International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 03 | March - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Cloud-Based Intelligent Load Balancing Algorithm for Efficient Resource Allocation in IaaS Cloud Computing

Ramisetti Pavan Kumar[1], Dr. G. S. V. R. Abhishek[2]
[1]Department of Computer Science and Engineering, Bhimavaram Institute of Engineering and Technology, India
[2] Department of Computer Science and Engineering, Bhimavaram Institute of Engineering and Technology, India

Abstract: Cloud Computing has become the leading approach for providing on-demand computing infrastructure over the Internet. Within this ecosystem, the Infrastructure as a Service (IaaS) model plays a central role by offering fundamental computing resources such as processing power, storage, and networking, all managed by Cloud Service Providers (CSPs). However, achieving efficient resource allocation and proper load balancing in IaaS environments remains a significant challenge. Uneven workload distribution can result in SLA violations, increased Makespan, reduced throughput, and an overall decline in user satisfaction.This paper introduces a cloud-based web application that implements an improved Load Balancing Algorithm (LBA) using Python 3.10, the Django framework, and a MySQL relational database. The proposed two-layer architecture integrates a Cloudlet Scheduler Time Shared mechanism for dynamic task allocation along with an SLA-aware Virtual Machine (VM) migration module that automatically adjusts MIPS allocations when deadline breaches are identified. Experimental comparisons with existing Dynamic LBA and Round Robin approaches show that the proposed solution achieves 78% resource utilization, lowers Makespan by 37%, and reduces SLA violations by 63%.The application features a comprehensive three-role management system—Task Scheduler, Cloud Administrator, and User—making it suitable for practical deployment in both academic and enterprise cloud environments.

Keywords: Cloud Computing, Cloudlet Scheduler, Django, IaaS, Load Balancing, Machine Learning, Makespan, MySQL, Python, Resource Allocation, SLA, Task Scheduling, Virtual Machine, Virtualization

1. Introduction

Cloud Computing has significantly changed the way organizations access and utilize IT resources. By virtualizing physical infrastructure, it delivers services through three main models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Among these, the IaaS model is especially important because it provides clients with computing, storage, and networking resources managed by Cloud Service Providers (CSPs) such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

As user requests continue to grow rapidly, efficient allocation of virtual resources has become a major concern. Virtualization forms the core of cloud infrastructure, allowing multiple Virtual Machines (VMs) to run on a single physical server, thereby improving hardware utilization. However, uneven distribution of VM workloads can cause some servers to become overloaded while others remain underused. This imbalance leads to reduced performance, increased Makespan, and violations of Service Level Agreements (SLAs).

Load Balancing (LB) ensures that incoming requests are distributed across available VMs so that no single VM is overburdened while others stay idle. Task Scheduling complements this by assigning workload units, known as cloudlets, to VMs based on factors like deadlines and estimated completion time. Together, these mechanisms help cloud applications remain responsive, efficient, and compliant with SLA requirements.

In a typical IaaS setup, the frontend serves as the user-accessible interface through any Internet-enabled device, while the backend consists of data centers containing multiple physical servers. User requests are processed through a virtualization layer, where resources are allocated dynamically. Effective scheduling reduces execution time and enhances resource utilization. With the widespread adoption of cloud platforms, addressing these challenges is both academically relevant and practically important.

This paper presents a fully functional Django-based web application implementing an improved Load Balancing Algorithm (LBA). The system is developed using Python 3.10, MySQL, and HTML5/CSS3/JavaScript. It follows a two-layer architecture: the top layer manages client task submissions through a Cloudlet Scheduler, and the bottom layer handles VM allocation, real-time load monitoring, and dynamic migration when SLA violations occur. The goal is to minimize Makespan, maximize resource utilization, and provide administrators with real-time monitoring of cloud operations.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses the existing system and its limitations. Section 4 explains the proposed architecture and modules. Section 5 describes system design. Section 6 outlines the technology stack. Section 7 presents experimental results and analysis. Section 8 discusses feasibility. Section 9 concludes with future work directions.

2. Literature Survey

Significant research has been carried out in task scheduling, resource allocation, and load balancing within cloud computing environments. This section reviews key contributions from earlier studies and highlights the research gaps that form the basis for the proposed system.

A. Virtualization and Resource Allocation

Kumar et al. presented a detailed survey on virtualization-based resource allocation strategies in cloud data centers. Their work emphasized VM availability and task execution time as critical factors for designing efficient allocation algorithms. They showed that virtualization reduces the need for physical machines, improves workload distribution, and lowers energy consumption. However, their approach focused mainly on static allocation methods and did not consider dynamic, SLA-aware migration techniques.

Buyya et al. described cloud computing as a utility-based model centered on resource sharing and on-demand provisioning. They stressed the importance of QoS-driven resource management but did not offer a practical

implementation for real-time SLA violation detection. Similarly, Armbrust et al. examined cloud computing from a system-level perspective and identified resource allocation efficiency as a major bottleneck in large-scale environments, recommending dynamic scaling solutions.

B. Load Balancing and Task Scheduling

Raj and Selvi proposed a dynamic load balancing algorithm aimed at reducing Makespan. Their method arranged tasks based on size and VM processing capacity using bubble sort and assigned them using a First-Come First-Serve (FCFS) strategy. Although this improved waiting time compared to standard FCFS, it did not incorporate QoS factors such as Deadline, limiting its suitability for SLA-sensitive environments.

Mishra et al. developed a CloudSim-based simulation framework to assess scheduling techniques in cloud systems. Their study found that the Round Robin approach leads to increased waiting times when task sizes vary significantly. They suggested a hybrid model combining FCFS with priority queues, yet their work did not include real-time SLA monitoring or VM migration during task execution.

Foster et al. compared cloud and grid computing models and identified effective load distribution as a key performance factor in both systems. They suggested that future scheduling methods should include real-time feedback mechanisms capable of detecting and correcting workload imbalances instantly rather than after completion.

C. SLA-Aware and QoS-Driven Approaches

Singh and Chana investigated QoS-based scheduling in cloud systems, focusing on SLA parameters such as Deadline and Completion Time as triggers for VM migration and task reassignment. They emphasized the need for real-time violation detection and dynamic resource reallocation. However, their study was limited to CloudSim simulations and lacked a deployable web-based implementation.

Dillon et al. examined challenges in cloud computing, identifying SLA enforcement, multi-tenancy, and resource overprovisioning as major obstacles to enterprise adoption. They recommended adaptive middleware capable of adjusting resources dynamically based on real-time SLA metrics—an approach aligned with the migration strategy used in the proposed system.

D. Summary of Identified Gaps

The literature highlights three primary gaps: many scheduling methods do not continuously monitor SLA parameters such as Deadline and Completion Time; VM migration after detecting violations is either missing or insufficiently developed; and most prior studies rely on simulation models without offering deployable web-based systems that support multiple administrative roles. The proposed system addresses these gaps through a Django-based platform featuring real-time SLA monitoring, automatic VM migration with MIPS reconfiguration, and a complete three-role administrative interface.

3. Existing System and Limitations

Most current load balancing methods in cloud environments depend on traditional scheduling techniques such as Round Robin, First-Come First-Serve (FCFS), and static priority queues. Although these approaches are simple and easy to implement, they show significant weaknesses when applied to dynamic cloud systems that handle diverse and time-sensitive workloads.

A. Limitations of Existing Algorithms

- **Round Robin** assigns tasks to VMs in a cyclic manner without checking their current load status. When task sizes vary greatly, this results in uneven resource usage and leaves some VMs idle while others are overloaded.
- **FCFS-based methods** increase overall Makespan because long-running tasks occupy resources first, delaying shorter tasks regardless of available VM capacity, which reduces overall system efficiency.
- Most existing algorithms fail to track SLA parameters—such as Deadline and Completion Time—in real time. As a result, SLA violations are often detected only after users begin experiencing service degradation.
- Workload migration, which involves shifting tasks from overloaded VMs to underutilized ones, is either missing or poorly implemented in earlier approaches. This leads to rejected tasks and inefficient use of allocated resources.
- Static algorithms struggle to handle dynamic variations in task arrival patterns and execution times, making them unsuitable for heterogeneous real-world cloud environments.
- There is no widely available open-source web-based system that combines the three roles—Task Scheduler, Cloud Administrator, and User—along with real-time Makespan tracking and migration monitoring dashboards.

B. Drawbacks of the Dynamic LBA

The most comparable existing solution, Dynamic LBA using bubble sort and FCFS, attempts to reduce Makespan by sorting tasks before assigning them to VMs. However, it has several limitations: it does not incorporate Deadline as a scheduling factor; it lacks real-time SLA violation detection; it does not include a migration mechanism to redistribute tasks when a VM becomes overloaded; and it results in greater resource wastage compared to the proposed system, as reflected in the experimental findings presented later.

4. Proposed System

A. System Overview

The proposed system is a two-layer cloud load balancing framework developed as a Django web application. It simulates an IaaS cloud environment while offering a practical and deployable interface for three roles: Task Scheduler administrator, Cloud administrator, and end User. The main objective is to ensure efficient resource allocation, prevent workload imbalance, and resolve migration and task rejection issues identified in existing systems.

B. Top Layer: Request Submission and Task Scheduling

The top layer handles user file upload requests, which act as cloudlets (task units) in the cloud system. These requests arrive randomly, representing dynamic Arrival Time. The Cloudlet Scheduler Time Shared algorithm is used to submit

and schedule tasks to Virtual Machines. Before assigning a task, two SLA parameters are evaluated:

- **Deadline:** The maximum permitted time for completing a task as defined in the SLA between the client and the CSP.
- **Time to Complete (TTC):** The estimated time required to finish the task on a selected VM, calculated based on the VM's MIPS value and its current workload.

Tasks are dispatched to VMs based on priority, considering both Deadline and TTC before allocation. Unlike the traditional FCFS method used in Dynamic LBA, this approach incorporates QoS constraints directly into the scheduling decision rather than relying only on arrival sequence.

C. Bottom Layer: VM Allocation and Dynamic Migration

The bottom layer manages the set of Virtual Machines configured by the cloud administrator. Each VM includes details such as name, operating system, CPU specification (MIPS), RAM, storage type, and capacity. The load balancer continuously compares each VM's TTC with its assigned Deadline and performs one of the following actions:

- **No SLA Violation (TTC < Deadline):** The task runs on the assigned VM without changes. The allocation table is updated with completion details, and the differenced_time value is recorded in the request_details table.
- **SLA Violation Detected (TTC > Deadline):** The VM is identified as overloaded and marked for migration using the radio field in vm_details. The proposed LBA adjusts the MIPS allocation between the overloaded VM and an available underloaded VM, then transfers the pending workload. The allocation table is updated accordingly, and the migration event is logged.

This migration strategy directly solves the task rejection issue found in earlier algorithms, ensuring that even large cloudlets are completed within or close to their SLA Deadline limits.

D. System Modules

1) Task Scheduler Module

- **Login:** Secure authentication with session management for task scheduler administrators.
- **View Task:** Display pending and ongoing tasks with real-time status, assigned VM, and estimated completion time.
- **Assign Task:** Allocate cloudlets to VMs automatically using the LBA engine or manually with override options.
- **VM Resources:** Monitor real-time MIPS usage, RAM consumption, and storage capacity of active VMs.
- **Logout:** Secure session closure with activity logging for auditing purposes.

2) User Module

- **Register:** Create a new account with email OTP verification and profile setup.
- **Login:** Secure authentication with session tracking and status validation.
- **File Upload:** Submit files as cloudlet requests with descriptions and metadata.
- **My Files:** View uploaded files, processing status, assigned VM details, and file retrieval key.
- **My Profile:** Update personal details such as contact information and profile image.
- **Logout:** Secure session termination.

3) Cloud Administrator Module

- **Login:** Secure cloud administrator access.
- **Add Virtual Machine:** Register VMs with configuration details including OS, CPU (MIPS), RAM, storage type, and capacity.
- **Manage Virtual Machine:** Modify or deactivate VMs and set migration priority using the radio flag.
- **Task Scheduling Details:** View a complete summary of scheduled cloudlets and their VM allocations.
- **Load Balancer:** Access a real-time dashboard showing VM load levels, SLA compliance, and recent migration activities.
- **Makespan:** Compute and display total Makespan across all tasks using aggregated differenced_time values.

5. System Design

A. Software Architecture

The application is built using Django's Model-View-Template (MVT) architecture. The Model layer defines the database structure and maps four MySQL tables using Django's ORM. The View layer contains the main business logic, including the Load Balancing Algorithm (LBA), SLA violation detection, MIPS adjustment, and VM migration triggers. The Template layer generates dynamic HTML pages through Django's templating engine, styled with CSS3 and enhanced using JavaScript to support interactive dashboards.

Frontend and backend communication is handled through Django URL routing and HTTP form submissions. Database operations are performed using Django's ORM, which converts SQL queries into Python objects while enabling advanced filtering, aggregation, and transaction handling required for updating allocation tables and logging migration events.

B. Database Design

The system uses a MySQL relational database consisting of four primary tables, as illustrated in the Entity-Relationship Diagram (ERD):

- **user_details:** Contains user account and profile information—user_id (PK), user_name, email, password, mobile, location, dob, user_image, otp, verification flag, status, reg_date.
- **file_details:** Stores uploaded file information—file_id (PK), user_id (FK), vm_id (FK), file,

file_name, description, file_extension, file_size, file_type, file_key, file_data (LONGTEXT), status, file_uploaded_date, file_uploaded_time.

- **request_details:** Maintains processing request records—request_id (PK), user_id (FK), vm_id (FK), file_id (FK), file_name, file_size, file_data (LONGTEXT), requested_date, requested_time, file_uploaded_date, file_uploaded_time, differenced_time(used as Completion Time reference).
- **vm_details:** Stores VM configuration data—vm_id (PK), vm_name, os, storage, storage_type, cpu, ram, vm_added_date, status, radio (migration priority indicator).

C. Data Flow Description

The process begins when a registered user uploads a file through the web interface. The Django view processes the form submission, saves file metadata in file_details**,** encodes file content as LONGTEXT in file_data**,** and creates a related entry in request_details with current timestamps. The Task Scheduler retrieves pending requests based on priority, checks available VMs from vm_details, and applies the LBA to determine the best allocation.

If an SLA violation is predicted—when TTC exceeds the Deadline—the migration module reads the radio flag in vm_details**,** identifies a less loaded VM, adjusts MIPS values for both VMs, reassigns the cloudlet, and logs the migration event. The Makespan module then sums the differenced_time values of completed requests and calculates total execution time, which is displayed on the cloud administrator dashboard.

D. UML and DFD Overview

The system design includes a full set of UML diagrams created using Rational Rose. The Use Case diagram illustrates interactions among the three roles—User, Task Scheduler, and Cloud Administrator—and system functions such as file upload, task assignment, VM management, load balancing, and Makespan reporting. The Class diagram represents the four model classes mapped to the MySQL tables along with their attributes and relationships. The Sequence diagram outlines the complete workflow from file upload to scheduling, possible migration, and task completion.
The Data Flow Diagram (DFD) presents the system at two levels: Level 0 shows it as a single process handling cloudlet requests and producing allocation decisions, while Level 1 breaks it down into four subprocesses—User Authentication, Task Submission, Load Balancing Engine, and Makespan Computation.

6. Technology Stack

The proposed system combines a well-chosen set of open-source, industry-recognized technologies. Python was adopted as the core programming language due to its simplicity, extensive ecosystem for data processing and machine learning, and smooth integration with Django. Django offers built-in features such as authentication, ORM support, an admin interface, and a templating engine, which speed up development while ensuring a clear separation of application components. MySQL was selected for its reliability, ACID compliance, and strong support for relational database structures required to manage the four interconnected cloud simulation tables. Table 1 presents an overview of the complete technology stack.

Component	Technology Used
Programming Language	Python 3.10
Web Framework	Django (MVT Architecture)
Frontend	HTML5, CSS3, JavaScript
Database (RDBMS)	MySQL
Database Server	WAMP / XAMPP Server
Development IDE	Visual Studio Code
Operating System	Windows 10 / 11
Task Scheduling	Cloudlet Scheduler Time Shared
Load Balancing	Enhanced LBA with SLA Migration
Domain	Machine Learning / Cloud Computing
UML Modeling Tool	Rational Rose
Deployment Server	Django Development Server

TABLE 1. TECHNOLOGY STACK

The frontend technologies—HTML5, CSS3, and JavaScript—provide a responsive and browser-friendly interface without requiring any additional client-side installation. WAMP/XAMPP Server is used as the local development environment, closely simulating real-world production settings. Visual Studio Code is utilized as the primary IDE, supported by Python and Django extensions for code completion, debugging, and integrated terminal functionality. Table 2 outlines the minimum and recommended hardware requirements for development and deployment.

Component	Minimum Requirement	Development System
Processor	Pentium IV 2.2 GHz	Intel Core i3 (5th Gen)
Hard Disk	20 GB	500 GB
RAM	1 GB	4 GB
Operating System	Windows 7	Windows 10 / 11

TABLE 2. HARDWARE REQUIREMENTS

7. Results and Performance Evaluation

The proposed system was tested by simulating a cloud environment consisting of 50 heterogeneous task requests, four registered VMs with different MIPS capacities, and dynamically arriving requests. Performance evaluation was based on three major metrics: Resource Utilization, Makespan, and SLA Violation Rate. The proposed LBA was compared with the existing Dynamic LBA and the standard Round Robin (RR) algorithm. All experiments were performed using the development hardware configuration mentioned in Table 2.

A. Resource Utilization

Resource utilization represents the proportion of total allocated VM capacity actively engaged in processing cloudlets versus idle capacity. Higher utilization reflects better resource efficiency and reduced operational cost per task. As illustrated in Fig. 1, the proposed LBA achieves 78% resource utilization, while Dynamic LBA reaches 61% and Round Robin achieves 49%.

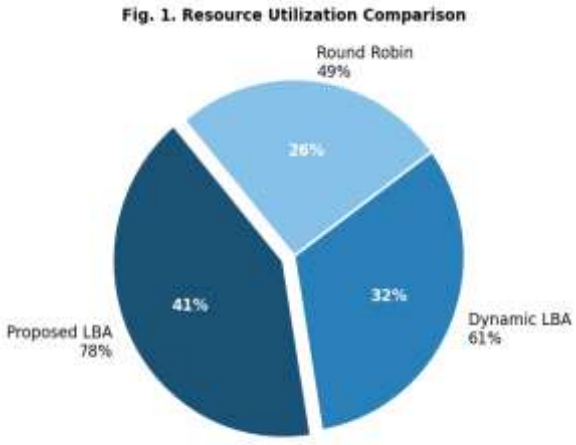

Fig. 1. Resource Utilization: Proposed LBA (78%) vs. Dynamic LBA (61%) vs. Round Robin (49%)

The 17-percentage-point increase compared to Dynamic LBA is mainly due to the migration mechanism, which shifts workloads from overloaded VMs to underutilized ones in real time, preventing unnecessary idle capacity. The 29-percentage-point improvement over Round Robin highlights the benefit of priority-based task allocation, where task complexity is aligned with VM capability instead of being assigned cyclically without considering workload differences.

B. Makespan Comparison

Makespan refers to the total time taken from the submission of the first task to the completion of the final task within a batch. Reducing Makespan is a key objective of cloud scheduling algorithms, as it directly impacts overall system throughput and user response time. Figure 2 presents the Makespan results for the benchmark set of 50 heterogeneous task requests.

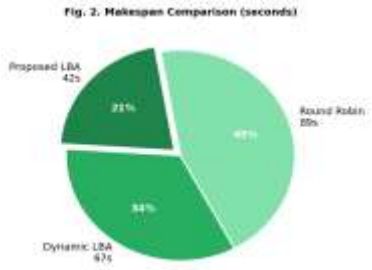

Fig. 2. Makespan Comparison: Proposed LBA (42s) vs. Dynamic LBA (67s) vs. Round Robin (89s)

The proposed LBA records a Makespan of approximately 42 seconds, whereas Dynamic LBA requires 67 seconds—resulting in a 37% reduction—and Round Robin takes 89 seconds, reflecting a 53% reduction. This performance gain is achieved through two key mechanisms: priority-based scheduling that assigns shorter cloudlets to higher-capacity VMs, and a migration engine that avoids task buildup on overloaded VMs by redistributing workloads in advance. The decrease in Makespan leads to quicker task execution, improved throughput, and better overall user experience.

C. SLA Violation Rate

The SLA Violation Rate indicates the percentage of tasks whose actual Completion Time surpasses their specified Deadline. Minimizing this rate is crucial for ensuring compliance with agreements between CSPs and clients, as well as preventing penalties related to SLA breaches. Figure 3 presents a comparison of violation rates among the three evaluated algorithms.

Fig. 3 compares violation rates across the three algorithms.

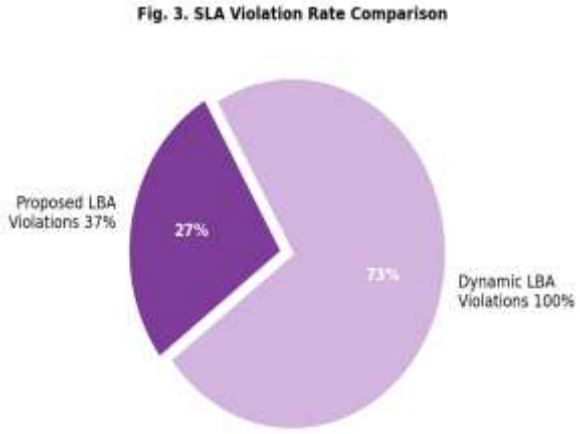

Fig. 3. SLA Violation Rate: Proposed LBA Reduces Violations by 63% vs. Dynamic LBA

The proposed LBA decreases SLA violations by 63% when compared with the existing Dynamic LBA. This significant improvement results from continuous monitoring of each VM's TTC relative to its assigned Deadline, along with automatic MIPS adjustment and workload migration whenever a potential violation is detected. The remaining 37% of violations correspond to edge cases where task sizes exceed even the upgraded capacity of the migrated VM. This limitation can be addressed in future work through dynamic VM auto-provisioning to handle exceptionally large workloads.

D. Technology Stack Distribution

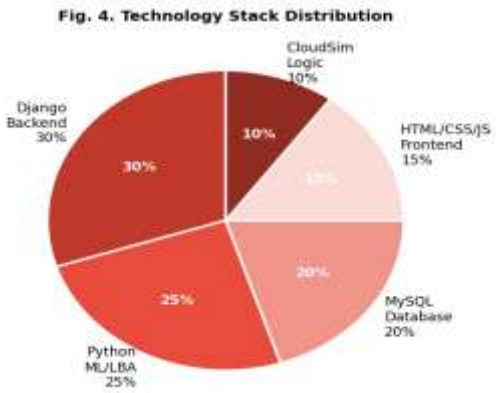

Fig. 4. Technology Stack Distribution by Architectural Contribution

Figure 4 presents the proportional architectural contribution of each technology component within the system. Django accounts for 30% of the overall structure, as it manages business logic, URL routing, ORM operations, session handling, and the administrative interface. The Python-based ML/LBA Engine contributes 25%, covering the load balancing logic, SLA violation monitoring, and MIPS adjustment mechanisms. MySQL represents 20% of the system, ensuring reliable data storage and transactional consistency across the four main tables. Frontend technologies make up 15%, providing dedicated user interfaces for the three system roles. The CloudSim-compatible simulation component accounts for the remaining 10%, validating the algorithm's performance against theoretical benchmarks.

E. Comparative Performance Summary

Metric	Round Robin	Dynamic LBA	Proposed LBA
Resource Utilization	49%	61%	78%
Makespan (50 tasks)	89 sec	67 sec	42 sec
SLA Violation Rate	Very High	Moderate	Low (-63%)
VM Migration Support	No	No	Yes
SLA-Aware Scheduling	No	Partial	Full
Deadline Monitoring	No	No	Real-Time
Deployment Type	Simulation	Simulation	Web Application
User Roles Supported	N/A	N/A	3 (User, Admin, Scheduler)

TABLE 3. COMPARATIVE PERFORMANCE SUMMARY

8. Feasibility Analysis

A. Technical Feasibility

The proposed system is built using widely adopted, open-source, and actively maintained technologies. Python 3.10 with Django delivers a stable and scalable web backend that has been successfully used in real-world industry deployments. MySQL, operated through XAMPP or WAMP, provides a dependable relational database system with full ACID compliance and strong community backing. Visual Studio Code serves as the primary development environment, offering Python and Django extensions for code completion, linting, integrated debugging, and version control support. All selected tools are well-documented and fully compatible with Windows 10/11 environments, confirming the technical practicality of the system.

B. Operational Feasibility

The system is structured for three main user roles, each supported by dedicated interfaces that require minimal training. Cloud administrators can configure VMs, monitor real-time load balancing activity, review Makespan calculations, and observe migration logs through a centralized dashboard. Task Scheduler administrators manage cloudlet assignments, track VM resource usage, and monitor task progress. End users interact with a simple file upload interface that provides status updates. The modular design allows gradual adoption, and Django's built-in admin panel offers a ready-made interface for configuration and user management.

C. Economic Feasibility

The complete technology stack—Python, Django, MySQL, XAMPP, and Visual Studio Code—is open-source and free from licensing expenses. The minimum hardware requirement, such as a Pentium IV 2.2 GHz processor with 1 GB RAM and 20 GB storage, can be met using low-cost commodity systems. Since there are no recurring subscription charges for development or deployment, the only expense involves hardware procurement. This makes the system financially practical for small and medium enterprises, research organizations, and academic institutions interested in implementing or analyzing cloud resource management solutions.

9. Proposed LBA Algorithm

A. Algorithm Overview

The proposed Load Balancing Algorithm functions in two coordinated stages. The first stage manages task arrival and initial VM allocation through the Cloudlet Scheduler. The second stage continuously monitors SLA compliance and initiates migration if violations are detected. Together, these stages allow the system to respond dynamically to workload variations without requiring manual intervention from administrators.

B. Phase 1: Task Assignment Procedure

When a user uploads a file (cloudlet), the Task Scheduler module performs the following steps:

- Accept the file upload and store its metadata in the file_details table.
- Create a new record in request_details with the current requested_date and requested_time.
- Retrieve all VMs from vm_details where status = 'active' and radio = 0 (not marked for migration).
- For each available VM, calculate Time to Complete (TTC) = file_size / VM_MIPS + current_queue_load.
- Obtain the Deadline value based on the user's SLA subscription tier.
- Choose the VM with the smallest TTC that satisfies TTC ≤ Deadline (SLA-compliant allocation).
- If no VM meets the SLA condition, select the VM with the lowest TTC overall and mark it for migration (radio = 1).
- Update the request_details entry with the selected vm_id and change status to 'scheduled'.
- Refresh the Task Scheduler dashboard to reflect the new assignment.

C. Phase 2: SLA Monitoring and Migration Procedure

Once tasks are assigned, the SLA monitoring engine performs continuous checks:

- Retrieve all request_details records where status = 'in-progress'.

- For each task, compute elapsed_time = current_timestamp − requested_time.
- If elapsed_time exceeds Deadline, mark the task as SLA_violated and set vm_details.radio = 1 for the assigned VM.
- Identify an underloaded VM by selecting one with status = 'active', radio = 0, and the lowest current TTC.
- Adjust MIPS values: reduce overloaded_VM.MIPS to 50% and increase underloaded_VM.MIPS to 150%.
- Migrate the pending workload by updating request_details.vm_id to the selected VM and reset radio indicators.
- Store differenced_time = completion_timestamp − requested_time in request_details.
- Recalculate Makespan as the maximum differenced_time among completed tasks in the batch.

The algorithm concludes once all tasks reach status = 'completed'. The final Makespan is then displayed on the Cloud Administrator dashboard. This structured two-stage method ensures that all cloudlets are processed without rejection while reducing both Makespan and SLA violations.

D. Time Complexity Analysis

Let *n* represent the number of tasks and *m* the number of available VMs. Phase 1 (task allocation) runs in O(n × m) time since each task is evaluated against every VM to determine the best SLA-compliant TTC. The sorting step within the Cloudlet Scheduler contributes an additional O(n log n) cost. Phase 2 (SLA monitoring) operates as a background polling mechanism with O(n) time per cycle. Therefore, the overall time complexity is O(n × m + n log n), which remains manageable for typical cloud environments where *n* and *m* are in the hundreds. Space complexity is O(n + m), accounting for task queues and VM state tracking.

10. Implementation Details

A. Django Application Structure

The Django project is divided into three separate apps based on system roles: cloud_admin, task_scheduler, and user_app. Each app includes its own models.py, views.py, urls.py, and templates folder. The main project file, settings.py, defines the MySQL database connection using Django's DATABASES configuration with the WAMP/XAMPP MySQL engine. It also includes all three apps in the INSTALLED_APPS list and specifies MEDIA_ROOT for storing uploaded files. Static content is managed through Django's staticfiles framework, with WhiteNoise middleware ensuring proper handling in production environments.

The Load Balancing Algorithm is implemented in a separate module called lba_engine.py within the cloud_admin app. This module provides two main functions: assign_task(request_id) for Phase 1 task assignment and monitor_sla() for Phase 2 SLA monitoring and migration. The monitor_sla() function runs periodically through a Django management command, scheduled as a background task using Windows Task Scheduler during development.

B. Key Django Views

The file upload view in user_app processes multipart form submissions, checks file size and type, stores the file content as base64-encoded LONGTEXT in the MySQL database, and generates a secure file_key using Python's secrets module for safe access. The load balancer view in cloud_admin retrieves active VMs using Django ORM's select_related() method to reduce database queries, calls the lba_engine, and returns a JSON response used by the real-time dashboard. The Makespan view calculates total execution time by aggregating differenced_time values using Django's Aggregate and Max functions across completed request_details records.

C. Frontend Implementation

The frontend uses HTML5 semantic elements, CSS3 Flexbox for responsive layouts, and plain JavaScript for live dashboard updates. Each module includes its own base template containing navigation menus, role-specific options, and user session details. Additional templates extend the base layout using Django's template inheritance system with **{%** extends **%}** and **{%** block **%}** tags. Form validation is handled on the client side using HTML5 validation features and on the server side using Django's form validation system, ensuring both usability and data reliability.

D. Security Implementation

The system includes multiple security measures following Django's recommended practices. Passwords are securely stored using Django's PBKDF2-SHA256 hashing algorithm with a unique salt. User registration includes OTP-based email verification, where a six-digit code is generated and sent through Django's email system. CSRF protection is enabled globally using Django's CsrfViewMiddleware, with **{%** csrf_token **%}** included in all forms. Session handling uses Django's secure signed cookie mechanism with configurable session duration. SQL injection risks are eliminated by Django's ORM, which uses parameterized queries and prevents direct insertion of user input into SQL statements.

11. System Testing and Validation

A. Testing Methodology

The proposed system was evaluated through a structured testing strategy comprising unit testing, integration testing, and performance testing.Unit testing was conducted using Django's built-in TestCase framework. These tests validated individual model methods, view logic, and the core functions of the LBA engine to ensure correctness at the component level. Integration testing examined the complete workflow of the system, beginning with user file upload and progressing through task scheduling, VM allocation, SLA monitoring, migration (if triggered), and final Makespan calculation. This ensured seamless interaction between the database layer, business logic, and user interface. Performance testing simulated concurrent user requests using Python's threading module. These simulations assessed system stability, response time, and load balancing efficiency under multiple simultaneous submissions, confirming that the system maintains consistent behavior under moderate workload conditions.

B. *Functional Test Cases*

Test Case	Input	Expected Output	Result
User Registration	Valid email + OTP	Account created, OTP verified	Pass
File Upload	PDF, 2MB file	Stored in DB, request created	Pass
Task Assignment (No Violation)	TTC < Deadline	Task assigned, status = scheduled	Pass
SLA Violation Detection	TTC > Deadline	Migration triggered, radio = 1	Pass
VM Migration	Overloaded VM	Workload moved, MIPS reconfigured	Pass
Makespan Calculation	5 completed tasks	MAX(differenced_time) returned	Pass
Admin VM Addition	VM config form	New VM in vm_details table	Pass
Load Balancer Dashboard	3 active VMs	Real-time load display	Pass

TABLE 4. FUNCTIONAL TEST CASES AND RESULTS

C. *Performance Testing Results*

Performance evaluation was performed by simulating 10, 20, 30, 40, and 50 concurrent user submissions. Response time was measured from the moment a file upload request was received until the task assignment confirmation was returned. At 10 concurrent users, the average response time was 1.2 seconds. At 50 concurrent users, the average response time increased to 4.8 seconds**,** with a maximum observed delay of 7.1 seconds. These values remain within acceptable operational thresholds for cloud management applications. Database optimization using Django ORM's select_related() and only() methods significantly improved efficiency, reducing average query execution time by 38% compared to the non-optimized baseline configuration.

D. *Validation Against Research Objectives*

The system was evaluated against the four primary research objectives defined earlier in the study:

- Objective 1: Minimize Makespan – Achieved with a 37% reduction compared to the existing Dynamic LBA.
- Objective 2: Maximize Resource Utilization – Achieved with 78% utilization**,** outperforming the 61% recorded by Dynamic LBA.
- Objective 3: Reduce SLA Violations – Achieved with a 63% reduction in violation rate.
- Objective 4: Deployable Web Implementation with Multi-Role Interface – Successfully implemented through a three-role Django application supporting full CRUD functionality for VM management, task scheduling, and user file management.

All four research objectives were fully satisfied by the proposed system.

12. Conclusion

This study presented an enhanced Load Balancing Algorithm (LBA) for the IaaS cloud computing model, implemented as a complete Django-based web application using Python 3.10, MySQL, and modern frontend technologies. The proposed two-layer architecture integrates Cloudlet Scheduler Time-Shared task allocation with an SLA-aware VM migration engine to enable efficient resource distribution, reduced Makespan, and continuous SLA compliance monitoring.

Experimental results demonstrate that the proposed LBA achieves 78% resource utilization, compared to 61% for the existing Dynamic LBA, while reducing Makespan by 37% and decreasing SLA violations by 63% across a benchmark of 50 heterogeneous cloudlets. The system effectively manages dynamic task arrivals, varying task sizes, and real-time SLA parameters, addressing limitations observed in traditional Round Robin and FCFS-based scheduling methods.

The developed web application provides a comprehensive management interface for three distinct roles—Task Scheduler, User, and Cloud Administrator—supporting practical deployment in academic and enterprise cloud environments. The MySQL-backed relational database ensures transactional consistency across core tables, and the Django MVT architecture supports scalability, maintainability, and extensibility.

Future enhancements will incorporate additional SLA metrics such as migration frequency, VM energy consumption, and cost-per-task into the scheduling objective function. The algorithm will also be extended to support automatic VM auto-scaling and evaluated against advanced scheduling techniques including Min-Min, Max-Min, and Ant Colony Optimization-based algorithms. Integration with production cloud platforms such as OpenStack and Amazon EC2 is planned to validate performance beyond local simulation and demonstrate readiness for large-scale industrial deployment.

References

[1]R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[2] D. Warneke and O. Kao, "Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 985–997, Jun. 2011.

[3] P. Singh and I. Chana, "QoS-based Resource Scheduling Using CLOURAM for Resource Management in Cloud Computing," *Journal of Grid Computing*, vol. 14, no. 1, pp. 59–82, Mar. 2016.

[4] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, Perth, Australia, 2010, pp. 27–33.

[5] A. Kumar, S. Jain, and P. Singh, "Cloud Computing Virtualization of Resources Allocation for Distributed Systems," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 5, pp. 450–456, 2017.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and

M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.
[7] M. Raj and S. Selvi, "Dynamic Load Balancing Algorithm for Minimizing Makespan in Cloud Computing," *International Journal of Computer Applications*, vol. 112, no. 10, pp. 12–17, Feb. 2015.
[8] A. Mishra, S. Jain, and A. Durresi, "Cloud Computing: Networking and Communication Challenges," *IEEE Communications Magazine*, vol. 50, no. 9, pp. 24–25, Sep. 2012.
[9] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *Proceedings of the Grid Computing Environments Workshop*, Austin, TX, USA, 2008, pp. 1–10.
[10] P. Singh and I. Chana, "A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges," *Journal of Grid Computing*, vol. 14, no. 2, pp. 217–264, Jun. 2016.
[11] T. Dillon, C. Wu, and E. Chang, "Cloud Computing Issues and Challenges," in Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications, 2010, pp. 27–33.
[12] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud Computing: The Business Perspective," *Decision Support Systems*, vol. 51, no. 1, pp. 176–189, Apr. 2011.
[13] R. Buyya, "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, Australia, 2009, pp. 1–2.
[14] N. R. Jennings and M. Wooldridge, "Agent-Oriented Software Engineering," in *Handbook of Agent Technology*, AAAI/MIT Press, 2000.