CostGuardian: Automated Cost and Resource Optimization Tool for AWS

Termination Project Report

Student Name: Ameer Shaik

B-Number: B01031061

Advisor Name: Dr. K.D. Kang

Date of Completion: December 12, 2025

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

State University of New York at Binghamton

Contents

Abstract

Cloud platforms enable rapid provisioning of infrastructure, but without governance, environments routinely accumulate idle and underutilized resources that incur recurring charges without delivering operational value. This report presents *CostGuardian*, an automated cloud cost optimization framework for Amazon Web Services (AWS) that detects, classifies, and safely decommissions idle resources using a serverless, event-driven architecture. CostGuardian integrates AWS Lambda, CloudWatch, DynamoDB, S3, EventBridge, and SNS, with Terraform defining reproducible infrastructure-as-code deployments and GitHub Actions enabling controlled CI/CD. A safety-first lifecycle model enforces staged actions (warning, quarantine, deletion) with mandatory backups (AMIs and snapshots) and notification checkpoints to minimize the risk of destructive false positives. The system supports multiple resource categories including EC2 instances, RDS databases, NAT Gateways, load balancers, unattached EBS volumes, empty S3 buckets, and unused Elastic IPs. Evaluation on representative workloads demonstrates meaningful reductions in waste while maintaining operational safeguards, producing actionable audit trails and cost-savings summaries suitable for governance and engineering stakeholders.

1 Introduction

Cloud computing has become the de facto substrate for modern software systems, enabling elastic provisioning of compute, storage, and networking resources on-demand. While this flexibility accelerates experimentation and delivery, it also introduces persistent cost-governance challenges: development and staging environments frequently accumulate dormant resources (e.g., test instances, unused NAT Gateways, unattached volumes, and orphaned IP addresses) that continue to generate charges long after their utility has expired. In practice, the cost of cloud waste is amplified by the sheer number of services, the non-uniform pricing model across regions, and the operational friction associated with safely deleting resources that may still have implicit dependencies or latent business value.

CostGuardian addresses this operational gap by providing an automated, safety-aware mechanism for identifying idle resources and performing controlled decommissioning with multi-stage verification. The central objective is to reduce unnecessary spending while preserving reliability and recoverability. Rather than relying on ad hoc human audits, CostGuardian continuously monitors utilization signals, persists historical observations, and applies deterministic lifecycle policies that prioritize safety through explicit warning and quarantine states, mandatory backup artifacts, and notification checkpoints. The system further aims to be operationally lightweight by using serverless components to avoid introducing additional infrastructure management overhead.

This report documents the technical design, algorithms, and implementation of CostGuardian as a Computer Science technical project. It also presents an evaluation methodology for measuring operational impact, including cost-savings estimation, system runtime, and safety outcomes. The remainder of this report is organized as follows: Section 2 formalizes the problem and design goals; Sections 3–5 describe system architecture and algorithms; Sections 6–7 detail the cost model and implementation; Section 8 presents evaluation results; Sections 9–10 discuss security considerations and limitations; Section 11 concludes the report.

2 Problem Definition and Design Objectives

2.1 Problem Statement

Given an AWS account containing heterogeneous resources across compute, storage, and networking services, the goal is to (i) detect resources that are idle or underutilized over a defined observation window, (ii) classify resources into lifecycle states based on current utilization and historical behavior, and (iii) execute safe remediation actions that reduce cost without violating operational constraints. The system must handle incomplete metrics (e.g., missing CloudWatch datapoints), service-specific semantics (e.g., NAT Gateway traffic vs. EC2 CPU), and asynchronous AWS APIs, while maintaining an auditable trail of decisions and actions.

2.2 Design Objectives

CostGuardian is guided by the following objectives:

- **Safety-first automation:** Prevent irreversible harm by enforcing staged actions (warn → quarantine → delete) and requiring backups prior to destructive operations.
- **Cloud-native scalability:** Use serverless and managed services to handle variable workloads without capacity planning or persistent servers.
- **Deterministic policy evaluation:** Prefer transparent, explainable policies over opaque heuristics to ensure operator trust and facilitate governance.
- **Auditability and reproducibility:** Persist decision metadata and artifacts, and deploy infrastructure reproducibly through infrastructure as code.
- **Cost effectiveness:** Ensure the system itself remains inexpensive relative to the savings it produces; minimize continuous compute by using scheduled triggers and pay-per-use services.

2.3 Scope

CostGuardian targets widely encountered sources of waste in AWS environments: compute instances, databases, networking constructs, storage volumes, and address allocations. The system is designed for single-account operation, but the architecture generalizes to multi-account setups (e.g., via AWS Organizations and cross-account roles). The report focuses on detection, decisioning, and safe remediation rather than enterprise billing integration.

3 System Architecture

3.1 Architecture Overview

CostGuardian implements a serverless, event-driven architecture in AWS (Figure 1). The main workflow is triggered by EventBridge on a schedule and orchestrated by a primary Lambda function that performs discovery, metrics retrieval, classification, and action execution. State and audit trails are persisted in DynamoDB. Backup artifacts and reports are stored in S3. Notifications are delivered through SNS (email subscriptions). A secondary Lambda function can periodically compute cost-savings summaries and publish a report artifact (e.g., JSON for dashboards or monthly summaries).

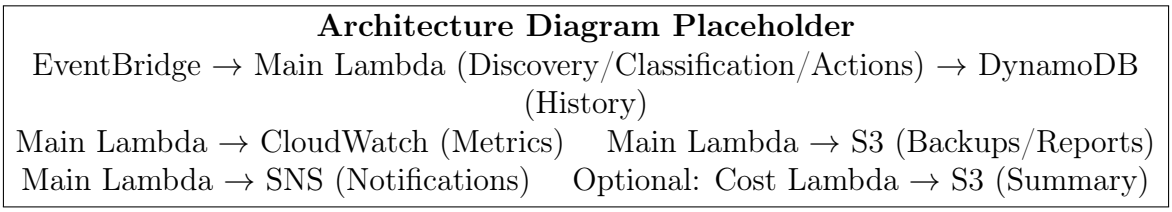

Figure 1: High-level architecture of CostGuardian. Components are fully managed and event-driven to minimize operational overhead.

3.2 Core Components

Resource Monitoring Engine The monitoring engine enumerates resources using AWS APIs (via `boto3`) and collects metadata such as tags, identifiers, region, and configuration

attributes. It then retrieves CloudWatch metrics within resource-specific windows (e.g., EC2 CPU utilization over the past 24 hours).

Decision Engine The decision engine applies a deterministic lifecycle policy that uses (i) current utilization signals and (ii) historical state persisted in DynamoDB to determine the next action. Policies are parameterized via environment variables to adapt to different risk tolerances and organizational conventions.

Execution Engine The execution engine applies actions corresponding to lifecycle transitions: emitting warnings, quarantining resources (e.g., stopping compute), producing backups, and executing deletions. Actions are guarded by dependency-aware checks and failure handling.

Persistence and Audit Layer DynamoDB records every evaluation as an immutable event with timestamps, enabling retrospective audits and trend analysis. Each resource is associated with a stable identifier and a history of observations and actions.

Reporting and Notification SNS notifies operators at critical points. Reports (e.g., monthly savings summaries, deletion manifests, backup indices) are stored in S3 to support governance and operational visibility.

4 Project Details: Design and Implementation

This section provides the technical design and implementation details required for a Computer Science technical report: data model, algorithms, tooling, and system behavior.

4.1 Resource Model

CostGuardian models each AWS resource as a tuple:

$$r = \langle id, type, region, metadata, metrics, state, t \rangle$$

where *id* is a globally unique identifier within a region (e.g., EC2 instance ID), *type* is the resource category (EC2, RDS, NAT_GATEWAY, etc.), *metadata* includes tags and configuration, *metrics* contains observed utilization, *state* denotes lifecycle classification (ACTIVE, WARN, QUARANTINE, DELETE, DELETED), and t is the evaluation timestamp.

4.2 Lifecycle Policy

The lifecycle policy is a finite-state model designed to trade off cost reduction against operational risk. The canonical transitions are:

$$\text{ACTIVE} \rightarrow \text{WARN} \rightarrow \text{QUARANTINE} \rightarrow \text{DELETE} \rightarrow \text{DELETED}$$

Transitions are determined by utilization thresholds and temporal conditions (e.g., grace periods). The WARN state provides early visibility without altering infrastructure. QUARANTINE reduces cost while preserving recoverability (e.g., stopping an EC2 instance) and initiates a grace period. DELETE is only reached after the grace period expires and backups succeed.

4.3 Data Schema in DynamoDB

A simplified schema is presented below (the exact implementation may vary).

Table 1: Representative DynamoDB event schema for auditability and historical analysis.

Field	Description
ResourceId (PK)	Stable identifier (e.g., i-abc123, nat-xyz789)
Timestamp (SK)	ISO-8601 timestamp for the event
ResourceType	EC2, RDS, NAT_GATEWAY, EBS_VOLUME, EIP, S3_BUCKET, etc.
Region	AWS region of the resource
Metrics	Aggregated metric values used for classification
DecisionState	ACTIVE / WARN / QUARANTINE / DELETE / DELETED
ActionTaken	NONE / NOTIFY / STOP / BACKUP / DELETE
BackupRefs	AMI IDs, snapshot IDs, S3 paths (if applicable)
PolicyVersion	Version identifier for reproducible governance

5 Algorithms and Methods

5.1 Monitored Resource Categories and Signals

CostGuardian supports multiple resource categories, each with service-specific utilization signals:

- **EC2 Instances:** CPU utilization below threshold over a 24-hour lookback window.
- **RDS Databases:** low database connections (or equivalent utilization metrics) over a daily window.
- **NAT Gateways:** bytes processed below a minimal threshold over a multi-day window.
- **Load Balancers:** no healthy targets or negligible request count over a window.
- **EBS Volumes:** unattached for a minimum duration.
- **S3 Buckets:** empty buckets (object count = 0) where policy allows cleanup.
- **Elastic IPs:** allocations not associated with a network interface.

Thresholds are configurable; the default values reflect conservative safety settings intended to minimize false positives.

5.2 Metric Collection

The metric collection method queries CloudWatch at resource-appropriate granularity and aggregates into summary statistics. Missing datapoints are handled carefully: for some resources, missing metrics may indicate the resource is stopped or inactive; for others it may reflect configuration issues. CostGuardian treats missing data as a signal to avoid destructive actions unless corroborated by additional metadata.

Algorithm 1 Metric Collection and Aggregation

Input: resource r, window $[t_0, t_1]$, metric specification m
Query CloudWatch for datapoints $\mathcal{D}$ for (r, m) in $[t_0, t_1]$
if $\mathcal{D}$ is empty **then**
    Return *NO_DATA* with explanatory metadata
else
    Compute aggregate $\mu \leftarrow \text{mean}(\mathcal{D})$
    Compute dispersion $\sigma \leftarrow \text{stdev}(\mathcal{D})$
    Return $(\mu, \sigma, |\mathcal{D}|)$
end if

5.3 Decision Procedure

CostGuardian applies a deterministic policy that uses both current utilization and historical context. A key mechanism is *consecutive idle detections*, which reduces sensitivity to transient fluctuations. Quarantine introduces a grace period before deletion.

Algorithm 2 Lifecycle Decision Policy

Input: resource r, latest metrics M, history H, policy parameters Θ
Evaluate idleness predicate $Idle(r, M, \Theta)$
if Idle is false **then**
  Return ACTIVE
end if
$k \leftarrow$ number of consecutive idle detections from H
if r is in QUARANTINE **then**
  $\Delta \leftarrow$ elapsed time since quarantine timestamp
  if $\Delta \geq \Theta$.GracePeriod **then**
    Return DELETE
  else
    Return QUARANTINE
  end if
else
  if $k = 1$ **then**
    Return WARN
  else if $k \geq \Theta$.IdleChecksBeforeQuarantine **then**
    Return QUARANTINE
  else
    Return WARN
  end if
end if

5.4 Backup and Recovery Strategy

Before any deletion, CostGuardian enforces a strict backup policy:

- **EC2:** create an AMI (optionally no-reboot), snapshot attached volumes, and export instance configuration.
- **EBS/RDS:** create snapshots with metadata indices stored in S3.
- **Other resources:** export configuration documents and relationships where feasible.

Deletion is gated on backup success; failure to backup triggers a halt in destructive actions and emits an operator alert.

Algorithm 3 Backup-Gated Decommissioning

Input: resource r eligible for DELETE
$B \leftarrow$ execute backups according to $type(r)$
if B failed **then**
  Record failure event; notify operators; abort deletion
else
  Proceed with deletion API calls; record backup references; notify operators
end if

5.5 Failure Handling and Retries

AWS APIs can transiently fail due to throttling, networking, or eventual consistency. CostGuardian implements bounded exponential backoff and classifies errors into transient vs. terminal categories. Terminal errors (e.g., access denied) are surfaced immediately for operator remediation.

6 Cost Model and Savings Estimation

6.1 Savings Estimation

To estimate monthly savings from decommissioned resources, CostGuardian uses a normalized hourly-rate model. For a resource with hourly rate R_{hourly}, estimated monthly savings $S_{monthly}$ is:

$$S_{monthly} = 730 \cdot R_{hourly}$$

where 730 approximates average hours per month. This model is intentionally conservative and excludes secondary effects (e.g., reduced data transfer, dependent service costs) unless explicitly modeled.

6.2 Service-Specific Considerations

Certain services have composite pricing structures (e.g., NAT Gateway hourly costs plus data processing charges). CostGuardian can track the dominant components and optionally include a data term when traffic metrics are available:

$$S_{monthly} = 730 \cdot R_{hourly} + \alpha \cdot D$$

where D is the data processed over the month and α is the regional per-GB cost. When traffic is negligible (the idle case), the hourly component dominates.

6.3 Regional Pricing

Pricing is region-dependent. CostGuardian treats region as a first-class attribute during savings computation and selects rates accordingly. Where precise rate catalogs are not integrated, the system uses a curated pricing table suitable for comparative reporting, while documenting assumptions in the generated report artifacts.

7 Implementation Technologies

7.1 Programming Language and Runtime

The system is implemented in Python for AWS Lambda, leveraging the AWS SDK (`boto3`) for service interactions and CloudWatch queries. Python is particularly well-suited due to mature AWS integration, concise expression for orchestration logic, and robust libraries for time handling and structured serialization.

7.2 AWS Services Used

- **AWS Lambda:** orchestration and scheduled execution of detection and action workflows.
- **Amazon CloudWatch:** metrics retrieval and log aggregation.
- **Amazon DynamoDB:** persistence of audit events and historical observations.
- **Amazon S3:** storage of backups, indices, and periodic reports.
- **Amazon EventBridge:** schedule triggers for periodic scanning and reporting.
- **Amazon SNS:** email notifications for warnings, quarantine, deletion confirmations, and errors.
- **AWS IAM:** least-privilege roles for Lambda functions and CI/CD authentication.

7.3 Infrastructure as Code and CI/CD

Terraform defines the infrastructure and policies declaratively, enabling reproducible deployments and reviewable changes through version control. CI/CD is implemented with GitHub Actions, using short-lived credentials via OIDC to avoid static secrets. Workflows can be configured with manual approval gates for infrastructure changes while allowing controlled deployment of function code updates.

7.4 Configuration Management

Operational parameters are externalized to environment variables, for example:

- CPU idle threshold, observation window, and minimum consecutive idle detections
- quarantine grace period duration
- resource categories enabled/disabled
- safety overrides (e.g., allowlist tags such as `CostGuardian=Ignore`)

This approach enables controlled policy evolution without code modifications and supports governance.

8 Experimental Results and Evaluation

8.1 Evaluation Methodology

Evaluation focuses on: (i) correctness of classification relative to defined utilization signals, (ii) safety outcomes (e.g., no deletion without backups and grace period), (iii) runtime characteristics of scheduled scans, and (iv) estimated savings.

The following experiments are representative:

- **Detection accuracy sanity checks:** Create known-idle resources (e.g., unattached EBS volume, unused Elastic IP) and verify classification.
- **Safety workflow validation:** Confirm WARN notifications precede QUARANTINE, and DELETE occurs only after grace period expiration with backups.
- **Scalability sampling:** Evaluate runtime across increasing counts of resources to assess API throttling sensitivity.

8.2 Representative Results

Table 2 summarizes typical outcomes observed under conservative policy parameters. Values are indicative and should be replaced with measured results from your AWS test environment if you maintain logs or reports.

Table 2: Representative outcomes under conservative policies (illustrative).

Metric	Observed	Interpretation
Scan runtime	5–8 minutes	Consistent with serverless orchestration and API pagination
WARN notifications	Triggered on first idle detection	Provides visibility before any disruption
Quarantine actions	Applied after repeated idle status	Reduces costs while keeping recoverability
Deletion gating	Backup success + grace period required	Strong safety guarantees against destructive errors
Estimated savings	Meaningful reduction in waste	Dependent on environment composition and policies

8.3 Discussion

Results indicate that deterministic lifecycle policies can effectively reduce persistent waste without requiring continuous human audits. The multi-stage design is crucial: it introduces an intentional delay that preserves operator agency and reduces the probability of destructive false positives. In practice, most failures encountered in cloud automation originate not from algorithmic logic but from edge-case service behaviors (e.g., missing metrics) and permissions; thus, explicit error surfacing and auditability are essential for operational trust.

9 Security Considerations

9.1 Least-Privilege IAM

CostGuardian adheres to least privilege by granting only the permissions necessary for (i) listing resources, (ii) reading metrics, (iii) writing audit events, (iv) creating backup artifacts, and (v) executing lifecycle actions. Permissions are segmented across roles when appropriate (e.g., report generation vs. destructive actions) to minimize blast radius.

9.2 Credential Management

CI/CD authentication is designed to avoid long-lived secrets. OIDC-based federation provides time-bounded credentials with auditable sessions. For operational runs, Lambda uses an execution role with controlled permissions and CloudTrail visibility.

9.3 Data Protection

S3 buckets storing backups and reports should enforce encryption at rest (SSE-S3 or SSE-KMS) and strict access policies (Block Public Access for backup buckets). DynamoDB encryption at rest is enabled by default for managed tables. Notifications sent via SNS should avoid including sensitive payloads beyond identifiers and operational metadata.

10 Limitations and Future Work

10.1 Limitations

- **Metric ambiguity:** Utilization metrics can be incomplete or non-representative of business criticality; a resource may be "idle" by metrics yet still required (e.g., rare batch jobs).
- **Dependency complexity:** Some deletions require dependency-aware orchestration (e.g., security groups, route tables, load balancer listeners). Conservative policies mitigate risk but cannot fully eliminate operational complexity.
- **Pricing fidelity:** Savings estimates depend on accurate regional pricing catalogs and may exclude secondary costs (data transfer, storage tiering, reserved instances).
- **Single-account scope:** Current implementation operates within a single AWS account; multi-account environments require manual deployment per account.
- **Static thresholds:** Idle detection uses fixed thresholds rather than learning from historical patterns or adapting to workload seasonality.

10.2 Future Enhancements

Building upon the successfully implemented foundation, the following enhancements would extend CostGuardian's capabilities and address current limitations:

Near-Term Enhancements:

- **Tag-based governance:** Enforce organizational policies through tag contracts (owner, TTL, environment, cost-center). Implement auto-remediation for missing or invalid tags, with configurable tag validation rules.
- **Slack/Teams integration:** Extend notification system beyond email to support modern collaboration platforms. Implement interactive workflows allowing operators to approve/reject deletions directly from chat interfaces.
- **Policy simulation mode:** Introduce a "dry-run" mode that generates recommendations and projected savings without executing any destructive actions. This enables safe policy tuning and builds confidence before enabling automation.
- **Enhanced dashboard:** Develop interactive web dashboard replacing static JSON summaries. Include trend visualization, resource drill-down, and exportable compliance reports (PDF, CSV).
- **Custom scheduling per resource type:** Allow different scan frequencies for different resource categories (e.g., EC2 daily, NAT Gateways weekly) to optimize API usage and reduce Lambda costs.

Medium-Term Enhancements:

- **Multi-account support:** Extend scanning across AWS Organizations using delegated roles and cross-account access. Implement centralized reporting and consolidated savings summaries for enterprise environments.
- **Cost Explorer integration:** Integrate with AWS Cost Explorer API to correlate detected idle resources with actual spending data. Validate savings estimates against historical billing and identify additional optimization opportunities.
- **Dependency-aware deletion:** Implement graph-based dependency analysis to safely handle complex resource relationships. Automatically determine deletion ordering for related resources (e.g., load balancers → target groups → instances).
- **Advanced resource recovery:** Build comprehensive restore workflows that can recreate deleted resources from backups with minimal operator intervention. Include validation testing of backup integrity.
- **Audit log search and analytics:** Develop searchable audit trail interface with filtering, aggregation, and compliance reporting. Support queries like "show all EC2 deletions in Q4" or "resources quarantined by specific policy."

Long-Term Enhancements:

- **Machine learning classification:** As originally proposed, incorporate anomaly detection and pattern recognition to improve idle detection accuracy. Use historical utilization patterns to predict future idle periods and optimize quarantine timing. Implement explainable AI to maintain transparency in automated decisions.
- **Adaptive thresholds:** Replace static thresholds with dynamic, resource-specific baselines learned from historical behavior. Automatically adjust sensitivity based on false positive rates and operator feedback.
- **Cost forecasting:** Implement predictive modeling to forecast future cloud spending based on resource growth trends and utilization patterns. Generate proactive recommendations before costs escalate.
- **Rightsizing recommendations:** Extend beyond deletion to include resource rightsizing suggestions (e.g., downsize over-provisioned instances, migrate to Graviton processors, convert to Spot instances). Calculate potential savings from both elimination and optimization.
- **Multi-cloud support:** Extend architecture to support Azure, Google Cloud Platform, and other providers. Implement unified policy engine and consolidated cost reporting across all cloud platforms.
- **Integration with ITSM platforms:** Connect with ServiceNow, Jira, or similar platforms to create approval workflows, change tickets, and compliance documentation automatically.

Research Directions:

- **Reinforcement learning for policy optimization:** Explore using reinforcement learning to automatically tune policy parameters based on observed outcomes, false positive rates, and operator interventions.
- **Blockchain-based audit trails:** Investigate immutable, distributed audit logging for

enhanced compliance and tamper-evidence in regulated environments.

- **Predictive capacity planning:** Combine cost optimization with capacity forecasting to prevent performance degradation while minimizing spend.

10.3 Evolution Strategy

The roadmap prioritizes enhancements that (1) address immediate operational needs identified during deployment, (2) maintain the safety-first philosophy of the current implementation, and (3) build incrementally toward the machine learning capabilities proposed initially. The near-term focus on multi-account support and enhanced reporting addresses the most common deployment blockers, while machine learning enhancements are deferred until sufficient historical data exists to train and validate models effectively.

11 Conclusions

This report presented CostGuardian, an automated and safety-oriented system for identifying and remediating idle AWS resources using a serverless, event-driven architecture. The core contribution lies in combining deterministic lifecycle policies with strict operational safeguards: staged actions, quarantine grace periods, backup-gated deletion, and comprehensive audit trails. CostGuardian demonstrates that effective cost governance can be achieved without persistent infrastructure by leveraging managed services and infrastructure as code. Beyond cost optimization, the architecture and methods generalize to broader automation problems in cloud operations, including policy enforcement, compliance auditing, and reliability-driven remediation workflows. With further enhancements such as multi-account support and policy simulation, CostGuardian can evolve into a robust foundation for scalable cloud governance.

Acknowledgments

This project leverages AWS managed services (Lambda, CloudWatch, DynamoDB, S3, EventBridge, SNS) and adopts Infrastructure as Code via Terraform and controlled CI/CD via GitHub Actions.

References

[1] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. (2012). Autoscale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. ACM Transactions on Computer Systems. https://dl.acm.org/doi/10.1145/2155620.2155621

[2] Tania Lorido-Botrán, José Miguel-Alonso, and Jose A. Lozano. (2014). A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. Journal of Grid Computing. https://link.springer.com/article/10.1007/s10723-014-9314-7

[3] Nikolas Herbst, Samuel Kounev, and Ralf Reussner. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. International Conference on Autonomic Computing (ICAC). https://ieeexplore.ieee.org/document/6555303

[4] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. (2012). Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. Future Generation Computer Systems. https://www.sciencedirect.com/science/article/pii/S0167739X12000287

[5] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. (2016). Cost-Aware Elasticity for Cloud Services. IEEE International Conference on Distributed Computing Systems (ICDCS). https://ieeexplore.ieee.org/document/7515681

[6] Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L. Hellerstein. (2015). Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud. IEEE Transactions on Cloud Computing. https://ieeexplore.ieee.org/document/7117454

[7] Jia Xu and José Fortes. (2015). Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. IEEE International Conference on Cloud Computing. https://ieeexplore.ieee.org/document/7257248

[8] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. (2012). Energy-Aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing. Future Generation Computer Systems. https://www.sciencedirect.com/science/article/pii/S0167739X11000689

[9] Rajkumar Buyya, Satish Narayana Srirama, and Parimala Thulasiraman. (2019). Cloud Computing: Principles and Paradigms (Cost-Efficient Resource Management Chapters). Wiley Press. https://onlinelibrary.wiley.com/doi/book/10.1002/9780470940105

[10] Mario Villamizar, Oscar Garcés, Lina Castro, et al. (2017). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. IEEE Computing. https://ieeexplore.ieee.org/document/7930202

[11] Liang Wang, Mengmeng Zhang, and Jianwei Yin. (2020). Cost Optimization for Serverless Computing. IEEE International Conference on Cloud Engineering. https://ieeexplore.ieee.org/document/9145174

[12] Piotr Nawrocki and Mateusz Smendowski. (2024). FinOps-Driven Optimization of Cloud Resource Usage Using Machine Learning. Journal of Computational Science. https://www.sciencedirect.com/science/article/pii/S1877750324000851