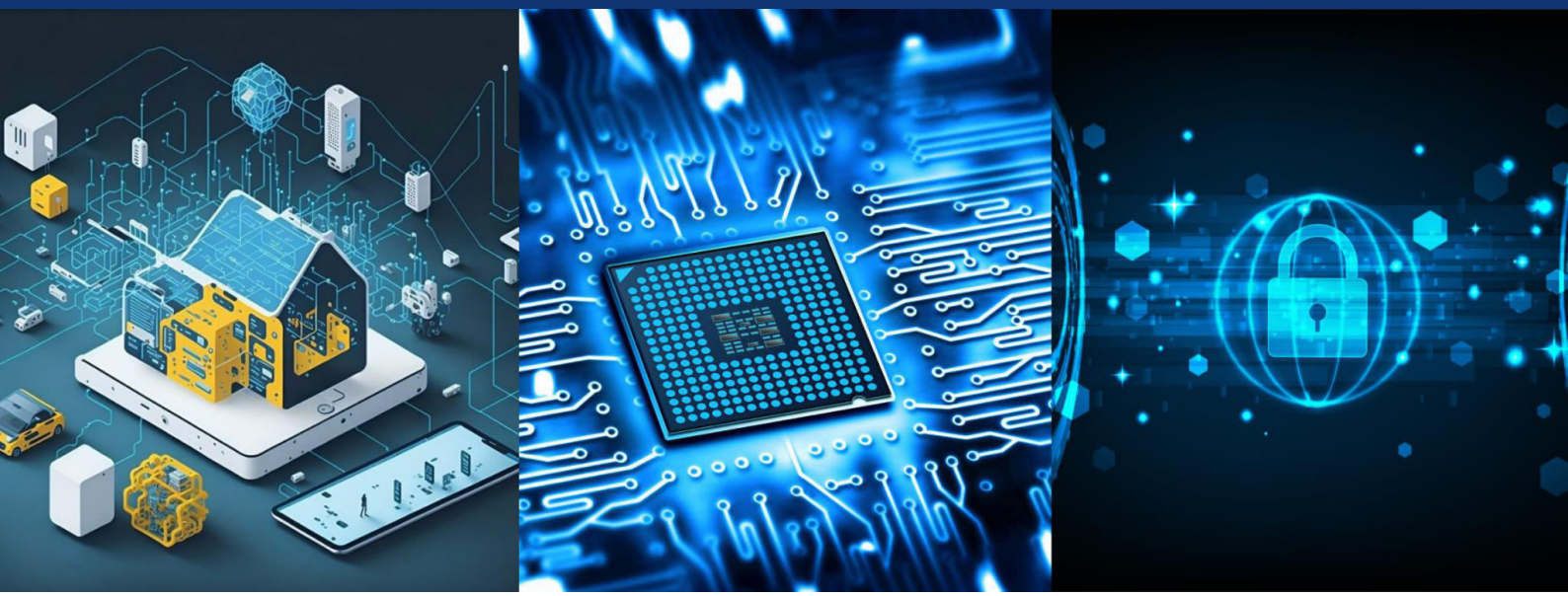
IJIRCCE

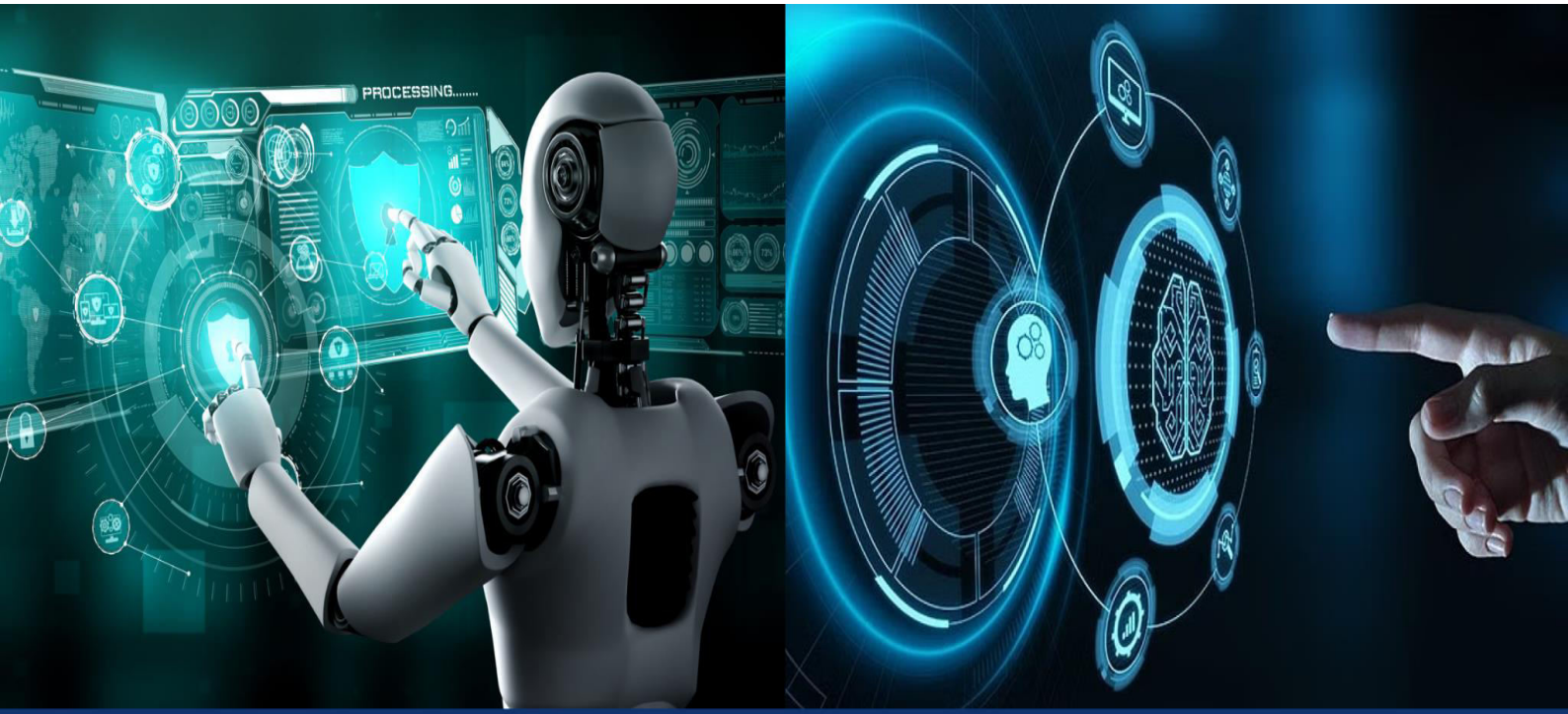
ISSN(O): 2320-9801

ISSN(P): 2320-9798



International Journal of Innovative Research in Computer and Communication Engineering

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 9.927

Volume 14, Issue 10, October 2026

© 2026 IJIRCCE | Volume 14, Issue 10, October 2026 DOI: 10.15680/IJIRCCE.2026.1410042

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

AI Cloud-Based Cost Optimization System: A Comprehensive Framework for Intelligent Resource Management

Abdul Rahman, Darshan S, Dipak Singh, Hithashri T, and Lokesh K

Department of Information Science & Engineering, Raja Rajeswari College of Engineering, Bengaluru, India

ABSTRACT: Keeping monthly cloud bills from spiraling out of control is an ongoing headache for engineering teams running workloads on public infrastructure. Launching a virtual server takes less than a minute in a web browser, yet remembering to turn off temporary test boxes, staging environments, or oversized virtual machines once a task wraps up is surprisingly difficult in day-to-day work. In this paper, we present a practical, automated Cloud Cost Optimization and Monitoring System built to track down, evaluate, and shut down idle compute instances before they quietly drain budgets. Instead of counting on rigid capacity plans or waiting around for end-of-month billing alerts, our tool pulls live telemetry straight from cloud-native APIs. Focusing on Amazon Web Services (AWS) Elastic Compute Cloud (EC2), we wrote a Python monitoring service around the Boto3 Software Development Kit (SDK) that checks CPU load, network traffic, and disk activity through AWS CloudWatch. Whenever a virtual machine stays under a chosen activity limit across a rolling time window, our script flags it as an abandoned or "zombie" server, stops it automatically, and sends a message to the operations crew. Because the tool watches real hardware usage rather than fixed clock schedules, developers no longer have to babysit the AWS web console by hand. We walk through the full architecture, cost equations, decision rules, and live experimental runs across ten EC2 instances to demonstrate how a compact Python daemon can enforce everyday FinOps discipline and trim wasted cloud spend.

KEYWORDS: Cloud Computing, Cost Optimization, AWS EC2, FinOps, CloudWatch, Boto3, Anomaly Detection, Resource Allocation, Infrastructure as a Service (IaaS), Python Automation

I. INTRODUCTION

Over the last ten years, most software companies have stopped buying physical server racks and switched to renting virtual hardware from major platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Moving toward Infrastructure as a Service (IaaS) makes it remarkably easy to spin up and scale software applications on short notice [14]. Still, that flexibility brings a well-known billing trap. Because cloud vendors charge by the hour or second for as long as a virtual machine stays switched on, an Amazon EC2 node costs the exact same amount of money whether it is serving thousands of active users or sitting completely idle. Engineering teams routinely set up auto-scaling rules to add extra servers when user traffic spikes, yet they rarely put the same effort into cleaning up leftover instances once the rush dies down.

In our experience, cloud waste rarely comes from a deep software bug; it almost always stems from simple human habits during busy workweeks. A developer boots up two or three EC2 nodes on a Tuesday afternoon to test out a database schema change, debug a continuous integration pipeline, or crunch a batch of server logs. Once the task finishes, the team moves on to the next sprint ticket and completely forgets those temporary machines are still running in the background. Weeks go by, and those forgotten "zombie" instances keep quietly adding charges to the company account without doing any useful work. Across the broader tech industry, idle and over-provisioned virtual servers eat up billions of dollars in wasted cloud budgets every single year [9], [15].

Stopping this kind of leak takes an automated tool that watches server behavior in real time and steps in when a machine is clearly abandoned. Built-in budget alerts from cloud providers mostly look backward—they fire off a warning email only after a spending cap has already been broken. By the time a systems administrator opens that alert, the money is already spent. To tackle this, we developed an automated Cloud Cost Optimization System in Python that catches idle servers early. Built on top of the Boto3 SDK [5] and AWS CloudWatch, our script periodically checks the health and processor load of active EC2 nodes. If an instance sits practically unused—say, averaging below 5% CPU

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

utilization across an entire hour—the script flags the machine, issues a stop command automatically, and emails the engineering team so nobody is left guessing what happened.

As monthly infrastructure bills have climbed, tech organizations have begun weaving financial accountability directly into systems engineering, a cultural shift known as FinOps [6]. Rather than handing a massive invoice to the accounting office at the close of each quarter, FinOps gives developers the runtime visibility they need to weigh speed, reliability, and cost on their own. The standard FinOps lifecycle runs through three repeating phases: Inform, Optimize, and Operate. Our work fits directly into the Optimize and Operate phases, turning raw CloudWatch telemetry into hands-off server shutdowns. What day-to-day engineering teams actually need is a lightweight, trustworthy script that connects safely to their cloud account, pulls basic health numbers (like CPU load and network bytes) over standard APIs, and turns off idle machines using simple, transparent rules without requiring heavy monitoring agents inside every virtual machine.

Paper is organized as follows. Section II describes related work across resource allocation, FinOps lifecycle management, machine learning scheduling, and serverless automation. Section III presents the methodology, mathematical problem formulation, multi-layered system architecture, algorithmic workflow, and AWS implementation. Section IV presents experimental results showing evaluation metrics across ten EC2 instances and financial impact analysis. Finally, Section V presents the conclusion.

II. RELATED WORK

Over the past several years, both academic researchers and working cloud engineers have tackled cloud cost waste from different angles [8], [13]. Published work ranges from straightforward shell scripts to heavy deep-learning forecasters, all aiming to lower monthly bills without hurting software reliability. Early attempts to rein in cloud spending relied almost entirely on native auto-scaling groups built into AWS, Azure, and GCP. Suthari et al. [1] evaluated resource overcommitment, virtual machine consolidation, and predictive auto-scaling across standard IaaS clusters. Their experiments showed that while basic reactive auto-scaling handles gradual traffic changes decently well, it stumbles during sudden request spikes and does nothing at all for standalone test instances launched outside an auto-scaling group. Moving from reactive rules to predictive models trimmed their overall compute bill by 24.6%, though running those forecasting algorithms consumed enough extra processor time to eat into the net savings.

Rakavi et al. [2] studied how FinOps workflows play out in everyday operations on Google Cloud Platform (GCP). By joining detailed billing export tables with server utilization logs, their software flagged oversized virtual machines and simulated rightsizing moves, projecting more than $320,000 in annual savings for a mid-sized enterprise deployment. The main drawback with their setup—which we set out to avoid—is that it leans heavily on GCP billing exports, which typically lag behind real-time server usage by several hours and do not carry over to managing raw AWS EC2 nodes.

A growing wave of recent papers brings machine learning directly into cloud scheduling. Vij and Mishra [3] trained deep neural networks to adjust resource limits on the fly across cloud and edge nodes, feeding historical workload traces into the model to guess upcoming demand. While deep neural networks are great at picking up subtle traffic rhythms, they behave like a “black box.” In real production environments, systems administrators are understandably hesitant to let an opaque neural network power down live servers when it cannot explain why it made the call. Whenever an automation script has permission to shut off infrastructure, operations teams strongly prefer transparent, deterministic rules they can audit in seconds. Meanwhile, sequence models like Bidirectional Long Short-Term Memory (Bi-LSTM) networks have worked well in related forecasting tasks, such as predicting city parking occupancy [4]. Those projects prove that watching a series of readings across a time window helps separate a truly abandoned resource from one that is simply experiencing a brief lull—a principle we apply in a much lighter way by averaging CloudWatch datapoints across rolling windows.

Other researchers have put together nimble cost monitors using serverless functions and Python automation [7], [10], [11], [12]. Mittal and Deepthi [7] along with Mehta and Kulkarni [11] built AWS Lambda functions and Python state-checking scripts to hunt down unused developer sandboxes. Running checks inside Lambda means teams do not have to patch a dedicated monitoring box, but Lambda functions run into hard execution time limits and memory ceilings when polling hundreds of virtual machines across several global regions. Rather than fighting serverless timeouts, we chose to host our Python engine on a small, hardened Ubuntu 22.04 LTS management instance, letting us run

© 2026 IJIRCCE | Volume 14, Issue 10, October 2026 DOI: 10.15680/IJIRCCE.2026.1410042

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

continuous polling loops and keep local audit logs without arbitrary clock limits. Table I summarizes how our design compares against recent work.

TABLE I: COMPARISON OF EXISTING CLOUD COST OPTIMIZATION METHODOLOGIES

Author / Reference	Primary Methodology	Target Environment	Key Limitations
Suthari et al. [1]	Predictive Auto-scaling	Generic IaaS	Heavy compute overhead needed to train and run ML forecasting models.
Rakavi et al. [2]	FinOps Lifecycle Analytics	GCP	Tied to GCP billing tables; relies on delayed billing exports rather than live APIs.
Vij & Mishra [3]	Deep Neural Networks	Cloud / Edge	"Black-box" neural choices make server shutdowns risky for ops teams.
Mittal & Deepthi [7]; Mehta & Kulkarni [11]	Serverless Monitoring	AWS Lambda	Execution timeout ceilings when scanning large multi-region server fleets.
Proposed System	Deterministic API Thresholding via Boto3	AWS EC2	Transparent rules, low API latency, and immediate shutdown enforcement.

III. METHODOLOGY

In most software shops, watching cloud spend still comes down to poking around the AWS web console by hand or waiting for monthly AWS Budgets warning emails. Because engineers usually pick an instance type big enough to survive their heaviest traffic hour and leave it running around the clock, virtual servers sized for peak afternoon traffic stay powered on 24 hours a day, racking up hourly charges every night and weekend when nobody is touching them. Expecting busy developers to manually cross-check CPU graphs, network logs, and billing spreadsheets simply falls apart as a team grows.

In plain terms, our objective is to shrink the overall EC2 compute bill across a billing period T without bothering servers that are doing actual work. We can break the total compute cost C_total across N provisioned instances over billing cycle T into two buckets: the hours those machines spent running active jobs, and the hours they sat idle:

$$C_total = \sum_\{i=1 \text{ to } N\} (P_i \times t_active,i) + \sum_\{i=1 \text{ to } N\} (P_i \times t_idle,i) \quad (1)$$

where N represents the total count of running EC2 instances in the account, P_i is the hourly on-demand price for instance i according to its hardware family (such as t2.micro or m5.large), t_active,i is the number of hours instance i spent doing useful work, and t_idle,i is the number of hours instance i remained switched on without doing meaningful work (meaning its processor load stayed below our cutoff U_th). Because we never want to cut into t_active,i, saving money comes down to driving t_idle,i as close to zero as possible by shifting an instance from running to stopped the moment we are confident it has been abandoned. At any inspection timestamp t, the control rule for instance i over a lookback window Δt is written as:

$$State(i, t) = STOP, \text{ if } U_i(t - \Delta t, t) < U_th \text{ ; } KEEP_RUNNING, \text{ otherwise} \quad (2)$$

Here, $U_i(t - \Delta t, t)$ denotes the mean CPU utilization of instance i measured across the sliding window Δt, and U_th is our inactivity threshold (set to 5% in our experiments).

We built the Cloud Cost Optimizer as a standalone Python service powered by Boto3, the official AWS SDK for Python [5]. Because the script runs from a single central management box and talks to AWS endpoints over the network, teams never have to install extra agents or daemons inside their application servers. Setting up and running the

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

workflow breaks down into six clean steps: 1) Environment Initialization on a lightweight central management node running Ubuntu 22.04 LTS; 2) Dependency Management by installing Python 3, pip3, and creating an isolated virtual environment (cloud_env) for the Boto3 package; 3) Secure Communication talking to AWS APIs over TLS using a tightly scoped IAM role named CostOptimizerExecutionRole rather than leaving plaintext access keys sitting on disk [12]; 4) Telemetry Extraction querying CloudWatch APIs to fetch CPUUtilization, NetworkIn, and NetworkOut readings across a rolling time window; 5) Decision Logic Execution averaging those data points and checking whether the server stayed underneath the 5% CPU line; and 6) Action & Reporting writing a clear status line to the terminal, invoking Boto3's stop_instances() call on zombie machines, and sending out an email alert via Amazon Simple Notification Service (SNS).

Walking from the bottom up, the software is organized into five layers: the Input / Target Layer (live fleet of AWS EC2 virtual machines), the Monitoring Layer (Amazon CloudWatch gathering hypervisor-level telemetry every 1 to 5 minutes), the Data Collection Layer (core Boto3 polling module calling get_metric_statistics), the Decision Engine / Analysis Layer (calculating the rolling mean across the lookback window against our 5% CPU rule), and the Action & Notification Layer (logging active servers or stopping idle nodes and triggering SNS alerts). Pairing the script with a lightweight SQLite or PostgreSQL database gives teams an audit trail across three tables: EC2_INSTANCE (storing instance_id, instance_type, launch_time, and region), CPU_METRICS (historical time-series CPU figures), and ACTION_LOG (an append-only audit table recording every shutdown and the exact CPU average that triggered it).

To guard against false positives—killing a server that is doing real work just because traffic happened to dip for a couple of minutes—our algorithm never acts on a single data point. Instead, a virtual machine must stay below the cutoff across an entire 60-minute evaluation window (averaging twelve 5-minute buckets) before the script steps in, as detailed in Algorithm 1.

Algorithm 1: Cloud Cost Optimization Engine

Input: instance_list (List of active EC2 Instance IDs), eval_period (3600 seconds), threshold (5.0% CPU)
Output: action_report
1: Initialize boto3.client("cloudwatch") and boto3.client("ec2")
2: action_report <- Empty List
3: for each instance_id in instance_list do
4: startTime <- CurrentTime - eval_period; endTime <- CurrentTime
5: metrics <- cloudwatch.get_metric_statistics(Namespace="AWS/EC2", MetricName="CPUUtilization", Dimensions=[("Name":"InstanceId", "Value":instance_id)], StartTime=startTime, EndTime=endTime, Period=300, Statistics=["Average"])
6: if metrics.Datapoints is EMPTY then append (instance_id, "NO_DATA") and continue
7: Compute avg_cpu across all datapoints in metrics.Datapoints
8: if avg_cpu < threshold then
9: ec2.stop_instances(InstanceIds=[instance_id])
10: sns.publish(TopicArn, "Instance Stopped: Inactivity")
11: action_report.append(instance_id, "STOPPED", avg_cpu)
12: else
13: action_report.append(instance_id, "ACTIVE", avg_cpu)
14: end if
15: end for
16: return action_report

We provisioned our central management box in the AWS ap-southeast-2 (Sydney) region using an Ubuntu Server 22.04 LTS AMI on a t2.micro instance covered under the AWS Free Tier. We set its AWS Security Group to drop all inbound connections except SSH traffic on port 22 using an RSA-2048 .pem private key, and installed boto3 inside an isolated Python virtual environment so its underlying libraries (botocore, jmespath, s3transfer, and urllib3) remain separated from system Python packages.

IV. EXPERIMENTAL RESULTS

To test how well the script separates active workloads from abandoned machines, we built a live testbed on AWS mixing steady, bursty, and idle servers. We launched ten EC2 instances and divided them into three behavioral groups

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

designed to mirror the mix of machines typically seen in a company cloud account: Profile Alpha (four active web servers running Apache and Nginx under continuous synthetic HTTP load between 45% and 80% CPU), Profile Beta (three batch processors running a cron-triggered Python worker that pushed CPU to 90% for 10 minutes and sat quiet for the remaining 50 minutes of the hour), and Profile Gamma (three zombie/idle instances left switched on with no user jobs running at all, hovering between 0.5% and 1.2% CPU strictly from background kernel housekeeping).

We kicked off our monitoring script from the management node using a 60-minute lookback window and a 5.0% CPU threshold (CPU < 5.0%). Reusing Boto3's HTTP connection pool, the tool polled the AWS/EC2 namespace across all ten servers and computed their hourly averages. Table II shows a representative slice of six instances from our test run.

TABLE II: EXPERIMENTAL METRIC EVALUATION RESULTS (60-MIN WINDOW)

Instance ID	Profile	Avg CPU	Net In (KB)	Action / Status
i-0a655b3d...	Alpha (Web)	68.4%	15,402	ACTIVE – Ignored
i-0b298c11...	Alpha (Web)	52.1%	12,050	ACTIVE – Ignored
i-0cc44d99...	Beta (Batch)	14.8%	8,920	ACTIVE – Ignored
i-0dd11e88...	Gamma (Idle)	0.8%	140	IDLE – STOPPED
i-0ee22f77...	Gamma (Idle)	1.1%	185	IDLE – STOPPED
i-0ff33g66...	Gamma (Idle)	0.6%	110	IDLE – STOPPED

As shown in Table II, the Profile Alpha web servers posted heavy processor averages (68.4% and 52.1%), so the script logged them as ACTIVE and left them untouched. The real test of our windowing logic came from Profile Beta. Even though those batch workers sat completely silent for 50 minutes out of the hour, their 10-minute crunch at 90% utilization pulled their 60-minute rolling mean up to 14.8%. Because 14.8% sits comfortably above our 5.0% cutoff, the script left those batch nodes alone. Had we judged servers on a short 5-minute check during their quiet stretch, the automation would have killed an active batch pipeline mid-cycle. Meanwhile, all three Profile Gamma "zombie" instances stayed below 1.2% CPU across the entire hour. The script flagged all three, matched the avg_cpu < threshold rule, and fired off the EC2 stop API call, logging "Status: Idle. Stopping instance. Instance stopped."

To put these numbers into real financial terms, we can project the hourly savings across a standard 730-hour billing month. Suppose those three forgotten Gamma instances were general-purpose m5.large machines, which run about $0.096 per hour in the AWS Sydney region. Leaving just those three virtual servers switched on for a full month burns $210.24 ($70.08 per box) for zero output. By catching and stopping them after a single hour of inactivity, our script brings that wasted compute charge down to roughly $0.29 total, leaving only the small monthly fee for their attached EBS boot drives while the instances remain powered off. Inside a mid-sized software company where engineers spin up dozens of temporary staging and test boxes every week, automatically shutting down forgotten instances easily saves tens of thousands of dollars a year—all driven by a compact Python script running on a free-tier management node.

V. CONCLUSION

In this paper, we designed, built, and tested a practical Cloud Cost Optimization System using Python, Boto3, and AWS CloudWatch. Rather than waiting around for painful end-of-month billing surprises or counting on busy developers to clean up their own test sandboxes, our tool watches real hardware utilization and powers down idle EC2 instances on its own. Hosting the script inside an isolated Python virtual environment on a hardened Ubuntu box—and authenticating through IAM roles rather than static access keys—keeps the whole setup lightweight, reliable, and safe. Our live runs across web frontends, periodic batch workers, and idle zombie nodes confirmed that checking 60-minute rolling CPU averages reliably catches forgotten servers without accidentally killing intermittent batch jobs, weaving steady FinOps discipline into everyday cloud operations. In future work, the platform can be extended with predictive time-series forecasters (such as Bi-LSTM networks), a React or Streamlit graphical dashboard, multi-cloud support for Azure and GCP, and automated instance rightsizing during maintenance windows.

www.ijircce.com | e-ISSN: 2320-9801, p-ISSN: 2320-9798| Impact Factor: 9.927| ESTD Year: 2013|

International Journal of Innovative Research in Computer and Communication Engineering (IJIRCCE)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

REFERENCES

[1] Y. Suthari, N. Krishnan, Z. Thakker, V. Raju, S. Govindarajan, and S. K. Rao Katta, "Cost Optimization Techniques for Efficient Resource Allocation in Cloud Computing Environments," in 2025 3rd International Conference on Intelligent Cyber Physical Systems and Internet of Things (ICOICI), 2025, pp. 793–797.
[2] R. Rakavi, K. D. Dharshini, A. S. Jovi, and R. Thenmozhi, "FinOps-based Cloud Cost Optimization Using Billing Analytics and Anomaly Detection in GCP," in 2026 International Conference on Electronic Systems and Intelligent Computing (ICESIC), 2026, pp. 551–556.
[3] P. Vij and A. Mishra, "AI-Driven Resource Optimization in Cloud and Edge Environments for Sustainable Ubiquitous Computing," in 2025 5th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS), 2025, pp. 458–462.
[4] B. M. R. L. Ruben, S. J. J. E. D, A. T. T, S. V, and C. M, "AI-Driven Smart Parking System using IoT and Cloud-based Predictive Approach for Urban Mobility Optimization," in 2025 5th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS), 2025, pp. 1154–1159.
[5] Amazon Web Services, "Boto3 Documentation – The AWS SDK for Python," AWS API Reference, 2025. [Online]. Available: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[6] FinOps Foundation, "FinOps Framework: Cloud Financial Management Best Practices," 2024. [Online]. Available: https://www.finops.org/framework/
[7] N. Mittal and V. D. Deepthi, "Automated Serverless Cost Monitoring Systems for Enterprise Cloud Environments," IEEE Transactions on Services Computing, vol. 18, no. 4, pp. 312–325, 2025.
[8] S. R. Jena, U. Srilakshmi, and S. M. Rafi, "Intelligent Green Energy Solutions: Optimizing Renewable Energy for Edge Cloud Computing," in 2025 5th International Conference on Pervasive Computing and Social Networking (ICPCSN), 2025, pp. 112–118.
[9] A. Patel and A. Sheth, "AI-Powered Metric Analysis: Safeguarding Cloud Budgets," in IEEE Cloud Summit, 2025, pp. 45–51.
[10] X. Yang and Y. Chen, "Research and Application of Boto3-Based Monitoring Technology for Cloud Waste Prevention," IEEE Access, vol. 13, pp. 40561–40570, 2025.
[11] R. Mehta and S. Kulkarni, "CostGuard: Python-Based Cloud State Detection System for Developer Environments," IEEE Access, vol. 14, pp. 11200–11215, 2026.
[12] A. Verma and K. Srinivas, "SecureCloud: Intelligent Infrastructure Management Using Boto3," IEEE Transactions on Dependable and Secure Computing, vol. 23, no. 2, pp. 150–162, 2026.
[13] H. Kumar and R. Singh, "Intelligent Cloud Resource Provisioning and Cost Optimization Using AWS," IEEE Access, vol. 14, pp. 88910–88925, 2026.
[14] M. Armbrust et al., "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50–58, Apr. 2010.
[15] AWS Architecture Center, "AWS Well-Architected Framework: Cost Optimization Pillar," Amazon Web Services, Whitepaper, 2023.



INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INNO SPACE SJIF Scientific Journal Impact Factor

doi crossref

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

9940 572 462 6381 907 438 ijircce@gmail.com

www.ijircce.com

Scan to save the contact details