution time
for each task $t_i$ in Σ:
        shift_task_to_underloaded_VM($t_i$)
        update_makespan()
        makespan.overloaded_VM = makespan. overloaded_VM
 - execution_time($t_i$)
        makespan.underloaded_VM = makespan.underloaded_VM
+execution_time($t_i$)
if(makespan.underloaded_VM < makespan. overloaded_VM):
            continue        // Take another task from Σ
        else:
            rollback (makespan.underloaded_VM, makespan. overloaded_VM)
// To the previous state
            $t_i$= $t_{i+1}$      //take next task from Σ
end for
Estimate the QoS parameters Makespan, and UT
                    // as per Eq.(8) and Eq. (9)
```

### B. COMPUTATIONAL COMPLEXITY OF HFSLM

To compute the complexity of HFSLM, the basic operations are analyzed as a function of input size. We will express the complexity of the presented HFSLM in Big Onotation:

*Task Sorting and VM Sorting:*

Sorting a list of n tasks concerning task size using Quick-Sort, the complexity is typically O (n log n)

*Task Mapping:*

- For n tasks and m VMs, we need to iterate through all VMs in the worst case for a suitable mapping. i.e., O (n * m) iterations
- Operations of each iteration will take constant time.
- Complexity is O (n * m)

*Dynamically arriving Task Mapping (for n arriving tasks):*

- For identifying neighboring tasks of the arrived task, the algorithm takes constant time.
- Total number of iterations equals the number of arriving tasks (n).
- Complexity is O (n)

*Neighbouring-based Reservation Algorithm for Fault Tolerance:*

- For q failed tasks, the algorithm iterates for each failed task and performs constant time operations.
- The total sum of iterations depends on the number of failed tasks (q).
- Complexity is O (q)

Similarly, for the load balancing algorithm, if we assume k tasks in Σ have execution time less than €, the complexity will be O (k)

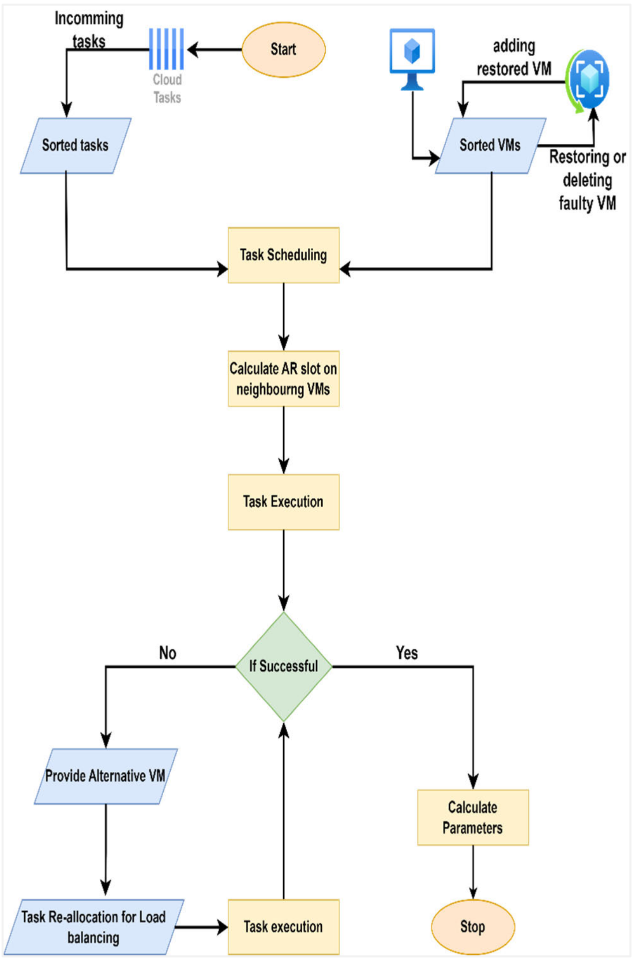

**FIGURE 5. Flow chart of the Proposed HFSLM.**

*HFSLM Algorithm Complexity:*

The total complexity of the model is taken by adding all the individual complexities:

O (n log n) + O (n * m) + O (n) + O (q) + O (k)

As we can observe O (n * m) dominates other runtime operations because it depends on both the number of tasks and VMs. Therefore, the complexity of the model can be estimated as O (n * m).

## V. ILLUSTRATIVE EXAMPLE

This section demonstrates an explanatory and motivational example where the working of the proposed reservation-based fault tolerance and load balancing model has been expressed. An example to illustrate the model has been taken from the most recent paper where FTHRM [39] has been proposed and we have related our proposed model with FTHRM based on the same example.

Nine different independent tasks and three VMs have been taken to demonstrate the working of the proposed model. (*Note: the proposed model supports run-time dealing with both tasks and VMs as shown in Figures 3 and 4*). But for simplicity of an example, we are taking the instance of tasks

**TABLE 2.** Instance of tasks and VMs.

| *Task modeling for the illustration* | | *VM modeling for illustration* | |
|---|---|---|---|
| Task (*ti*) | Task_size | Virtual Machine(Vi) | VM_speed (s) |
| *t*0 | 120 MI | V1 | 10 MIPS, |
| *t*1 | 260 MI | | Ready time = 2 µs |
| *t*2 | 380 MI | | |
| *t*3 | 90 MI | V2 | 12 MIPS, |
| *t*4 | 100 MI | | Ready time = 4 µs |
| *t*5 | 220 MI | | |
| *t*6 | 400 MI | V3 | 14 MIPS, |
| *t*7 | 280 MI | | Ready time = 6 µs |
| *t*8 | 350 MI | | |

and VMs as shown in Table 2. The Ready time of each VM is the previous load on the VM. Now, the allocation of tasks to the VMs has been done by the proposed strategy where the tasks and VMs are sorted initially according to increased *task_size* and VM_speed respectively as shown in Figure 6.

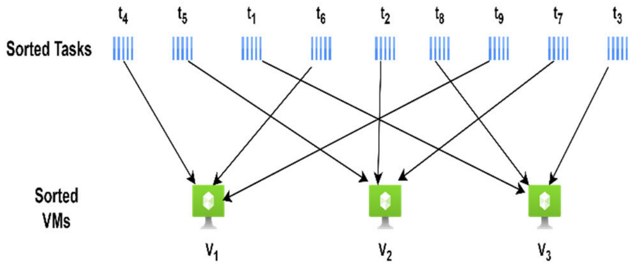

**FIGURE 6.** Task allocation concerning the proposed strategy.

The proposed allocation is illustrated in Figure 7. $\text{EST}_{ij}$ is the time when the execution of $t_i$ on $v_j$ starts and $\text{AFT}_{ij}$ is the time when the execution of $t_i$ on $v_j$ completes. AFT of the previously executed task on any VM becomes the EST of the next task on the same VM as shown in Figure 7. Furthermore, $\text{AFT}_{ij}$ is computed by adding the execution time ($t_p(t_i, v_j)$) to the $\text{EST}_{ij}$. Here, $t_p(t_i, v_j)$ is computed as shown in equation 8. The TET of each VM is initialized to zero which indicates that no task has been executed on the particular VM.

However, in this example, each VM has its Ready time i.e., $RT(V_1) = 2$, $RT(V_2) = 4$, and $RT(V_3) = 6$. Now, the Ready time of $V_j$ will be assigned to $\text{TET}_j$. In other words, $\text{TET}(V_1) = 2$, $\text{TET}(V_2) = 4$, $\text{TET}(V_3) = 6$.

## A. TASK ALLOCATION

After sorting tasks and VMs, VMs are allocated to the tasks in the sorting order as shown in Figure 6. Initially, $t_4$ is allocated to $\text{VM}_1$ with EST = 2, $t_5$ is allocated to $V_2$ with EST=4, and $t_1$ is allocated to $\text{VM}_3$ with EST = 6. For $t_4$, $\text{EST}_{41} = 2$ because Ready time of $\text{VM}_1$ is 2, now to compute $\text{AFT}_{41},\ t_p(t_4, v_1) = \frac{90}{10} = 9$ will be added to the $\text{EST}_{41}$. In other words, $\text{AFT}_{41} = 2{+}9 = 11$. After the execution of $t_4$ is over, $\text{RT}(V_1)$ will be updated to 11 and is the EST of the next task.

Similarly, for $t_5$, $\text{EST}_{52} = 4$ because Ready time of $\text{VM}_2$ is 4, now to compute $\text{AFT}_{52},\ t_p(t_5, v_2) = \frac{100}{12} = 8.3$ will be added to the $\text{EST}_{52}$. In other words, $AFT_{52} = 4{+}8.3 = 12.3$. After the execution of $t_5$ is over, $\text{RT}(V_2)$ will be updated to 12.3 and is the EST of the next task.

For, $t_1$, the same thing happens, and $\text{RT}(V_3)$ will be updated to 14.5 and is the EST of the next task as illustrated in Figure 7.

The next task i.e., $t_6$ will be allocated to that VM whose RT has been already computed i.e., the VM which is available or free. If RT has been computed for more than one VM, then the next task will be allocated to the VM having minimum ready time. (*Note: as the VMs and task are sorted, RT for all the VMs will also come to be sorted. The same happens with this example also, there will be some variation in case little or minimum variation in arriving task size i.e., in low task heterogeneity cases*). Finally, after the allocation of all tasks is over, we compute the Makespan and Average VM Utilization.

Since, $TET(v_1) = 68,\ TET(v_2) = 71.2,\ TET(v_3) = 73..7$ as shown in Figure 7.

Now, Makespan is computed as $max(TET_j),\ \forall v_j$ $i.e., max(68,\ 71.2,\ 73.7)$

$$Makespan = 73.7$$

*Average VM Utilization(U) is computed as:*

$$\frac{sum\ of\ all\ TETs}{Total\ number\ of\ VMs\ X\ Makespan}$$

$$U = \frac{\mathbf{68 + 71.2 + 73.7}}{\mathbf{3X73.7}} = \frac{\mathbf{213.1}}{\mathbf{221.1}} = \mathbf{96.3\%}$$

Comparing the Makespan and Average VM Utilization with FTHRM, the Makespan of FTHRM was found to be as 80 and utilization was 84.76% for the same example. The allocation in FTHRM was done according to the MCT strategy. In other words, the proposed allocation used in the model surpasses the MCT strategy as well on both makespan and average VM utilization.

## B. FAULT-TOLERANCE

As per the algorithmic flow of the model, after allocation, we are performing fault tolerance of the system using a neighboring-based advance reservation. Before reservation-tolerance, we have assumed and illustrated the random fault tolerance first so that we can compare our neighbouring reservation fault tolerance with the assumed random fault tolerance. The implementation results are also compared with FTHRM in the results section.

### 1) RANDOM FAULT TOLERANCE

Now, let's suppose $V_3$ failed at 34.5 as shown in Figure 8. Now if we randomly assign an alternative VM to the affected task which here is $t_3$. In Random allocation, we randomly pick

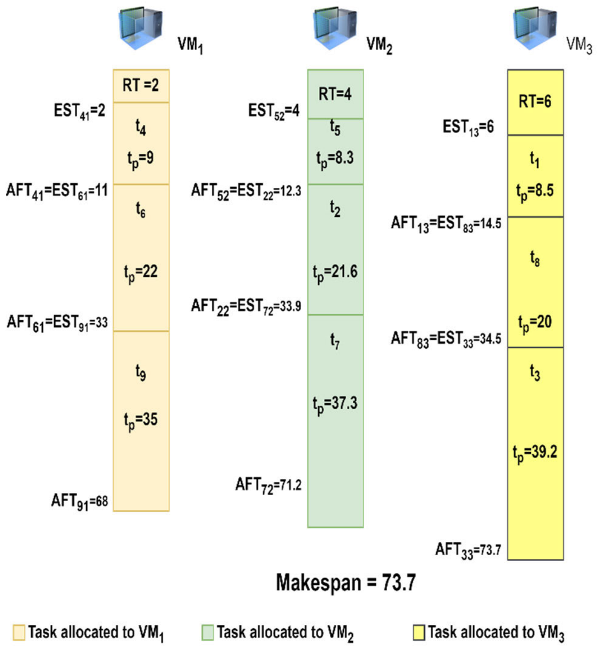

**FIGURE 7.** Allocation of tasks concerning the proposed sorting algorithm-based fault.

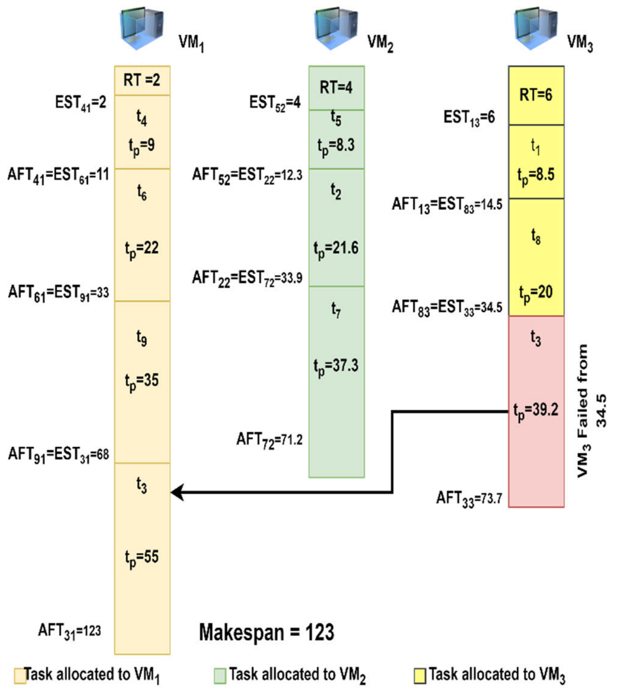

**FIGURE 8.** Random fault tolerance without reserving neighboring VMs.

any of the VMs and allocate them to t$_3$. Here we are migrating t$_3$ to VM$_1$ till t$_3$ completes its execution.

Since

So, $AFT_{31} = 68 + 55 = 123$.

Hence, $TET(v_1) = 123$, $TET(v_2) = 71.2$, $TET(v_3) = 73.7 - t_p(t_3, v_3) = 73.7 - 39.2 = 34.5$ as shown in Figure 8.

Now, Makespan is computed as $max(TET_j),\ \forall v_j$

$i.e., max(123.\ 71.2,\ 34.5)$

$$Makespan = 123$$

Average VM Utilization(U) is computed as:

$$\frac{sum\ of\ all\ TET}{Total\ number\ of\ VMs\ X\ Makespan}$$

$i.e.,\ U = \frac{123+71.2+34.5}{3\times 123} = \frac{194.2}{246} = 78.9\%$

It is clear because of fault in any of the VMs, the makespan increases and utilization decreases.

### 2) PROPOSED NEIGHBOURING BASED RESERVATION

Now, we are going to use a neighboring-based reservation to provide an alternative VM to the affected task. Neighboring-based reservation strategy selects the neighboring VM as an alternative VM for the affected tasks. As we have already sorted the tasks and VMs, therefore, reserving the neighboring VM will ensure that the same capacity of an alternative VM is reserved for the affected task. For using an advance reservation, we need to estimate the advance reservation slot represented as an AR$_{ij}$ slot i.e., advance reservation slot for t$_i$ on some VM$_j$ and we are reserving the neighboring VM for the same AR slot to ensure task execution till completion of the task. AR$_{ij}$ is estimated as the difference between AFT$_{ij}$ and EST$_{ij}$ given in equation (11).

$$AR_{IJ} = AFT_{IJ} - EST_{IJ} \tag{11}$$

Furthermore, all the information regarding tasks and VMs including AR slots are stored in a Matrix known as ARM as shown in Table 3.

The illustration of fault tolerance by reserving Neighbouring VMs is given in Figure 9. It is clear from the illustration, that, unlike random fault tolerance with reservation, VM$_2$ has been selected as an alternative VM because VM$_2$ is the neighbor of failed VM i.e., VM$_3$. The affected task i.e., t$_3$ has been migrated to VM$_2$ which will be of the approximately same capacity as that of VM$_3$. The computational flow is shown in Figure 9.

Again, let's suppose V$_3$ failed at 34.5 as shown in Figure 9. Now, using a Neighbouring-based reservation, we will select the Neighbouring VM for the affected task.

Here, we are migrating t$_3$ to its neighbouring VM (VM$_2$) till t$_3$ completes its execution.

Since, $t_p(t_3, v_2) = \frac{550}{12} = 45.8$.

So, $AFT_{32} = 71.2 + 45.8 = 117$.

Hence, $TET(v_1) = 68$, $TET(v_2) = 117$, $TET(v_3) = 73.7 - t_p(t_3, v_3) = 73.7 - 39.2 = 34.5$ as shown in Figure 9.

Now, Makespan is computed as $max(TET_j),\ \forall v_j$

$i.e., max(68,\ 117,\ 34.5)$

$$Makespan = 117$$

**TABLE 3.** Advance reservation matrix (ARM).

| Task ($t_i$) | VM ($V_j$) | Task_size ($w(t_i)$) | speed (s) | $EST(ij)$ | $AFT(ij)$ | AR slot ($ARs_j$) | Status |
|---|---|---|---|---|---|---|---|
| $t1$ | VM$_3$ | 120 | 14 | 6 | 14.5 | 8.5 | 1 |
| $t2$ | VM$_2$ | 260 | 12 | 12.3 | 33.9 | 21.9 | 1 |
| $t3$ | VM$_3$ | 380 | 14 | 71.2 | 117 | 45.8 | 1 |
| $t4$ | VM$_1$ | 90 | 10 | 2 | 11 | 9 | 1 |
| $t5$ | VM$_2$ | 100 | 12 | 4 | 12.3 | 8.3 | 1 |
| $t6$ | VM$_1$ | 220 | 10 | 11 | 33 | 22 | 1 |
| $t7$ | VM$_2$ | 400 | 12 | 33.9 | 71.2 | 37.3 | 1 |
| $t8$ | VM$_3$ | 280 | 14 | 14.5 | 34.5 | 20 | 1 |
| $t9$ | VM$_1$ | 350 | 10 | 33 | 68 | 35 | 1 |

(*Note: here the failed VM$_3$ has only one neighbouring VM i.e., VM$_2$. When the failing VM has both neighbours that time an alternative VM will be selected with the least Ready time*)

Average VM Utilization(U) is computed as:

$$\frac{sum\ of\ all\ TET}{Total\ number\ of\ VMs\ X\ Makespan}$$

*i.e.,* $U = \frac{68+117}{2\times117} = \frac{185}{234} = 79.5\%$

It is clear from the illustration that using reservation makespan and utilization improved. Although, in this example, the utilization is found to be increased by only 1%. This is because we have only three VMs here. Furthermore, the capacity of VMs varies only by 2MIPS. In real-time where we have a large number of VMs of extremely different capacities, this strategy will show huge improvements in both makespans as well as utilization.

## C. LOAD BALANCING

After performing fault tolerance, the model escorts the whole system with load balancing for further optimization of the makespan and utilization. The load balancing algorithm focuses on the maximum overloaded and minimum underloaded VM. After identifying the overloaded and underloaded VM, the load is shifted according to the given algorithm and depicted in Figure 10. In this example, after fault tolerance, the overloaded and underloaded VMs are identified as VM$_2$ and VM$_1$ respectively. Tasks executing on maximum overloaded VM and minimum underloaded VM are denoted as sets O and U respectively. Then, the average execution time (€) of tasks allocated over the maximum overloaded VM is calculated. The tasks having execution time less than the average execution time are taken as separate sets ($\Sigma$). I.e., $\Sigma$=t$_i$| $E(t_i, v_j) <$€

Overloaded and underloaded VMs are VM$_2$ and VM$_1$ respectively.

$O$:{$t_5$, $t_2$, $t_7$, $t_3$}&& |$O$| = $o$, $U$: {$t_4$, $t_6$, $t_9$}&& |$U$| = $u$

$$€ = \sum_{i=1}^{o} \frac{E\ (ti,\ vo)\ \&\&\ ti\ \bar{e}\ O}{|O|}$$

$$= \frac{E\ (t5,\ v2) +\ E\ (t2,\ v2) + E\ (t7,\ v2) + E(t3,\ v2)}{4}$$

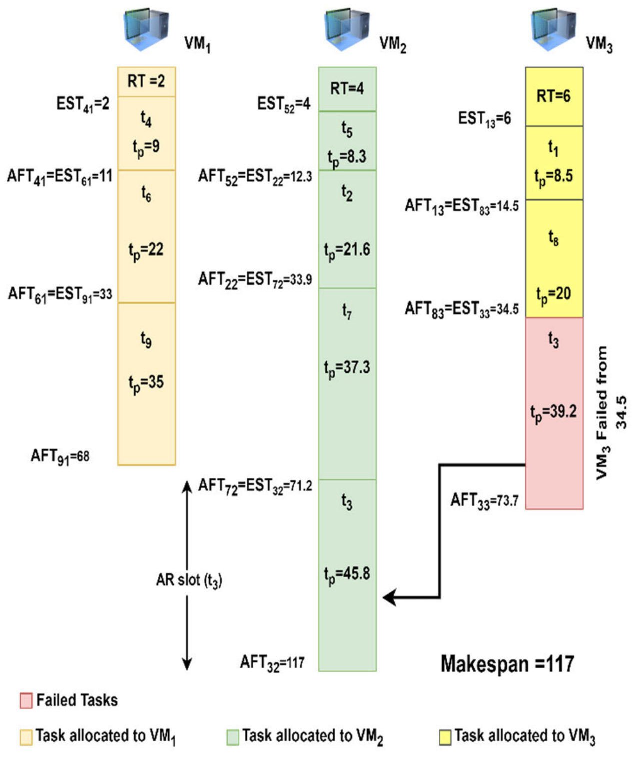

**FIGURE 9.** Fault tolerance by reserving neighbouring VMs.

$$= \frac{8.3 + 21.6 + 37.3 + 45.8)}{4} = \frac{113}{4}$$

€ = 28.5

$\Sigma$=$t_2$, $t_5$ E($t5$, $v2$) = 8.3 *and E* ($t2$, $v2$)= *21.6*(**both are less than €**)

Now $\Sigma$ will be sorted in descending order of their execution time.

$\Sigma$={t$_5$, t$_2$}

Now the algorithm will shift the load from VM$_2$ to VM$_1$ till the makespan of underloaded VM < makespan of overloaded VM (see load balancing algorithm). Here, we are migrating t$_2$ from VM$_2$ to VM$_1$. Now, Makespan is computed as:

*max* $(TET_j)$,$\forall V_j$*i.e., max (94, 95.4, 34.5)*

Makespan = 95.4

Average VM Utilization(U) is computed as:

$$\frac{sum\ of\ all\ TET}{Total\ number\ of\ VMs\ X\ Makespan}$$

$U = \frac{94+95.4}{2\times95.4} = \frac{189.4}{190.8} = 99.2\%$

Since,$t_p(t_2, v_1) = \frac{260}{10} = 26.$

So, $AFT_{21} = 68 + 26 = 94.$

Hence, $TET(v_1) = 94$, $TET(v_2) = TET(v_2) - t_p(t_2, v_2) = 117 - 21.6 = 95.4$, $TET(v_3)$ will be the same i.e., 34.4 as it is a faulty VM as shown in Figure 10.

Now, Makespan is computed as $max(TET_j)$, $\forall v_j$ *i.e.,max*(94, 95.4, 34.5)

$$Makespan = 95.4$$

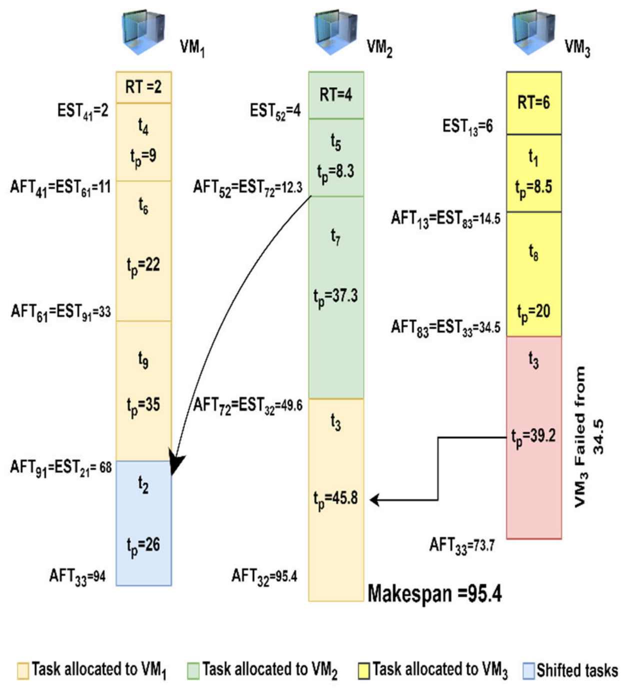

**FIGURE 10.** Load balancing after fault tolerance.

Average VM Utilization(U) is computed as:

$$\frac{sum\ of\ all\ TET}{Total\ number\ of\ VMs\ X\ Makespan}$$

$i.e.,\ U = \frac{94+95.4}{2\times 95.4} = \frac{189.4}{190.8} = 99.2\%$

The above illustration demonstrates the working of the whole hybrid model. Comparing the proposed model with FTHRM. FTHRM shows the final utilization as 84.76 on the other hand proposed model shows the final utilization as 99.2% in the same example. It is because of escorting the proposed model with load balancing. Furthermore, FTHRM shows a makespan of 80 but the proposed model shows a makespan of 95.4. It is because the proposed model has reserved the neighbouring VMs for the affected tasks and has also migrated the affected task from the failed VM to the neighbouring reserved VM. It is clear from the illustration that after load balancing the makespan and utilization have been optimized up to 18.8% and 20% respectively than before load balancing.

## VI. RESULTS AND DISCUSSIONS

The findings were evaluated over MSI GF 65 Thin (15-inch, 2021 Model) Machine, having 4 GHz Hexa-core Intel Core i5 Processor, 816GB 2133 MHz DDR4 RAM, Intel Iris Plus Graphics 650 1536 MB Graphics, Nvidia Gtx 1660ti (6gb vram), 512 GB SSD Storage. The used Operating System was Windows 11.

Python 3.7 is the used programming language and environment.

The tools included in Python 3.7 are:

- *Excel - for producing and managing the CSV that the Python program uses as input that contains the job sizes and resource capacities.*
- *Jupyter Notebook for Python development.*
- *The used Python Libraries (Library - Version) are:*

Numpy - 1.19.2 (used for vector algebra)

Pandas - 1.1.2 (used for manipulating CSV and data frames, (both loading and storing))

Operator - used for sorting dictionary.

Random - To produce arbitrary or random numbers.

The proposed HFSLM is compared with four different approaches based on two main parameters i.e., makespan and average utilization. HFSLM is evaluated by comparing it with FTHRM [22], OLB [23], MIN-MIN [24], and MAX-MIN [25] for less than 1000 tasks. Also, compared with ELISA and MELISA with greater than 10,000 tasks. The conclusions were observed on varying the number of tasks and VMs. Furthermore, task and machine heterogeneity are also varied to analyze the results of the proposed model more clearly. As mentioned in [5], the range of Expected Time to Compute (ETC) for ti on $v_j$ is variable as heterogeneity varies from low to high for both the arriving tasks and VMs. By altering the heterogeneities of the incoming tasks and VMs, HFSLM is evaluated in this section.

### A. VARYING HETEROGENEITY OVER SMALL TASK SCALE

The evaluation has been done by adjusting the number of tasks and VMs, size of tasks, and capacities of VMs in four different heterogeneities given by [5] i.e., HH, HL, LH, LL. For all these four cases the working efficiency of the proposed model and compared strategies have been analyzed and depicted graphically in the given figures. In comparison, the tasks have been taken on a small scale varying from 250 to 1000. On the other hand, the VMs have varied from 16 to 128. Additionally, the input parameters taken to analyze the considered model are further shown in Table 4.

**TABLE 4.** Simulation parameter used in HFSLM.

| S.no. | Input parameter | Range |
|---|---|---|
| | Task and VM Number | |
| 1 | No of tasks (n) | 250 to 1000 |
| 2 | No of VMs (m) | 16 to 128 |
| | Task size (t_size) | |
| 3 | Low task heterogeneity | 1 MI to 100 MI |
| 4 | High task heterogeneity | 100 MI to 3000 MI |
| | VM Speed (S) | |
| 5 | Low task heterogeneity | 1 MIPS to 10 MIPS |
| 6 | High task heterogeneity | 10 MIPS to 100 MIPS |

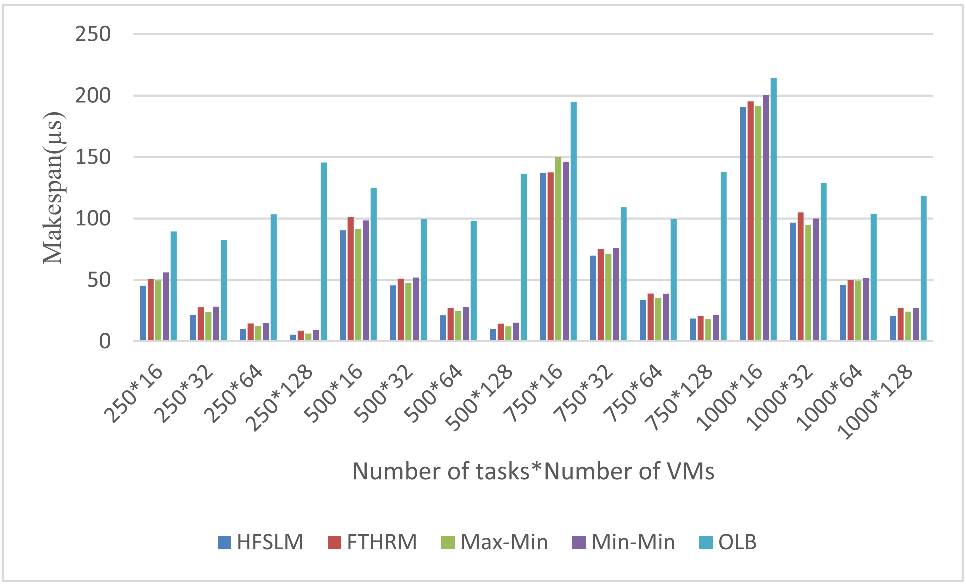

**FIGURE 11.** Makespan for varying tasks and VM (HH).

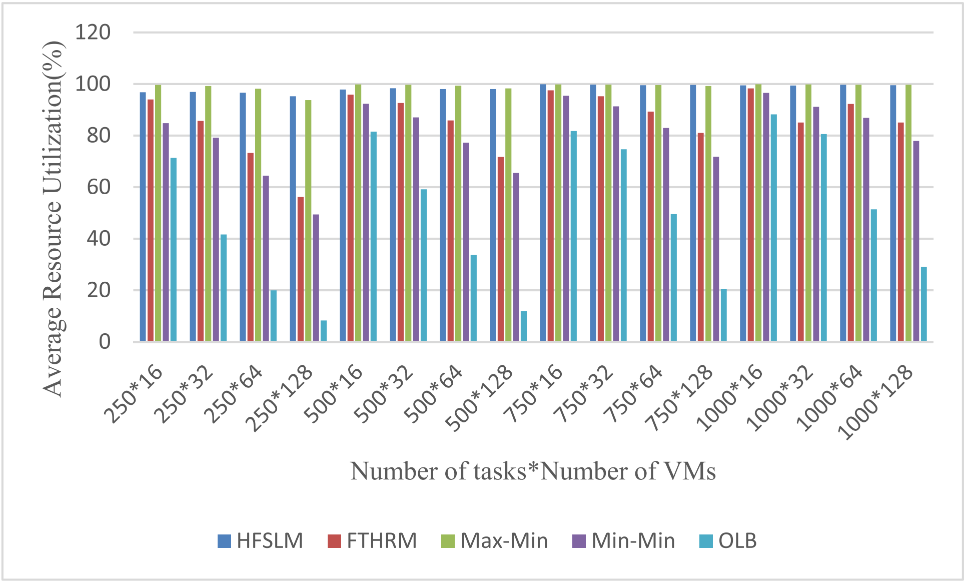

**FIGURE 12.** Average resource utilization for varying task and VM (HH).

### 1) HIGH TASK – HIGH MACHINE HETEROGENEITY (HH)

In high task heterogeneity, the task size ranges from 100 MI to 3000 MI, and high machine heterogeneity ranges from 10 MIPS to 100 MIPS. A few observations regarding the considered parameters i.e., makespan and average resource utilization are depicted in Figures 11 and 12. Further details of the observations are as follows:

- *The proposed HFSLM is enhancing the makespan because of its planned features. For all the ranges of task number and VM number considered, the model surpasses other strategies by offering a minimum makespan. Apart from this, it is seen in Figure 11, that at large task scale and small VM scale, the makespan shown is quite large but in that case also, HFSLM offers an optimized makespan.*
- *Furthermore, on comparing average resource utilization, HFSLM beats about 80% of the compared approaches. Additionally, for small-scale tasks, HFSLM shows better utilization than MAX-MIN and as the number of tasks and VMs are going towards extremely large scales, HFSLM and MAX-MIN go almost equally.*
- *Out of all the compared approaches, OLB performs worst in both makespan and utilization. It is probably because OLB does not follow any plans and strategies.*

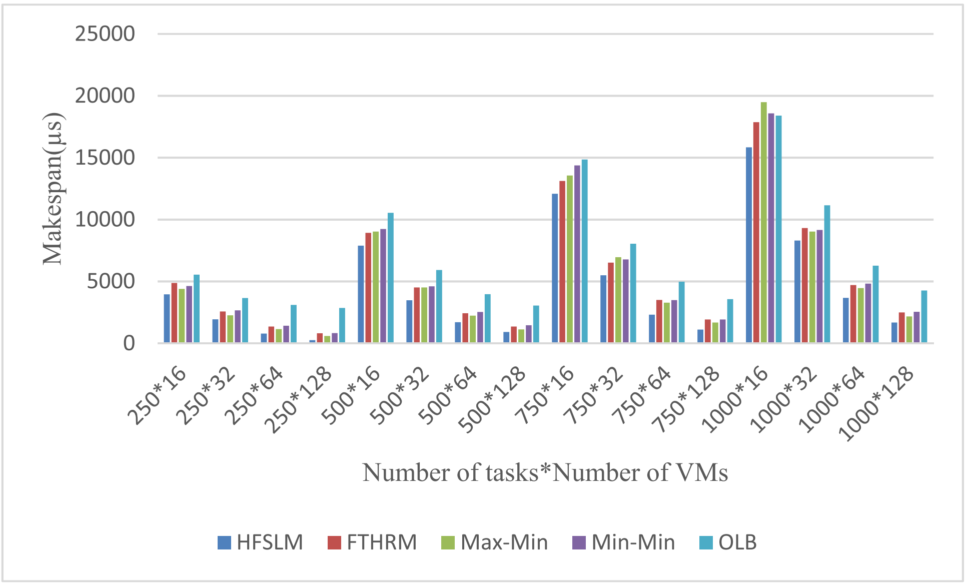

**FIGURE 13.** Makespan for varying tasks and VM (HL).

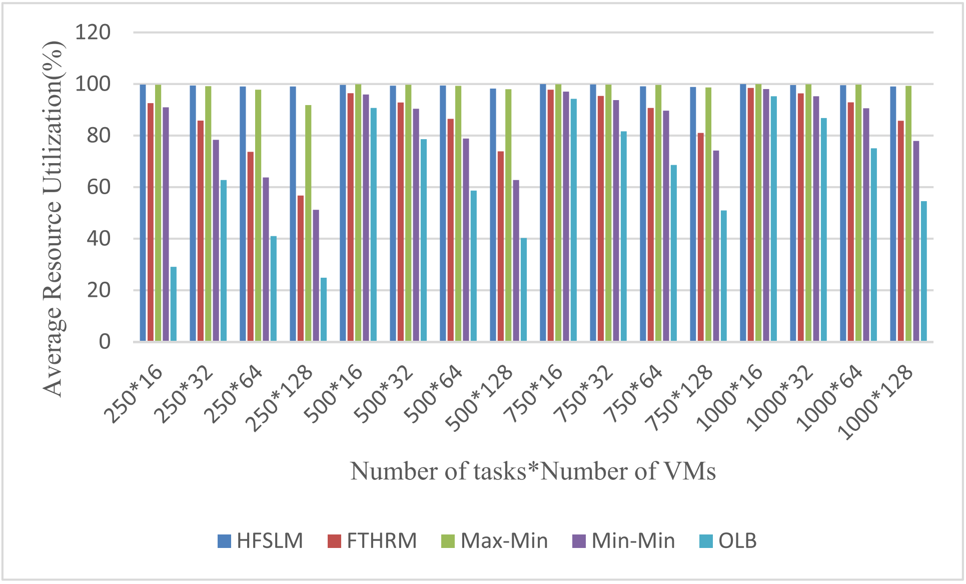

**FIGURE 14.** Average resource utilization for varying task and VM (HL).

### 2) HIGH TASK – LOW MACHINE HETEROGENEITY (HL)

In high task heterogeneity, the task size ranges from 100 MI to 3000 MI, and low machine heterogeneity ranges from 1 MIPS to 10 MIPS. A few observations regarding the considered parameters i.e., makespan and average resource utilization are depicted in Figures 13 and 14. Further details of the observations are as follows:

- *The proposed HFSLM is enhancing the makespan in HL because of its planned features. For all the ranges of task number and VM number considered, the model surpasses other strategies by offering a minimum makespan. Moreover, as depicted in Figure 13, HFSLM provides an optimized makespan even in the case of large task numbers and small available VMs. However, in such cases, MAX-MIN offers the highest makespan.*
- *On comparing average resource utilization, HFSLM again efficiently beats about 80% of the compared approaches in HL also. In HL, the proposed model also beats MAX-MIN in the case of a small task scale, and as the task scale goes up, both HFSLM and MAX-MIN go with almost tie.*
- *Out of all the compared approaches, OLB provides the worst makespan. However, in some cases, MAX-MIN also did not show an optimized makespan.*

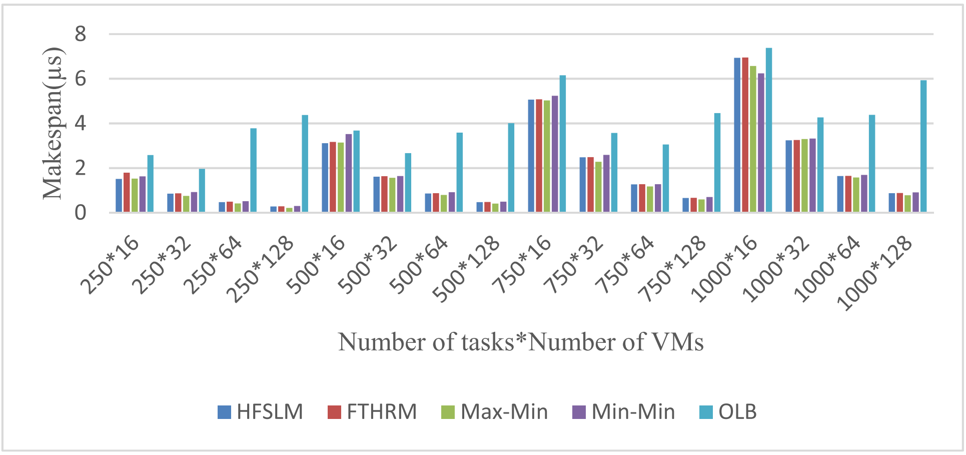

**FIGURE 15.** Makespan for varying tasks and VM (LH).

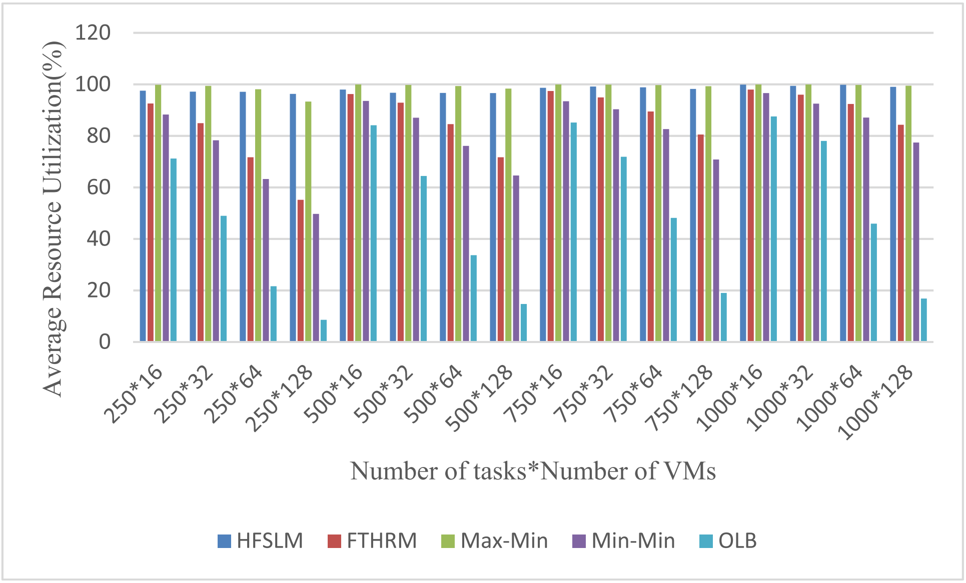

**FIGURE 16.** Average resource utilization for varying task and VM (LH).

*Additionally, in the case of average resource utilization, OLB offers very limited utilization than all the considered approaches.*

### 3) LOW TASK – HIGH MACHINE HETEROGENEITY (LH)

In low task heterogeneity, the task size ranges from 1 MI to 100 MI, and high machine heterogeneity ranges from 10 MIPS to 100 MIPS. A few observations regarding the considered parameters i.e., makespan and average resource utilization are depicted in Figures 15 and 16. Further details of the observations are as follows:

- *In this particular case, the proposed HFSLM cannot beat MAX-MIN concerning the makespan. As depicted in Figure 15, the makespan provided by HFSLM and MAX-MIN provided either equal makespan or MAX-MIN outperformed HFSLM. However, there are rare cases where HFSLM showed a little better makespan than MAX-MIN.*
- *On comparing average resource utilization, HFSLM again efficiently beats all the compared approaches except MAX-MIN. However, the proposed model also beats MAX-MIN in the case of a small task scale. The performance of HFSLM in the mid-scale goes down more than MAX-MIN but as the task scale goes up, HFSLM shows optimized utilizations.*
- *Like in the case of HH, out of all the compared approaches, OLB performs worst in both makespan and utilization in LH also. The reason for this can again be the lack of significant strategies in OLB.*

### 4) LOW TASK – LOW MACHINE HETEROGENEITY (LL)

In low task heterogeneity, the task size ranges from 1 MI to 100 MI, and low machine heterogeneity ranges

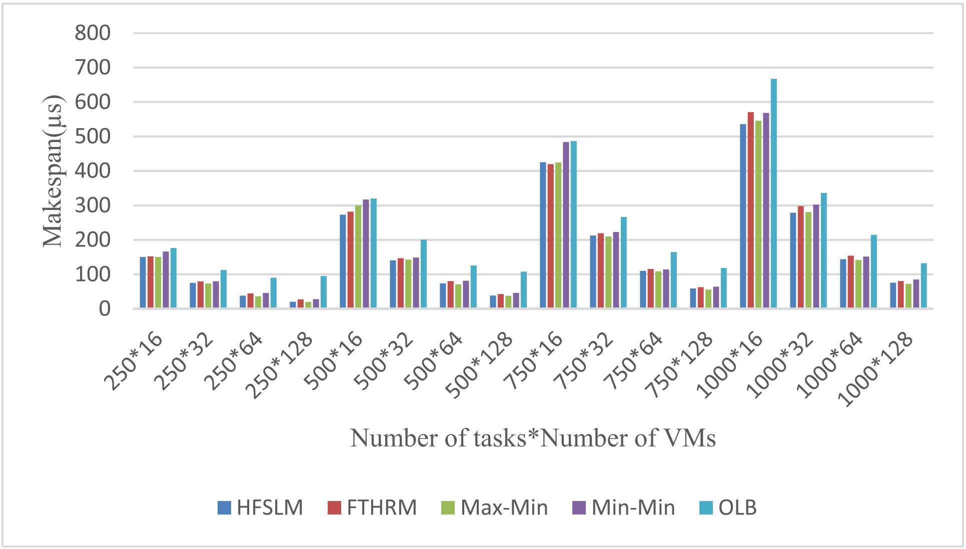

**FIGURE 17.** Makespan for varying tasks and VM (LL).

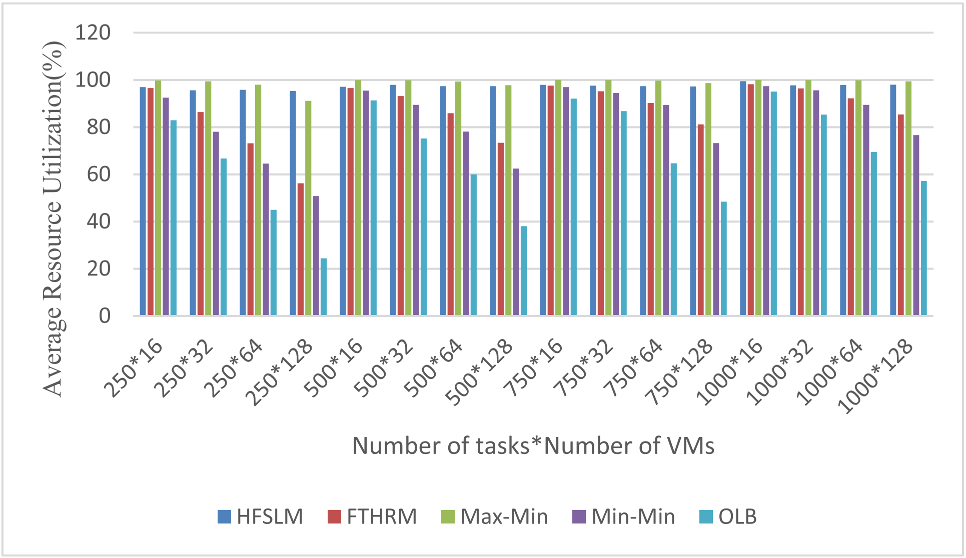

**FIGURE 18.** Average resource utilization for varying task and VM (LL).

from 1 MIPS to 10 MIPS. A few observations regarding the considered parameters i.e., makespan and average resource utilization are depicted in Figures 17 and 18. Further details of the observations are as follows:

- *In the case of LL, the proposed HFSLM beats MAX-MIN concerning the makespan. As depicted in Figure 17, the makespan provided by HFSLM for small task scales is more optimal than MAX-MIN. However, as the task scale goes up, both HFSLM and MAX-MIN are in a tie.*
- *On comparing average resource utilization, HFSLM utilizes the resource efficiently. However, MAX-MIN and HFSLM behave almost the same in all task scales.*
- *As depicted in Figure 17, OLB and MIN-MIN both show poor makespan for small task scales. Additionally, as the task scales grow, the makespan of MIN-MIN improves. Furthermore, as depicted in Figure 18, OLB shows poor resource utilization in all cases.*

*Observations:* The suggested technique outperforms FTHRM in terms of makespan and utilization, which go from 0.72% to 10.8% and 1.01% to more than 50%, respectively. When compared to MAX-MIN, HFSLM exhibits makespan improvements of −3.03% to 8.8% and average resource utilization gains of −2.15% to 6.7%. While comparing the suggested approach with MIN-MIN, the model shows an improvement of 0.6% to 19% in makespan and 1.09% to more than 45% in utilization. However, OLB was seen

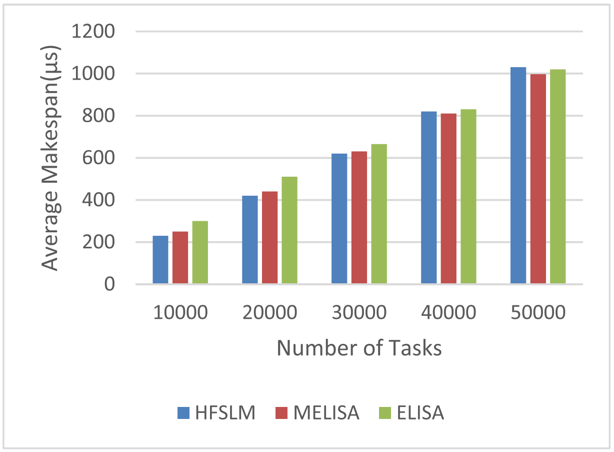

**FIGURE 19.** **Average makespan on varying heterogeneity.**

to perform very weakly among all approaches where the suggested model shows improvements of more than 50% in both makespan and utilization than OLB. Furthermore, it was observed that all the models perform almost equal optimization in makespan in LH heterogeneity. However, in that case, also OLB performs weakly among the compared approaches.

### B. VARYING HETEROGENEITY OVER LARGE TASK SCALE

Besides, the suggested model has also been contrasted with two other popular load-balancing models. i.e., ELISA and MELISA, and was evaluated based on makespan and average resource utilization on an extremely large task scale. The tasks were varied from 10000 to 50000. This is because the literature shows that these two models are more significant for large task scales. The average results of the makespan are depicted in Figure 19. The outcomes confirm that the makespan of the proposed HFSLM is superior to ELISA and MELISA. However, as the number of tasks grows beyond 40000 the average makespan of ELISA and MELISA is optimized. This shows that ELISA and MELISA are more optimal at a very high task scale.

Comparing average utilization, the three models are compared based on Min, Avg, and Max average resource utilization. In Figure 20, it can be noted that there is a significant variation in the range of VM utilization for ELISA and MELISA. However, for HFSLM, the range of variations between minimum, average, and maximum utilization is almost negligible.

The proposed model was seen to perform optimally in the case of utilization in all cases of heterogeneity. However, on makespan, the model could not perform optimally in a few cases.

*Observations:* Additionally, Comparing HFSLM with ELISA and MELISA on a large tasks scale, HFSLM shows improvements from −0.98% to 23.33% and from −3% to 8% on makespan respectively. Besides, HFSLM shows 1.42% and 1.22% improvements in minimum resource utilization as

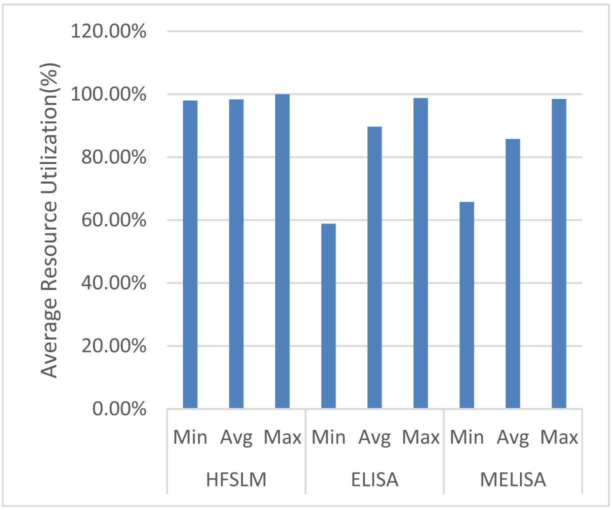

**FIGURE 20.** **Average resource utilization on varying heterogeneity.**

compared to ELISA and MELISA respectively. On maximum resource utilization, the proposed model shows improvements of 39.1% and 48.8% respectively.

## VII. CONCLUSION

In the proposed study, a Hybrid Fault-tolerant Scheduling and Load balancing Model is introduced employing neighboring-based VM to control failure in the cloud system with high computational demands. HFSLM uses a proficient task allocation strategy and distributes the arriving tasks among VMs at the arrival. In case of fault, the model uses the neighboring VMs of the faulty VM as a substitute and allocates an alternate VM to the affected task. Moreover, the proposed model escorts the whole system with an efficient load-balancing algorithm and maintains load equilibrium post-to-fault tolerance. After the implementation of the model in Python, performance evaluation was carried out by comparing HFSLM with FTHRM, MIN-MIN, MAX-MIN, and OLB on a low task scale by varying the task and VM in four different heterogeneities. The evaluations were performed based on makespan and average VM utilization. On very large task scales, the model was also contrasted with two other emerging models i.e., ELISA and MELISA.

The suggested approach outperformed other considered strategies for QoS parameters. A few reasons are listed below:

- *The proposed allocation considers both the upcoming tasks and newly added and deleted VMs. Additionally, optimal load distribution and effective average resource utilization occur simultaneously. As a result, it provides significant enhancement in all considered parameters.*
- *As can be seen from the overall results the utilization of the proposed approach remains optimized on varying the number of tasks and VMs. This is because the proposed allocation strategy focuses on distributing the arriving tasks throughout the available VMs.*

*Moreover, various strategic advancements in the proposed HFSLM play a significant role in the same.*

- *Furthermore, the proposed model outperforms all the compared approaches in HH and HL cases. It is because in high task heterogeneity the ready time of all the available VMs will always be sorted in other words, whenever we have high task heterogeneity, the ready time of all the VMs in the VM list will always be sorted. The sorted ready time of VMs is the best case for the proposed allocation.*

## DECLARATION

- **Competing Interests:**

The authors have no affiliation with any entity regarding any financial interest or non-financial interest in the material discussed in this manuscript.

## REFERENCES

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, Sep. 2013, doi: 10.1016/j.future.2013.01.010.

[2] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Softw. Eng.*, vol. SE-14, no. 2, pp. 141–154, Feb. 1988, doi: 10.1109/32.4634.

[3] A. Olteanu, F. Pop, C. Dobre, and V. Cristea, "A dynamic rescheduling algorithm for resource management in large scale dependable distributed systems," *Comput. Math. Appl.*, vol. 63, no. 9, pp. 1409–1423, May 2012, doi: 10.1016/j.camwa.2012.02.066.

[4] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions," *ACM Comput. Surveys*, vol. 53, no. 3, pp. 1–41, May 2021, doi: 10.1145/3355399.

[5] T. D. Braun, H. J. Siegel, N. Beck, L.L.Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001, doi: 10.1006/jpdc.2000.1714.

[6] J. C. Patni, M. S. Aswal, O. P. Pal, and A. Gupta, "Load balancing strategies for grid computing," in *Proc. 3rd Int. Conf. Electron. Comput. Technol.*, vol. 3. Kanyakumari, India: IEEE, Apr. 2011, pp. 239–243, doi: 10.1109/ICECTECH.2011.5941745.

[7] F. Yu and G.-S. Xueyuan, *Proceedings/The Third International Symposium on Electronic Commerce and Security Workshops (ISECS)*, Oulu, Finland: Academy, Jul. 2010.

[8] J.-T. Tsai, J.-C. Fang, and J.-H. Chou, "Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm," *Comput. Operations Res.*, vol. 40, no. 12, pp. 3045–3055, Dec. 2013, doi: 10.1016/j.cor.2013.06.012.

[9] C. Cheng, J. Li, and Y. Wang, "An energy-saving task scheduling strategy based on vacation queuing theory in cloud computing," *Tsinghua Sci. Technol.*, vol. 20, no. 1, pp. 28–39, Feb. 2015, doi: 10.1109/TST.2015.7040511.

[10] X. Zhu, L.T. Yang, H. Chen, J. Wang, S. Yin, and X. Liu, "Real-time tasks oriented energy-aware scheduling in virtualized clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 168–180, Apr. 2014, doi: 10.1109/TCC.2014.2310452.

[11] X. Liu, Y. Zha, Q. Yin, Y. Peng, and L. Qin, "Scheduling parallel jobs with tentative runs and consolidation in the cloud," *J. Syst. Softw.*, vol. 104, pp. 141–151, Jun. 2015, doi: 10.1016/j.jss.2015.03.007.

[12] M. B. Gawali and S. K. Shinde, "Task scheduling and resource allocation in cloud computing using a heuristic approach," *J. Cloud Comput.*, vol. 7, no. 1, p. 4, Dec. 2018, doi: 10.1186/s13677-018-0105-8.

[13] Z. Tong, X. Deng, H. Chen, J. Mei, and H. Liu, "QL-HEFT: A novel machine learning scheduling scheme base on cloud computing environment," *Neural Comput. Appl.*, vol. 32, no. 10, pp. 5553–5570, May 2020, doi: 10.1007/s00521-019-04118-8.

[14] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," in *Proc. 7th Heterogeneous Comput. Workshop (HCW98)*. Orlando, FL, USA: IEEE Comput. Soc, 1998, pp. 79–87, doi: 10.1109/HCW.1998.666547.

[15] P. Banerjee, "MTD-DHJS: Makespan-optimized task scheduling algorithm for cloud computing with dynamic computational time prediction," *IEEE Access*, vol. 11, pp. 105578–105618, 2023, doi: 10.1109/ACCESS.2023.3318553.

[16] K. Sharma, M. M. Hassan, S. K. Pandey, M. Saini, R. Doriya, M. K. Ojha, A. Sinha, C. Kaushal, A. K. Bairagi, and N. F. Soliman, "Cloud based multi-robot task scheduling using PMW algorithm," *IEEE Access*, vol. 11, pp. 146003–146013, 2023, doi: 10.1109/ACCESS.2023.3344459.

[17] S. A. Murad, Z. R. M. Azmi, A. J. M. Muzahid, M. M. H. Sarker, M. S. U. Miah, M. K. B. Bhuiyan, N. Rahimi, and A. K. Bairagi, "Priority based job scheduling technique that utilizes gaps to increase the efficiency of job distribution in cloud computing," *Sustain. Computing: Informat. Syst.*, vol. 41, Jan. 2024, Art. no. 100942, doi: 10.1016/j.suscom.2023.100942.

[18] S. A. Murad, Z. R. M. Azmi, A. J. M. Muzahid, M. K. B. Bhuiyan, M. Saib, N. Rahimi, N. J. Prottasha, and A. K. Bairagi, "SG-PBFS: Shortest gap-priority based fair scheduling technique for job scheduling in cloud environment," *Future Gener. Comput. Syst.*, vol. 150, pp. 232–242, Jan. 2024, doi: 10.1016/j.future.2023.09.005.

[19] G. Yao, Q. Ren, X. Li, S. Zhao, and R. Ruiz, "A hybrid fault-tolerant scheduling for deadline-constrained tasks in cloud systems," *IEEE Trans. Services Comput.*, vol. 15, no. 3, pp. 1371–1384, May 2022, doi: 10.1109/TSC.2020.2992928.

[20] K. Rezaei Kalantari, A. Ebrahimnejad, and H. Motameni, "Presenting a new fuzzy system for web service selection aimed at dynamic software rejuvenation," *Complex Intell. Syst.*, vol. 6, no. 3, pp. 697–710, Oct. 2020, doi: 10.1007/s40747-020-00168-x.

[21] M. A. Shahid, N. Islam, M. M. Alam, M. S. Mazliham, and S. Musa, "Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment," *Comput. Sci. Rev.*, vol. 40, May 2021, Art. no. 100398, doi: 10.1016/j.cosrev.2021.100398.

[22] Y. Liu, Z. Wang, and D. Zhou, "Resilient actuator fault estimation for discrete-time complex networks: A distributed approach," *IEEE Trans. Autom. Control*, vol. 66, no. 9, pp. 4214–4221, Sep. 2021, doi: 10.1109/TAC.2020.3033710.

[23] J. A. Liakath, P. Krishnadoss, and G. Natesan, "DCCWOA: A multi-heuristic fault tolerant scheduling technique for cloud computing environment," *Peer-to-Peer Netw. Appl.*, vol. 16, no. 2, pp. 785–802, Mar. 2023, doi: 10.1007/s12083-022-01445-x.

[24] G. Chen, N. Guan, K. Huang, and W. Yi, "Fault-tolerant real-time tasks scheduling with dynamic fault handling," *J. Syst. Archit.*, vol. 102, Jan. 2020, Art. no. 101688, doi: 10.1016/j.sysarc.2019.101688.

[25] A. Sheeba and B. Uma Maheswari, "An efficient fault tolerance scheme based enhanced firefly optimization for virtual machine placement in cloud computing," *Concurrency Computation: Pract. Exper.*, vol. 35, no. 7, p. e7610, Mar. 2023, doi: 10.1002/cpe.7610.

[26] S. U. Mushtaq, S. Sheikh, and A. Nain, "The response rank based fault-tolerant task scheduling for cloud system," in *Proc. 1st Int. Conf. Adv. Inform. Intell. Inf. Syst. (ICAI3S)*, in Advances in Intelligent Systems Research, vol. 181, A. P. Suryotomo and H. C. Rustamaji, Eds. Dordrecht, The Netherlands: Atlantis Press International BV, 2024, pp. 37–48, doi: 10.2991/978-94-6463-366-5_5.

[27] A. Semmoud, M. Hakem, B. Benmammar, and J. Charr, "Load balancing in cloud computing environments based on adaptive starvation threshold," *Concurrency Computation: Pract. Exper.*, vol. 32, no. 11, p. e5652, Jun. 2020, doi: 10.1002/cpe.5652.

[28] S. Negi, M. M. S. Rauthan, K. S. Vaisla, and N. Panwar, "CMODLB: An efficient load balancing approach in cloud computing environment," *J. Supercomput.*, vol. 77, no. 8, pp. 8787–8839, Aug. 2021, doi: 10.1007/s11227-020-03601-7.

[29] M. Mani and L. Kaur, "Efficient enhanced sleep awake scheduling using fuzzy logic and neural networks : A review," *Int. J. Scientific Res. Sci., Eng. Technol.*, vol. 8, pp. 18–24, Mar. 2021, doi: 10.32628/ijsrset21828.

[30] A. H. El-Gamal, R. R. Mostafa, and N. A. Hikal, "Load balancing enhanced technique for static task scheduling in cloud computing environments," in *Internet of Things—Applications and Future* (Lecture Notes in Networks and Systems), vol. 114, A. Z. Ghalwash, N. E. Khameesy, D. A. Magdi, and A. Joshi, Eds. Singapore: Springer 2020, pp. 411–430, doi: 10.1007/978-981-15-3075-3_28.

[31] H. Mahmoud, M. Thabet, M. H. Khafagy, and F. A. Omara, "An efficient load balancing technique for task scheduling in heterogeneous cloud environment," *Cluster Comput.*, vol. 24, no. 4, pp. 3405–3419, Dec. 2021, doi: 10.1007/s10586-021-03334-z.

[32] S. Shiekh, M. Shahid, M. Sambare, R. A. Haidri, and D. K. Yadav, "A load-balanced hybrid heuristic for allocation of batch of tasks in cloud computing environment," *Int. J. Pervasive Comput. Commun.*, vol. 19, no. 5, pp. 756–781, Nov. 2023, doi: 10.1108/ijpcc-06-2022-0220.

[33] A. N. Singh and S. Prakash, "WAMLB: Weighted active monitoring load balancing in cloud computing," in *Big Data Analytics* (Advances in Intelligent Systems and Computing), vol. 654, V.B. Aggarwal, V. Bhatnagar, and D. K. Mishra, Eds. Singapore: Springer, 2018, pp. 677–685, doi: 10.1007/978-981-10-6620-7_65.

[34] S.-C. Wang, K.-Q. Yan, S.-S. Wang, and C.-W. Chen, "A three-phases scheduling in a hierarchical cloud computing network," in *Proc. 3rd Int. Conf. Commun. Mobile Comput.* Qingdao, China: IEEE, Apr. 2011, pp. 114–117, doi: 10.1109/CMC.2011.28.

[35] L. Anand, D. Ghose, and V. Mani, "ELISA: An estimated load information scheduling algorithm for distributed computing systems," *Comput. Math. Appl.*, vol. 37, no. 8, pp. 57–85, Apr. 1999, doi: 10.1016/s0898-1221(99)00101-7.

[36] R. Shah, B. Veeravalli, and M. Misra, "On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 12, pp. 1675–1686, Dec. 2007, doi: 10.1109/TPDS.2007.1115.

[37] S. Sheikh, M. Shahid, M. Sambare, R. A. Haidri, and S. Prakash, "A load distribution based resource allocation strategy for bag of tasks (BoT) in computational grid environment," *Wireless Pers. Commun.*, vol. 135, no. 1, pp. 47–80, Apr. 2024, doi: 10.1007/s11277-024-10951-5.

[38] B. Radojević and M. žagar, "Analysis of issues with load balancing algorithms in hosted (cloud) environments," in *Proc. 34th Int. Conv. MIPRO*, 2011, pp. 416–420.

[39] M. Rahman, R. Ranjan, R. Buyya, and B. Benatallah, "A taxonomy and survey on autonomic management of applications in grid computing environments," *Concurrency Computation: Pract. Exper.*, vol. 23, no. 16, pp. 1990–2019, Nov. 2011, doi: 10.1002/cpe.1734.

[40] D. Poola, M. A. Salehi, K. Ramamohanarao, and R. Buyya, "A taxonomy and survey of fault-tolerant workflow management systems in cloud and distributed computing environments," in *Software Architecture for Big Data and the Cloud*. Amsterdam, The Netherlands: Elsevier, 2017, pp. 285–320, doi: 10.1016/B978-0-12-805467-3.00015-6.

[41] F. G. Khan, K. Qureshi, and B. Nazir, "Performance evaluation of fault tolerance techniques in grid computing system," *Comput. Electr. Eng.*, vol. 36, no. 6, pp. 1110–1122, Nov. 2010, doi: 10.1016/j.compeleceng.2010.04.004.

[42] S. Sheikh, A. Nagaraju, and M. Shahid, "A fault-tolerant hybrid resource allocation model for dynamic computational grid," *J. Comput. Sci.*, vol. 48, Jan. 2021, Art. no. 101268, doi: 10.1016/j.jocs.2020.101268.

[43] T. Tamilvizhi and B. Parvathavarthini, "A novel method for adaptive fault tolerance during load balancing in cloud computing," *Cluster Comput.*, vol. 22, no. S5, pp. 10425–10438, Sep. 2019, doi: 10.1007/s10586-017-1038-6.

[44] S. M. A. Attallah, M. B. Fayek, S. M. Nassar, and E. E. Hemayed, "Proactive load balancing fault tolerance algorithm in cloud computing," *Concurrency Comput., Pract. Exper.*, vol. 33, no. 10, p. e6172, May 2021, doi: 10.1002/cpe.6172.

[45] T. Mohmmed and N. Abdalrahman, "A load balancing with fault tolerance algorithm for cloud computing," in *Proc. Int. Conf. Comput., Control, Electr., Electron. Eng. (ICCCEEE)*. Khartoum, Sudan: IEEE, Feb. 2021, pp. 1–6, doi: 10.1109/ICCCEEE49695.2021.9429597.

[46] M. A. Shahid, N. Islam, M. M. Alam, M. M. Su'ud, and S. Musa, "A comprehensive study of load balancing approaches in the cloud computing environment and a novel fault tolerance approach," *IEEE Access*, vol. 8, pp. 130500–130526, 2020, doi: 10.1109/ACCESS.2020.3009184.

**SHEIKH UMAR MUSHTAQ** received the B.C.A. and M.C.A. degrees in computer applications from the University of Kashmir, Srinagar, India. He is currently a Research Scholar of computer application with Lovely Professional University, Phagwara, Punjab. He has presented and published various conferences with the best presentation awards. His research interests include scheduling, load balancing, fault tolerance, and cloud computing.

**SOPHIYA SHEIKH** received the B.Sc. degree in information technology from Maharshi Dayanand Saraswati University, Ajmer, India, the master's degree in computer application from Rajasthan Technical University, Kota, and the Ph.D. degree from the Department of Computer Science, Central University of Rajasthan, India. She is currently an Associate Professor with Lovely Professional University, Phagwara, Punjab, India. She regularly writes articles and research papers in reputed national and international magazines and journals. Her research interests include grid/distributed computing and cloud computing. She is an editor of various books. She has organized various international conferences and workshops. She is a Potential Reviewer in various reputed journals, such as IEEE Systems Journal, *Cluster Computing*, *Scientific Reports*, *Journal of Supercomputing*, *Journal of Cloud Computing*, and *Concurrency and Computation*.

**SHEIKH MOHAMMAD IDREES** received the Ph.D. degree in computer science. He is currently a Researcher with the Decentralized Systems Engineering Laboratory, Department of Computer Science, Norwegian University of Science and Technology (NTNU), Norway. He is also a Computer Science Researcher and an Educator. With a steadfast dedication to advancing computer science, he has made significant contributions to the academic community throughout his career. Renowned for his expertise in blockchain, DeFi, DLT, fintech, data analytics, cloud computing, and machine learning, he has authored numerous papers in esteemed journals and conferences. Moreover, he frequently contributes as a book editor for leading publishing units, such as Springer, Taylor, and Francis. He was a recipient of the ERCIM Postdoctoral Research Fellowship by the European Research Consortium for Informatics and Mathematics at NTNU, Norway.

Open Access funding provided by 'Norwegian University of Science and Technology' within the CRUI CARE Agreement