# Final Report: Modified Min-Min Heuristic for Workflow Scheduling in Cloud Computing

Muneeba Badar and Shifa Shah

April 2025

## 1 Background and Motivation

In the context of workflow scheduling, load balancing plays a crucial role in ensuring optimal resource utilization and minimizing execution time. The challenge of managing server overload when handling numerous user requests, leading to performance bottlenecks and delayed response times, is commonly known as the load balancing problem. To address this, load balancing techniques are employed to distribute the task load across processing units, preventing the overloading or underutilization of machines, thereby improving throughput, response time, and overall makespan.
Numerous scheduling algorithms exist, including the original Min-Min heuristic, which is effective for independent tasks. However, the original Min-Min heuristic overlooks the distribution of load among resources, which can lead to imbalanced resource utilization and prolonged execution times. This presents a research gap for algorithms that can dynamically adjust to fluctuating workloads, consider heterogeneous environments, optimize multiple Quality of Service (QoS) factors, and ensure efficient load balancing in CCE.
To bridge this gap, the paper proposes a modified Min-Min heuristic that integrates load-balancing principles. This approach seeks to achieve optimal load distribution and minimize overall execution time by considering both task completion times and resource loads. The primary objective of workflow scheduling is to minimize the makespan, or the total execution time of the workflow. The proposed method is particularly effective for workflows with dependent tasks (represented as DAGs) and adheres to precedence constraints. Its significance lies in its ability to handle real-world workflows—such as scientific computations (e.g., Montage)—with enhanced performance compared to existing methods like HEFT and PETS. This approach ensures more efficient use of resources, offering a promising solution for real-world applications in cloud computing environments.

## 2 Algorithm Overview

### 2.1 Core idea

The paper proposes a modified Min-Min heuristic for workflow scheduling in cloud computing environments, introducing two key algorithmic innovations:

- Load-Balanced Task Scheduling: The modified version of the algorithm incorporates load-balancing principles by considering both task execution time and current resource load, ensuring more equitable distribution of tasks across virtual machines (VMs).
- Handling Dependent Tasks and Precedence Constraints (PC): The proposed algorithm handles dependent tasks in workflows modeled as Directed Acyclic Graphs (DAGs). It

duplicates the entry task across all VMs to reduce communication delays between dependent tasks, improving makespan. Task prioritization accounts for precedence constraints (PC), ensuring dependencies are respected during scheduling

The modified Min-Min heuristic enhances the traditional Min-Min approach by incorporating task dependencies, load balancing, and entry task duplication. It operates in two phases:

- **Phase 1: Task Selection.** Task Selection based on Priority and Precedence Constraints (PC): Instead of simply selecting the task with the overall minimum completion time, the modified approach calculates a priority for each task. This priority considers the minimum Estimated Completion Time (ECT) of the task across all available virtual machines while ensuring that the precedence constraints of the workflow (dependencies between tasks) are satisfied. Tasks whose predecessors have been completed are eligible for selection and are placed into a priority queue (PQ).
- **Phase 2: Resource Allocation.** Resource Selection with Entry Task Duplication: Tasks are dequeued from the priority queue based on their assigned priority. For entry task, the proposed algorithm duplicates the entry task and allocates it to all available virtual machines to reduce the communication time between the entry task and its immediate successor tasks. For subsequent tasks, the algorithm determines the Earliest Start Time (EST) and Earliest Finish Time (EFT) on each virtual machine, considering the completion times of their predecessors and the communication times. The task is then allocated to the virtual machine that results in the earliest finish time, aiming to minimize the overall makespan.

**Inputs:** A DAG representing the workflow, an ECT matrix for tasks on VMs, and VM configurations.
**Outputs:** Makespan, load balancing metrics, speedup, efficiency, and resource utilization.
The main idea is to balance task completion time and resource load while minimizing communication delays through strategic task duplication.

## 3 Implementation Summary

The implementation, developed in Python, consists of modular components:

- **Workflow Parser:** Reads DAGs, dependencies, and ECT tables from input files.
- **Priority Queue:** Manages task selection based on priority and precedence.
- **Scheduler:** Allocates tasks to VMs, duplicating the entry task and computing EST/EFT for others.
- **Metrics Calculator:** Computes QoS metrics (makespan, speedup, efficiency, load balancing, resource utilization).

The implementation strategy prioritized modularity for easy updates and reproducibility, using file-based inputs/outputs. Key challenges included:

- **Dependency Management:** Ensured precedence constraints via explicit checks before task scheduling.
- **Communication Delays:** Incorporated inter-VM communication times in EFT calculations.
- **Numerical Precision:** Used double-precision arithmetic to mitigate floating-point errors.

The implementation adheres to the original algorithm but includes enhancements, such as support for additional datasets and parameter tuning, as described later.

# 4 Evaluation

## 4.1 Correctness

The implementation was validated using Montage and random DAG workflows (10 to 2000 tasks) and a 10-task example from the paper. For the following the workflow exmaple:

- **Tasks:** J1 to J10
- **Entry Task:** J1
- **Exit Task:** J10
- **Cloud Infrastructure:** 2 servers (R1, R2) with 3 virtual machines (R1_V1, R1_V2, R2_V3).
- **ECT Table:** Each task has distinct execution times on each VM to reflect heterogeneity.
- **Communication Delays:** Applied between dependent tasks based on VM-to-VM communication across servers.

Testing included:

1. **Dependency Verification:** Each task was only scheduled once all its dependencies were satisfied. For example, J5 was scheduled only after both J2 and J3 were completed, and J10 followed a complete chain from J1 to J9.
2. **Entry Task Duplication:** J1, as the entry task, was correctly duplicated across all three VMs (R1_V1, R1_V2, R2_V3), consistent with the algorithm's goal to reduce communication latency in subsequent tasks.
3. **EST/EFT Validation:** EST and EFT values were manually verified. For example:
   - J3 on R1_V1 had an EST of 14 (after J1) and EFT of 25, considering the execution time and any necessary delays.
   - J10 on R1_V2 started at EST 63 and finished at EFT 70.
4. **Correct Resource Allocation:**
   - R1_V1 was assigned tasks J1, J3, J6, J2, J5, J7, J8 in an order consistent with precedence and execution availability.
   - J9 and J10 were allocated to R1_V2 for faster makespan.
5. **QoS Metrics Validation:**
   - Makespan = 70
   - Load Balancing = 0.605
   - Speedup = 1.3; Efficiency = 43.3%
   - Resource Utilization = 0.604

All tests confirmed correct scheduling and metric calculations, aligning with the paper's results.

### 4.2 Runtime & Complexity

**Theoretical Complexity:** Each of $n$ tasks is evaluated across $m$ VMs, yielding $\mathcal{O}(n \cdot m)$ per scheduling round. Priority queue operations add minimal overhead.
**Empirical Performance:** On an i5 CPU with 8GB RAM, scheduling 100–500 tasks across 20–25 VMs completed in under 1 second, consistent with the paper's scalability claims. Larger workflows (400 tasks) showed slight delays due to dependency checks, but performance remained practical. For 1000 and 2000 tasks dataset, the runtime increased significantly. Empirical time for 1000 tasks dataset was 4.7 seconds and for 2000 tasks dataset was 31.9 seconds.

### 4.3 Comparisons

The modified Min-Min heuristic was compared against HEFT and PETS using Montage and random workflows.

- **Makespan:** 5.09% better than HEFT/PETS.
- **Load Balancing:** Improved resource utilization (0.604 vs. 0.55 for HEFT).
- **Speedup/Efficiency:** Higher due to reduced communication overhead via entry task duplication.

The traditional Min-Min heuristic, tested as a baseline, showed poorer load balancing and longer makespans due to its lack of dependency handling.

## 5 Enhancements

Beyond the original implementation, we introduced:

- **New Datasets:** New Datasets: We additionally tested on CyberShake workflows (seismic hazard analysis) featuring 100–400 tasks, which are characterized by denser dependency structures compared to Montage. To further validate robustness, we also evaluated on randomly generated workflows with 1000 and 2000 tasks, as well as specifically designed edge case workflows to stress-test performance under extreme conditions.
- **Modular and Reproducible Code Structure:** The code is implemented in separate parts, each handling a specific task, making it easy to manage and update. It uses input and output files, so the process can be repeated with consistent results, and the structure allows easy modifications without affecting the whole system
- **Structured Output:** Output structured to include task allocation, EST/EFT timings, and detailed metrics.

**Motivation:**

- **New Datasets:** We wanted to ensure that our system is robust and adaptable to different types of workflows. By testing on CyberShake workflows, which involve seismic hazard analysis and feature more complex dependency structures, we can better understand how well the system performs with varying levels of task complexity. Additionally, by testing with larger, randomly generated workflows and edge cases, we aimed to push the limits of our system and verify that it can handle large-scale and extreme conditions without performance degradation.
- **Modular and Reproducible Code Structure:** We designed the code in a modular way to make it easier to manage and update. Each part of the code is responsible for a specific task, so if any part needs improvement or adjustment, it can be done without

affecting the rest of the system. Additionally, using input and output files ensures that the process can be repeated consistently, making the system more reliable and reproducible for different use cases or further development.

- **Structured Output:** Providing output in a clear, structured format allows for better tracking and analysis of the workflow scheduling process. By including task allocation, EST (Earliest Start Time) and EFT (Earliest Finish Time) timings, and detailed performance metrics, we make it easier to evaluate the effectiveness of the scheduling algorithm and identify any potential areas for improvement.

# 6 Reflection

**Challenges:** The complexity of managing task dependencies and communication delays required thoughtful data structure choices, such as priority queues and ECT matrices, to maintain efficiency. While we addressed numerical precision issues, they still remained a concern when handling larger workflows. For workflows involving 500+ tasks, scalability became an issue, requiring further optimization to avoid potential memory bottlenecks.
**Learning Outcomes:** This project helped deepen our understanding of workflow scheduling and DAG processing. Also, gained expertise in managing cloud system constraints and learned how to balance performance with system limitations. Improved skills in Python modularity, testing, and performance analysis.
**Future Work:** Explore adaptive duplication strategies based on runtime VM loads, integrate machine learning for priority prediction, and test on larger real-world datasets (e.g., LIGO workflows). Energy-aware scheduling could be further refined to meet green computing goals.

# 7 References


- Choudhary, A., Rajak, R. (2024). A novel strategy for deterministic workflow scheduling with load balancing using modified min-min heuristic in cloud computing environment. *Cluster Computing*, 27, 6985–7006. Link

- Subhadeep Choudhuri. (2023). Min-Min Algorithm in Grid Computing with Code in C. Medium Article