Vol. 10, No. 02, July 2026, pp. 42-57

INTERNATIONAL JOURNAL OF INNOVATION IN ENTERPRISE SYSTEM

Journal homepage: https://ijies.sie.telkomuniversity.ac.id
e-ISSN: 2580-3050, DOI: https://doi.org/10.25124/ijies.v10i02.10429

Implementation of Auto-Scaling and Load Balancing on Proxmox VE Using Python and REST API

Naufal Hanif[1*], Dading Oktaviadi Resmiranta[2], M. Khaerul Ihsan[3], Tanwir[4], I Putu Hariyadi[5], Ondi Asroni[6]

[1] Teknologi Informasi
Universitas Bumigora, Mataram, Indonesia
naufal@universitasbumigora.ac.id

[2] Ilmu Komputer
Universitas Bumigora, Mataram, Indonesia
dading@universitasbumigora.ac.id

[3] Rekayasa Perangkat Lunak
Universitas Bumigora, Mataram, Indonesia
ihsan@universitasbumigora.ac.id

[4] Ilmu Komputer
Universitas Bumigora, Mataram, Indonesia
tanwir@universitasbumigora.ac.id

[5] Ilmu Komputer
Universitas Bumigora, Mataram, Indonesia
putu.hariyadi@universitasbumigora.ac.id

[6] Ilmu Komputer
Universitas Bumigora, Mataram, Indonesia
ondisembahulun@universitasbumigora.ac.id

[*]Corresponding Author

ARTICLE INFO

Article history:
Received 10 April 2026
Accepted 27 July 2026
Published 09 August 2026

Keywords:
Auto-scaling;
Load Balancing;
Proxmox VE;
Python API;
Virtualization

ABSTRACT

Auto-scaling and load balancing are essential for maintaining web service availability and performance under fluctuating workloads. This research implements an automated auto-scaling and load balancing system on Proxmox Virtual Environment (Proxmox VE) leveraging Proxmox REST API through Python scripts. The system monitors web container CPU usage at 5-second intervals, triggering scale-out by cloning LXC containers when average CPU exceeds 75% and scale-in when it falls below 30%, while ensuring at least one container remains active. Testing on a laboratory topology consisting of 1 template container, 1 load balancer, and 4 backend containers (IP range: 192.168.100.251-192.168.100.254) demonstrates average provisioning time of 96.0 seconds (range: 85.5-117.7s) with breakdown: API Clone (82.4s), Network Config (0.0s), Container Start (6.8s), Content Customization (3.1s), and Load Balancer Update (3.6s). Comparative analysis between always-on (4 containers) and auto-scaling scenarios reveals 36.1% CPU savings (31.66 vs 49.57 CPU-hours), idle time reduction from 54.2% to 28.3%, and efficient resource utilization with the system operating predominantly on 1 container (85% uptime). This implementation proves that Proxmox API integration with Python-based automation provides a practical auto-scaling solution for private clouds without complex orchestration platforms like Kubernetes.

1. INTRODUCTION

Elasticity is one of the most critical aspects in the adoption of cloud computing. The concept of elasticity enables applications running in cloud environments to automatically adjust their capacity in response to workload fluctuations, thereby maintaining performance. Consequently, elasticity allows applications to scale resources up or down efficiently while minimizing infrastructure usage costs [1]. Data centers, computing devices, and networking equipment consume substantial amounts of energy. Recent reports indicate that global energy consumption for data centers could more than double, rising from 460 TWh in 2022 to 1000 TWh by 2026 due to the expansion of AI and high-density computing. Furthermore, global data creation is projected to reach 175 trillion gigabytes by 2025, further driving the need for energy-efficient infrastructures [2]. Consequently, most of this energy is still generated by thermal power plants. These power plants predominantly rely on fossil fuels, such as coal and petroleum, which contribute to various environmental issues, including global warming, ozone layer depletion, and increased health risks that may lead to premature mortality among living organisms. As global awareness of these environmental impacts increases, recent research trends have shifted toward optimizing energy consumption and developing environmentally friendly technologies [3].

The selection of effective virtualization technologies plays a crucial role in mitigating the negative environmental impact of information technology usage in the public sector by optimizing resource utilization and improving the efficiency of IT infrastructure. Virtualization serves as a key mechanism in supporting the implementation of green computing concepts within organizations [4]. While public cloud services typically provide built-in auto-scaling mechanisms, these mechanisms often operate as 'black boxes' with limited user control over the scaling logic, forcing users to rely on vendor-specific metrics [5]. Furthermore, to strictly meet Service Level Objectives (SLOs), public providers frequently resort to aggressive resource over-provisioning. For instance, cloud providers tend to define conservative CPU utilization thresholds to prevent service degradation, particularly under highly fluctuating workloads. This practice leads to unnecessary costs and energy waste [6]. Similarly, standard lightweight orchestrators often lack the granularity required for complex workload patterns in private infrastructure [7]. Public cloud services typically provide built-in auto-scaling mechanisms; however, public cloud service providers often rely on resource over-provisioning to ensure that Service Level Objectives (SLOs) are consistently met. Cloud providers tend to define conservative CPU utilization thresholds to prevent service degradation, particularly under highly fluctuating workloads. This practice not only leads to resource wastage but also potentially increases energy consumption inefficiency, especially in large-scale cloud deployments [8].

Proxmox functions as the primary platform for managing all virtual machines (VMs) and the networks interconnecting them. Through Proxmox, the infrastructure gains a stable and feature-rich virtualization layer, including controllable storage and backup mechanisms that can be programmatically accessed via REST APIs or command-line interfaces (CLI). These capabilities facilitate seamless integration of Proxmox into Infrastructure as Code (IaC) workflows, enabling more efficient infrastructure automation and orchestration processes [9], [10]. Load balancing represents a critical issue in cloud computing, as it directly affects service availability, Quality of Service (QoS), performance evaluation, and the fulfillment of service contracts that cloud service providers (CSPs) must guarantee to client organizations. The primary objective of load balancing is to optimally distribute workloads across multiple computing resources, thereby improving resource utilization and significantly enhancing overall system performance [11].

In cloud environment, applications are run on different servers in a distributed mode [12]. Server needs load balancing remains a common challenge in cloud environments, complicating the ability of service providers to maintain application performance that complies with Quality of Service (QoS) parameters and Service Level Agreement (SLA) requirements agreed upon with customers. The primary challenge lies in evenly distributing workloads among available servers or virtual machines (VMs). Therefore, efficient load balancing techniques are required to optimize VM resource utilization while ensuring high levels of user satisfaction [13].

Python programming can be effectively utilized for network automation and abstraction. The use of Python enables faster and more efficient configuration processes with a reduced likelihood of human error compared to manual configuration. Furthermore, Python-based automation enhances network security by facilitating rapid detection and remediation of vulnerabilities, thereby contributing to improved system stability. Automation also allows system upgrades to be performed centrally and simultaneously, significantly accelerating network device management processes [14].

*Corresponding author: naufal@universitasbumigora.ac.id

This study designs and implements an automated auto-scaling and load balancing system using Python and the Proxmox API. The main contributions of this research are formulated as follows:

1. Development of a Lightweight Auto-scaler Architecture: Unlike complex orchestrators (e.g., Kubernetes) that require significant overhead, this study designs a lightweight auto-scaler specifically leveraging the Proxmox API. This approach offers a resource-efficient alternative suitable for private cloud environments with limited hardware resources.

2. Automated Provisioning for LXC Containers: We implement an automated provisioning mechanism for Linux Containers (LXC) and network configurations. In contrast to traditional VM-based scaling which entails slower boot times, this container-based approach ensures rapid response to workload spikes.

3. Dynamic Nginx Integration for Zero-Downtime: The system features dynamic integration of Nginx configurations within a container-based load balancer. This explicitly addresses the limitation of static load balancers by enabling seamless backend updates without service interruptions during scaling events.

2. METHOD

The method employed in this study adopts an Experimental Design approach as the foundation for the development and evaluation of the computational resource auto-scaling system. This approach was selected to ensure that each stage of the research is conducted in a systematic, measurable, and scientifically verifiable manner through experimental testing. The research framework consists of several sequential main phases, including the design phase, implementation phase, testing phase, and analysis phase. Each phase plays a critical role in ensuring the successful design, implementation, and performance evaluation of the developed system.

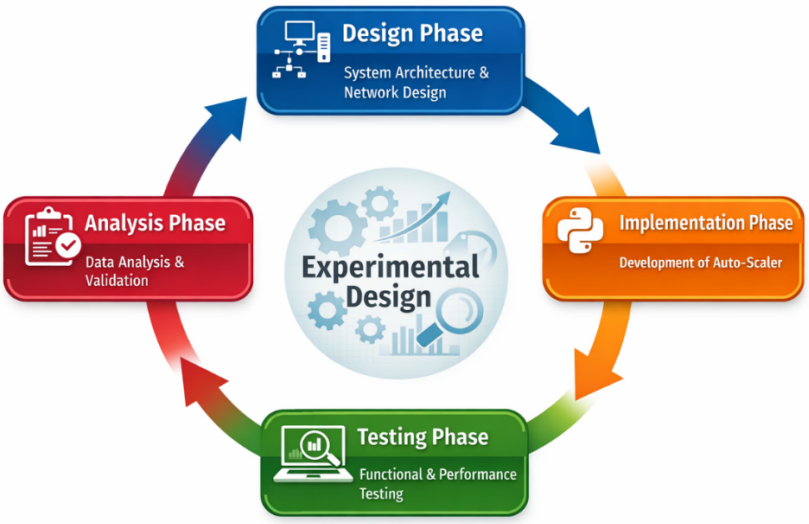

Figure 1 - Research Approach Method

In the design phase, the system architecture and network topology that serve as the operational foundation of the auto-scaler mechanism are designed. This phase includes the identification of system components, data flow definition, and the determination of testing scenarios to be employed in the study. Subsequently, the implementation phase focuses on the development of the Python-based auto-scaler system in accordance with the predefined design. During this phase, all system components are implemented and integrated to ensure that the system operates functionally as intended.

In the testing phase, functional testing and performance evaluation are conducted by applying specific workloads to observe the auto-scaler's response to changes in resource conditions. This testing aims to obtain relevant quantitative data to serve as the basis for further analysis. The final stage is the analysis phase, in which the experimental results are analyzed to evaluate the effectiveness of the system and to validate the research hypotheses. This analysis is used to assess whether the developed system is capable of achieving the research objectives optimally. Overall, this research framework ensures that the entire process, from system design to result analysis, is carried out in a structured and comprehensive manner, thereby enabling the research outcomes to be scientifically accountable.

*Corresponding author: naufal@universitasbumigora.ac.id

2.1. System Architecture

The system architecture employed in this study is designed to support automated provisioning of Linux container–based web server services within a Proxmox VE virtualization environment with auto-scaling capabilities. The system consists of several main components that are tightly integrated within a single local network, as described below.

1) Proxmox VE Node (pve)

Proxmox VE is a Debian-based distribution that has been modified and equipped with a kernel optimized for virtualization requirements. Although similar in concept and functionality to other virtualization platforms, Proxmox VE remains based on a distinct Linux distribution, resulting in certain differences in system implementation and configuration [15]. In this research, Proxmox VE functions as the virtualization host that runs all LXC containers. The node used has the hostname pve, with an API access address at 192.168.100.2:8006. All automation processes are performed through the Proxmox REST API, which is accessed using token-based authentication mechanisms (PVEAuthCookie and CSRFPreventionToken).

2) LXC Template (ID 100)

LXC provides an isolated environment that allows applications and processes to run independently of the host operating system. Containerization represents a form of operating system–level virtualization in which applications can run in isolated spaces without requiring full virtualization of the entire operating system. By encapsulating all required dependencies and files, containers offer high portability, compatibility, and independence from specific operating system types. Containers commonly used in cloud environments are generally lightweight and highly portable, enabling applications developed for one operating system to run seamlessly on another without code modification. To achieve high availability in container-based environments, operating system–level virtualization is employed, allowing multiple Linux instances to run concurrently on a single physical host [16]. In this study, a container template with ID 100 (TEMPLATE_CT_ID = 100) is used as the cloning base to accelerate the provisioning of new web servers. The template is preconfigured with a web server, PHP runtime, and minimal system packages. Full cloning from this template ensures configuration consistency across all generated instances.

3) Load Balancer Container (ID 102)

Cloud computing has rapidly evolved into a flexible and effective paradigm for addressing large-scale computing challenges. As the number of users and service demands increases, system workloads may become unbalanced, leading to issues such as increased response time or excessive energy consumption. Load balancing plays a crucial role in mitigating these issues and improving the performance of cloud-based servers. In general, load balancing techniques aim to identify underutilized or overloaded nodes and distribute workloads evenly among them [17]. A dedicated container with ID 102 (LOAD_BALANCER_CT_ID = 102) and IP address 192.168.100.3 runs Nginx as a reverse proxy and load balancer. Nginx acts as a single entry point that distributes incoming HTTP traffic to backend web servers using a round-robin algorithm. The Nginx upstream configuration is dynamically updated whenever backend servers are added or removed.

4) Backend Web Server Containers

Server clustering can be employed to address issues arising from high workloads on a particular service. In addition to functioning as a web server, Nginx also provides reverse proxy and load balancing capabilities. When used as a reverse proxy, Nginx can distribute requests to backend servers, reduce the load on the primary server, and cache frequently accessed pages to improve response times. Furthermore, Nginx forwards client requests to backend servers and returns their responses to users through the web interface. Both proxy and backend servers may operate on the same physical machine or on different infrastructures, depending on implementation requirements [18].

The backend web server containers are dynamic components assigned IP addresses within the range 192.168.100.251–192.168.100.254, providing a maximum capacity of four servers. Each web server is customized with an HTML page displaying container information (ID, IP address, and timestamp) and includes a health check endpoint for monitoring purposes. The container with IP address 192.168.100.251 is designated as the base server and is protected from automatic removal to maintain high availability.

5) Automation Engine (Python)

Python offers significant potential as an automation tool due to its flexibility and extensive library ecosystem, enabling improved decision-making, faster innovation processes, and simplified operational activities [19]. The Automation Engine serves as the core component of the system and is implemented in Python using a modular, class-based architecture. This engine is responsible for monitoring system load, cloning container templates, configuring networking, customizing web content, and automatically updating load balancer configurations.

6) Network Configuration
All system components are deployed within a single subnet, 192.168.100.0/24, which simplifies network management and inter-device communication. The network configuration uses 192.168.100.1 as the default gateway for external network access and 8.8.8.8 as the public DNS server for domain name resolution. Additionally, the search domain ubg.local is applied to support internal name resolution, enabling containers and services within the environment to communicate using hostnames without explicitly specifying IP addresses. This configuration provides a well-structured, manageable network environment that supports the automation processes implemented in the system.

The system architecture diagram is presented in Figure 2, illustrating the communication flow among the Proxmox API, Automation Engine, Nginx Load Balancer, Backend Containers, and Clients, as well as the auto-scaling mechanism based on predefined CPU utilization thresholds.

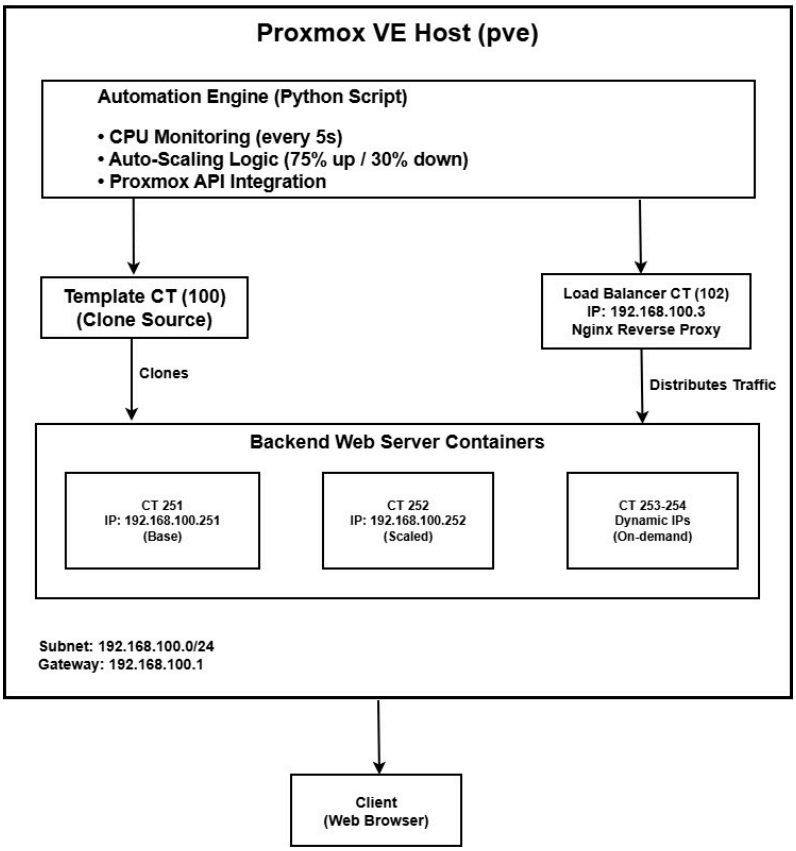

Figure 2 - System Architecture Diagram

2.2. Network Topology
This study employs a virtualization-based network topology. The topology is designed to represent a distributed service environment consisting of a load balancer, backend web servers, and an automated resource management mechanism. The network topology is designed with careful consideration of inter-component connectivity, ease of management, and flexibility in conducting controlled workload testing and system performance evaluation.

*Corresponding author: naufal@universitasbumigora.ac.id

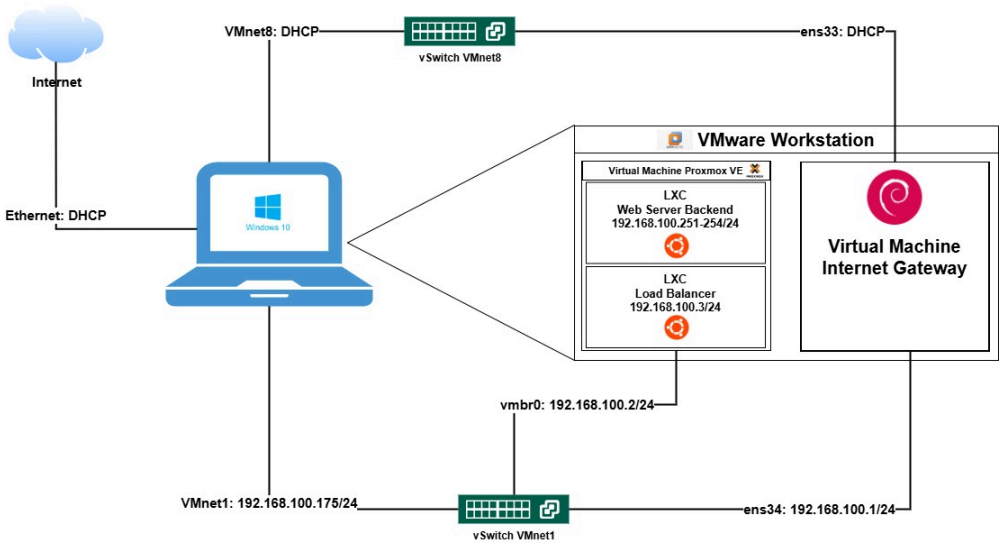

Figure 3 - Network Architecture Design

The network topology used in this study is illustrated in Figure 3 and is deployed within a virtualized environment using VMware Workstation running on Windows 10 as the host operating system. The host system is connected to the Internet via an Ethernet interface configured using the Dynamic Host Configuration Protocol (DHCP). VMware Workstation utilizes two virtual networks, namely VMnet8 and VMnet1, which function as the external connectivity path and the internal laboratory network, respectively. The VMnet8 network is configured in NAT (DHCP) mode and is connected through vSwitch VMnet8, allowing virtual machines to obtain indirect Internet access via the host. The network interface ens33 on the virtual machines is configured using DHCP and serves as the outbound communication path to the Internet. To support this external connectivity, a Debian-based Internet Gateway virtual machine is deployed to act as a gateway bridging the internal network and the external network.

The internal network uses VMnet1, which is configured with the network address 192.168.100.0/24 and connected via vSwitch VMnet1. Within this network, a Proxmox Virtual Environment (Proxmox VE) virtual machine operates as the primary virtualization platform. Proxmox VE is connected to the internal network through the ens34 interface with the IP address 192.168.100.1/24, and it provides a network bridge vmbr0 with the IP address 192.168.100.2/24, which is used as the communication channel among containers. Inside Proxmox VE, several Linux Containers (LXC) are deployed to form the core service architecture of the system. A dedicated container functions as an Nginx-based load balancer with the IP address 192.168.100.3/24, responsible for distributing client traffic to multiple backend containers. The backend web servers are hosted on multiple LXC containers with IP addresses ranging from 192.168.100.251 to 192.168.100.254, representing a maximum capacity of four active backend containers.

All backend containers are created from a single container template, enabling rapid and consistent cloning and provisioning processes. Automation and container management, including the dynamic addition and removal of backend servers, are controlled by an automation engine implemented as a Python script running on the Proxmox VE host. This automation engine leverages the Proxmox API to perform resource monitoring and execute auto-scaling actions in response to system load conditions. The architecture and network topology are designed to simulate a distributed service environment in a controlled manner, allowing the auto-scaling mechanism to be evaluated systematically and quantitatively in accordance with the research objectives.

The IP addressing scheme is designed to separate internal and external networks while ensuring stable and controlled communication among system components. Detailed IP address configuration used in the network topology of this study is presented in Table 1.

*Corresponding author: naufal@universitasbumigora.ac.id

Table 1 - IP Address Design Used

Host	Interface	IP Address	Subnet Mask
Host Machine	Ethernet	DHCP	DHCP
	VMnet8	DHCP	DHCP
	VMnet1	192.168.100.175	255.255.255.0
Proxmox VE	vmbr0	192.168.100.2	255.255.255.0
Internet Gateway	ens33	DHCP	DHCP
	ens34	192.168.100.1	255.255.255.0
Load Balancer	ens33	192.168.100.3	255.255.255.0
Web Server Backend	ens33	192.168.100.251-192.168.100.254	255.255.255.0

Based on Table 1, the internal network employs a static IP addressing scheme within the 192.168.100.0/24 subnet for all core service components, including Proxmox VE, the Internet Gateway, the load balancer, and the backend web servers. The use of static IP addresses in the internal network is intended to maintain connectivity consistency and to facilitate system monitoring and automation processes. In contrast, network interfaces connected to the external network are configured using DHCP to dynamically obtain IP addresses. This configuration supports flexible Internet access while maintaining isolation between the internal and external networks, thereby ensuring that the testing environment operates in a secure and well-structured manner.

2.3. System Implementation

Python is a powerful scripting language with a continuously evolving ecosystem of libraries, frameworks, and tools. These libraries and tools provide pre-written code that enables users to perform various operations while significantly reducing development time [20]. The proposed system is implemented using Python 3, utilizing the requests library for REST API communication and the subprocess module for executing pct (Proxmox Container Toolkit) commands. The implementation consists of approximately 670 lines of code organized in a modular structure that separates concerns among API communication, load balancer management, and scaling logic.

1) ProxmoxAPI Class

This class handles all communication with Proxmox VE through its REST API. Authentication is performed by sending a POST request to the /api2/json/access/ticket endpoint with a username and password payload. The response contains a ticket and a CSRFPreventionToken, which are subsequently set as cookies and headers for all subsequent API requests.

Container discovery is performed via a GET request to /nodes/{node}/lxc, which returns a list of all containers along with their status. To retrieve detailed network configuration, a GET request is issued to /nodes/{node}/lxc/{vmid}/config, which returns the net0 parameter containing the IP address, bridge, and gateway information.

Real-time monitoring is conducted through a GET request to /nodes/{node}/lxc/{vmid}/status/current, which returns CPU usage metrics in floating-point format (0.0–1.0), memory usage, and container status. CPU utilization values are converted into percentages by multiplying by 100. Container cloning is performed via a POST request to /nodes/{node}/lxc/{template_vmid}/clone with parameters specifying the target VMID (newid), full=1 for full cloning, and an optional hostname. The response returns a task_id, which is polled using a GET request to /nodes/{node}/tasks/{task_id}/status at 2-second intervals until the task reaches status=stopped with exitstatus=OK or a timeout of 300 seconds is reached.

Network configuration is applied using a PUT request to /nodes/{node}/lxc/{vmid}/config with a request body specifying the net0 parameter in the format name=eth0,bridge=vmbr0,ip={ip}/24,gw={gateway}, along with nameserver for DNS and searchdomain for domain resolution. Container lifecycle operations are managed using POST requests to start (/status/start) and stop (/status/stop) containers, as well as DELETE requests to remove containers and their associated storage.

2) LoadBalancerManager Class

This class manages Nginx configuration within the load balancer container and implements a three-layer fallback mechanism to ensure robustness. Load balancer discovery is performed by identifying a container with a predefined ID (102) or, as a fallback, by locating a container with the IP address 192.168.100.3 based on its net0 configuration.

Nginx configuration files are generated by constructing an upstream block containing the list of backend servers and a server block with proxy settings. The configuration follows standard Nginx syntax and includes proxy headers to preserve client information, such as Host, X-Real-IP, X-Forwarded-For, and X-Forwarded-Proto.

*Corresponding author: naufal@universitasbumigora.ac.id

The primary configuration update method uses the pct push command to transfer configuration files from the host to the container, followed by nginx -t for syntax validation and systemctl reload nginx for activation. If this method fails due to permission or file path issues, the system falls back to a second method based on Base64 encoding. In this method, the configuration content is Base64-encoded, transferred using pct exec, and decoded inside the container using a pipe to base64 -d, making it more robust against shell escaping issues.

The third and final fallback method writes the configuration line by line using echo commands with proper escaping for special characters, appending each line using the >> operator. Each method concludes with nginx -t validation to ensure configuration correctness and prevent service downtime. If validation fails, changes are discarded and the previous configuration remains active.

3) AutoScaler Class
This class coordinates the entire auto-scaling logic through a workflow consisting of initialization, discovery, monitoring loops, and scaling operations. During initialization, the system authenticates with the Proxmox API, discovers the load balancer container, and scans existing containers within the defined IP range to build the initial system state. The discovery mechanism allows the system to resume monitoring after restarts without disruption by identifying already running containers based on IP address matching within the net0 configuration.

The monitoring loop executes every 5 seconds, collecting CPU utilization data for all active containers via API calls. CPU values are aggregated, and the average CPU utilization is computed for decision-making. Containers that are unresponsive or not in a running state are automatically removed from the tracking list.

The scale-up procedure begins by identifying available IP addresses from the predefined pool (192.168.100.251–192.168.100.254) and available VMIDs from the range 251–254. The cloning process is initiated with a timeout of 300 seconds, followed by network configuration, container startup, a 10-second boot grace period, content customization using Base64-encoded HTML and PHP files, and an update to the load balancer configuration.

The scale-down procedure selects candidate containers by excluding the base server (192.168.100.251) and sorting remaining containers using a Last-In-First-Out (LIFO) policy based on the highest IP address. The selected container is removed from the tracking list, the load balancer configuration is updated first to ensure graceful traffic redirection, and the container is then stopped and deleted along with its storage. Content customization includes uploading an index.html file displaying container information with responsive styling and an endpoint returning JSON-based status information.

4) Configuration Parameters
The system utilizes configurable parameters defined in the script header, including Proxmox connection settings (host, port, username, password, node), load balancer settings (container ID, IP address, configuration path), container settings (template ID, IP range, gateway, DNS, domain), and auto-scaling thresholds (high CPU threshold: 75%, low CPU threshold: 30%, monitoring interval: 5 seconds).

5) Auto-Scaling Policy
Elasticity is a fundamental feature of cloud computing that enables users to acquire or release computing resources on demand, allowing web application providers to automatically adjust resource allocation under dynamic workloads while minimizing costs and meeting Quality of Service (QoS) requirements [21]. The proposed auto-scaling system implements a CPU threshold–based policy with three operational states derived from the average CPU utilization across all active containers.

A scale-out trigger occurs when the average CPU utilization exceeds 75% (HIGH_CPU_THRESHOLD). The system validates the availability of IP addresses and VMIDs before initiating a container clone from the template. Upon completion, the system configures networking, starts the container, waits for the boot process, customizes web content, registers the container in the tracking system, updates the load balancer configuration, and reloads the Nginx service. Protection mechanisms prevent scale-out actions when IP pools or VMID ranges are exhausted, generating warning logs instead.

A scale-in trigger occurs when the average CPU utilization falls below 30% (LOW_CPU_THRESHOLD) and more than one container is active. The system selects a removal candidate using a LIFO policy while excluding the base server. The load balancer configuration is updated first to ensure graceful traffic redirection, after which the container is shut down and deleted. Protection mechanisms ensure that at least one container remains active and that the base server is never removed to maintain service availability.

The normal operation state applies when CPU utilization remains between 30% and 75%. In this state, the system performs monitoring and logging only, without executing scaling actions, while continuously collecting metrics for analysis.

A recovery mechanism enables the system to resume normal operation after restarts or failures. During initialization, all running containers within the IP range 192.168.100.251–192.168.100.254 are identified and registered in the tracking system based on their network configuration, allowing the system to continue monitoring without creating duplicate containers or disrupting existing services. A hysteresis mechanism with a 45% gap between scale-up and scale-down thresholds (75%–30%) is applied to prevent thrashing behavior, ensuring sufficient stabilization time for workload adjustment without unnecessary resource churn.

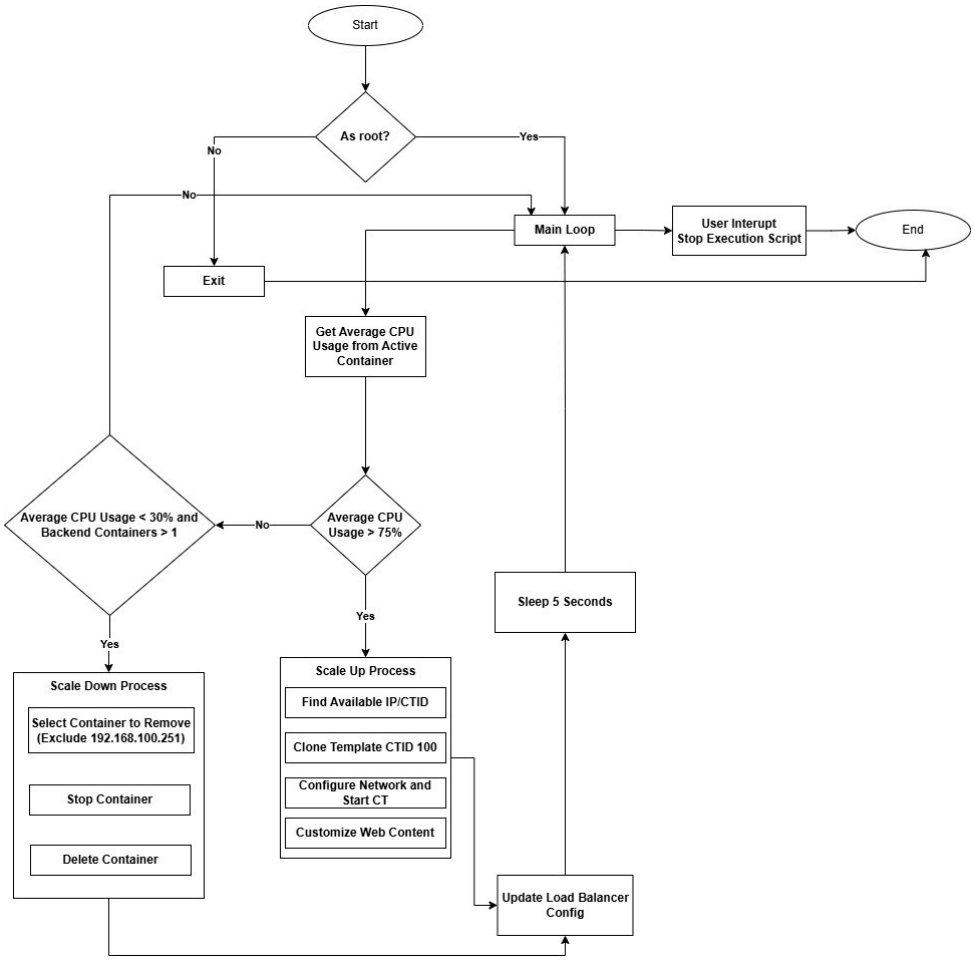

Figure 4 - Python Automation Engine Flowchart

2.4. Testing Scenario

System testing was conducted through three main scenarios designed to comprehensively evaluate functionality, performance, and resource efficiency.

The first scenario was a functional testing phase aimed at validating the core system functionalities. The testing procedure consisted of seven sequential stages, starting with Proxmox API authentication testing, discovery of existing containers, clone operations with timeout control, network configuration, content customization, load balancer updates using three fallback methods, and finally scale-up and scale-down operation testing. The success criteria required that all API calls return a response code of 200, clone completion time remain under 300 seconds, network configurations be correctly applied, load balancer configurations be updated without errors, and Nginx reload operations be executed without causing service downtime.

The second scenario focused on measuring end-to-end provisioning time to evaluate the system's speed in delivering new containers. The testing procedure began by triggering a scale-up event from one to two containers, followed by timestamp recording for each phase, including the start of the API clone call (t1), clone task completion (t2), network

configuration completion (t3), container startup (t4), content customization completion (t5), and load balancer update completion (t6). This experiment was repeated for ten iterations to ensure statistical significance. The measured metrics included per-phase execution time with calculations of mean, minimum, maximum, and standard deviation, total end-to-end provisioning time, and the success rate of each phase.

The third scenario was designed to compare resource consumption between two different approaches. This scenario consisted of two sub-scenarios: the always-on scenario and the auto-scaling scenario. In the always-on scenario, four containers were kept continuously active for one hour, while CPU usage was monitored every 30 seconds, producing 120 data samples, along with measurements of CPU utilization percentage, memory usage, and idle time. In contrast, the auto-scaling scenario started with a single container, and CPU workload was simulated using a stress tool executed directly on the backend web server container to trigger the automatic scaling mechanism within a range of one to four containers. The workload simulation was performed by executing stress commands on active containers to gradually increase CPU usage, forcing the system to scale up when CPU utilization exceeded 75% and scale down when it dropped below 30%. CPU usage monitoring was conducted at 30-second intervals for the same one-hour duration as the always-on scenario. The measured metrics in this scenario included total CPU-hours consumed, total RAM-hours utilized, the average number of running containers, the percentage of idle time with CPU utilization below 10%, and the frequency of scaling events, both scale-up and scale-down.

2.5. Evaluation Metrics

System performance evaluation was conducted by measuring four main categories of metrics, encompassing provisioning efficiency, resource utilization, scaling behavior, and workload performance.

The first category is provisioning performance, which assesses the system's speed in delivering new containers. The evaluated metrics include clone time (in seconds), calculated from the difference between API response timestamps; network configuration time, measured based on the duration of the PUT request; container start time, obtained through status polling via the API; content customization time, measured from the execution duration of the pct exec command; and load balancer update time, recorded from the duration of the Nginx reload process. The total end-to-end provisioning time is computed as the sum of all these phases, providing a comprehensive view of container readiness from initiation until it is fully capable of serving traffic.

The second category is resource efficiency, which evaluates the system's computational resource consumption. CPU consumption is measured in CPU-hours using the formula $\sum$(CPU% × sampling interval) to accumulate CPU usage over the observation period, while memory usage in GB-hours is calculated using the formula $\sum$(RAM_GB × sampling interval). The idle time percentage is determined by the ratio of samples with CPU utilization below 10% to the total number of samples, allowing identification of resource underutilization. The average number of running containers is calculated as the mean container count during the observation period to indicate overall system capacity utilization.

The third category is scaling behavior, which examines the dynamics of the auto-scaling mechanism. Scale-up frequency is measured in events per hour by counting the number of container addition operations, while scale-down frequency represents the number of container removal events within the same period. Thrashing events are recorded as occurrences of rapid scale-up and scale-down cycles, indicating potential system instability caused by overly sensitive thresholds or highly fluctuating workloads.

3. RESULT AND DISCUSSION

The implementation of the auto-scaling system on Proxmox VE resulted in an infrastructure composed of multiple LXC containers centrally managed through the Proxmox web interface. In the experimental environment, four primary containers were deployed, each serving a distinct role within the auto-scaling ecosystem. The container with ID 100 functions as the template for the cloning process, container 102 runs Nginx as the load balancer, while container 251 serves as the backend web server and acts as a base container that is not removed during scale-down operations. The containers subject to dynamic provisioning during scaling activities are containers 252 to 254.

The network configuration for all containers utilizes the vmbr0 bridge with IP address allocation within the 192.168.100.0/24 subnet and a default gateway at 192.168.100.1. The Proxmox VE management interface provides comprehensive visibility into the operational status of each container, including IP address information, MAC addresses, firewall configurations, VLAN tagging, and the execution history of tasks performed on the node. Figure 4 illustrates the Proxmox VE web interface displaying container 251 in a running state along with its detailed network configuration.

*Corresponding author: naufal@universitasbumigora.ac.id

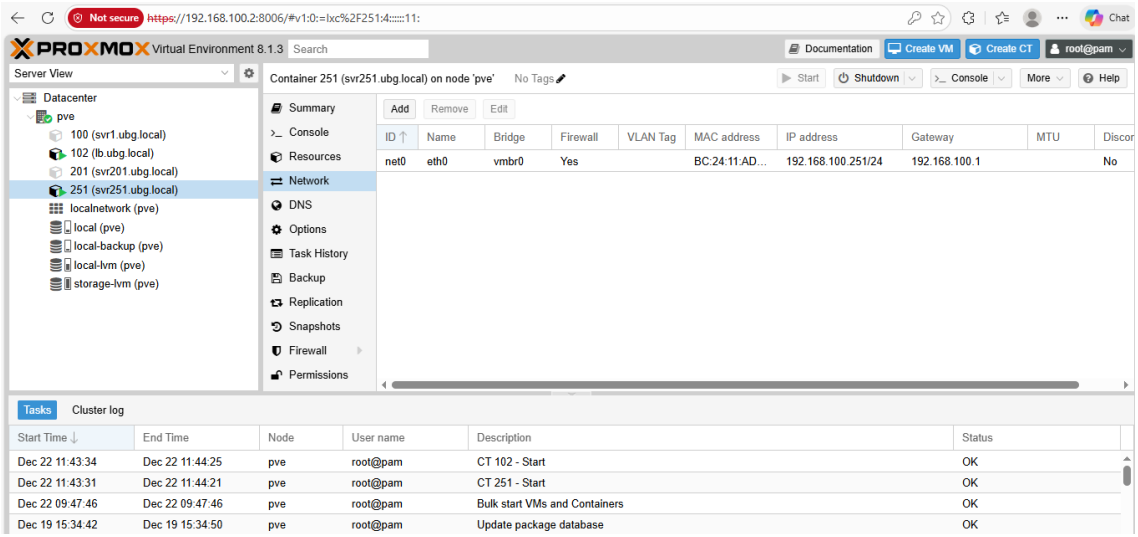

Figure 5 - Proxmox VE Web-Based Interface

3.1. Container Provisioning Time

The experimental results indicate that the provisioning time of a new container, from the scaling trigger until it is ready to serve traffic, exhibits consistent characteristics, with a per-stage time breakdown as presented in Table 2.

Table 2 - Provisioning Time Breakdown

Stage	Average Time	Minimum Time	Maximum Time
API Clone Task	82,4 Second	72,6 Second	105,3 Second
Network Configuration	0,0 Second	0,0 Second	0,1 Second
Container Start & Boot	6,8 Second	6,0 Second	10,9 Second
Content Customization	3,1 Second	2,7 Second	3,6 Second
Load Balancer Update	3,6 Second	3,0 Second	5,3 Second
Total End-to-End	96,0 Second	85,5 Second	117,7 Second

Based on Table 2, the cloning phase constitutes the primary bottleneck, accounting for 85.9% of the total provisioning time due to the full copy operation from the template storage. The substantial average clone time of 82.4 seconds is influenced by several key factors. First, the experimental environment employs nested virtualization on VMware Workstation with limited resource allocation, namely 1 CPU core, 4 GB of RAM, and a 32 GB virtual disk. This configuration introduces significant I/O overhead, as each disk operation must traverse multiple virtualization layers, including VMware, Proxmox VE, and the LXC container layer. Second, the allocation of a single CPU core prevents the inherently I/O-intensive cloning operation from exploiting parallel processing, forcing the process to execute sequentially. Third, the use of virtual disk storage (VMDK) adds latency to each I/O operation compared to bare-metal storage.

The observed time variation ranging from 85.5 to 117.7 seconds, with a standard deviation of 9.36 seconds (coefficient of variation of 9.8%), indicates good measurement consistency. This variation is primarily affected by disk I/O load on the VMware host during concurrent operations and background processes. A coefficient of variation below 10% suggests that the measurements are reliable and reproducible, despite the absolute provisioning time being higher than that of a bare-metal deployment. In contrast, the network configuration phase demonstrates excellent performance, with near-zero execution time (0.0 seconds), as it involves only metadata updates via the Proxmox API without significant disk I/O operations. This result highlights the efficiency of API-based configuration management for networking parameters.

The container boot time shows consistent performance, with an average of 6.8 seconds and a range of 6.0–10.9 seconds, reflecting a key advantage of LXC over traditional virtual machines. Because LXC containers share the host kernel, they do not require a full operating system boot sequence, which typically takes 30–60 seconds for virtual machines. The observed 6–7 second boot time includes the initialization of systemd services, activation of network interfaces, and startup of the web server inside the container. Content customization requires an average of 3.1 seconds to upload HTML files and a PHP health-check endpoint using base64 encoding through the pct exec command. This duration is relatively consistent (2.7–3.6 seconds) and reflects the overhead associated with encoding/decoding processes and filesystem write operations within the container.

*Corresponding author: naufal@universitasbumigora.ac.id

Updating the load balancer requires an average of 3.6 seconds, encompassing Nginx configuration generation, transfer to the load balancer container, syntax validation using nginx -t, and a graceful service reload. This duration is higher than the ideal expected range of 0.5–1 second due to the overhead of command execution via pct exec in a nested virtualization environment. Nevertheless, the update process is performed without service downtime, as it relies on systemctl reload, which preserves active connections. A comparison with manual provisioning highlights a substantial efficiency gain despite the constrained experimental environment. Manual provisioning, which includes container creation, package installation (Apache/Nginx and PHP), network configuration, and web server setup, typically requires 5–10 minutes depending on configuration complexity and package download speed. In contrast, the automated approach using pre-configured templates and scripted workflows reduces the average provisioning time to 96.0 seconds (approximately 1.6 minutes), yielding an efficiency improvement of 3–6× over manual methods. In a bare-metal deployment with adequate resources, the projected improvement could reach 16–32×, with an estimated provisioning time of 20–30 seconds.

3.2. Resource Efficiency

The resource efficiency analysis was conducted by comparing resource consumption between the auto-scaling scenario and the always-on scenario (maintaining four containers continuously) through monitoring over a full 1-hour period under varying workloads. The results are presented in Table 3.

Table 3 - Resource Consumption During One Hour of Testing

Scenario	CPU-Hours	RAM-Hours	Average Container	Idle Time (%)
Always-On (4 Container)	49,57	0,44 GB	4,0	54,2
Auto-Scaling	31,66	0,56 GB	1,1	28,3
Savings	36,1%	-27,3%	-	-25,9pp

Based on Table 3, the auto-scaling approach achieves 36.1% CPU savings over the one-hour testing period. The always-on scenario maintains four containers continuously, resulting in a total consumption of 49.57 CPU-hours, whereas the auto-scaling scenario, with an average of only 1.1 active containers, consumes 31.66 CPU-hours. The difference of 17.91 CPU-hours clearly demonstrates the significant effectiveness of dynamic scaling in optimizing CPU utilization.

An important note regarding the CPU-hours calculation is that the reported values of 49.57 and 31.66 reflect the accumulated CPU utilization rather than a simple multiplication of the number of containers by time. In a nested virtualization environment with a single physical CPU core shared among containers, the actual measured CPU consumption can exceed the theoretical maximum (4 containers × 1 hour = 4 CPU-hours). This occurs because Proxmox LXC reports CPU usage as a percentage of the available core, and the reported values can exceed 100% when multiple containers compete for CPU time. Additionally, nested virtualization overhead causes the VMware host to perform CPU scheduling that is reflected in the measurements. Sampling artifacts further contribute to observed anomalies, such as CPU usage spikes up to 112.2% in the always-on scenario and anomalous values of −351.7% in the auto-scaling scenario, which can occur due to measurement timing and context switching effects.

To further evaluate the performance of both approaches, CPU and memory usage were monitored throughout the testing period. Figure 5 presents a comparison of resource utilization between the Always-On and Auto-Scaling methods. Measurements were collected at 30-second intervals to provide an accurate representation of resource consumption patterns in both systems. The upper graph illustrates the average CPU utilization in percentage, while the lower graph shows the total memory usage in gigabytes (GB). Data were collected over the time range from 15:42 to 17:45, covering various workload conditions, including idle, normal load, and peak load scenarios.

*Corresponding author: naufal@universitasbumigora.ac.id

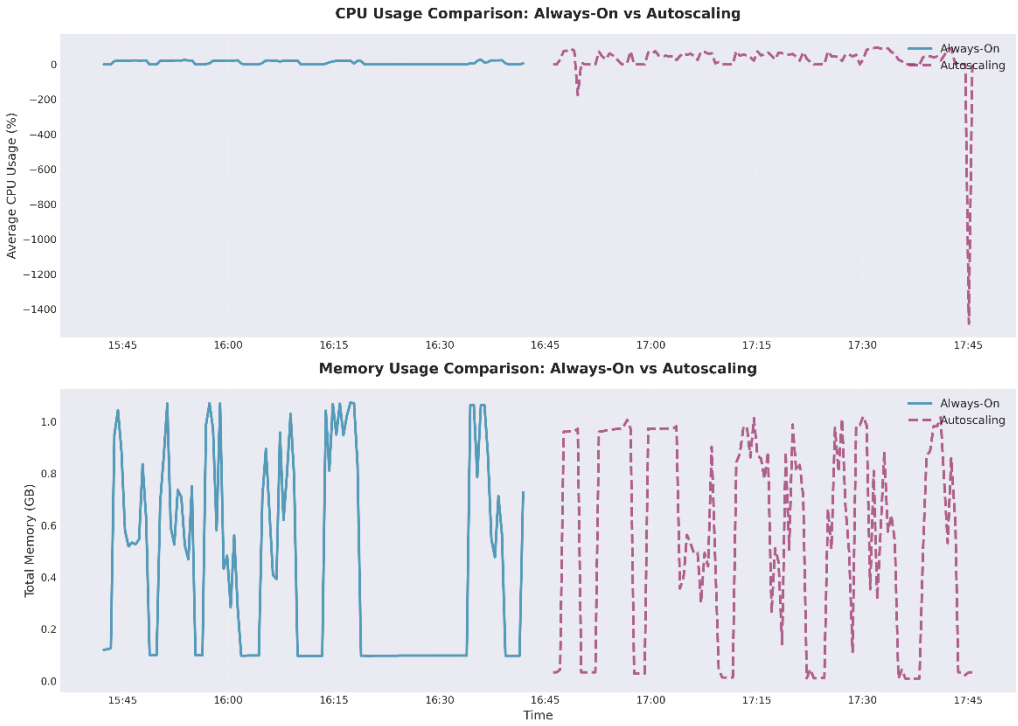

Figure 6 - Comparison of Resource Utilization

Based on Figure 6, a significant difference can be observed in the resource utilization patterns between the two approaches. In the Always-On method (solid blue line), CPU usage tends to remain stable and very low, with an average close to 0%, while memory usage exhibits relatively high fluctuations, ranging from approximately 0.1 GB to 1.0 GB. This pattern indicates that containers remain continuously running and allocate memory even when there is no active workload, which implies resource inefficiency and waste.

In contrast, the Autoscaling method (dashed purple line) demonstrates markedly different characteristics. During the initial period (before 16:45), no activity is observed because the system has not yet detected any workload. Once the workload begins to be detected (after 16:45), the system automatically provisions containers according to demand, as reflected by the increase in CPU and memory usage. Notably, an anomaly in the form of extreme negative CPU values appears toward the end of the measurement period, which is most likely caused by transition processes during scale-down operations or container termination. Despite this anomaly, the overall memory utilization pattern under Autoscaling shows better efficiency, with dynamic allocation that closely follows the actual workload.

The fundamental difference is particularly evident in memory utilization. The Always-On approach maintains consistently high memory allocation even in the absence of workload, whereas Autoscaling allocates memory only when it is truly required, exhibiting a rising and falling pattern that follows the workload cycle. This behavior indicates that the Autoscaling approach has greater potential for resource savings, especially in scenarios where workloads are non-uniform or sporadic.

To provide a more comprehensive view of the operational characteristics of both approaches, further analysis was conducted on the number of active containers and key performance metrics throughout the testing period. Figure 6 presents two important aspects: the upper graph illustrates the dynamics of the number of active containers over time, while the lower graph compares key performance metrics, including average CPU utilization, maximum CPU utilization, average memory usage, and the average number of running containers. These metrics were selected because they represent resource efficiency and the system's ability to respond to workload variations. Measurements were taken over the same time interval as the previous experiments to ensure data consistency and the validity of the comparison.

*Corresponding author: naufal@universitasbumigora.ac.id

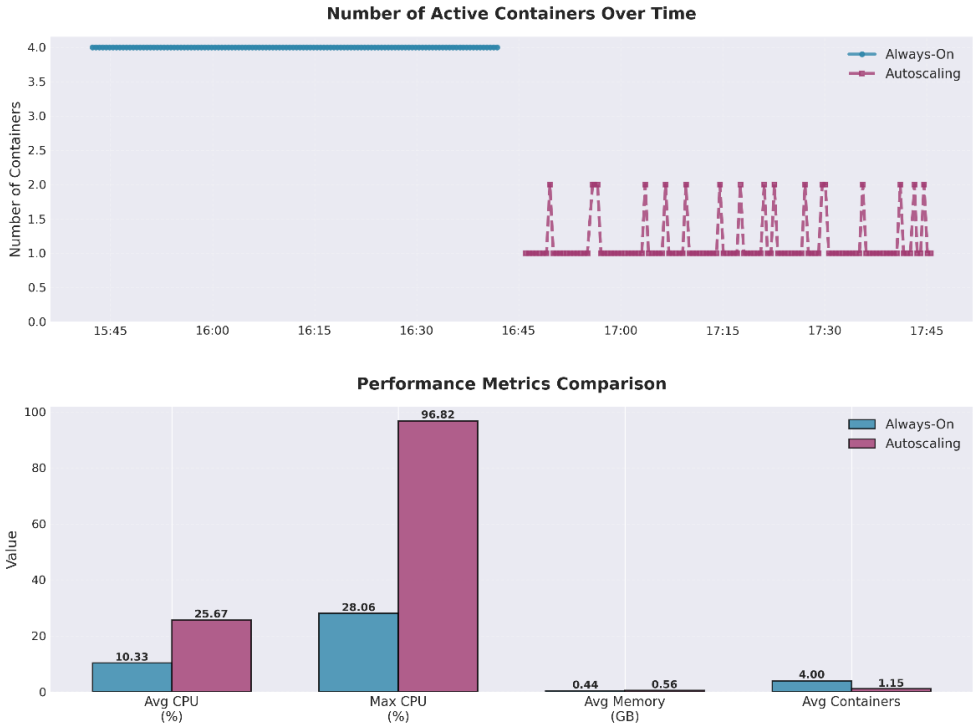

Figure 7 - Container Metrics

Based on Figure 7, a fundamental difference in resource allocation strategies between the two approaches can be observed. In the active container count graph (upper section), the Always-On method consistently maintains four containers throughout the measurement period (solid blue line), reflecting a static approach in which all containers run continuously regardless of the actual workload. In contrast, the Autoscaling method (dashed purple line) exhibits a highly dynamic pattern, starting with a single container under idle conditions and increasing to two containers when workload is detected (after 16:45), with fluctuations that follow workload intensity.

The performance metrics comparison graph (lower section) provides several important insights. Although the Always-On method shows a lower average CPU utilization (10.33% compared to 25.67%), this actually indicates resource underutilization, as containers remain idle without performing productive work. The maximum CPU utilization under Autoscaling reaches 96.82%, significantly higher than the 28.06% observed in the Always-On approach, demonstrating that Autoscaling is able to exploit available computational capacity more effectively when needed.

The most significant differences are observed in memory efficiency and container count. The average memory usage in the Always-On approach is recorded at 0.44 GB, slightly lower than the 0.56 GB observed under Autoscaling. However, when considering the average number of containers, Always-On operates with 4.00 containers, whereas Autoscaling runs only 1.15 containers on average. This results in substantially better per-container efficiency under Autoscaling: memory usage per container in the Always-On approach is 0.11 GB/container (0.44 ÷ 4), while Autoscaling reaches 0.49 GB/container (0.56 ÷ 1.15). Although memory usage per container is higher under Autoscaling, the overall system-level resource consumption is lower because only the containers that are actually required are running.

These findings confirm that the Autoscaling approach is not only responsive to workload variations but also significantly more efficient in terms of infrastructure resource utilization. The reduction in the average number of containers from four to 1.15 (a decrease of 71.25%) represents a substantial potential for operational cost savings, particularly in cloud computing environments where costs are directly tied to actual resource consumption.

To gain a deeper understanding of the performance characteristics of both approaches, a comprehensive statistical analysis was conducted on the collected resource utilization data. Figure 7 presents this statistical analysis, which consists of four main components: CPU utilization statistics (upper left), memory utilization statistics (upper right), and resource distribution in the form of pie charts for the Always-On method (lower left) and the Autoscaling method (lower right). The statistical analysis includes mean, maximum, minimum, and standard deviation values to provide a holistic view of performance variability and consistency. The pie charts illustrate the relative proportions of average CPU usage, average memory usage, and the number of containers with respect to total system resources, normalized to facilitate a clear comparison of resource distribution between the two approaches.

*Corresponding author: naufal@universitasbumigora.ac.id

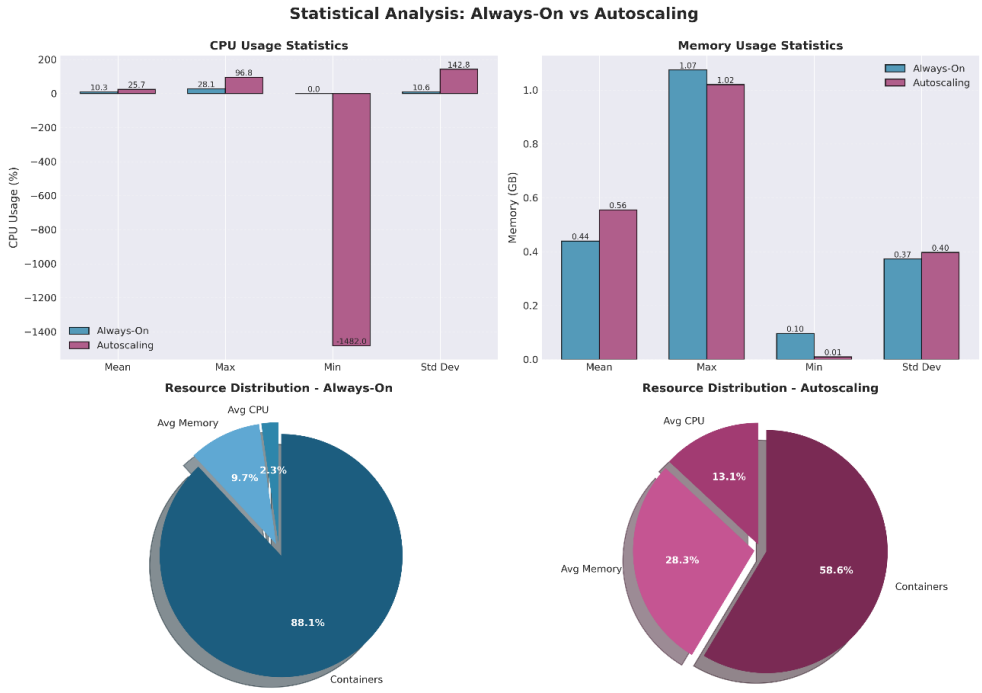

Figure 8 - Statistical Summary

Based on Figure 8, the statistical analysis reveals fundamental differences between the Always-On and Autoscaling methods in terms of operational characteristics and resource efficiency. The Always-On approach exhibits low and stable CPU utilization (mean of 10.3% with a standard deviation of 10.6%), indicating that many containers operate in an idle state. In contrast, Autoscaling shows a higher average CPU utilization (25.7%) with substantial variability (standard deviation of 142.8%), reflecting the system's responsiveness to workload fluctuations, although anomalies occur during scale-down processes.

In terms of memory usage, Autoscaling is slightly higher on average (0.56 GB compared to 0.44 GB for Always-On), yet it is capable of reducing memory consumption to much lower minimum levels during idle periods, demonstrating better overall efficiency. The resource distribution further reinforces these findings: the Always-On approach is dominated by container allocation (88.1%), much of which is non-productive, whereas Autoscaling presents a more balanced distribution among containers, CPU, and memory.

Overall, these results confirm that Autoscaling is more adaptive and efficient in resource allocation, while the Always-On approach tends to be rigid and prone to over-provisioning.

4. CONCLUSION

This study successfully demonstrates that an LXC container–based auto-scaling and load balancing system on Proxmox VE, implemented using Python and REST APIs, can significantly improve resource utilization efficiency compared to the Always-On approach. The results show an increase in average CPU utilization from 10.3% to 25.7% and a reduction in the number of active containers by up to 71.25%. Although memory usage is slightly higher, the Autoscaling system exhibits better adaptability to workload fluctuations, effectively reducing resource consumption during idle periods and achieving optimal utilization under peak loads without compromising service availability. This is enabled by the multi-layer fallback mechanism in Nginx, which ensures zero downtime during scaling operations.

These findings indicate that the Autoscaling approach not only minimizes over-provisioning and resource waste but also has the potential to reduce operational costs, improve energy efficiency, and lower carbon emissions. Furthermore, the study confirms that auto-scaling can be implemented effectively and cost-efficiently on Proxmox VE without the need for complex orchestration platforms such as Kubernetes, making it particularly suitable for educational institutions and small-to-medium enterprises (SMEs). Future work will focus on establishing a more concrete technical roadmap to enhance system intelligence and scalability. Specifically, we aim to integrate Machine Learning-based predictive scaling (e.g., using LSTM or ARIMA models) to anticipate workload spikes before they occur. Additionally, the research will

*Corresponding author: naufal@universitasbumigora.ac.id

be extended to support Multi-node Proxmox Clusters, enabling high availability and fault tolerance beyond the current single-node implementation.

REFERENCES

[1] M. Catillo, U. Villano, and M. Rak, "A survey on auto-scaling: how to exploit cloud elasticity," *International Journal of Grid and Utility Computing*, vol. 14, no. 1, p. 37, 2023, doi: 10.1504/IJGUC.2023.129702.

[2] A. d'Orgeval, S. Sheehan, Q. Avenas, E. Assoumou, and V. Sessa, "Generative AI impact assessment through a life cycle analysis of multiple data center typologies," *Appl. Energy*, vol. 406, p. 127288, Mar. 2026, doi: 10.1016/j.apenergy.2025.127288.

[3] S. Bharany *et al.*, "A Systematic Survey on Energy-Efficient Techniques in Sustainable Cloud Computing," *Sustainability*, vol. 14, no. 10, p. 6256, May 2022, doi: 10.3390/su14106256.

[4] M. Y. Jusoh, H. Haron, and J. Kaur, "Virtualization Technology to Support Green Computing Among IT Personnel in the Public Sector," 2021, pp. 676–688. doi: 10.1007/978-3-030-90235-3_58.

[5] M. Catillo, L. Ocone, U. Villano, and M. Rak, "Black-box load testing to support auto-scaling web applications in the cloud," *International Journal of Grid and Utility Computing*, vol. 12, no. 2, p. 139, 2021, doi: 10.1504/IJGUC.2021.114823.

[6] D. Zou *et al.*, "OptScaler: A Collaborative Framework for Robust Autoscaling in the Cloud," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 4090–4103, Aug. 2024, doi: 10.14778/3685800.3685829.

[7] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano, "KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes," 2021, pp. 821–829. doi: 10.1007/978-3-030-91431-8_59.

[8] Z. Wang *et al.*, "DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proceedings of the 13th Symposium on Cloud Computing*, New York, NY, USA: ACM, Nov. 2022, pp. 16–30. doi: 10.1145/3542929.3563469.

[9] R. Evenn, I. Hasanov, and P. Sainio, "Securing Company Infrastructure with Modern Automation Tools," University of Turku, 2025. Accessed: Nov. 18, 2025. [Online]. Available: https://www.utupub.fi/bitstream/handle/10024/194200/Master_Thesis_Resnais_Evenn.pdf?sequence=1

[10] V. P. Oleksiuk and O. R. Oleksiuk, "The practice of developing the academic cloud using the Proxmox VE platform," *Educational Technology Quarterly*, vol. 2021, no. 4, pp. 605–616, Dec. 2021, doi: 10.55056/etq.36.

[11] J. Zhou *et al.*, "Comparative analysis of metaheuristic load balancing algorithms for efficient load balancing in cloud computing," *Journal of Cloud Computing*, vol. 12, no. 1, p. 85, Jun. 2023, doi: 10.1186/s13677-023-00453-3.

[12] M. H. Hersyah, "A Proposed Model of Digital Forensic on Cloud Computing Security Infrastructure," *International Journal of Innovation in Enterprise System*, vol. 2, no. 02, pp. 18–23, Jul. 2018, doi: 10.25124/ijies.v2i02.21.

[13] D. A. Shafiq, N. Z. Jhanjhi, and A. Abdullah, "Load balancing techniques in cloud computing environment: A review," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3910–3933, Jul. 2022, doi: 10.1016/j.jksuci.2021.02.007.

[14] A. M. Mazin, R. A. Rahman, M. Kassim, and A. R. Mahmud, "Performance Analysis on Network Automation Interaction with Network Devices Using Python," in *2021 IEEE 11th IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, IEEE, Apr. 2021, pp. 360–366. doi: 10.1109/ISCAIE51753.2021.9431823.

[15] M. Kondoj, H. Langi, Y. Putung, and V. Lengkong, "Performance Analysis of Cloud Computing Based E-Commerce Server Using PROXMOX Virtual Environment," in *Proceedings of the 5th International Conference on Applied Science and Technology on Engineering Science*, SCITEPRESS - Science and Technology Publications, 2022, pp. 741–745. doi: 10.5220/0011876000003575.

[16] M. Šimon, L. Huraj, and N. Búčik, "A Comparative Analysis of High Availability for Linux Container Infrastructures," *Future Internet*, vol. 15, no. 8, p. 253, Jul. 2023, doi: 10.3390/fi15080253.

[17] V. Mohammadian, N. J. Navimipour, M. Hosseinzadeh, and A. Darwesh, "Fault-Tolerant Load Balancing in Cloud Computing: A Systematic Literature Review," *IEEE Access*, vol. 10, pp. 12714–12731, 2022, doi: 10.1109/ACCESS.2021.3139730.

[18] C. Ma and Y. Chi, "Evaluation Test and Improvement of Load Balancing Algorithms of Nginx," *IEEE Access*, vol. 10, pp. 14311–14324, 2022, doi: 10.1109/ACCESS.2022.3146422.

[19] A. M. Khan, W. Salah Alaloul, M. A. Musarat, and M. Hassaan Farooq Khan, "Python: An Automation Tool for Unlocking Innovation and Efficiency in the AEC Sector," in *2023 4th International Conference on Data Analytics for Business and Industry (ICDABI)*, IEEE, Oct. 2023, pp. 89–94. doi: 10.1109/ICDABI60145.2023.10629478.

[20] H. Bin Mehare, J. P. Anilkumar, and N. A. Usmani, "The Python Programming Language," in *A Guide to Applied Machine Learning for Biologists*, Cham: Springer International Publishing, 2023, pp. 27–60. doi: 10.1007/978-3-031-22206-1_2.

[21] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, Jul. 2019, doi: 10.1145/3148149.

*Corresponding author: naufal@universitasbumigora.ac.id