Vol. 8, No. 2, April 2026

Journal of Information Systems and Informatics

Published By Asosiasi Doktor Sistem Informasi Indonesia

Dependency- and Trust-Aware Task Scheduling Framework for Efficient Internet of Things Edge Systems

Fulufhelo Hopewell Mamidza[1], Bassey Isong[2]

[1,2]Computer Science Department, North-West University, Mafikeng, South Africa

Received:
October 3, 2025
Revised:
March 10, 2026
Accepted:
April 1, 2026
Published:
April 26, 2026

Corresponding Author:
Author Name*:
Bassey Isong
Email*:
bassey.isong@nwu.ac.za

DOI:
10.63158/journalisi.v8i2.1489

Abstract. The rapid growth of the Internet of Things (IoT) has significantly increased the number of connected devices, generating massive volumes of data and placing substantial demands on edge and fog computing infrastructures. Traditional resource management approaches often overlook task dependencies, which can lead to inefficient resource utilization, increased execution delays, reduced reliability, and potential security risks in distributed IoT environments. To address these challenges, this paper proposes an improved dependency-aware task scheduling framework designed to operate between edge devices and edge servers. The framework employs directed acyclic graph (DAG) modeling to represent task dependencies and execution order, trust-aware node selection to avoid malicious, overloaded, or unreliable nodes, and Particle Swarm Optimization (PSO) to support adaptive resource allocation under dynamic and heterogeneous workloads. Experimental results demonstrate that the proposed framework achieves an average latency of 50 ms, throughput of approximately 500 transactions per second (tps), and a task completion rate of 98%. These findings indicate that the proposed approach outperforms conventional scheduling methods by improving latency, throughput, reliability, security, and overall task execution efficiency in IoT-enabled edge computing environments.

Keywords: Internet of Things, Edge computing, DAG-based Scheduling, Trust-Aware Scheduling, Particle Swarm Optimization

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

1. INTRODUCTION

With the rise of the Internet of Things (IoT), the number of connected smart devices is increasing. In fact, these devices work together to improve ease and functionality, as well as overall sustainability, in home environments, health-care settings (HCOs), transport, and more. Consequently, they generate huge amounts of data [1]. Traditionally, IoT-generated data has been processed on centralized cloud servers. However, due to its centralized architecture, several problems, including high latency, high network bandwidth, and privacy challenges, are faced, which restrict real-time responsiveness and reliability [2], [3].

To overcome these limitations, Edge computing has emerged as a paradigm that brings computation closer to IoT devices. Reducing communication distance improves latency and processing efficiency while enabling scalable system operation [4]. It also mitigates resource constraints on IoT devices by offloading computationally intensive tasks to nearby edge nodes, resulting in more stable, scalable systems [5],[6]. Consequently, numerous task scheduling algorithms have been proposed to optimize resource utilization in edge environments [7], [8].

Traditional resource allocation and task scheduling in IoT-edge-cloud computing often rely on static or simple heuristics, which are poorly suited to dynamic conditions [9]. In recent years, intelligent optimization models have been proposed to adapt to environmental changes, such as fuzzy logic–based scheduling [10] and genetic algorithms [11]. However, they often overlook task dependencies or have slow convergence, making them unsuitable for real-time processing. Deep reinforcement learning (DRL)-based approaches [12] can adapt to network dynamics but are computationally intensive.

Furthermore, the growing volume of data in edge computing motivates the use of DAG-based modelling. DAG-based methods [13-16] improve task execution by representing dependencies and deadlines, with some incorporating DRL or UAV optimization. However, most rely on static assumptions, ignore pipeline parallelism, or incur high computational overhead. Dependency-aware approaches further address scheduling challenges in heterogeneous and time-varying environments. Workflow and edge–cloud schedulers [17, 18] improve execution efficiency and service quality but often handle dependencies in a

largely static or centralized manner. Equally, fog–edge–cloud methods [22-25] employ dynamic queues, hybrid architectures, or learning-based strategies, achieving reductions in latency and response time, improved throughput, and higher task success rates. Despite these advances, trade-offs remain in complexity, scalability, and resource balancing. Table 1 summarizes existing approaches, showing their dependency awareness, adaptability, and limitations.

Table 1. Summary of existing task dependency scheduling approaches

Ref.	Approach/ Technique	Dependency Awareness	Adaptivity to Dynamic Environment	Limitations
[10]	Fuzzy logic–based task scheduling	No	Moderate	Does not consider task interdependencies
[11]	Genetic algorithm–based scheduling	Partial	Low	Slow convergence; unsuitable for real-time processing
[12]	DRL-based scheduling	Partial	High	High computational complexity; difficult to deploy on IoT devices
[13]	DAG-based dependency & deadline optimization	Yes	Moderate	Limited scalability; static modelling; does not parallel dependencies
[14]	DRL-based DAG scheduling with UAV optimization	Yes	High	High computational overhead; complex deployment
[15]	DAG-based heuristic scheduling	Yes (Static)	Low	Assumes homogeneous dependencies; ignores data dependency and parallelism
[16]	Multi-dependency workflow scheduling	Yes	Moderate	Limited parallelism and resource balancing
[17]	Predictive Pareto-based edge–cloud scheduling	Partial	Moderate	Dependency handling is largely static
[18]	Edge–cloud scheduling with prediction & optimization	Partial	High	Dependency handling is not fully dynamic or distributed
[22]	Dynamic fog scheduling with queue management	Yes	High	Focused on latency reduction; moderate throughput trade-offs
[23]	Hybrid edge–cloud scheduling (DSOTS + TSGS)	Yes	High	Cost and SLA improvements may increase computational complexity
[24]	PSTBA for healthcare IoT	Yes	Moderate	Specific to healthcare IoT, may not generalize to other domains
[25]	EASA-MORU with Dung Beetle Optimization	Yes	Moderate	Optimized for 500 requests; high computational cost

This study addresses the lack of a balanced solution that simultaneously ensures efficiency, flexibility, and reliability in existing task-scheduling approaches. To bridge this

gap, it proposes a unified, dependency- and trust-aware scheduling framework that integrates DAG-based dependency modeling, PSO-based resource allocation, and dynamic trust-aware node selection. The primary objective is to design and evaluate an integrated scheduling mechanism that improves latency, throughput, and robustness in large-scale IoT-edge environments.

2. METHODS

This section outlines the proposed framework and its design. We followed the design science research (DSR) approach to develop and evaluate DTINT. The rationale is that it involves creating artifacts to solve real-world problems and support the mixed-method approach [29], [30]. This study used a mixed-methods approach, combining both qualitative and quantitative data collection methods. The quantitative method involved empirically simulating various scheduling algorithms (e.g., through simulation), enabling important factors to be measured and their relationships tested. The qualitative method focused on careful observation and analysis of phenomena in the situation, drawing on insights from literature reviews.

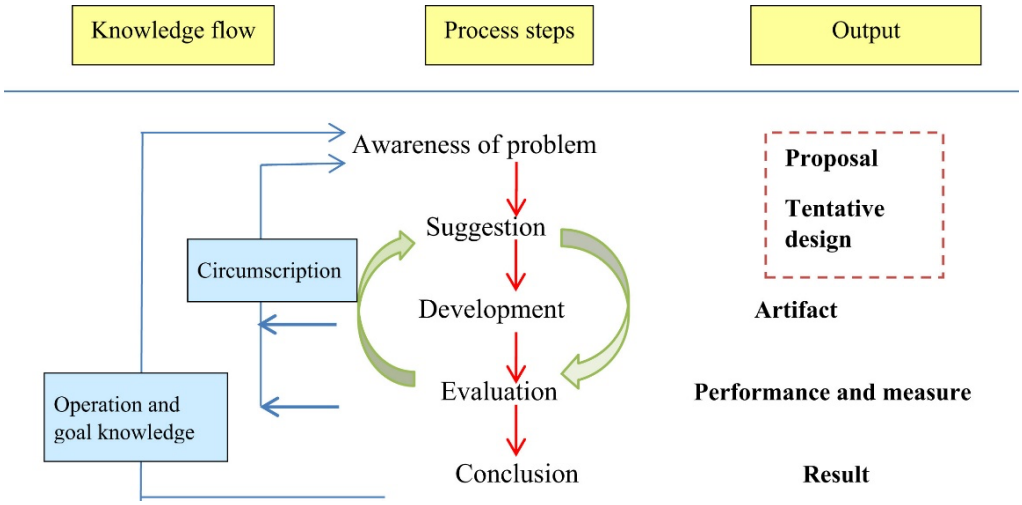

Figure 1. Design Science Methodology

As shown in Figure 1, the process started with a problem formulation to identify the research gap and develop the research questions. An empirical evaluation of existing algorithms was conducted, and a literature review was used to examine suggested solutions and validate their selection. Insights from the review and comparative study

guided the development of a solution model, which was implemented as a proof-of-concept to demonstrate feasibility. Numerous simulations were conducted to evaluate the suitability and performance of the artifact. Data obtained from these simulations were subsequently analysed and validated to ensure reliability. The process concludes by highlighting the contributions in research and summarizing the results of this comprehensive evaluative stage. This study follows a structured step-by-step workflow:

1) Problem Formulation:

   We define the dependency-aware task scheduling problem in IoT-edge environments, considering task dependencies, resource collaboration, trust, and dynamic workloads.

2) Suggestion

   We employed a literature review to identify the most effective scheduling algorithm and conducted a comparative analysis to evaluate their performance. The findings from this study were subsequently utilized to inform the design of our model.

3) Framework Design:

   The DTINT architecture was designed to integrate DAG-based dependency modelling, adaptive batching, trust-aware node selection, and PSO-based optimization. It includes three main algorithms: DAG construction and dependency analysis, batch processing and parallel execution, and trust-aware task allocation with retry and fault tolerance.

4) Simulation Setup:

   In a simulated IoT-edge environment with collaborative nodes, varying workloads, and dynamic task arrival rates, the proposed algorithms process tasks with adaptive load balancing and trust-aware decision-making.

5) Results and discussion

   System performance is evaluated using throughput, latency, trust score, and task success rate, compared against the baseline approach. PSO is selected due to its fast convergence, low computational complexity, and effectiveness in dynamic environments. Compared to Genetic Algorithms and Deep Reinforcement Learning, PSO offers a balance between efficiency and adaptability, making it suitable for real-time edge scheduling.

2.1. Proposed System Overview

Figure 2 illustrates the overall architecture of the proposed DTINT framework. The framework addresses task dependencies, resource collaboration, load balancing, and trust-aware node selection through modelling, design, and scheduling. DTINT operates in an IoT-edge system with resource-constrained devices offloading computation to nearby edge servers. It enables collaborative execution across edge nodes, considering task dependencies and workloads.

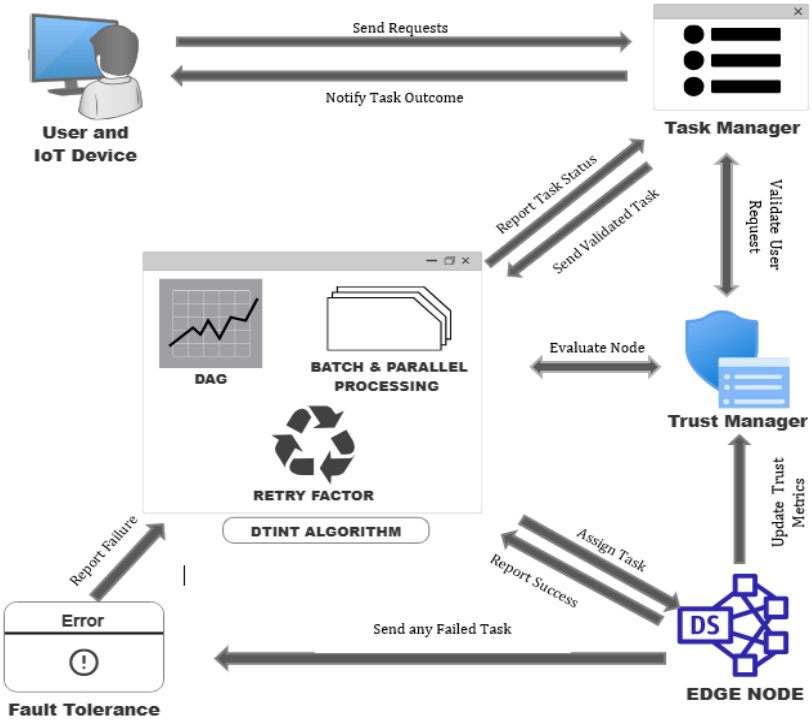

Figure 2. Proposed scheduling framework

Users request via IoT devices, and the Task Manager (TM) splits applications into DAG tasks based on priority. Tasks are organized into a queue by dependencies and sent to the DTINT module, which uses a dependency-aware PSO to reduce latency and energy use. A lightweight heuristic retains PSO's greedy, adaptive behaviour, removing metaheuristic overhead. Our scheduler combines PSO's fitness-based learning with trust-based filtering and retries, using DAG topological sorting for correct execution. This balances performance and explainability in edge environments, even with potentially malicious nodes.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

2.2. Application Model

We consider an IoT application composed of a set of computational tasks $\{v_1, v_2, \ldots, v_j\}$ to be executed in edge systems. To accurately reflect real-world task dependencies, workloads were modeled using Directed Acyclic Graphs (DAGs). Each DAG represented an application composed of interdependent tasks with precedence constraints. The generated DAGs exhibited controlled structural properties, including average chain depth, parallelism level, and dependency patterns, as shown in Figure 3. A topological order will be utilized to create a dependency model.

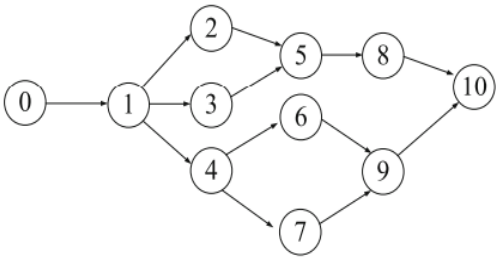

Figure 3: A dependency model

A topological order $G = (V, E)$ *where: V is the set of tasks ND E is the set of directed edges representing dependencies*

A topological order ensures that if $(v_i, v_j) \in E$, *then* the task v_i needs to be processed before the task v_j. The link weight $D_{weight} \geq 0$ represents the amount of data to be produced from v_i to v_j. In each DAG, a task with 0 indegree is called a User Request (e.g., "User and IoT Device" in Figure 2). Typically, IoT applications receive initial data for each user's request (UR), and their computation results are the DTINT algorithm's combined outputs. When an IoT device offloads an UR to a particular edge server, it transfers the initial data to that server and requires the application's computation result to be returned. To model such a requirement, we add a dummy entry task. v_0 that connects to each actual entry task with a link whose weight is set as the initial data amount and a dummy final task (v_{j+1}) pointed by each actual exit task with a link weight of the output data amount. The processing time and computation resource required by the dummy tasks are set to 0, and the two dummy tasks should be placed on the same edge server. Each task exactly maps to one function that needs to be configured on servers to execute it. Without ambiguity, we denote both the task and its corresponding function as $v_{i,}$ $i \in$

$\{0,1,\ldots,n+1\}$. In addition, we adopt the DAG G of the tasks in an application to denote the application itself.

Edge System: We consider an edge system K with one collaborative edge server, denoted as $S = \{S_1, S_2, \ldots, S_j\}$. The data rate between S_i and S_j is $d_{i,j}$. For each edge node N with limited resources, the system must dynamically manage them to accommodate changes in tasks and load variations. In contrast to previous studies [19], which addressed tasks sequentially, our methodology facilitates the concurrent processing of multiple tasks via batch processing as follows:

$$B_i = \left\lceil \frac{|V|}{W} \right\rceil \quad (1)$$

where V = total number of tasks and W = number of parallel processors

To address resource limitations and load imbalance, multiple edge computing (EC) nodes can work together through various strategies:

1) Task Migration

When a node lacks sufficient resources to manage its current task load, it can transfer some tasks to other nodes for execution. This transfer occurs over network bandwidth, necessitating careful consideration of network latency and bandwidth limitations. Unlike previous studies that utilize fixed migration thresholds, our approach employs an adaptive threshold:

$$\emptyset_p(t) = \int (B(t), H(t), \lambda(t)) \quad (2)$$

where $\lambda(t)$ = current task arrival rate

which dynamically adjusts based on network variance σ(t), enabling better load balancing. The decision to migrate is usually based on the current node's load (e.g., task queue length, computational load), network latency and bandwidth (the overhead of task migration), and the target node's resources (e.g., computational power, storage capacity).

2) Parallel Task

When the computational demand of a task is excessively high, a node can divide the task into several subtasks and distribute these subtasks among other nodes for concurrent processing. This strategy can enhance computational efficiency and optimize resource

use, though it is necessary to account for dependencies between subtasks. Assume the task $v_{i,}$ is divided into P subtasks, where the computational requirement of each subtask is represented as C_i, allowing each subtask to be processed in parallel.

$$k_i = \sum_{p=0}^{p_i} C_i, n \quad (3)$$

The time required for the execution of each subtask is as follows:

$$\mathrm{V}_{i,np} = \frac{C_{i,n}}{P_j} \quad (4)$$

In designing a task-splitting strategy, it is crucial to address the challenges of allocating subtasks to nodes appropriately and managing inter-task dependencies effectively. Collaborative Strategy: Relationship among nodes depends not only on computing power and storage capabilities but also on network bandwidth and latency. As the system scales, communication costs between nodes can become a significant performance factor. Consequently, strategies for inter-node communication must account for bandwidth and latency to reduce the time overhead associated with task migration and data transfer.

Load Balancing: To prevent the overburdening of specific nodes, the system must maintain an equitable distribution of load across all nodes through effective task allocation and resource scheduling. Load-balancing techniques may include task migration and task allocation optimization. Suppose the loads of the node. $N_{j,k}$ are: $Q_t(t)$and $Q_k(t)$. The objective of load balancing is to minimize the load difference between nodes. Its optimization objective can be expressed as:

Consider nodes $N_{j,k}$ with respective loads denoted as, $Q_t(t)$and $Q_k(t)$. Load balancing aims to minimize the load imbalance across these nodes. This optimization objective can be articulated as:

$$min \sum_{p=0}^{p} \left(Q_t(t) - \frac{1}{p} \sum_{k=0}^{p} Q_k(t) \right)^2 \quad (5)$$

The objective of this optimization is to align each node's load as closely as possible with the global load average, thereby improving the overall system's resource utilization efficiency.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

2.3. Problem formulation

Consider an IoT-edge computing environment comprising a set of IoT users, a Task Manager, a Trust Manager, and a set of collaborative edge nodes $E = \{e_1, e_2, \ldots, e_n\}$. An application is modelled as a DAG, $G = (v_i, v_j)$ where v_i represents tasks and v_j denotes precedence constraints. Each task must be executed on a trusted, available edge node, subject to resource capacity limits and dependency constraints. Independent tasks may be processed in parallel, while dependent tasks must follow the DAG order. Node reliability is assessed through dynamic trust metrics, and failed tasks are reassigned via a retry-based fault-tolerance mechanism. The task dependency scheduling problem is to jointly determine task-to-node assignments, dependency-aware execution orders, parallel batching of independent tasks, and the reallocation of failed tasks.

2.4. Trust Score

The trust score mechanism is explicitly implemented to ensure secure and reliable node participation. Each edge node is assigned a dynamic trust value computed from historical performance metrics, including task completion success rate, execution delay, failure frequency, and response consistency. A weighted-aggregation model is used to iteratively update trust scores, and nodes with trust values below a predefined threshold are excluded from task allocation, as shown in Figure 4.

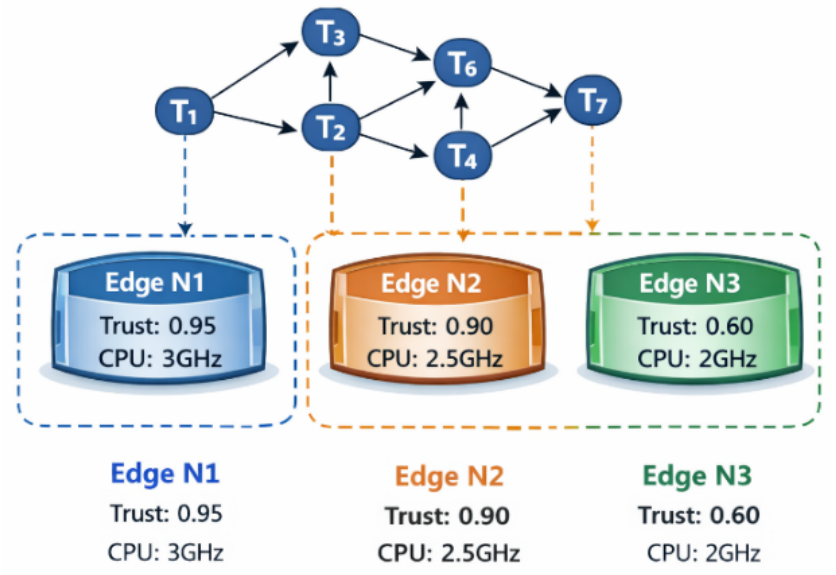

Figure 4. DAG diagram integrated into the Edge Nodes Cluster

2.5. Dependency-trust aware algorithms

We present a fully dependent, trust-aware scheduling algorithm for task offloading that minimizes latency and maximizes throughput while satisfying dependencies and priority constraints. Figure 5 presents the activity diagram of the proposed scheduling. Tasks are organized into multiple queues based on readiness and deadlines. Applications may be delay-tolerant or dependency-constrained; therefore, task priority is derived from the application dependencies and deadline. For tasks within the same application, execution is governed by three factors: task dependencies, priority, and deadlines. A task can only start after all its predecessor tasks are completed, and tasks with earlier deadlines or higher priority are scheduled first.

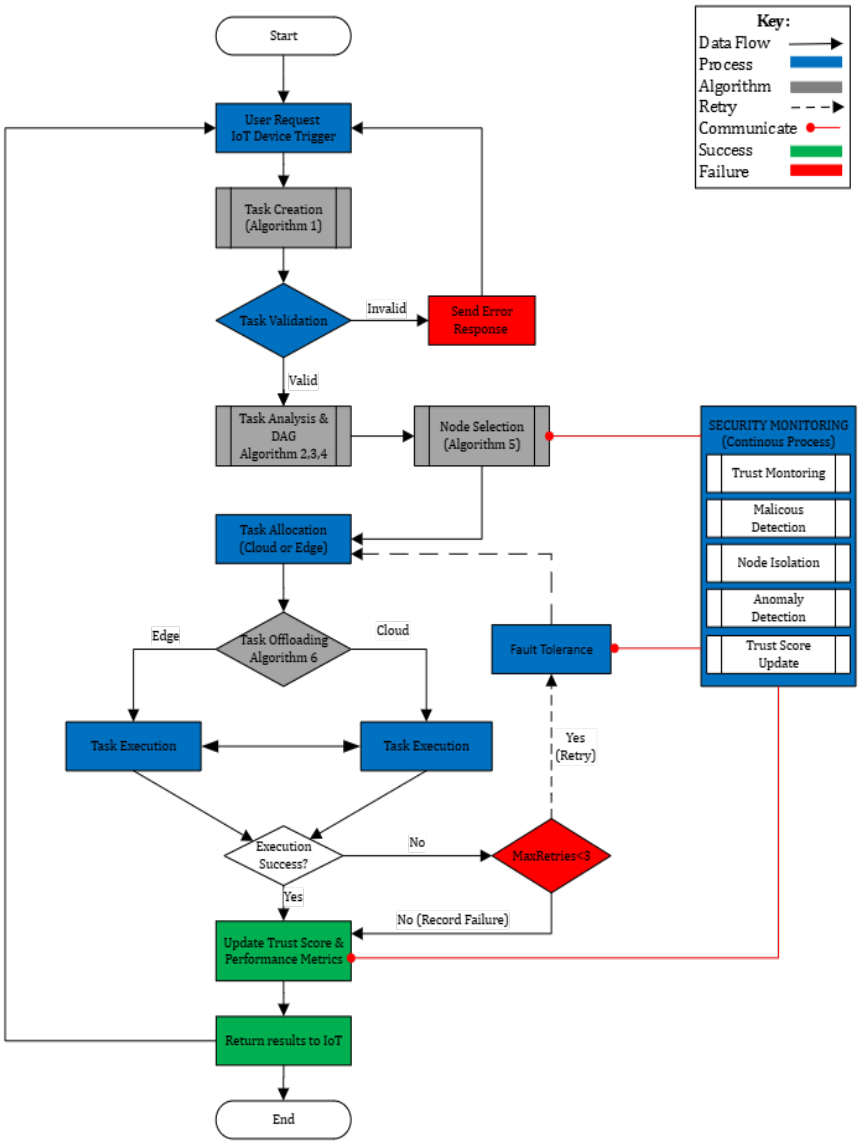

Figure 5. Task Dependency Scheduling Process

For each task $V_i \in V$ with predecessor set $pred(T_i)$, the start time must satisfy:

$$S_i \geq \max_{V_j \in pred(V_i)} C_j \quad (6)$$

where S_i and C_j denote the start and completion times, respectively.

Task priority is determined using its deadline (latest completion time, LCT_i), defined as

$$LCT_i = \min_{T_k \in suc(T_i)} (C_k - ET_k) \quad (7)$$

where $suc(T_i)$represents successor tasks, ET_k is is the execution time.

Tasks with smaller LCT_i values are assigned to a higher priority.

Figure 6 shows the algorithm for constructing a DAG from the Tasks Queue, with tasks as nodes and the dependencies as edges. Computing involves calculating dependency weights, examining graph characteristics such as density and degree, and using a priority queue to order tasks by priority and readiness. Tasks with few or no unmet dependencies are scheduled first, and then a topological sort is performed, scheduling them after their dependencies. We derived a lightweight heuristic that retains the greedy and adaptive behaviour of the PSOs without their metaheuristic overhead. Our scheduler mimics PSO's fitness-based learning through trust-based filtering and retries while ensuring execution-order correctness via DAG topological sorting. This allowed us to achieve a balance between performance and explainability in edge environments with potentially malicious nodes.

Figure 7 processes a queue of tasks by dividing them into batches. It initializes parameters, calculates an optimal batch size, and iteratively forms task batches based on a topological order. Each batch is created by taking a segment of tasks and adding it to the collection of task batches until all tasks are processed. Once tasks have been grouped into batches based on dependencies and execution feasibility, the next logical step is to decide where each task should run. Batch processing organizes the workload into

manageable, dependency-aware segments, but optimal performance depends on assigning those batches (or the tasks within them) to the most suitable computing nodes.

Topological Ordering Algorithm

Input: $Task_{Assigned}(T_1, T_2, T_3, \ldots, T_N)$
Output: $Task_{Dag}$
Steps:
1. **Initialization**
2. **For** each Task V in $Task_{Assigned}$ **do** //Create vertices
3. Vertex P → Create $Task_{Vertex}$(T)
4. P.Task$_{\text{Assigned}}$ → T.$Task_{Assigned}$
5. $Task_{Dag}$.addVertex(V)
6. **End For**
7. D_M → Dependency Matrix, D Dependency, T Task, T_{PQ} → $Task\ Priority\ Queue$
8 D_M(Task_ID) → Initialize D_{Row}() // create dependency edges
9. **For** all D_{Task_ID} in D_{Row} **do**
10. If $Task_{Dag}$.Vertex(V) has D_{Task_ID} then
11. D_{weight} → Calculate $Task_{\text{Dweight}}(D_{Task\_ID}, Task\_ID)$
12. Edge E → $Create\ D_{Edge}(D_{Task\_ID}, Task\_ID, D_{Weight})$
13. **End For**
14. V_P Number of Vertices, N_eNumber of Edges, V_C $Vertex\ Count$
15. V_P → $Task_{Dag}.getV_C()$ // Calculate basic graph metric
16. V_E → $Task_{Dag}.getV_E()$
17. Average In-Degree → V_E /V_v
18. Density → (2 × V_E) / (numVertices × (numVertices - 1))
19. **Return** Average In-Degree, Density
20. Priority Queue → Initialise T_{PQ}() //Initialise priority queue
21. **For** all Vertex P in the Task DAG. vertices **do** // Topological Sorting
22. If Average In-Degree[P_{Task_ID}] ≤ 2 then
23. T_{Score} → Calculate T_{Score} (V)
24. PQ. enqueue(V, T_{Score})
25. **End For**
26. Return $Task_{Dag}$, Dependency Analysis

Figure 6. Task DAG Construction and Analysis

Creating task batches involves initializing parameters, choosing batch size, and iteratively grouping tasks in topological order until all are batched. Batching maintains dependencies and splits responsibilities. From this point, identifying suitable computing nodes based on task performance is crucial; performance depends on task clusters and assigning tasks to nodes that handle their load well.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Batch Processing and Task Organization for Resource Optimization

Input: Task_{Dag}

Output: $\text{Task}_{\text{Batches}}$

Steps:

1. **Initialization:**
2. $V_N \rightarrow Number\ of\ Tasks, T_B \rightarrow Task\ Batches$
3. $T_N \rightarrow \emptyset$
4. **Calculate `Batch Size` using the formula** BatchSize = $min\left(5, \frac{T_N}{4}\right)$
5. **Initialize** an empty list `Task Batches` to store the batches of tasks.
6. $T_B \rightarrow \emptyset$
7. **For**(i $=$ 1, i $<$ length. Task_{Dag}, i $++$): *// Calculate batch end index*
8. **For** i in range(len(topoOrder)):
9. **endIdx** $=$ min(i $+$ batchSize, len(topoOrder) *// Create batch from Task DAG*
10. Batch $=$ topoOrder[i: endIdx]
11. **End For**
12. **Parallel Group** = **For** T in Batch **do** // Form parallel group
13. T. update (processed)
14. **End For**
15. **Return** $\text{Task}_{\text{Batches}}$

Figure 7. Batch and Parallel Processing Allocation Algorithm

The algorithm in Figure 8 processes tasks in parallel batches, assigning each to a trusted node while respecting dependencies and retry limits. It checks if dependencies are met; low-priority tasks with unmet dependencies are skipped. It then selects a node based on trust scores, or retries if no suitable node. Once a node is chosen, it calculates execution timing with jitter, executes, and updates system states, trust levels, and metrics. If execution fails, the task is retried until max retries; persistent failures are logged. After each batch, node loads are rebalanced. The system computes metrics like throughput, latency, trust levels, and task times. It uses an algorithm based on Particle Swarm Optimization to dynamically adjust task allocation and resources for varying loads.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Task Allocation and Execution

Input: $\text{Task}_{\text{Batches}}$, Nodes, $Task_{Dag}$
Output: Metrics
Steps:
1. **Initialization:**
2. $Task_{End\ Times}$ → Array[Task. length] *// Initialize tracking variables*
3. Utilized Nodes → [ ]
4. Max Retries → 3
5. **For** (ID = 1, ID < length(TaskBatches), ID + +) **do** *// Process each batch sequentially*
6. Current Batch → taskBatches[Batch_ID]
7. **For** (Task ID in Current Batch) **do** *// Process each task in current batch*
8. Task → tasks[Task ID]
9. assigned →FALSE
10. attempt → 0
11. **While** (Attempt < Max Retries && Assigned← FALSE) **do** *// Retry loop*
12. attempt → attempt + 1
13. **If** not isEmpty($Task_{Dag}$) **Then** *// Dependency Management*
14. Satisfied → checkDependencySatisfaction(Task, $Task_{End\ Times}$)
15. **If** ($Dep_{\text{Satisfied}}$ < $Dep_{\text{Threshold}}$) **Then**
16. **If** ($Task_{Priority}$ ≥ 2) **Then** *// Not a critical task*
17. **Continue**
18. **End If**
19. **End if**
20. Selected Node → Nodes *// Trust-Based Node Selection with Fallbacks*
21. **If** selected Node = Null **Then**
22. **Continue**
23. **End If**
24. Timing → Calculate Task Timing(Task, Selected Node, $Task_{End\ Times}$)
25. **Execution Result** → Execute Task on Node(Task, Selected Node, Timing
26. **If** (Execution Result == Success) **Then** // Update States and Metric
27. **Update** System State(task, selectedNode, executionResult, taskEndTimes)
28. **Update**Performance Metrics(throughput, latency_total, trust_total)
29. **Task Assigned** → True
30. **End If**
31. **End While**
32. **If (Task** not assigned) **Then** *// Handle Failed assignments*
33. Failed → failed + 1
34. Print "Task " + TaskID + " FAILED after " + maxRetries + " attempts"
35. **End If**
36. **End For**
37. **If** (Batch ID < length(TaskBatches)) **Then** *// Inter-Batch Load Rebalancing*
38. Rebalance Node Loads(nodes)
39. **End If**
40. **End For**
41.metrics → computeFinalMetrics (throughput, latency_total, trust_total, Failed, Utilized Nodes)
42. **returns** $Task_{End\ Times}$, Nodes, Metrics, Failed

Figure 8. Dependency-aware trust scheduling model

2.6. Evaluation

The proposed DTINT framework is evaluated against standard scheduling approaches, including a Genetic Algorithm and a conventional PSO, without dependency or trust awareness. All baseline methods operate under identical simulation settings, including network topology, task characteristics, resource availability, and workload distributions, ensuring that performance improvements are solely attributable to the proposed enhancements.

1) Average Latency: The average time taken to complete a transaction from submission to response.

$$\text{Average Latency} = \frac{\sum(T_{\text{response}} - T_{\text{submission}})}{N} \quad (8)$$

2) Response Time: The time required by the system to process a request after it is received.

$$\text{Response Time} = V_{\text{completion}} - V_{\text{arrival}} \quad (9)$$

3) Throughput: The number of transactions processed successfully within a given time period.

$$\text{Throughput} = \frac{N}{V} \quad (10)$$

4) Node Utilization: The proportion of nodes actively involved in processing transactions.

$$\text{Node Utilization} = \frac{N_{\text{active}}}{N_{\text{total}}} \quad (11)$$

3. RESULTS AND DISCUSSION

This section presents the experiments conducted with the proposed system to evaluate its effectiveness and performance.

3.1. Experiment setup

This study conducted a series of simulations to evaluate the performance of the DTINT framework after its implementation. Performance metrics included Throughput, Latency, Node Utilization, and Load Balance Score. All algorithm runs were simulated over 0 to

500 task scheduling iterations to reduce statistical variability and ensure reliable results. Simulations were performed on a Windows 10 PC with an Intel(R) Core (TM) i7-8750H processor at 2.20 GHz and 16 GB of RAM, providing sufficient computational power for complex tasks. All evaluation parameters are listed in Table 2. MATLAB R2023 was used to implement the scheduling algorithm and visualize the results. Task loads were generated using Poisson arrival processes with varying intensity parameters to mimic real-world IoT workload patterns [20].

Table 2. Simulation parameters

Domains	Value
Scheduling approach	PSO algorithm, Parallel Scheduling
Simulator	MATLAB (Python kernel), PyCharm
The total number of task iterations	0-500
Simulation Area	100*100 meters
Simulation Time	1000-2000 rounds/iteration
Network Topology	Ring
Number of Nodes(s)	10-100

3.2. Experiment Results and Analysis

This section presents the simulation results and their analysis. The analysis is based on three simulated scenarios: 1, 2, and 3. Scenario 1 evaluates the framework's handling of varying workloads (Load Testing), and Scenario 2 assesses the efficiency of the proposed framework under different operating conditions (Performance Testing). Scenario 3 compares the proposed framework to the traditional scheduling system across throughput, latency, and node utilization.

1) Scenario 1: Load testing

The results of DTINT's performance with integrated resource sharing mechanisms on MATLAB are presented in Figure 9. As shown, the system performance improves as the workload increases, with throughput rising from 200 transactions per second (tps) at 50 tasks to 500 tps at 500 tasks, yielding an average throughput of approximately 295 tps. Latency peaks at around 200 ms during the initial workload range (50100 tasks) before declining to 57 ms, indicating an initial processing overhead that stabilizes as the task load increases. Node utilization grows steadily with the number of tasks and reaches full

capacity (100%) at approximately 350-400 tasks, demonstrating efficient resource usage under heavy workloads. Meanwhile, the average transaction processing time increases from 50 ms at low load to 200 ms at 500 tasks, reflecting the cost of scaling at higher task volumes.

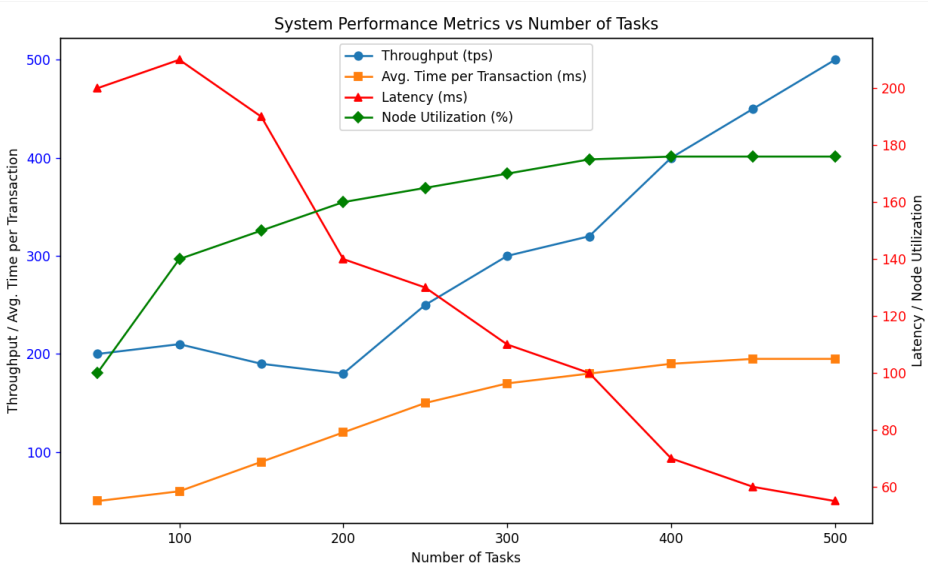

Figure 9. Load testing with an integrated resource sharing mechanism

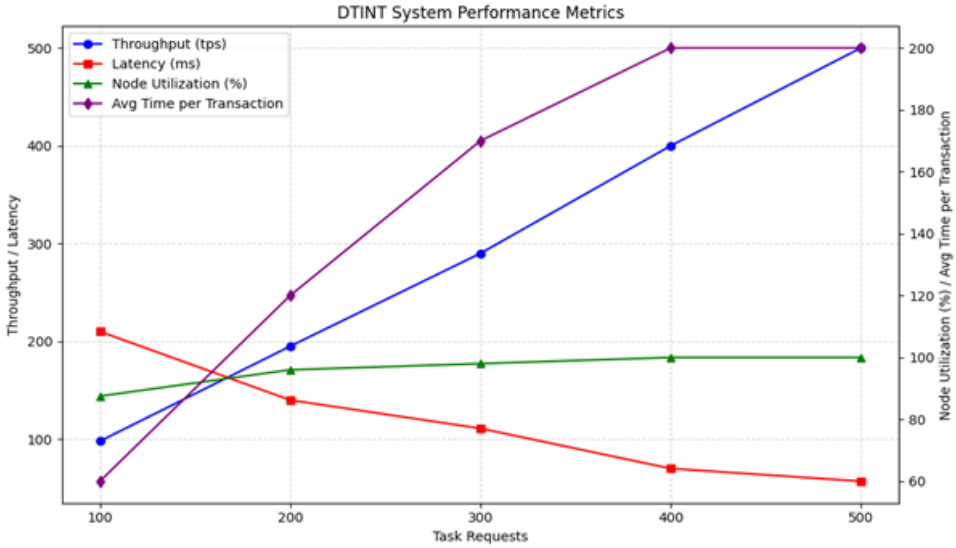

Figure 10. Load Testing without a proposed resource sharing mechanism

The performance of the system without using shared resources is depicted in Figure 10 and is significantly slower than that of the proposed DTINT framework. These higher

latencies are due to inefficiencies in resource management caused by a lack of resource sharing. Lagged results in observed delays arise from a lack of mechanisms for optimizing resource utilization and insufficient concurrency, as observed in all previous studies.

2) Scenario 2: Performance testing

DTINT significantly reduces dependency duration by effectively mitigating dependency-related delays, as illustrated in Figure 11. Experimental results show that DTINT-based systems achieve a 70–75% reduction in dependency-induced latency. The framework sustains a high processing flow between 80% and 100%, with peak throughput observed at 100-150 work units. The average processing time per transaction remains stable within 10-15 units across all user requests. Task failures are negligible, with a completion rate exceeding 99%, demonstrating the framework's robustness and reliability. Overall, DTINT exhibits near-zero failure rates, linear scalability in task scheduling, and consistent response times despite the challenges of parallel execution. These results indicate that the proposed mechanism can efficiently accommodate additional user workloads while maintaining stable and dependable performance.

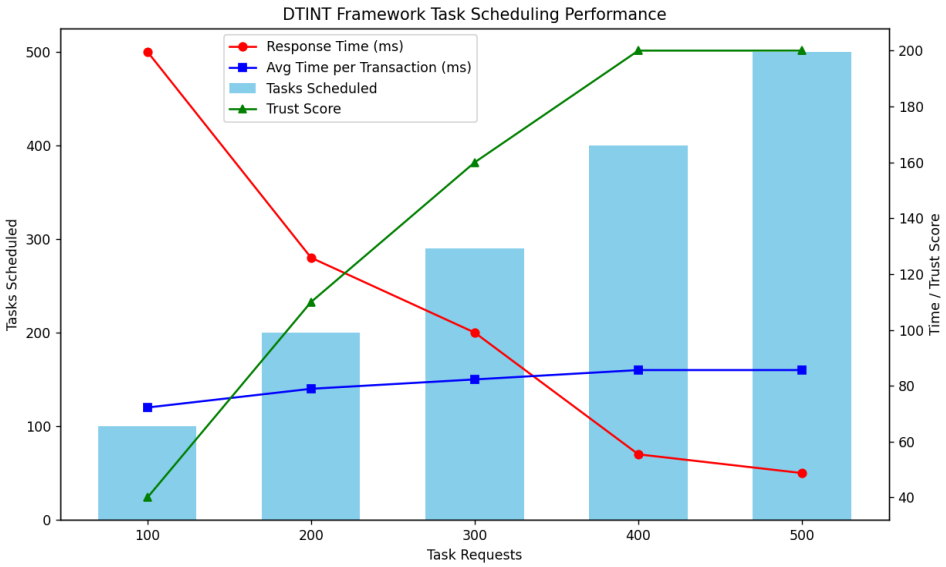

Figure 11. Collaborative DTINT scheduling process

3) Scenario 3: Proposed DTINT vs Traditional Framework

Figure 12 shows that the DTINT framework processes more tasks with lower latency, improved resource utilization, and enhanced coordination through trust-aware node selection. It employs a sequencing process based on DAG scheduling, which avoids

dependency violations and delays, resulting in throughput that is 2-2.5 times better than that of traditional methods. With task sequencing accuracy, it ensures zero re-execution, which means predictable task processing. DTINT maintains node utilization at 70%-100%, whereas traditional methods exhibit a wider range of 20%-60% under high task load. The intelligent batching and splitting mechanisms of DTINT enable nearly linear scale-up to approximately 500 tps, compared to the 200 tps throughput achieved in traditional methods. Its flexible and proactive collaboration framework has enhanced coordination efficiency, as evidenced by steady throughput and consistent resource utilization.

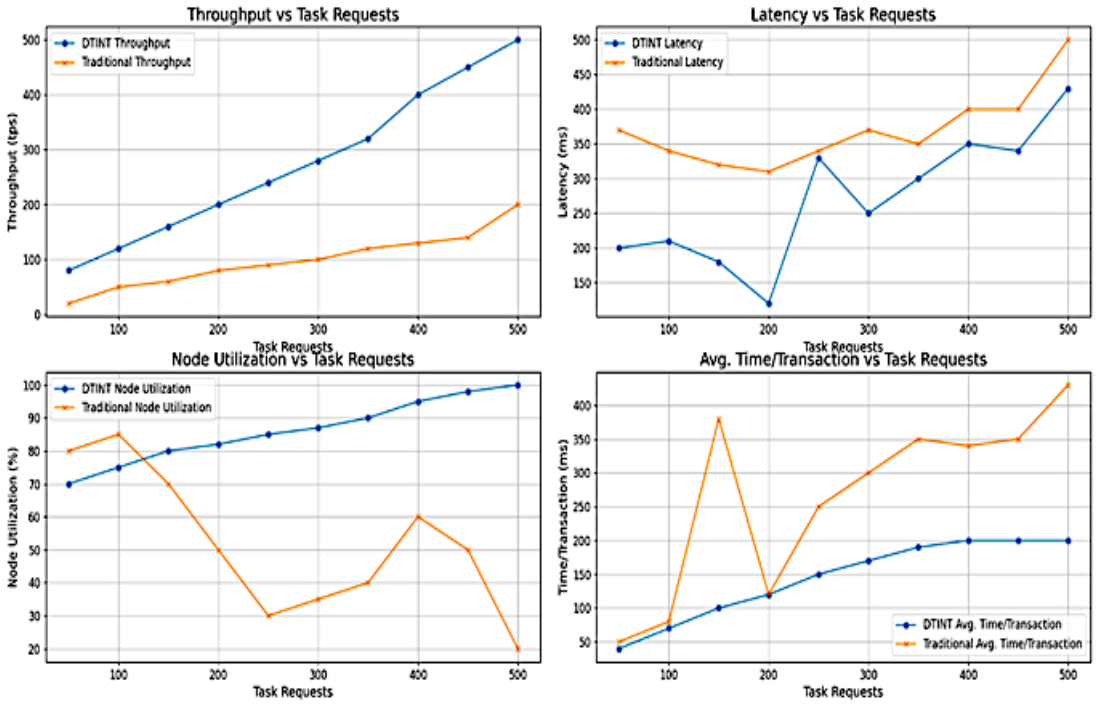

Figure 12. Traditional vs proposed framework

Table 3. Comparative performance of scheduling algorithms

Scheduling Algorithm	Average Throughput	Average Latency	Average Node Utilization	Average Time Per Transaction	Average Trust Score
DTINT	500	50	98	60	0.989
PSO	150	97	80	120	0.841
GA	102	111	85	170	0.782

As shown in Table 3 and Figure 13, the performance of traditional approaches, such as PSO and GA, is compared with the proposed approach using the baseline approach under the same setup conditions. The results show that it has deteriorated significantly in the non-resource-sharing system. Its total throughput decreased by around 2.5 (200 tps against 500 tps), latency was 6-8 times higher (380 ms to 60 ms at peak), and overall node utilization fell to 30%, compared to 50% in the resource-sharing system. Although the resource-sharing system offers linear scalability, providing predictable and steady operation, the non-sharing system suffers from bottlenecks, inefficient resource consumption, and variable transaction intervals. These results clearly demonstrate that efficient resource allocation is essential for achieving better system performance and scalability.

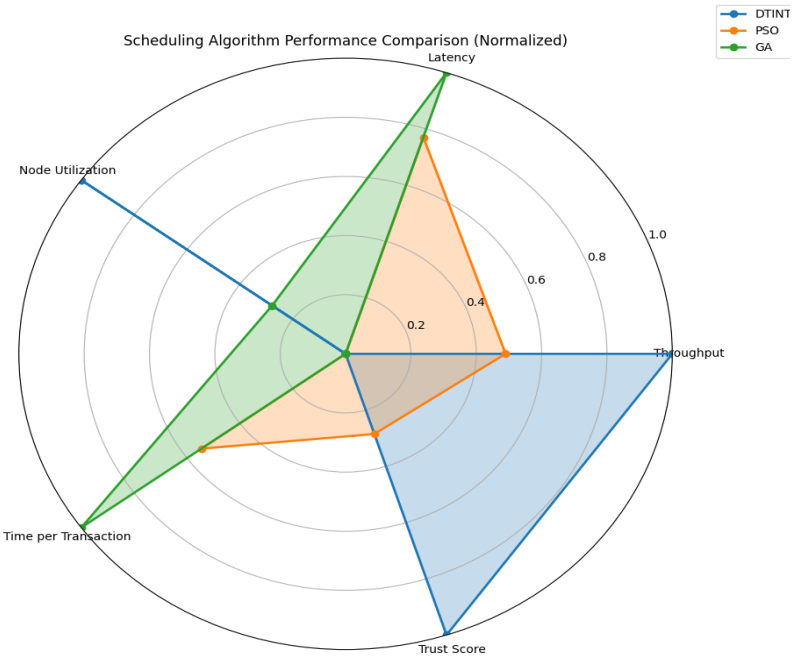

Figure 13. Comparative performance

3.3. Comparison with existing studies

This subsection compares our study with selected existing task dependency and resource scheduling across fog, edge, and cloud environments. We focus on performance metrics such as throughput, latency, average response time, and task completion rate. Table 4 summarizes the performance of representative methods alongside the proposed system.

Table 4. Comparison with existing studies

Method	Throughput (tps)	Latency (ms)	Average response time(ms)	Task Completion Rate (%)
Dynamic [21]	600	-	476	80
Deadline-aware [22]	-	70	250	-
Prioritized [23]	370	5690	-	82
Evolutionary [24]	380	1000	1046	91.6
RL-based [25]	260	78.5	7320	88.2
Time Sensitive [26]	350	210	1500	92
Latency-aware [27]	240	2100	-	56.81
Proposed method	500	50	60	98

As shown in Table 4, the comparison results of a cross-study with other published work results are displayed. The result shows that the proposed method achieves balanced, superior performance across all metrics. While some existing approaches achieve higher throughput, they generally incur higher latency or response times or lower task completion rates. For instance, methods such as dynamic [21] or evolutionary [24] exhibit moderate-to-high throughput but comparatively higher latency or average response times. In contrast, the proposed system achieves a throughput of 500 tps, a latency of 50 ms, an average response time of 60 ms, and a task completion rate of 98%, representing improvements in responsiveness and reliability without severely compromising throughput.

Furthermore, latency and response time are particularly low in the proposed system relative to all baseline methods, including time-sensitive [27] and RL-based [26] approaches. This suggests that the proposed dependency-aware scheduling effectively manages task execution order and resource allocation, minimizing delays even in heterogeneous, multi-level deployments. Finally, the task completion rate of 98% indicates strong handling of task dependencies and resource contention, outperforming all compared approaches. In general, the proposed system achieves a balanced performance profile, combining low latency, fast response, and high reliability, which is essential for fog–edge–cloud environments with complex task dependencies.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

3.4. Discussion

The experimental results demonstrate that the DTINT framework significantly enhances task scheduling efficiency by effectively integrating resource sharing, dependency-aware scheduling, and trust-based node selection. Among these components, dependency-aware scheduling based on Directed Acyclic Graph (DAG) modeling contributes most to overall performance improvement, as it ensures correct task execution order, eliminates dependency violations, and minimizes idle time between interdependent tasks. This directly reduces latency and improves throughput. The resource-sharing mechanism further strengthens performance by enabling efficient utilization of distributed edge resources, while trust-based node selection improves reliability by assigning tasks to dependable nodes, thereby reducing failures and re-execution overhead.

Internal baseline comparisons focus on the performance differences between the proposed DTINT framework and a traditional, non-resource-sharing system under identical simulation conditions. Across all scenarios, the baseline performs worse. Scenario 1 shows efficient parallelization raises throughput and resource use, with initial latency spikes leveling off. Scenario 2 confirms the framework's strength, cutting dependency latency by 70–75% with DAG-based task sequencing. It proves stable with low processing times and near-zero failure rates. In Scenario 3, DTINT outperforms traditional methods with higher throughput, lower latency, and stable node use by reducing re-execution, improving coordination, and balancing load.

In contrast, cross-studied comparisons evaluate DTINT against previously published methods under different experimental setups and assumptions. While such comparisons provide useful context, they are inherently less controlled due to variations in system models, workloads, and evaluation environments. Many existing approaches—such as heuristic, evolutionary, or learning-based methods—tend to optimize these metrics in isolation, without fully accounting for task dependencies or system-wide coordination. In contrast, DTINT adopts a holistic approach by jointly optimizing dependency handling, resource allocation, and node reliability. This integrated design explains its ability to achieve a balanced performance profile, combining high throughput, low latency, and strong reliability.

4. CONCLUSION

This paper proposed and presented a dependency- and trust-aware resource management framework for IoT-edge computing. By integrating DAG-based dependency modelling, PSO-based optimization, and trust-aware node selection. The results demonstrate significant improvements in latency, throughput, and task completion rates, confirming the effectiveness of the proposed approach. Future work will focus on incorporating machine learning techniques to enable adaptive and predictive scheduling in dynamic IoT-edge environments.

REFERENCES

[1] B. Rathi, S. Thapaswi, M. Kambhampati *et al*, "Realizing the potential of Internet of Things (IoT) in industrial applications," *Discover Internet of Things*, vol. 5, no. 1, p. 45, Apr. 2025, doi: 10.1007/s43926-025-00141-5.

[2] G. Jeon, M. Albertini, V. Bellandi, and A. Chehri, "Intelligent mobile edge computing for IoT big data," *Complex & Intelligent Systems*, vol. 8, no. 5, pp. 3595–3601, Oct. 2022, doi: 10.1007/s40747-022-00821-7.

[3] U. Islam, M. N. Alatawi, A. Alqazzaz, S. Alamro, B. Shah, and F. Moreira, "A hybrid fog-edge computing architecture for real-time health monitoring in IoMT systems with optimized latency and threat resilience," *Scientific Reports*, vol. 15, no. 1, p. 25655, Jul. 2025, doi: 10.1038/s41598-025-09696-3.

[4] B. Isong, F. H. Mamidza, and R. Molose, "The role of dynamic trust and resource management in improving IoT-based edge computing: A review," in *Proc. AI Revolution: Research, Ethics, and Society (AIR-RES 2025)*, *Communications in Computer and Information Science*, vol. 2723. Cham, Switzerland: Springer, 2026, pp. 337–355, doi: 10.1007/978-3-032-13056-3_25.

[5] T. Wang, Y. Liang, X. Shen, X. Zheng, A. Mahmood, and Q. Z. Sheng, "Edge computing and sensor-cloud: Overview, solutions, and directions," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–37, Dec. 2023, doi: 10.1145/3582270.

[6] Y. Chen, J. Zhao, J. Hu, S. Wan, and J. Huang, "Distributed task offloading and resource purchasing in NOMA-enabled mobile edge computing: Hierarchical game theoretical approaches," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–28, Jan. 2024, doi: 10.1145/3597023.

[7] H. Zou, J. Guo, J. Zeng, Y. Li, J. Cao, and T. Wang, "Fine-grained service lifetime optimization for energy-constrained edge-edge collaboration," in *Proc. IEEE 44th Int. Conf. Distributed Computing Systems (ICDCS)*, Jersey City, NJ, USA, Jul. 2024, pp. 565–576, doi: 10.1109/ICDCS60910.2024.00059.

[8] Y. Liang, W. Wang, X. Zheng, Q. Liu, L. Wang, and T. Wang, "Collaborative edge service placement for maximizing QoS with distributed data cleaning," in *Proc. IEEE/ACM 31st Int. Symp. Quality of Service (IWQoS)*, Orlando, FL, USA, Jun. 2023, pp. 1–4, doi: 10.1109/IWQoS57198.2023.10188694.

[9] R. Qing, H. Rao, G. Jia, Y. Xu, W. Wei, and G. Xie, "Task scheduling strategy based on resource constraint in edge computing system," in *Proc. IEEE 7th Int. Conf. Industrial Cyber-Physical Systems (ICPS)*, St. Louis, MO, USA, May 2024, pp. 1–7, doi: 10.1109/ICPS59941.2024.10639950.

[10] A. Khatoon, A. Ullah, A. U. Bello, A. Khan, M. B. Roslee, and H. Amin, "Fuzzy logic-based task scheduling for AI-enabled IoT edge devices in smart communication networks," in *Proc. Multimedia University Engineering Conf. (MECON)*, Cyberjaya, Malaysia, Jul. 2025, doi: 10.1109/MECON67253.2025.11276908.

[11] A. A. Al-Habob, O. A. Dobre, A. G. Armada, and S. Muhaidat, "Task scheduling for mobile edge computing using genetic algorithm and conflict graphs," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 8, pp. 8805–8819, Aug. 2020, doi: 10.1109/TVT.2020.2995146.

[12] G. Vijayasekaran and M. Duraipandian, "Resource scheduling in edge computing IoT networks using a hybrid deep learning algorithm," *System Research and Information Technologies*, no. 3, pp. 86–101, Oct. 2022, doi: 10.20535/SRIT.2308-8893.2022.3.06.

[13] A. M. Sheikh, M. R. Islam, M. H. Habaebi, S. A. Zabidi, A. R. Najeeb, and A. Kabbani, "A survey on edge computing security challenges: Classification, threats, and mitigation strategies," *Future Internet*, vol. 17, no. 4, p. 175, Apr. 2025, doi: 10.3390/fi17040175.

[14] M. Maray, E. Mustafa, J. Shuja, and M. Bilal, "Dependent task offloading with deadline-aware scheduling in mobile edge networks," *Internet of Things*, vol. 23, p. 100868, Oct. 2023, doi: 10.1016/j.iot.2023.100868.

[15] J. Li, Y. Pan, Y. Xia, Z. Fan, X. Wang, and J. Lv, "Optimizing DAG scheduling and deployment for IoT data analysis services in multi-UAV mobile edge computing systems," *Wireless Networks*, vol. 30, no. 7, pp. 6465–6479, Oct. 2024, doi: 10.1007/s11276-023-03451-0.

[16] G. Li *et al.*, "DAG scheduling in mobile edge computing," *ACM Transactions on Sensor Networks*, vol. 20, no. 1, Oct. 2023, doi: 10.1145/3616374.

[17] V. Prakash, S. Bawa, and L. Garg, "Multi-dependency and time-based resource scheduling algorithm for scientific applications in cloud computing," *Electronics*, vol. 10, no. 11, p. 1320, May 2021, doi: 10.3390/electronics10111320.

[18] M. Su, G. Wang, and K.-K. R. Choo, "Prediction-based resource deployment and task scheduling in edge-cloud collaborative computing," *Wireless Communications and Mobile Computing*, vol. 2022, no. 1, p. 2568503, Jan. 2022, doi: 10.1155/2022/2568503.

[19] S. A. Murad, A. J. M. Muzahid, Z. R. M. Azmi, M. I. Hoque, and M. Kowsher, "A review on job scheduling techniques in cloud computing and priority rule-based intelligent framework," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 6, pp. 2309–2331, Jun. 2022, doi: 10.1016/j.jksuci.2022.03.027.

[20] Q. Cai, M. R. Bajuri, K. E. Leong, L. Chen, L, "Multimodal Learning Interactions Using MATLAB Technology in a Multinational Statistical Classroom," *Multimodal Technologies and Interaction*, vol. 9, no. 10, pp. 106, Oct. 2025, doi: 10.3390/mti9100106.

[21] M. R. Saha, M. E. Kader, and R. Reaz, "Dependency, deadline, and priority-aware multi-queue dynamic task scheduling using heterogeneous resources in fog environment," in *Proc. IEEE/ACM 17th Int. Conf. Utility and Cloud Computing (UCC)*, Sharjah, UAE, Dec. 2024, pp. 9–16, doi: 10.1109/UCC63386.2024.00012.

[22] Y. Zhang, B. Tang, J. Luo, and J. Zhang, "Deadline-aware dynamic task scheduling in edge–cloud collaborative computing," *Electronics*, vol. 11, no. 15, p. 2464, Aug. 2022, doi: 10.3390/electronics11152464.

[23] E. M. Elshahed, R. M. Abdelmoneem, E. Shaaban, H. A. Elzahed, and S. M. Al-Tabbakh, "Prioritized scheduling technique for healthcare tasks in cloud computing," *Journal of Supercomputing*, vol. 79, no. 5, pp. 4895–4916, Mar. 2023, doi: 10.1007/s11227-022-04823-7.

[24] F. K. Karim, S. Ghorashi, S. Alkhalaf, S. H. A. Hamza, A. B. Ishak, and S. Abdel-Khalek, "Optimizing makespan and resource utilization in cloud computing environment via evolutionary scheduling approach," *PLOS ONE*, vol. 19, no. 11, p. e0311814, Nov. 2024, doi: 10.1371/journal.pone.0311814.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[25] Y. Wang, T. Tang, Z. Fang, Y. Deng, and Y. Duan, "Intelligent task scheduling for microservices via A3C-based reinforcement learning," in *Proc. IEEE Int. Conf. Communications, Information Systems and Computer Engineering (CISCE)*, Guangzhou, China, 2025, pp. 585–589, doi: 10.1109/CISCE65916.2025.11065827.

[26] H. Nie, Y. Wu, W. Zhu, J. Zhong, H. Yang, and Y. Zhou, "Time-triggered task offloading scheduling in TSN-based edge computing power networks," *IEEE Access*, vol. 13, pp. 85979–85996, 2025, doi: 10.1109/ACCESS.2025.3568848.

[27] A. Mahapatra, R. Pradhan, S. K. Majhi, and K. Mishra, "DELTa: Dynamic energy- and latency-aware task scheduling for fog-cloud paradigm," *IEEE Access*, vol. 13, pp. 74617–74633, 2025, doi: 10.1109/ACCESS.2025.3563103.