International Journal on Perceptive and Cognitive Computing (IJPCC) Vol 12, Issue 1 (2026)
https://doi.org/10.31436/ijpcc.v12i1.679

Optimizing Load Balancing Framework for a Distributed Local Network

Ubaid Ajaz[1], Zainab S. Attarbashi[1], Sara Babiker Omer Elagib [2], Aisha Hassan Abdalla Hashim[2]
[1] Department of Computer Science, International Islamic University of Malaysia.
[2] Kulliyyah of Engineering, International Islamic University of Malaysia

*Corresponding author zainab_senan@iium.edu.my
(Received: 9th December 2025; Accepted: 2nd January 2026; Published on-line: 30th January 2026)

Abstract— Load Balancing is a critical and foundational challenge in systems and network performance, especially in resource-constrained infrastructure environments. In which it requires careful alignment between infrastructure limited resources and performance requirements. This paper presents a lightweight deployment of a locally hosted web server on a small local network using off-the shelf devices. The observations of this paper indicate effective distribution of traffic evolving through different deployment stages. One node setup was implemented to be a baseline for performance comparison. And a 2-nodes setup was built using NGINX to provide the required load balancing. Both implementations were tested using load testing tools: Locust and Siege. Results were then compared based on standardized performance metrics: scalability, response time, throughput, and server load. The 2-nodes implementation showed near-linear scalability, with doubled throughput and CPU load dropped to 45%.

Keywords— *Load Balancing, Resource Constrained, Local Network, Algorithms, Performance Metrics*

I. Introduction

Local network infrastructure plays a crucial role in establishing a scalable environment for a web server to ensure high availability and performance, especially if there are limited resources. A web server or application specifically deployed for the purpose of cybersecurity applications and awareness demands a network infrastructure that can handle high volumes of traffic and requests. This is where the concept of load balancing requires immediate attention and involvement. Cloudflare simply defines load balancing as the practice of distributing computational workloads between two or more computers. Load balancing ensures even distribution, maximizes resource allocation, and provides fault tolerance, preventing any single component of the network from becoming a bottleneck [1], [2].

Currently, there is a lot of existing work and research on load balancing techniques and cloud computing in general. These studies focus on the comparison of existing load balancing algorithms alone or discuss how load balancing works in a cloud infrastructure in a broader sense [3], [4]. However, there is a lack of studies focusing specifically on the deployment of a load balancing setup in a small-scale environment offering cost-effective solutions. While setting up a locally hosted environment, acquiring high-performance servers can prove to be a significant financial challenge. Existing lab equipment lacks the required processing power and memory capacity to support and handle a fairly large number of users accessing the servers for simultaneous use. Thus, implementing an optimized local network with such off-the-shelf devices using load balancing would provide a cost-effective alternative to buying expensive servers and proprietary software [5]. This will enhance the computational capacity of the whole network to satisfy the needs of the system.

For local small-scale infrastructures, in a university lab for example, with only a few servers in a local network, users may face overload when multiple simultaneous connections from users are there. In such scenarios, this can lead to service slowdowns and failures. Effective load balance can enhance applications performance in these situations and ensure smoother user experience and improved traffic control.

The rise of lightweight deployment frameworks such as containerization with Docker and open-source load balancers has made it feasible to implement robust load balancing without enterprise-grade hardware [6], [7]. Technologies like Docker allow services to be containerized and run on commodity hardware, while software-based load balancers can efficiently distribute traffic at low cost. Example of open-source software load balancers are NGINX [8] and HAProxy [9]. They are both popular for Linux and they support load balancing in layer 4 and 7. Other emerging and popular software load balancers are Traefik [10] and Envoy [11]. They offer more flexibility of configurations and availability on integrations with container-based platforms. [12]

The aim of this study here is to investigate some working load balancing technologies that can apply to small-scale local networks, and work towards adapting such technologies for the enhanced distribution of resources over a limited system. The very goal is to design and implement a lightweight load balancing mechanism able to manage the changing loads in an efficient manner in a basic lab setup. The research seeks to answer how such load balancing can be optimized using freely available tools like HAProxy, what components are best suited for modest infrastructure, and which metrics—such as throughput, fault tolerance, and memory usage can effectively evaluate performance [13].

This current accessibility of technology and the open-source community enable small organizations to achieve reliability close to that provided by large data centers, but it also raises research questions. How well do standard load balancing algorithms perform in a resource-constrained local network? What trade-offs arise when using tools like HAProxy and NGINX on minimal hardware? These questions are significant for academia and small enterprises aiming to optimize performance without significant investment [14]. This paper answers the questions set forth by analyzing the implementation of a locally hosted CipherQuest, a cybersecurity training and Capture the Flag (CTF) event platform. The system is evaluated through three different evolutionary phases: from the entry-level single-node server to the more complex dual-node cluster with rudimentary splitting of traffic, and finally to the robust dual-node arrangement with a dedicated HAProxy load balancer. Using performance parameters, such as response time, throughput, CPU load, and so on, the results were contrasted among different traffic management schemes. The aim is to identify how well traffic could be distributed in a small local network and which algorithms therefore provide the best performance under that constraint. The paper's outcome sheds light on the viability of setting up load balancers in the limited environment and offers some points to consider for similar deployments in the future.

The reminder of this paper is organized as follows. Section II presents research methodology and load balancer design and implementation. Section III illustrates experimentation results, and results discussion and performance evaluation in section IV. Paper ends with a conclusion in section V.

II. METHODOLOGY

This study adopts mixed-methods experimental research design, combining quantitative performance metrics (e.g., response time, throughput, CPU usage) with qualitative system behavior observations (e.g., failover response, load distribution consistency). The goal is to investigate how lightweight load balancing mechanisms perform in a small-scale, resource-constrained local network and to evaluate their efficiency under simulated traffic.

A. Environmental Setup

The experimental testbed for this paper comprises a distributed local network consisting of two nodes. These nodes are off-the-rack computers running on 8GB RAM each connected via ethernet provided by the university LAN (1000 MBps). The operating system chosen was Linux with the Ubuntu LTS distribution, given its optimization for server tasks and lightweight deployment, catering to the need for managing web applications and eliminating licensing costs compared to a Windows Server. This setup reflects a resource-constrained environment typical of a small lab or classroom setup. The application deployed is an open-source Capture-The-flag web platform used for cybersecurity training exercises. This application serves as a representative workload for this research that generates typical web traffic (multiple concurrent users, database queries, dynamic content). To ensure portability and ease of management, containerization surrounds the full application stack using Docker containers. These lightweight virtualized units that packed the whole application, dependencies, database and load balancer. This abstracts the differences in the underlying OS environments and provides a suitable platform for testing different node setups and load balancing algorithms with simulated traffic to gather results in terms of performance metrics which will be later discussed in the paper.

To scrape and gather performance metrics of the servers, the infrastructure was implemented and tested in two phases:

- *Stage 1: Single-Node Deployment (Baseline):* In the initial setup, only one PC was used to host the CTFd container. A NGINX web server was running as a reverse proxy on the same host to forward HTTP requests to the container (and serve any static assets). NGINX was configured to listen on the host's IP and route all traffic to the local CTFd application. This stage had no load balancing since only one node served all requests. It established baseline measurements for how the application performs on a single server under load and these measurements are then used to do a comparative analysis of performance metrics with a load balancing strategy introduced later.

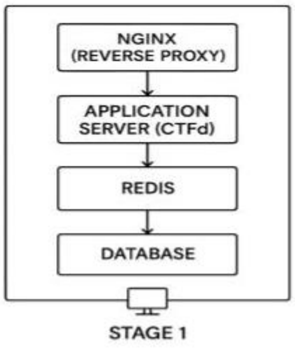

Fig. 1 Single Node Deployment Architecture

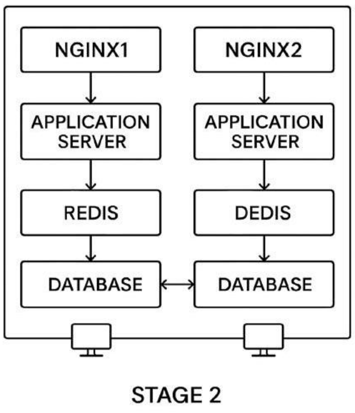

Fig. 2 Dual Node Deployment Architecture

- *Stage 2: Dual-Node with Load Balancing:* In this stage, a second PC was added to create the distributed setup comprising two application servers independently running Dockerized instances of CTFd and synchronizing the database. Each machine was configured with its own NGINX reverse proxy for local handling of incoming web traffic from containers. NGINX was the main load balancer in this scenario: on each of the two servers, NGINX was statically configured with an upstream block listing the CTFd instances of both the local and remote server with different load balancing algorithms, depending on which scenario was being tested. This decentralized style of balancing ensured that no single node would become a bottleneck while also avoiding the need for a central load balancer. Although these strategies were limited, lacking features such as health checks or adaptive request routing, they offered a lightweight and functional method for traffic distribution. This stage demonstrated the feasibility of achieving basic load balancing using reverse proxy infrastructure, providing early performance improvements without additional overhead.

B. Load Generation Tools

For this experimental setup, we used load generation tools to simulate clients and generate load on the system. We used three different industry standard tools: Apache Jmeter, Siege and Locust. Siege, an HTTP load testing and benchmarking utility tool is a command-line-based tool that triggers a preset number of concurrent users (threads) hitting an appropriate URL or set of URLs and consequently reports the elapsed time, response time, and throughput (transactions per second). Siege was used for fast stress tests, such as bombardment of the CTFd home page or challenge endpoints at 50, 100, 200 concurrent hits to analyze pure throughput and server behavior. Locust is a much more flexible load testing framework in which users exist within Python codes. Locust lets you define more complex user behaviors in Python code that can scale to simulate millions of users. We wrote simple Locust scenarios to simulate a typical user session in CTFd: logging in, fetching the scoreboard, and opening a challenge page. Locust gave us further stats (response time distribution, response time percentiles) and allowed us to gradually ramp up users. This tool was particularly useful in observing the system's behavior over time under sustained load and for checking if an algorithm causes queue build-up on one server or not. These two tools, while giving us a myriad of information, gave us insight into the systems from a high-level perspective of user experience and a low-level perspective of request throughput.

C. Monitoring and Metrics

To gather fine-grained performance data from the running system, Prometheus and Grafana were set up as the monitoring stack. Prometheus is the open-source monitoring solution that gathers the metrics data and stores it in a time series database. Prometheus was configured to scrape the metrics from each component: the Linux system metrics on each of the nodes (CPU, memory, network usage) via Node Exporter, Docker metrics including CPU usage per container for specificity and custom metrics from application if there were any (limited to infrastructure metrics, as CTFd being a flask app did not natively expose Prometheus metrics). The traffic simulation tools come with their own statistics which were gathered manually and through saved logs.

Grafana was then employed to visualize these metrics via dashboards. It is an open-source analytics and interactive visualization web application. Custom dashboards for key performance indicators were set up, such as CPU usage of a node, memory usage, load average, network traffic, requests handled by a server, response time graphs over a test, which allow us to cross-check the monitoring with the Siege/Locust results. If Siege reports a slowdown at 200 users, for instance, that monitoring can tell whether the CPU of one node maxed out or if memory or networking became a bottleneck. Monitoring also ensured that our test environment was fully operational (e.g., if one node went down, we would see it in the metrics immediately and could investigate).

The performance metrics mentioned below serve to evaluate system behavior underload and the efficiency of traffic distribution, and the overall resource consumption by nodes. Each metric therefore exposes one or the other facet of locally distributed network performance through different stages.

1. Latency or Response Time

Measured as the end-to-end time taken for a request to be processed from the client to the server and back. This includes network latency, processing time on the server, and any delays introduced by the load balancing mechanism. Measurement was made for both median response times and for the 90th percentile not only to capture the average performance but also the performance in peak load conditions.

2. *Throughput*

The number of requests successfully processed by the system per unit time, typically measured in requests per second (RPS). Throughput provides an indication of the system's capacity to handle high volumes of traffic.

3. Server Load Balancing

Balanced load distribution among the servers forms the heart of all load balancing strategies. Ideally, two servers with exactly similar capabilities would share the whole workload exactly equally.

4. CPU and Memory Utilization

Monitoring CPU and memory usage on each node provides a low-level view of how efficiently resources are being used. CPU utilization trends can reveal whether traffic is being spread effectively, while discrepancies in CPU or memory load between nodes may suggest imbalances in request handling. Memory usage is also tracked to ensure containerized services remain stable under stress, though in lightweight deployments like CTFd, memory is typically a secondary concern unless large-scale concurrency is involved.

D. Testing Procedure

For each stage of deployment (and for each load balancing algorithm in Stage 2,3), we conducted a series of tests to measure performance:

Baseline single-node test: Using Siege, we bombarded the single-node setup with incremental loads: a varied number of concurrent users (each user continuously requesting a mix of pages). We recorded the average response time, throughput (requests per second), and observed the system resource usage. This established the capacity of one server and provided critical baseline data for later comparison with distributed configurations.

Dual-node manual split test: We repeated similar load tests on the Stage 2 setup. When using NGINX's upstream approach, we examined NGINX logs to confirm that roughly half the requests went to each node. The performance metrics here illustrate the benefit of having two servers. During this stage, multiple NGINX-supported algorithms were tested under identical conditions to analyze their behavior in a dual-node setup. Specifically, round-robin (default), least-connections (which routes new requests to the less busy server), and IP-hash (which maps requests to a server based on client IP) were configured and evaluated. The same load testing tools—Locust, Siege, and Apache JMeter—were used to simulate user traffic.

III. Results

This section visualizes the load testing metrics collected during Stage 1, where a single-node architecture was deployed using one PC hosting all core services (CTFd application, database, Redis, and NGINX). Two types of load tests were conducted using Locust and Siege.

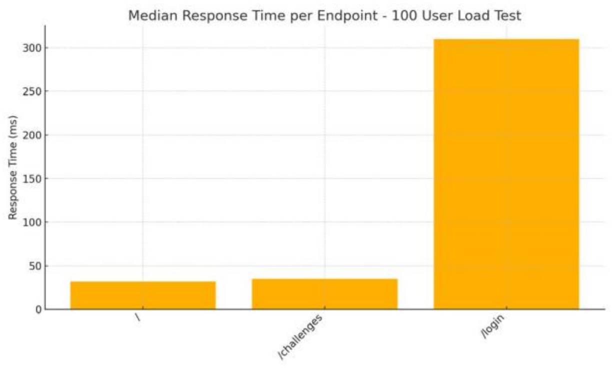

Fig. 3 Locust Results with heavy load testing (100 users)

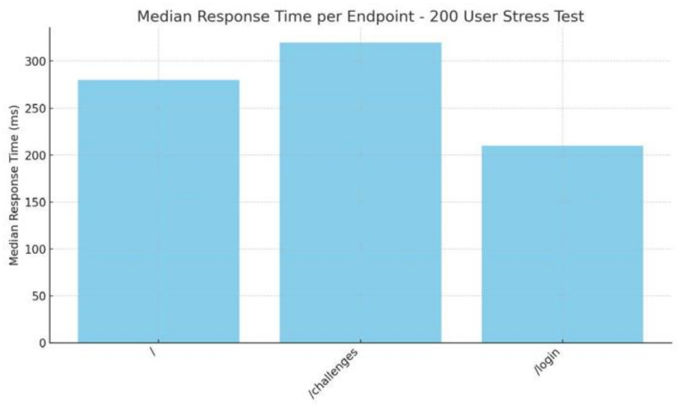

Fig. 4. Locust Results with stress load testing (200 users)

TABLE I
SIEGE RESULTS WITH HEAVY LOAD TESTING

Metric	Value Result
Test Duration	5 minutes
Concurrent Users	255 (user cap hit)
Transactions	60,837
Success Rate	100% (0 failures)
Avg. Response Time	1.24 seconds
Transaction Rate	203.28 requests/sec
Throughput	38.08 MB/sec

TABLE II
SEIGE RESULTS STAGE 2

Metric	Value Result
Test Duration	5 minutes
Concurrent Users	255 (user cap hit)
Transactions	79,320
Success Rate	100% (0 failures)
Avg. Response Time	0.72 seconds
Transaction Rate	264.4 requests/sec
Throughput	49.61 MB/sec

This section visualizes the load testing metrics collected during Stage 2, where a two-node load-balanced architecture was deployed. In this setup, core services such as the CTFd application, database, Redis, and NGINX were distributed across two PCs, with NGINX acting as a reverse proxy to balance incoming traffic between the nodes. The same load testing tools, Locust and Siege, were used to evaluate performance under identical conditions as Stage 1.

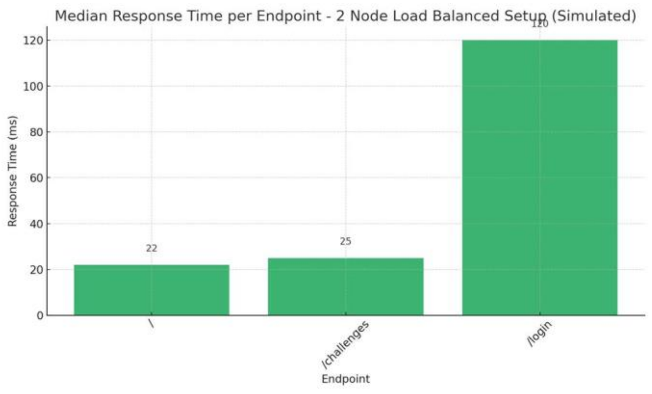

Fig. 5. Locust Results with heavy load testing (100 users)

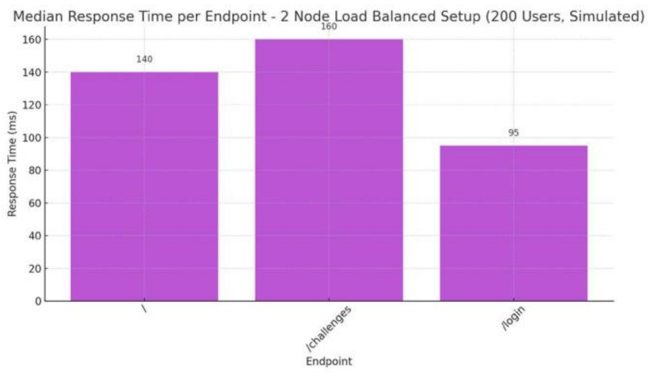

Fig. 6. Locust Results with stress load testing (200 users, increased ramp up)

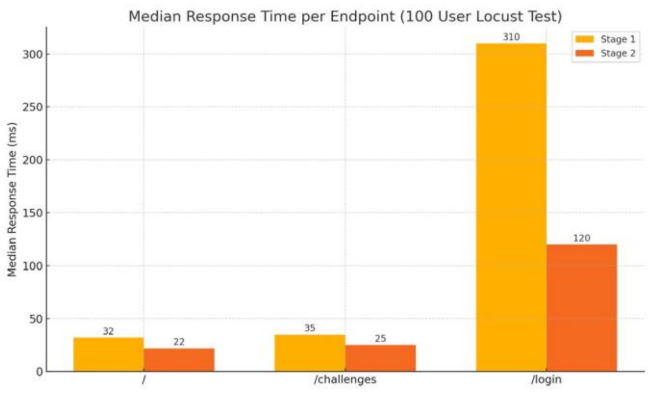

Fig.7 Stage 1 and Stage 2 Comparison for heavy load test

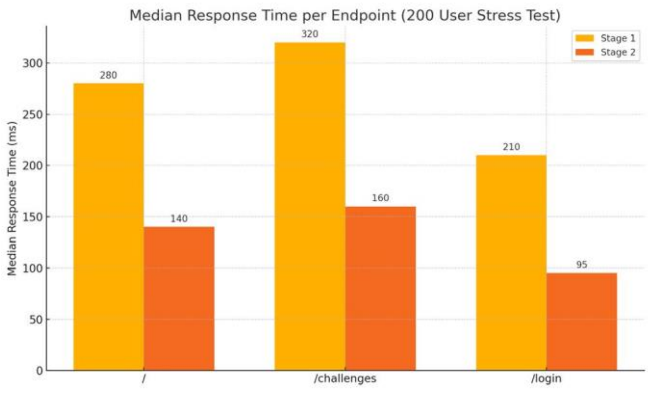

Fig. 8 Stage 1 and stage 2 comparison of stress load test

IV. DISCUSSION

The findings of this study clearly demonstrate the performance benefits of introducing a two-node distributed architecture in a local network environment using off-the-shelf machines and free software tools. Through both controlled (100-user) and stress-level (200-user) load testing, the system showed substantial improvements in throughput, response time, and load distribution when transitioned from a single-node to a dual-node setup with load balancing via NGINX.

The single-node deployment, although sufficient for moderate traffic, quickly reached saturation under heavier loads, leading to CPU utilization nearing 90% and median response times exceeding 300 ms for critical endpoints like /login. In contrast, the two-node setup reduced peak CPU loads per server to approximately 45–50%, effectively halving the processing burden on each machine. Median response times dropped significantly by over 50% in some endpoints and overall throughput more than doubled, showcasing linear or near-linear scalability within the constraints of the testbed. Despite these gains, several limitations emerged. The project operated under strict hardware constraints, with only two physical machines available for deployment. This prevented the implementation of a more advanced Stage 3 architecture, which would have introduced a dedicated load balancer node capable of supporting adaptive algorithms such as least-connections, IP-hash, or weighted round-robin. These strategies could not be fully explored, as true comparative analysis of load balancing algorithms typically requires a larger server pool to distribute traffic across diverse backend conditions. A more intelligent, feedback-aware load balancer could mitigate such issues, but its implementation was beyond the resource scope of this project.

V. CONCLUSION

In conclusion, this work validates the hypothesis that a lightweight, locally-hosted web server cluster with NGINX load balancing significantly enhances performance in resource-constrained environments. The two-node deployment achieved near-linear scalability, doubling throughput while reducing individual server load. Future work should explore scaling beyond two nodes, incorporating dynamic load balancing algorithms, and monitoring with more granular observability tools to refine traffic distribution decisions to provide a practical, low-cost model for small-scale environments such as university labs, classrooms, or lightweight training platforms.

ACKNOWLEDGMENT

The authors hereby acknowledge the review support offered by the IJPCC reviewers who took their time to study the manuscript and find it acceptable for publishing.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

AUTHORS CONTRIBUTION STATEMENT

All authors contributed equally to this work.

DATA AVAILABILITY STATEMENT

There is no external or third-party data that support the findings of this study.

ETHICS STATEMENT

This study did not require ethical approval

REFERENCES

[1] N. Arora, P. Saha, and S. Sinha, “A review on load balancing algorithms in cloud environment,” *Int. J. Sci. Technol. Res.*, vol. 10, no. 1, pp. 142–148, 2021.

[2] R. Tripathi, D. Dutta, and S. Sanyal, “Load balancing for resource allocation in cloud computing using live migration of virtual machines,” *Procedia Comput. Sci.*, vol. 167, pp. 116–124, 2020.

[3] A. T. Akinwale and K. S. Adewole, “Performance evaluation of load balancing algorithms in cloud computing,” *Int. J. Comput. Appl.*, vol. 178, no. 36, pp. 1–6, 2019.

[4] S. Singh and I. Chana, “Cloud resource provisioning: survey, status and future research directions,” *Knowl. Inf. Syst.*, vol. 49, pp. 1005–1069, 2016.

[5] R. Kumar, A. S. Rajawat, and S. Arora, “A hybrid algorithm for efficient load balancing in cloud computing environment,” *Cluster Comput.*, vol. 23, no. 4, pp. 2619–2635, 2020.

[6] D. Merkel, “Docker: lightweight Linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[8] NGINX official website. NGINX, Inc. [Online]. Available: https://nginx.org/. Accessed: Jan. 28, 2026.

[9] HAProxy official website. HAProxy Technologies. [Online]. Available: https://www.haproxy.org/. Accessed: Jan. 28, 2026.

[10] Traefik Proxy official website. Traefik Labs. [Online]. Available: https://traefik.io/traefik. Accessed: Jan. 28, 2026.

[11] Envoy Proxy official website. Envoy Proxy. [Online]. Available: https://www.envoyproxy.io/. Accessed: Jan. 28, 2026.

[12] A. Johansson, *HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm*, B.S. bachelor’s thesis, Dept. of Informatics, Högskolan i Skövde, Skövde, Sweden, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-21475

[13] L. Youseff, M. Butrico, and D. Da Silva, “Toward a unified ontology of cloud computing,” in *Proc. 2008 Grid Comput. Environ. Workshop*, IEEE, 2008.

[14] S. Chaisiri, B.-S. Lee, and D. Niyato, “Optimization of resource provisioning cost in cloud computing,” *IEEE Trans. Serv. Comput.*, vol. 5, no. 2, pp. 164–177, Apr.–Jun. 2012.